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*: $a \in set\ as \Longrightarrow \exists\, i.\ i < length\ as \wedge a = as\ !\ i$
unfolding *set-conv-nth* **by** *auto*

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

Every bijective function between two 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 −
have *finite B* **using** *assms* **by** (*metis bij-betw-finite*)
have [*simp*]: *card A* = *card B* **by** (*metis* ∗ *bij-betw-same-card*)
have *card* $(A - B) = card\ (B - A)$
proof −
have *card* $(A - B) = card\ A - card\ (A \cap B)$
by (*metis* ‹*finite A*› *card-Diff-subset-Int finite-Int*)
moreover have *card* $(B - A) = card\ B - card\ (A \cap B)$
by (*metis* ‹*finite A*› *card-Diff-subset-Int finite-Int inf-commute*)
ultimately show *?thesis* **by** *simp*
qed
then obtain *g* **where** ∗∗: *bij-betw g* $(B - A)$ $(A - B)$
by (*metis* ‹*finite A*› ‹*finite B*› *bij-betw-iff-card finite-Diff*)
define *h* **where** $h = (\lambda x.\ if\ x \in A\ then\ f\ x\ else\ if\ x \in B - A\ then\ g\ x\ else\ x)$
have *bij-betw h A B*
by (*metis* (*full-types*) ∗ *bij-betw-cong h-def*)
moreover have *bij-betw h* $(S - (A \cup B))$ $(S - (A \cup B))$
by (*auto simp*: *bij-betw-def h-def inj-on-def*)
moreover have $B \cap (S - (A \cup B)) = \{\}$ **by** *blast*
ultimately have *bij-betw h* $(A \cup (S - (A \cup B)))$ $(B \cup (S - (A \cup B)))$
by (*rule bij-betw-combine*)
moreover have $A \cup (S - (A \cup B)) = S - (B - A)$
and $B \cup (S - (A \cup B)) = S - (A - B)$
using ‹$A \subseteq S$› **and** ‹$B \subseteq S$› **by** *blast*+
ultimately have *bij-betw h* $(S - (B - A))$ $(S - (A - B))$ **by** *simp*
moreover have *bij-betw h* $(B - A)$ $(A - B)$

```
    using ** by (auto simp: bij-betw-def h-def inj-on-def)
  moreover have (S − (A − B)) ∩ (A − B) = {} by blast
  ultimately have bij-betw h ((S − (B − A)) ∪ (B − A)) ((S − (A − B)) ∪ (A
− B))
    by (rule bij-betw-combine)
  moreover have (S − (B − A)) ∪ (B − A) = S
    and (S − (A − B)) ∪ (A − B) = S
    using ‹A ⊆ S› and ‹B ⊆ S› by auto
  ultimately have bij-betw h S S by simp
  moreover have ∀ x∈A. h x = f x by (auto simp: h-def)
  moreover have finite {x. h x ≠ x}
  proof −
    have finite (A ∪ (B − A)) using ‹finite A› and ‹finite B› by auto
    moreover have {x. h x ≠ x} ⊆ (A ∪ (B − A)) by (auto simp: h-def)
    ultimately show ?thesis by (metis finite-subset)
  qed
  moreover have ∀ x∈UNIV − (A ∪ B). h x = x by (simp add: h-def)
  ultimately show ?thesis by blast
qed

lemma concat-nth:
  assumes m < length xs and n < length (xs ! m)
    and i = sum-list (map length (take m xs)) + n
  shows concat xs ! i = xs ! m ! n
  using assms
proof (induct xs arbitrary: m n i)
  case (Cons x xs)
  show ?case
  proof (cases m)
    case 0
    then show ?thesis using Cons by (simp add: nth-append)
  next
    case (Suc k)
    with Cons(1) [of k n i − length x] and Cons(2−)
    show ?thesis by (simp-all add: nth-append)
  qed
qed simp

lemma concat-nth-length:
  i < length uss ⟹ j < length (uss ! i) ⟹
    sum-list (map length (take i uss)) + j < length (concat uss)
proof (induct uss arbitrary: i j)
  case (Cons u uss i j)
  thus ?case by (cases i, auto)
qed auto

lemma less-length-concat:
  assumes i < length (concat xs)
  shows ∃ m n.
```

```
    i = sum-list (map length (take m xs)) + n ∧
    m < length xs ∧ n < length (xs ! m) ∧ concat xs ! i = xs ! m ! n
  using assms
proof (induct xs arbitrary: i rule: length-induct)
  case (1 xs)
  then show ?case
  proof (cases xs)
    case (Cons y ys)
    note [simp] = this
    { assume *: i < length y
      with 1 obtain n where i = n and n < length y
        and y ! i = y ! n by simp
      then have i = sum-list (map length (take 0 xs)) + n
        and 0 < length xs and n < length (xs ! 0)
        and concat xs ! i = xs ! 0 ! n
        using * by (auto simp: nth-append)
      then have ?thesis by blast }
    moreover
    { assume *: i ≥ length y
      define j where j = i − length y
      then have length ys < length xs j < length (concat ys)
        using * and 1.prems by auto
      with 1 obtain m n where j = sum-list (map length (take m ys)) + n
        and m < length ys and n < length (ys ! m)
        and concat ys ! j = ys ! m ! n by blast
      then have i = sum-list (map length (take (Suc m) xs)) + n
        and Suc m < length xs and n < length (xs ! Suc m)
        and concat xs ! i = xs ! Suc m ! n
        using * by (simp-all add: j-def nth-append)
      then have ?thesis by blast }
    ultimately show ?thesis by force
  qed simp
qed

lemma concat-remove-nth:
  assumes i < length sss
    and j < length (sss ! i)
  defines k ≡ sum-list (map length (take i sss)) + j
 shows concat (take i sss @ remove-nth j (sss ! i) # drop (Suc i) sss) = remove-nth
k (concat sss)
  using assms
  unfolding remove-nth-def
proof (induct sss rule: List.rev-induct)
  case Nil then show ?case by auto
next
  case (snoc ss sss)
  then have i = length sss ∨ i < length sss by auto
  then show ?case
  proof
```

```
    assume i:i = length sss
   have sum-list (map length sss) = length (concat sss) by (simp add: length-concat)
    with snoc i show ?thesis by simp
  next
    assume i:i < length sss
    then have nth:(sss @ [ss]) ! i = sss ! i by (simp add: nth-append)
    from i have drop:drop (Suc i) (sss @ [ss]) = drop (Suc i) sss @ [ss] by auto
    from i have take:take i (sss @ [ss]) = take i sss by auto
    from snoc(1)[OF i] snoc(2-) have 1:concat (take i (sss @ [ss]) @
      (take j ((sss @ [ss]) ! i) @ drop (Suc j) ((sss @ [ss]) ! i)) # drop (Suc i) (sss
@ [ss])) =
      take k (concat sss) @ drop (Suc k) (concat sss) @ ss unfolding take nth drop
by simp
    from snoc(4) take have k:k = sum-list (map length (take i sss)) + j by auto
    from nth snoc(3) have j: j < length (sss ! i) by auto
    have takek:take (sum-list (map length (take i sss)) + j) (concat (sss @ [ss])) =

      take (sum-list (map length (take i sss)) + j) (concat sss)
      using concat-nth-length[OF i j] by auto
    have dropk:drop (Suc (sum-list (map length (take i sss)) + j)) (concat sss) @
ss =
      drop (Suc (sum-list (map length (take i sss)) + j)) (concat (sss @ [ss]))
      using concat-nth-length[OF i j] by auto
    have take k (concat sss) @ drop (Suc k) (concat sss) @ ss =
      take k (concat (sss @ [ss])) @ drop (Suc k) (concat (sss @ [ss]))
      unfolding k takek dropk ..
    with 1 show ?thesis by auto
  qed
qed

lemma nth-append-Cons: (xs @ y # zs) ! i =
  (if i < length xs then xs ! i else if i = length xs then y else zs ! (i - Suc (length
xs)))
  by (cases i length xs rule: linorder-cases, auto simp: nth-append)

lemma finite-imp-eq [simp]:
  finite {x. P x ⟶ Q x} ⟷ finite {x. ¬ P x} ∧ finite {x. Q x}
  by (auto simp: Collect-imp-eq Collect-neg-eq)

lemma sum-list-take-eq:
  fixes xs :: nat list
  shows k < i ⟹ i < length xs ⟹ sum-list (take i xs) =
    sum-list (take k xs) + xs ! k + sum-list (take (i - Suc k) (drop (Suc k) xs))
  by (subst id-take-nth-drop [of k]) (auto simp: min-def drop-take)

lemma nth-equalityE:
  xs = ys ⟹ (length xs = length ys ⟹ (⋀i. i < length xs ⟹ xs ! i = ys ! i)
⟹ P) ⟹ P
  by simp
```

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

lemma *fold-map-cong* [*fundef-cong*]:
  assumes *a* = *b* **and** *xs* = *ys*
    and ⋀*x*. *x* ∈ *set xs* ⟹ *f x* = *g x*
  shows *fold-map f xs a* = *fold-map g ys b*
  using *assms* **by** (*induct ys arbitrary*: *a b xs*) *simp-all*

lemma *fold-map-map-conv*:
  assumes ⋀*x ys*. *x* ∈ *set xs* ⟹ *f* (*g x*) (*g' x* @ *ys*) = (*h x*, *ys*)
  shows *fold-map f* (*map g xs*) (*concat* (*map g' xs*) @ *ys*) = (*map h xs*, *ys*)
  using *assms* **by** (*induct xs*) *simp-all*

lemma *map-fst-fold-map*:
  map f (*fst* (*fold-map g xs y*)) = *fst* (*fold-map* (λ*a b*. *apfst f* (*g a b*)) *xs y*)
  by (*induct xs arbitrary*: *y*) (*auto split*: *prod.splits*, *metis fst-conv*)

lemma *not-Nil-imp-last*: *xs* ≠ [] ⟹ ∃ *ys y*. *xs* = *ys*@[*y*]
proof (*induct xs*)
  case *Nil* **then show** *?case* **by** *simp*
next
  case (*Cons x xs*) **show** *?case*
  proof (*cases xs*)
    assume *xs* = [] **with** *Cons* **show** *?thesis* **by** *simp*
  next
    fix *x' xs'* **assume** *xs* = *x'*#*xs'*
    then have *xs* ≠ [] **by** *simp*
    with *Cons* **obtain** *ys y* **where** *xs* = *ys*@[*y*] **by** *auto*
    then have *x*#*xs* = *x*#(*ys*@[*y*]) **by** *simp*
    then have *x*#*xs* = (*x*#*ys*)@[*y*] **by** *simp*
    then show *?thesis* **by** *auto*
  qed
qed

lemma *Nil-or-last*: *xs* = [] ∨ (∃ *ys y*. *xs* = *ys*@[*y*])
  using *not-Nil-imp-last* **by** *blast*

2.1 Combinators

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

definition *flip* :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow ('b \Rightarrow 'a \Rightarrow 'c)$ **where**
  flip f ≡ (λ*x y*. *f y x*)

declare *flip-def*[*simp*]

lemma *const-apply*[*simp*]: *const x y* = *x*
by (*simp add*: *const-def*)

lemma *foldr-Cons-append-conv* [*simp*]:
foldr (#) *xs ys* = *xs* @ *ys*
by (*induct xs*) *simp-all*

lemma *foldl-flip-Cons*[*simp*]:
foldl (*flip* (#)) *xs ys* = *rev ys* @ *xs*
by (*induct ys arbitrary*: *xs*) *simp-all*

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*
by (*simp add*: *foldr-conv-foldl*)

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*
by (*simp add*: *foldl-conv-foldr*)

fun *debug* :: (*String.literal* ⇒ *String.literal*) ⇒ *String.literal* ⇒ *'a* ⇒ *'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*
using *assms* **proof**(*induct xs*)
case (*Cons x xs*)
then have *IH*:*distinct xs*
by (*metis dual-order.trans filter.simps*(*2*) *impossible-Cons nle-le*)
from *Cons*(*2*) **have** *length* (*filter* ((=) *x*) *xs*) = *0*
by (*metis* (*mono-tags*) *One-nat-def add.right-neutral add-Suc-right filter.simps*(*2*)
le-less length-0-conv less-Suc0 list.simps(*3*) *list.size*(*4*) *nat.inject*)
then have *x* ∉ *set* (*xs*)
by (*metis* (*full-types*) *filter-empty-conv length-0-conv*)
with *IH* **show** *?case*
by *simp*
qed *simp*

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)
  using assms proof(induct xs)
  case (Cons x xs)
  {fix i j assume i < length xs j < length xs i ≠ j f (xs!i) f (xs!j)
    with Cons(2) have xs!i ≠ xs!j
      by (metis not-less-eq nth-Cons-Suc Suc-inject length-Cons)
  }
  with Cons(1) have IH:distinct (filter f xs)
    by presburger
  show ?case proof(cases f x)
    case True
    with Cons(2) have ∀ j<length xs. f (xs ! j) ⟶ x ≠ xs ! j by fastforce
   then have x ∉ set (filter f xs) by (metis filter-set in-set-conv-nth member-filter)
    then show ?thesis unfolding filter.simps using True IH by simp
  next
    case False
    then show ?thesis unfolding filter.simps using IH by presburger
  qed
qed simp

lemma distinct-is-partition:
  assumes distinct xs
  shows is-partition (map (λx. {x}) xs)
  using assms proof(induct xs)
  case (Cons x xs)
  then show ?case unfolding list.map(2) is-partition-Cons by force
qed (simp add: is-partition-Nil)

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)
  by (smt (verit, del-insts) add-diff-inverse-nat assms(1) assms(2) assms(3) dis-
joint-iff is-partition-alt is-partition-alt-def length-append mem-simps(9) nat-add-left-cancel-less
nth-append nth-mem)

lemma distinct-is-partitition-sets:
  assumes distinct xs
    and xs = concat ys
  shows is-partition (map set ys)
  using assms proof(induct ys arbitrary:xs)
  case (Cons y ys)
  have is-partition (map set ys) proof−
    from Cons(2,3) have distinct (concat ys)
      unfolding concat.simps by simp
    with Cons(1) show ?thesis by simp
  qed
  moreover from Cons(2,3) have set y ∩ ⋃ (set (map set ys)) = {}
    using distinct-append[of y concat ys]by simp
```

ultimately show *?case*
unfolding *is-partition-Cons list.map* **by** *simp*
qed (*simp add*: *is-partition-Nil*)

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*
by (*induct t*) *auto*

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*
using *assms* **by** (*cases a* = *b*) (*auto intro*: *subseq-Cons'*)

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 (*induct ys arbitrary*: *xs*)
case *Nil* **then show** *?case* **by** (*metis list-emb-Nil2*)

next
case (*Cons y ys'*)
note *Cons-ys* = *Cons*
note *sl* = *Cons*(*2*)
note *step-eq* = *Cons*(*4*)
note *step-neq* = *Cons*(*5*)
show *?case* **proof** (*cases xs*)
case *Nil* **show** *?thesis* **unfolding** *Nil* **using** *Cons.prems*(*2*) **by** *auto*
next
case (*Cons x xs'*)
have *sl'*: *subseq xs' ys'* **by** (*metis Cons sl subseq-Cons-Cons*)
from *sl'* **have** *P'*: *P xs' ys'* **using** *Cons-ys* **by** *auto*
show *?thesis* **proof** (*cases x* = *y*)
case *False*
have *sl''*: *subseq* (*x* # *xs'*) *ys'* **using** *sl* **unfolding** *Cons* **using** *False* **by** *auto*
then have *P''*: *P* (*x* # *xs'*) *ys'* **by** (*metis Cons.hyps Cons-ys*(*3*) *step-eq step-neq*)
show *?thesis* **using** *step-neq*[*OF False sl' sl'' P' P''*] **unfolding** *Cons* **by** *auto*
next
case *True*
show *?thesis* **using** *step-eq*[*OF sl' P'*] **unfolding** *Cons True*[*symmetric*] **by** *auto*
qed
qed
qed

lemma *subseq-submultiset*:
subseq xs ys ⟹ *mset xs* ⊆# *mset ys*
by (*induct rule*: *list-emb.induct*) (*auto intro*: *subset-mset.order-trans*)

lemma *subseq-subset*:
subseq xs ys ⟹ *set xs* ⊆ *set ys*
by (*induct rule*: *list-emb.induct*) *auto*

lemma *remove1-subseq*:
subseq (*remove1 x xs*) *xs*
by (*induct xs*) *auto*

lemma *subseq-concat*:
assumes ⋀*x*. *x* ∈ *set xs* ⟹ *subseq* (*f x*) (*g x*)
shows *subseq* (*concat* (*map f xs*)) (*concat* (*map g xs*))
using *assms* **by** (*induct xs*) (*auto intro*: *list-emb-append-mono*)

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 (*induct s*)
case (*Fun f ss*)
let *?z* = *zip* [*0*..<*length ss*] (*map poss-list ss*)
have (⋃ *a*∈*set ?z*. *set* (*case-prod* (λ*i*. *map* ((#) *i*)) *a*)) =
{*i* # *p* |*i p*. *i* < *length ss* ∧ *p* ∈ *poss* (*ss* ! *i*)} (**is** *?l* = *?r*)
proof (*rule set-eqI*)
fix *ip*
show (*ip* ∈ *?l*) = (*ip* ∈ *?r*)
proof
assume *ip* ∈ *?l*
from *this* **obtain** *ipI* **where**
z: *ipI* ∈ *set ?z* **and**
ip: *ip* ∈ *set* (*case-prod* (λ *i*. *map* ((#) *i*)) *ipI*)
by *auto*
from *z* **obtain** *i* **where** *i*: *i* < *length ?z* **and** *zi*: *?z* ! *i* = *ipI*
by (*force simp*: *set-conv-nth*)
with *ip Fun* **show** *ip* ∈ *?r* **by** *auto*
next
assume *ip* ∈ *?r*
from *this* **obtain** *i p* **where** *i*: *i* < *length ss* **and** *p* ∈ *poss* (*ss* ! *i*)
and *ip*: *ip* = *i* # *p* **by** *auto*
with *Fun* **have** *p*: *p* ∈ *set* (*poss-list* (*ss* ! *i*)) **and** *iz*: *i* < *length ?z* **by** *auto*
from *i* **have** *id*: *?z* ! *i* = (*i*, *poss-list* (*ss* ! *i*)) (**is** - = *?ipI*) **by** *auto*
from *iz* **have** *?z* ! *i* ∈ *set ?z* **by** (*rule nth-mem*)
with *id* **have** *inZ*: *?ipI* ∈ *set ?z* **by** *auto*

```
    from p have i # p ∈ set (case-prod (λ i. map ((#) i)) ?ipI) by auto
    with inZ ip show ip ∈ ?l by force
   qed
  qed
  with Fun show ?case by simp
qed simp

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 (induct s)
  case (Fun f ss)
  let ?z = zip [0..<length ss] (map var-poss-list ss)
  have (⋃ a∈set ?z. set (case-prod (λi. map ((#) i)) a)) =
       (⋃ i<length ss. {i # p |p. p ∈ var-poss (ss ! i)}) (is ?l = ?r)
  proof (rule set-eqI)
    fix ip
    show (ip ∈ ?l) = (ip ∈ ?r)
    proof
      assume ip ∈ ?l
      from this obtain ipI where
        z: ipI ∈ set ?z and
        ip: ip ∈ set (case-prod (λ i. map ((#) i)) ipI)
        by auto
      from z obtain i where i: i < length ?z and zi: ?z ! i = ipI
        by (force simp: set-conv-nth)
      with ip Fun show ip ∈ ?r by auto
    next
      assume ip ∈ ?r
      from this obtain i p where i: i < length ss and p ∈ var-poss (ss ! i)
        and ip: ip = i # p by auto
       with Fun have p: p ∈ set (var-poss-list (ss ! i)) and iz: i < length ?z by
auto
      from i have id: ?z ! i = (i, var-poss-list (ss ! i)) (is - = ?ipI) by auto
      from iz have  ?z ! i ∈ set ?z by (rule nth-mem)
      with id have inZ: ?ipI ∈ set ?z by auto
      from p have i # p ∈ set (case-prod (λ i. map ((#) i)) ?ipI) by auto
      with inZ ip show ip ∈ ?l by force
    qed
  qed
  with Fun show ?case unfolding var-poss-list.simps by simp
```

qed *simp*

lemma *length-var-poss-list*: *length* (*var-poss-list t*) = *length* (*vars-term-list t*)
proof(*induct t*)
case (*Var x*)
then show *?case* **unfolding** *var-poss-list.simps vars-term-list.simps* **by** *simp*
next
case (*Fun f ts*)
let *?xs*=*map2* (λ*x*. *map* ((#) *x*)) [*0*..<*length ts*] (*map var-poss-list ts*)
let *?ys*=*map vars-term-list ts*
have *l1*:*length ?xs* = *length ts*
by *simp*
have *l2*:*length ?ys* = *length ts*
by *simp*
{fix *i* **assume** *i*:*i* < *length ts*
then have (*zip* [*0*..<*length ts*] (*map var-poss-list ts*)) ! *i* = (*i*, *var-poss-list* (*ts*!*i*)) **by** *simp*
with *i* **have** *?xs*!*i* = (*map* ((#) *i*)) (*var-poss-list* (*ts*!*i*)) **by** *simp*
then have *length* (*?xs*!*i*) = *length* (*var-poss-list* (*ts*!*i*)) **by** *simp*
with *i* **have** *length* (*?xs*!*i*) = *length* (*?ys*!*i*) **using** *Fun.hyps* **by** *simp*
}
then show *?case*
unfolding *var-poss-list.simps vars-term-list.simps* **using** *eq-length-concat-nth*[*of ?xs ?ys*] *l1 l2* **by** *presburger*
qed

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*)
using *assms* **proof**(*induct t arbitrary*:*i*)
case (*Var x*)
then have *i*:*i* = *0*
unfolding *vars-term-list.simps* **by** *simp*
then show *?case* **unfolding** *i vars-term-list.simps poss-list.simps var-poss.simps* **by** *simp*
next
case (*Fun f ts*)
let *?xs*=(*map vars-term-list ts*)
let *?ys*=(*map2* (λ*i*. *map* ((#) *i*)) [*0*..<*length ts*] (*map var-poss-list ts*))
from *Fun*(*2*) **have** *1*:*i* < *length* (*concat ?xs*) **unfolding** *vars-term-list.simps* **by** *simp*
have *2*:*length ?ys* = *length ?xs* **unfolding** *length-map* **by** *simp*
{fix *i* **assume** *i*:*i* < *length ?xs*
then have ∗:(*map2* (λ*x*. *map* ((#) *x*)) [*0*..<*length ts*] (*map var-poss-list ts*) ! *i*) = *map* ((#) *i*) (*var-poss-list* (*ts*!*i*))
unfolding *length-map* **by** *simp*
with *i* **have** *length* (*?ys* ! *i*) = *length* (*?xs* ! *i*)
unfolding ∗ *length-map length-var-poss-list* **by** *simp*
}note *l*=*this*

```
  then obtain j k where j:j < length ?xs and k:k < length (?xs ! j)
    and concat:concat ?xs ! i = ?xs ! j ! k concat ?ys ! i = ?ys ! j ! k
    using nth-concat-two-lists[OF 1 2] by blast
  from Fun(1) j k have Var (vars-term-list (ts!j) ! k) = (ts!j) |- var-poss-list (ts!j)
! k
    unfolding length-map by force
  then have Var (vars-term-list (Fun f ts) ! i) = (Fun f ts)|-(j#(var-poss-list (ts!j)
! k))
    unfolding vars-term-list.simps concat(1) using j by auto
  moreover have j#(var-poss-list (ts!j) ! k) = ((var-poss-list (Fun f ts))!i) proof−
      from k have k < length (map2 (λi. map ((#) i)) [0..<length ts] (map
var-poss-list ts) ! j)
      using l j by presburger
    then show ?thesis
      unfolding var-poss-list.simps concat(2) using j unfolding length-map by
simp
  qed
  ultimately show ?case
    by presburger
qed

lemma var-poss-list-map-vars-term:
  shows var-poss-list (map-vars-term f t) = var-poss-list t
proof(induct t)
  case (Fun g ts)
  then have IH:map var-poss-list ts = map var-poss-list (map (map-vars-term f)
ts)
    by fastforce
 then show ?case unfolding map-vars-term-eq eval-term.simps IH var-poss-list.simps
    by force
qed simp

lemma distinct-var-poss-list:
  shows distinct (var-poss-list t)
proof(induct t)
  case (Fun f ts)
  let ?xs=(map2 (λi. map ((#) i)) [0..<length ts] (map var-poss-list ts))
  have l:length ?xs = length ts
    by force
  have d1:distinct (removeAll [] ?xs) proof−
    have xs':removeAll [] ?xs = filter (λx. x ≠ []) ?xs
      by (metis (mono-tags, lifting) filter-cong removeAll-filter-not-eq)
    {fix i j assume i:i < length ?xs ?xs!i ≠ [] and j:j < length ?xs ?xs!j ≠ [] and
ij:i ≠ j
      from i l obtain p where p:i#p ∈ set (?xs!i) using nth-mem
          by (smt (z3) add.left-neutral diff-zero length-greater-0-conv length-map
length-upt nth-map nth-upt nth-zip prod.simps(2))
      from l j(1) have ?xs ! j = map ((#) j) ((map var-poss-list ts)!j)
        by simp
```

```
    with p have ?xs!i ≠ ?xs!j using ij by force
   }
   then show ?thesis using distinct-filter2 xs' by (smt (verit))
  qed
  {fix x assume x ∈ set ?xs
   with l obtain i where i:i < length ts and x = ?xs!i by (metis in-set-idx)
   then have x:x = map ((#) i) (var-poss-list (ts!i)) by simp
   from Fun i have distinct (var-poss-list (ts!i)) by auto
   then have distinct x
    unfolding x by (simp add: distinct-map)
  }note d2=this
  {fix x y assume x ∈ set ?xs y ∈ set ?xs x ≠ y
   then obtain i j where i:i < length ?xs x = ?xs!i and j:j < length ?xs y =
?xs!j and ij:i ≠ j
    by (metis in-set-idx)
   from i have x:x = map ((#) i) (var-poss-list (ts!i)) by simp
   from j have y:y = map ((#) j) (var-poss-list (ts!j)) by simp
   {fix p q assume p:p ∈ set x and q:q ∈ set y
    from x p obtain p' where p':p = i#p' and p' ∈ set (var-poss-list (ts!i))
     by auto
    from y q obtain q' where q':q = j#q' and q' ∈ set (var-poss-list (ts!j))
     by auto
    from q' p' have p ≠ q by (simp add: ij)
   }
   then have set x ∩ set y = {} by auto
  }note d3=this
 from d1 d2 d3 show ?case unfolding var-poss-list.simps using distinct-concat-iff
by blast
qed simp

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
 by (induct t; auto simp: UNION-set-zip)

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* **by** (*induct rule*: *in-poss.induct*) *auto*
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*]
by (*simp add*: *vars-term-list.simps*(*1*))

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*
by (*metis assms in-set-idx nth-mem o-apply set-remdups set-rev*)

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*
by (*induct t*) *auto*

lemma *in-vars-term-code*[*code-unfold*]: *x* ∈ *vars-term t* = *contains-var-term x t*
by *simp*
end

3.1.4 Linear Terms

lemma *distinct-vars-linear-term*:
assumes *distinct* (*vars-term-list t*)
shows *linear-term t*
using *assms* **proof**(*induct t*)
case (*Fun f ts*)
{fix *t*′ **assume** *t*′:*t*′ ∈ *set ts*
with *Fun*(*2*) **have** *distinct* (*vars-term-list t*′)
unfolding *vars-term-list.simps* **by** (*simp add*: *distinct-concat-iff*)

```
    with Fun(1) t' have linear-term t' by auto
  }note IH=this
  have is-partition (map (set ∘ vars-term-list) ts)
    using distinct-is-partitition-sets vars-term-list.simps(2) Fun(2) by force
  then have is-partition (map vars-term ts) by (simp add: comp-def)
  with IH show ?case by simp
qed simp

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 −
  {
    note [simp] = is-partition-Nil is-partition-Cons
    fix xs ys
    let ?P = λ ys xs t. (linear-term-impl xs t = None ⟶ (xs ∩ vars-term t ≠ {}
∨ ¬ linear-term t)) ∧
        (linear-term-impl xs t = Some ys ⟶ (ys = (xs ∪ vars-term t)) ∧ xs ∩
vars-term t = {} ∧ linear-term t)
    have ?P ys xs t
   proof (induct rule: linear-term-impl.induct[of λ xs t. ∀ ys. ?P ys xs t, rule-format])
      case (3 xs f t ts zs)
      show ?case
      proof (cases linear-term-impl xs t)
        case None
        with 3 show ?thesis by auto
      next
        case (Some ys)
        note some = this
          with 3 have rec1: ys = xs ∪ vars-term t ∧ xs ∩ vars-term t = {} ∧
linear-term t by auto
        show ?thesis
        proof (cases linear-term-impl ys (Fun f ts))
          case None
           with rec1 Some have res: linear-term-impl xs (Fun f (t # ts)) = None
by simp
            from None 3(2) Some have rec2: ys ∩ vars-term (Fun f ts) ≠ {} ∨ ¬
linear-term (Fun f ts) by simp
          then have xs ∩ vars-term (Fun f (t # ts)) ≠ {} ∨ ¬ linear-term (Fun f
(t # ts))
          proof
            assume ys ∩ vars-term (Fun f ts) ≠ {}
```

```
        then obtain x where x1: x ∈ ys and x2: x ∈ vars-term (Fun f ts) by
auto
        show ?thesis
        proof (cases x ∈ xs)
          case True
          with x2 show ?thesis by auto
        next
          case False
          with x1 rec1 have x ∈ vars-term t by auto
          with x2 have ¬ linear-term (Fun f (t # ts)) by auto
          then show ?thesis ..
        qed
      next
        assume ¬ linear-term (Fun f ts) then have ¬ linear-term (Fun f (t #
ts)) by auto
        then show ?thesis ..
      qed
      with res show ?thesis by auto
    next
      case (Some us)
      with some have res: linear-term-impl xs (Fun f (t # ts)) = Some us by
auto
      {
        assume us: us = zs
        from Some[simplified us] 3(2) some
         have rec2: zs = ys ∪ vars-term (Fun f ts) ∧ ys ∩ vars-term (Fun f ts)
= {} ∧ linear-term (Fun f ts) by auto
        from rec1 rec2
       have part1: zs = xs ∪ vars-term (Fun f (t # ts)) ∧ xs ∩ vars-term (Fun
f (t # ts)) = {} (is ?part1) by auto
          from rec1 rec2 have vars-term t ∩ vars-term (Fun f ts) = {} and
linear-term t and linear-term (Fun f ts) by auto
        then have linear-term (Fun f (t # ts)) (is ?part2) by auto
        with part1 have ?part1 ∧ ?part2 ..
      }
      with res show ?thesis by auto
    qed
  qed
 qed auto
} note main = this
from main[of {}] show ?thesis by (cases linear-term-impl {} t, auto)
qed

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*
by (*simp add*: *check-linear-term-def*)

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 (*rule set-eqI*, *unfold mem-Collect-eq*)
fix *s*
show *s* ∈ *set*(*supteq-list t*) = (*t* ⊵ *s*)
proof (*induct t*)
case (*Fun f ss*)
show *?case*
proof (*cases Fun f ss* = *s*)
case *False*
show *?thesis*
proof
assume *Fun f ss* ⊵ *s*
with *False* **have** *sup*: *Fun f ss* ▷ *s* **using** *supteq-supt-conv* **by** *auto*
obtain *C* **where** *C* ≠ □ **and** *Fun f ss* = *C*⟨*s*⟩ **using** *sup* **by** *auto*
then obtain *b D a* **where** *Fun f ss* = *Fun f* (*b* @ *D*⟨*s*⟩ # *a*) **by** (*cases C*, *auto*)
then have *D*: *D*⟨*s*⟩ ∈ *set ss* **by** *auto*
with *Fun*[*OF D*] *ctxt-imp-supteq*[*of D s*] **obtain** *t* **where** *t* ∈ *set ss* **and** *s* ∈ *set* (*supteq-list t*) **by** *auto*
then show *s* ∈ *set* (*supteq-list* (*Fun f ss*)) **by** *auto*
next
assume *s* ∈ *set* (*supteq-list* (*Fun f ss*))
with *False* **obtain** *t* **where** *t*: *t* ∈ *set ss* **and** *s* ∈ *set* (*supteq-list t*) **by** *auto*
with *Fun*[*OF t*] **have** *t* ⊵ *s* **by** *auto*
with *t* **show** *Fun f ss* ⊵ *s* **by** *auto*
qed
qed *auto*
qed (*simp add*: *supteq-var-imp-eq*)
qed

lemma *supt-list-sound* [*simp*]:
set (*supt-list t*) = {*s*. *t* ▷ *s*}
by (*cases t*) *auto*

```
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) (λs. supt-impl s t)

lemma supt-impl [code-unfold]:
  s ▷ t ⟷ supt-impl s t
proof
  assume s ▷ t then show supt-impl s t
  proof (induct s)
    case (Var x) then show ?case by auto
  next
    case (Fun f ss) then show ?case
    proof (cases t ∈ set ss)
      case True then show ?thesis by (simp)
    next
      case False
      assume ⋀s. ⟦s ∈ set ss; s ▷ t⟧ ⟹ supt-impl s t
        and Fun f ss ▷ t and t ∉ set ss
      moreover from ‹Fun f ss ▷ t› obtain s where s ∈ set ss and s ▷ t
        by (cases rule: supt.cases) (simp-all add: ‹t ∉ set ss›)
      ultimately have supt-impl s t by simp
      with ‹s ∈ set ss› show ?thesis by auto
    qed
  qed
next
  assume supt-impl s t
  then show s ▷ t
  proof (induct s)
    case (Var x) then show ?case by simp
  next
    case (Fun f ss)
    then have t ∈ set ss ∨ (∃ s∈set ss. supt-impl s t) by simp
    then show ?case
    proof
      assume t ∈ set ss then show ?case by auto
    next
      assume ∃ s∈set ss. supt-impl s t
      then obtain s where s ∈ set ss and supt-impl s t by auto
      with Fun have s ▷ t by auto
      with ‹s ∈ set ss› show ?thesis by auto
    qed
  qed
qed

lemma supteq-impl[code-unfold]: s ⊵ t ⟷ s = t ∨ supt-impl s t
  unfolding supteq-supt-set-conv
  by (auto simp: supt-impl)
```

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*
unfolding *check-no-var-def* **by** *simp*

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*
by (*auto simp*: *check-supt-def*)

lemma *isOK-check-supteq* [*simp*]:
isOK (*check-supteq s t*) ⟷ *s* ⊵ *t*
by (*auto simp*: *check-supteq-def*)

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 (*induct t arbitrary*: *xs*)
case (*Fun f ts*)
then show *?case* **by** (*induct ts*) (*simp-all add*: *vars-term-list.simps*)
qed (*simp add*: *vars-term-list.simps*)

lemma *add-funs-term-funs-term-list-conv* [*simp*]:
add-funs-term t fs = *funs-term-list t* @ *fs*
proof (*induct t arbitrary*: *fs*)
case (*Fun f ts*)
then show *?case* **by** (*induct ts*) (*simp-all add*: *funs-term-list.simps*)
qed (*simp add*: *funs-term-list.simps*)

lemma *add-funas-term-funas-term-list-conv* [*simp*]:
add-funas-term t fs = *funas-term-list t* @ *fs*
proof (*induct t arbitrary*: *fs*)
case (*Fun f ts*)
then show *?case* **by** (*induct ts*) (*simp-all add*: *funas-term-list.simps*)
qed (*simp add*: *funas-term-list.simps*)

lemma *add-vars-term-vars-term-list-abs-conv* [*simp*]:
add-vars-term = (@) ∘ *vars-term-list*
by (*intro ext*) *simp*

lemma *add-funs-term-funs-term-list-abs-conv* [*simp*]:
add-funs-term = (@) ∘ *funs-term-list*
by (*intro ext*) *simp*

lemma *add-funas-term-funas-term-list-abs-conv* [*simp*]:
add-funas-term = (@) ∘ *funas-term-list*
by (*intro ext*) *simp*

lemma *add-funas-args-term-funas-args-term-list-conv* [*simp*]:
add-funas-args-term t fs = *funas-args-term-list t* @ *fs*
by (*simp add*: *add-funas-args-term-def funas-args-term-list-def concat-conv-foldr foldr-map*)

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 (*induct t arbitrary*: *xs*)
case (*Fun f ts*)
then show *?case* **by** (*induct ts*) *auto*
qed *simp*

lemma *set-insert-funs-term-funs-term* [*simp*]:
set (*insert-funs-term t fs*) = *funs-term t* ∪ *set fs*
proof (*induct t arbitrary*: *fs*)
case (*Fun f ts*)
then show *?case* **by** (*induct ts*) *auto*
qed *simp*

lemma *set-insert-funas-term-funas-term* [*simp*]:
set (*insert-funas-term t fs*) = *funas-term t* ∪ *set fs*
proof (*induct t arbitrary*: *fs*)
case (*Fun f ts*)
then have *set* (*foldr insert-funas-term ts fs*) = ⋃ (*funas-term* ‘ *set ts*) ∪ *set fs*
by (*induct ts*) *auto*
then show *?case* **by** *simp*
qed *simp*

lemma *set-insert-funas-args-term* [*simp*]:
set (*insert-funas-args-term t fs*) = ⋃ (*funas-term* ‘ *set* (*args t*)) ∪ *set fs*
proof (*induct t arbitrary*: *fs*)
case (*Fun f ts*)
then show *?case* **by** (*induct ts*) (*auto simp*: *insert-funas-args-term-def*)
qed (*simp add*: *insert-funas-args-term-def*)

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 []
  by simp-all



lemma with-arity-term-fold [code]:
  with-arity = Term-More.fold Var (λf ts. Fun (f, length ts) ts)
proof
  fix t :: ('f, 'v) term
  show with-arity t = Term-More.fold Var (λf ts. Fun (f, length ts) ts) t
    by (induct t) simp-all
qed

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 (induct t)
  case (Var x)
  show ?case (is - = ?R)
  proof −
    {
      fix t
      assume t ∈ ?R
      then have t = Var x by (cases t, auto)
    }
    then show ?thesis by auto
  qed
next
  case (Fun fs ts)
  show ?case (is ?L = ?R)
  proof −
    {
      fix i
      assume i < length ts
      then have ts ! i ∈ set ts by auto
      note Fun[OF this]
    } note ind = this
   have idL: ?L = {Fun g ss | g ss. g ∈ set fs  ∧ length ss = length ts ∧ (∀ i<length
```

```
ts. ss ! i ∈ set (flatten-term-enum (ts ! i)))} (is - = ?M1) by auto
    let ?R1 = {Fun f ss | f ss. f ∈ set fs ∧ length ss = length ts ∧ (∀ i<length ss.
instance-term (ss ! i) (map-funs-term set (ts ! i)))}
    {
      fix u
      assume u ∈ ?R
      then have u ∈ ?R1 by (cases u, auto)
    }
    then have idR: ?R = ?R1 by auto
    show ?case unfolding idL idR using ind by auto
  qed
qed

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
  by (simp add: check-ground-term-def)

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)
  by (induct t, auto)
```

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 −
 have ?I ⊆ ?R unfolding mk-subst-domain-def Let-def subst-domain-def by auto
 moreover
 {
  fix xt
  assume mem: xt ∈ ?R
  obtain x t where xt: xt = (x, t) by force
  from mem [unfolded xt]
  have x: x ∈ subst-domain (mk-subst Var σ) and t: t = mk-subst Var σ x by
auto
  then have mk-subst Var σ x ≠ Var x unfolding subst-domain-def by simp
  with t have l: map-of σ x = Some t and tx: t ≠ Var x
   unfolding mk-subst-def by (cases map-of σ x, auto)
 from map-of-SomeD[OF l] l t tx have (x,t) ∈ ?I unfolding mk-subst-domain-def
Let-def
   by force
  then have xt ∈ ?I unfolding xt .
 }
 ultimately show ?thesis by blast
qed

lemma finite-mk-subst: finite (subst-domain (mk-subst Var σ))
proof −
 have subst-domain (mk-subst Var σ) = fst ‘ set (mk-subst-domain σ)
  unfolding mk-subst-domain Let-def by force
 moreover have finite ...
  using finite-set by auto
 ultimately show ?thesis by simp
qed

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 −
 let ?σ = mk-subst Var σ
 let ?τ = mk-subst Var τ
 {
  assume id: ((λ x. (x, ?σ x)) ‘ subst-domain ?σ) = ((λ x. (x, ?τ x)) ‘
subst-domain ?τ) (is ?l = ?r)
  from arg-cong[OF id, of (‘) fst] have idd: subst-domain ?σ = subst-domain ?τ
by force
  have ?σ = ?τ
  proof (rule ext)
   fix x
   show ?σ x = ?τ x
   proof (cases x ∈ subst-domain ?σ)
```

```
      case False
      then show ?thesis using idd unfolding subst-domain-def by auto
    next
      case True
      with idd have x: (x,?σ x) ∈ ?l (x,?τ x) ∈ ?r by auto
      with id have x: (x,?τ x) ∈ ?l (x,?σ x) ∈ ?l by auto
      then show ?thesis by auto
    qed
  qed
 }
 then show ?thesis
  unfolding subst-eq-def Let-def
  unfolding mk-subst-domain by auto
qed

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)
 unfolding vars-subst-def vars-subst-impl-def Let-def
 by (auto simp: mk-subst-domain, force)

lemma range-vars-impl [simp]:
 set (range-vars-impl σ) = range-vars (mk-subst Var σ)
 unfolding range-vars-def range-vars-impl-def Let-def
 by (auto simp: mk-subst-domain)

lemma mk-subst-one [simp]: mk-subst Var [(x, t)] = subst x t
 unfolding mk-subst-def subst-def by auto

lemma fst-image [simp]: fst ` (λ x. (x,g x)) ` a = a by force

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 (intro ext)
  fix x
  {
    assume x: x ∉ subst-domain (mk-subst Var σ)
    then have σ: mk-subst Var σ x = Var x unfolding subst-domain-def by auto
     from x have x ∉ fst ‘ set (mk-subst-domain σ) unfolding mk-subst-domain
by auto
     then have look: map-of (mk-subst-domain σ) x = None by (cases map-of
(mk-subst-domain σ) x, insert map-of-SomeD[of mk-subst-domain σ x], force+)
    then have mk-subst Var (mk-subst-domain σ) x = mk-subst Var σ x unfolding
σ
      unfolding mk-subst-def by auto
  } note ndom = this
  {
    assume x ∈ subst-domain (mk-subst Var σ)
    then have x ∈ fst ‘ set (mk-subst-domain σ) unfolding mk-subst-domain by
auto
   then obtain t where look: map-of (mk-subst-domain σ) x = Some t by (cases
map-of (mk-subst-domain σ) x, (force simp: map-of-eq-None-iff)+)
    from map-of-SomeD[OF look, unfolded mk-subst-domain] have t: t = mk-subst
Var σ x by auto
    from look t have res: mk-subst Var (mk-subst-domain σ) x = mk-subst Var σ
x unfolding mk-subst-def by auto
  } note dom = this
  from ndom dom
  show mk-subst Var (mk-subst-domain σ) x = mk-subst Var σ x by auto
qed

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 (rule ext)
  fix x
  let ?σ = mk-subst Var σ
  let ?τ = mk-subst Var τ
 let ?s = map (λ (x, t). (x, t · mk-subst Var (mk-subst-domain τ))) (mk-subst-domain
σ)
  let ?t = [(x,t) ← mk-subst-domain τ. x ∉ set (map fst (mk-subst-domain σ))]
  note d = subst-compose-impl-def[unfolded Let-def]
  show ?l x = ?r x
  proof (cases x ∈ subst-domain (mk-subst Var σ))
    case True
    then have ?σ x ≠ Var x unfolding subst-domain-def by auto
    then obtain t where look: map-of σ x = Some t and σ: ?σ x = t
```

```
    unfolding mk-subst-def by (cases map-of σ x, auto)
   from σ have r: ?r x = t · ?τ unfolding subst-compose-def by simp
   from True have x ∈ subst-domain (mk-subst Var (mk-subst-domain σ))
    by simp
   from σ True  have mem: (x,t · ?τ) ∈ set ?s by (auto simp: mk-subst-domain)
   with map-of-eq-None-iff[of ?s x]
   obtain u where look2: map-of ?s x = Some u
    by (cases map-of ?s x, force+)
   from map-of-SomeD[OF this] σ have u: u = t · ?τ
    by (auto simp: mk-subst-domain)
   note look2 = map-of-append-Some[OF look2, of ?t]
   have l: ?l x = t · ?τ unfolding d mk-subst-def[of Var ?s @ ?t] look2 u
    by simp
   from l r show ?thesis by simp
  next
   case False
   then have σ: ?σ x = Var x unfolding subst-domain-def by auto
   from σ have r: ?r x = ?τ x unfolding subst-compose-def by simp
   from False have x ∉ subst-domain (mk-subst Var (mk-subst-domain σ))
    by simp
   from False  have mem: ⋀ y. (x,y) ∉ set ?s by (auto simp: mk-subst-domain)
   with map-of-SomeD[of ?s x] have look2: map-of ?s x = None
    by (cases map-of ?s x, auto)
   note look2 = map-of-append-None[OF look2, of ?t]
   have l: ?l x = (case map-of ?t x of None ⇒ Var x | Some t ⇒ t) unfolding d
mk-subst-def[of Var ?s @ ?t] look2 by simp
   also have ... = ?τ x
   proof (cases x ∈ subst-domain ?τ)
    case True
    then have ?τ x ≠ Var x unfolding subst-domain-def by auto
    then obtain t where look: map-of τ x = Some t and τ: ?τ x = t
     unfolding mk-subst-def by (cases map-of τ x, auto)
    from True have x ∈ subst-domain (mk-subst Var (mk-subst-domain τ))
     by simp
     from τ True  have mem: (x,?τ x) ∈ set ?t using False by (auto simp:
mk-subst-domain)
    with map-of-eq-None-iff[of ?t x] obtain u where look2: map-of ?t x = Some
u
     by (cases map-of ?t x, force+)
    from map-of-SomeD[OF this] τ have u: u = ?τ x
     by (auto simp: mk-subst-domain)
    show ?thesis using look2 u by simp
   next
    case False
    then have τ: ?τ x = Var x unfolding subst-domain-def by auto
    from False have x ∉ subst-domain (mk-subst Var (mk-subst-domain τ))
     by simp
    from False have mem: ⋀ y. (x,y) ∉ set ?t by (auto simp: mk-subst-domain)
    with map-of-SomeD[of ?t x] have look2: map-of ?t x = None
```

```
      by (cases map-of ?t x, auto)
    show ?thesis unfolding τ look2 by simp
  qed
  finally show ?thesis unfolding r by simp
 qed
qed

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
  by (induct n, auto)

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))
  unfolding commutes-impl-def by simp

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 (rule ext)
  fix x
  note d = subst-compose'-def subst-compose'-impl-def
  let ?σ = mk-subst Var σ
  let ?s = subst-compose'-impl σ ϱ
  show ?l x = ?r x
  proof (cases x ∈ subst-domain (mk-subst Var σ))
    case True
    then have r: ?r x = ?σ x · ϱ unfolding d by simp
   from True have (x, ?σ x) ∈ set (mk-subst-domain σ) unfolding mk-subst-domain
by auto
    then have (x, ?σ x · ϱ) ∈ set ?s unfolding d by auto
    with map-of-eq-None-iff[of ?s x] obtain u where look: map-of ?s x = Some u
      by (cases map-of ?s x, force+)
     from map-of-SomeD[OF this] have u: u = ?σ x · ϱ unfolding d using
mk-subst-domain[of σ] by auto
    then have l: ?l x = ?σ x · ϱ using look u unfolding mk-subst-def by auto
```

```
    from l r show ?thesis by simp
  next
    case False
    then have r: ?r x = Var x unfolding d by simp
    from False have ⋀ y. (x,y) ∉ set ?s unfolding d
      by (auto simp: mk-subst-domain)
    with map-of-SomeD[of ?s x] have look: map-of ?s x = None
      by (cases map-of ?s x, auto)
    then have l: ?l x = Var x unfolding mk-subst-def by simp
    from l r show ?thesis by simp
  qed
qed

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 (rule ext)
  fix y
  note d = subst-replace-impl-def
  show ?l y = ?r y
  proof (cases y = x)
    case True
    then show ?thesis unfolding d mk-subst-def by auto
  next
    case False
    let ?σ = mk-subst Var σ
    from False have r: ?r y = ?σ y by auto
     from False have l: ?l y =  mk-subst Var ([(y, t)←σ . y ≠ x]) y unfolding
mk-subst-def d
      by simp
    also have ... = ?σ y unfolding mk-subst-def
      using map-of-filter[of λ y. y ≠ x y σ, OF False] by simp
    finally show ?thesis using r by simp
  qed
qed

lemma mk-subst-domain-distinct: distinct (map fst (mk-subst-domain σ))
  unfolding mk-subst-domain-def Let-def distinct-map
  by (rule conjI[OF distinct-filter], auto simp: distinct-map inj-on-def)


definition is-renaming-impl :: ('f,'v) substL ⇒ bool where
  is-renaming-impl σ ≡
    let σ' = map snd (mk-subst-domain σ) in
```

$(\forall\ t \in set\ \sigma'.\ is\text{-}Var\ t) \wedge distinct\ \sigma'$

lemma *is-renaming-impl* [*simp*]:
is-renaming-impl σ = *is-renaming* (*mk-subst Var* σ) (**is** *?l* = *?r*)
proof −
let *?σ* = *mk-subst Var* σ
let *?d* = *mk-subst-domain* σ
let *?m* = *map snd ?d*
let *?k* = *map fst ?d*
have *?l* = ((∀ *t* ∈ *set ?m*. *is-Var t*) ∧ *distinct ?m*) **unfolding** *is-renaming-impl-def Let-def* **by** *auto*
also have (∀ *t* ∈ *set ?m*. *is-Var t*) = (∀ *x*. *is-Var* (*?σ x*))
by (*force simp*: *mk-subst-domain subst-domain-def*)
also have *distinct ?m* = *inj-on ?σ* (*subst-domain ?σ*)
proof
assume *inj*: *inj-on ?σ* (*subst-domain ?σ*)
show *distinct ?m* **unfolding** *distinct-conv-nth*
proof (*intro allI impI*)
fix *i j*
assume *i*: *i* < *length ?m* **and** *j*: *j* < *length ?m* **and** *ij*: *i* ≠ *j*
obtain *x t* **where** *di*: *?d* ! *i* = (*x*,*t*) **by** (*cases ?d* ! *i*, *auto*)
obtain *y s* **where** *dj*: *?d* ! *j* = (*y*,*s*) **by** (*cases ?d* ! *j*, *auto*)
from *di i* **have** *mi*: *?m* ! *i* = *t* **and** *ki*: *?k* ! *i* = *x* **by** *auto*
from *dj j* **have** *mj*: *?m* ! *j* = *s* **and** *kj*: *?k* ! *j* = *y* **by** *auto*
from *di i* **have** *xt*: (*x*,*t*) ∈ *set ?d* **unfolding** *set-conv-nth* **by** *force*
from *dj j* **have** *ys*: (*y*,*s*) ∈ *set ?d* **unfolding** *set-conv-nth* **by** *force*
from *xt ys* **have** *d*: *x* ∈ *subst-domain ?σ y* ∈ *subst-domain ?σ* **unfolding** *mk-subst-domain* **by** *auto*
have *dist*: *distinct ?k* **by** (*rule mk-subst-domain-distinct*)
from *ij i j* **have** *xy*: *x* ≠ *y* **unfolding** *ki*[*symmetric*] *kj*[*symmetric*]
using *dist*[*unfolded distinct-conv-nth*] **by** *auto*
from *xt ys* **have** *m*: *?σ x* = *t ?σ y* = *s* **unfolding** *mk-subst-domain* **by** *auto*
from *inj*[*unfolded inj-on-def*, *rule-format*, *OF d*]
show *?m* ! *i* ≠ *?m* ! *j* **unfolding** *m mi mj* **using** *xy* **by** *auto*
qed
next
assume *dist*: *distinct ?m*
show *inj-on ?σ* (*subst-domain ?σ*) **unfolding** *inj-on-def*
proof (*intro ballI impI*)
fix *x y*
assume *x*: *x* ∈ *subst-domain ?σ* **and** *y*: *y* ∈ *subst-domain ?σ*
and *id*: *?σ x* = *?σ y*
from *x y* **have** *x*: (*x*,*?σ x*) ∈ *set ?d* **and** *y*: (*y*,*?σ y*) ∈ *set ?d*
unfolding *mk-subst-domain* **by** *auto*
from *x* **obtain** *i* **where** *di*: *?d* ! *i* = (*x*,*?σ x*) **and** *i*: *i* < *length ?d* **unfolding** *set-conv-nth* **by** *auto*
from *y* **obtain** *j* **where** *dj*: *?d* ! *j* = (*y*,*?σ y*) **and** *j*: *j* < *length ?d* **unfolding** *set-conv-nth* **by** *auto*
from *di i* **have** *mi*: *?m* ! *i* = *?σ x* **by** *simp*

```
    from dj j have mj: ?m ! j = ?σ x unfolding id by simp
    from mi mj have id: ?m ! i = ?m ! j by simp
    from dist[unfolded distinct-conv-nth] i j id have id: i = j by auto
    with di dj
    show x = y by auto
   qed
  qed
  finally
  show ?thesis unfolding is-renaming-def by simp
qed

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 (rule ext)
  fix x
  let ?σ = mk-subst Var σ
  let ?σ' = mk-subst-domain σ
  let ?m = map (λ (x, y). (the-Var y, Var x :: ('f, 'v) term)) ?σ'
  let ?ran = subst-range ?σ
  note d = is-inverse-renaming-impl-def is-inverse-renaming-def
  {
    fix t
    assume (x,t) ∈ set ?m
    then obtain u z where id: (x,t) = (the-Var u,Var z) and mem: (z,u) ∈ set
?σ' by auto
    from var[unfolded is-renaming-def] mem obtain zz where u: u = Var zz
      unfolding mk-subst-domain by auto
    from id[unfolded u] have id: zz = x t = Var z by auto
    with mem u have (z,Var x) ∈ set ?σ' by auto
   then have ?σ z = Var x z ∈ subst-domain ?σ unfolding mk-subst-domain by
auto
    with id have ∃ z. t = Var z ∧ ?σ z = Var x ∧ z ∈ subst-domain ?σ by auto
  } note one = this
  have ?l x = mk-subst Var ?m x unfolding d by simp
  also have ... = ?r x
  proof (cases Var x ∈ ?ran)
    case False
    {
      fix t
      assume (x,t) ∈ set ?m
      from one[OF this] obtain z where t: t = Var z and z: ?σ z = Var x
```

```
      and dom: z ∈ subst-domain ?σ by auto
    from z dom False have False by force
  }
  from this[OF map-of-SomeD[of ?m x]] have look: map-of ?m x = None
    by (cases map-of ?m x, auto)
  then have mk-subst Var ?m x = Var x unfolding mk-subst-def by auto
  also have ... = ?r x using False unfolding d by simp
  finally show ?thesis .
 next
  case True
  then obtain y where y: y ∈ subst-domain ?σ and x: ?σ y = Var x by auto
  then have (y, Var x) ∈ set ?σ' unfolding mk-subst-domain by auto
  then have (x, Var y) ∈ set ?m by force
 then obtain u where look: map-of ?m x = Some u using map-of-eq-None-iff[of
?m x]
    by (cases map-of ?m x, force+)
  from map-of-SomeD[OF this] have xu: (x,u) ∈ set ?m by auto
  from one[OF this] obtain z where u: u = Var z and z: ?σ z = Var x and
dom: z ∈ subst-domain ?σ by auto
  have mk-subst Var ?m x = Var z unfolding mk-subst-def look u by simp
 also have ... = ?r x using is-renaming-inverse-domain[OF var dom] z by auto
  finally show ?thesis .
 qed
 finally show ?l x = ?r x .
qed

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 −
 let ?m = map (λ x. (x, σ x)) xs
 have id: mk-subst Var ?m = (λ x. if x ∈ set xs then σ x else Var x) (is ?l = ?r)
 proof (rule ext)
  fix x
  show ?l x = ?r x
  proof (cases x ∈ set xs)
   case True
   then have (x,σ x) ∈ set ?m by auto
  with map-of-eq-None-iff[of ?m x] obtain u where look: map-of ?m x = Some
u by auto
   from map-of-SomeD[OF look] have u: u = σ x by auto
   show ?thesis unfolding mk-subst-def look u using True by auto
  next
   case False
```

with *map-of-SomeD*[*of ?m x*]
have *look*: *map-of ?m x* = *None* **by** (*cases map-of ?m x*, *auto*)
show *?thesis* **unfolding** *mk-subst-def look* **using** *False* **by** *auto*
qed
qed
show *?thesis* **unfolding** *mk-subst-case-def subst-compose-impl id*
unfolding *subst-compose-def* **by** *auto*
qed

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*)
by (*simp add*: *subst-list-def subst-set-def*)

lemma *mgu-var-disjoint-right*:
fixes *s t* :: ('*f*, '*v*) *term* **and** σ τ :: ('*f*, '*v*) *subst* **and** *T*
assumes *s*: *vars-term s* $\subseteq$ *S*
and *inj*: *inj T*
and *ST*: *S* $\cap$ *range T* = {}
and *id*: *s* $\cdot$ σ = *t* $\cdot$ τ
shows $\exists$ μ δ. *mgu s* (*map-vars-term T t*) = *Some* μ $\wedge$
s $\cdot$ σ = *s* $\cdot$ μ $\cdot$ δ $\wedge$
($\forall$ *t*::('*f*, '*v*) *term*. *t* $\cdot$ τ = *map-vars-term T t* $\cdot$ μ $\cdot$ δ) $\wedge$
($\forall$ *x*$\in$*S*. σ *x* = μ *x* $\cdot$ δ)
proof −
let *?σ* = λ *x*. *if x* $\in$ *S then* σ *x else* τ ((*the-inv T*) *x*)
let *?t* = *map-vars-term T t*
have *ids*: *s* $\cdot$ σ = *s* $\cdot$ *?σ*
by (*rule term-subst-eq*, *insert s*, *auto*)
have *t* $\cdot$ τ = *map-vars-term* (*the-inv T*) *?t* $\cdot$ τ
unfolding *map-vars-term-compose o-def* **using** *the-inv-f-f*[*OF inj*] **by** (*auto simp*: *term.map-ident*)
also have ... = *?t* $\cdot$ (τ $\circ$ *the-inv T*) **unfolding** *apply-subst-map-vars-term* **..**
also have ... = *?t* $\cdot$ *?σ*
proof (*rule term-subst-eq*)
fix *x*
assume *x* $\in$ *vars-term ?t*
then have *x* $\in$ *T* ‘ *UNIV* **unfolding** *term.set-map* **by** *auto*
then have *x* $\notin$ *S* **using** *ST* **by** *auto*
then show (τ $\circ$ *the-inv T*) *x* = *?σ x* **by** *simp*

```
  qed
  finally have idt: t · τ = ?t · ?σ by simp
  from id[unfolded ids idt] have id: s · ?σ = ?t · ?σ .
  with mgu-complete[of s ?t] id obtain μ where μ: mgu s ?t = Some μ
    unfolding unifiers-def  by (cases mgu s ?t, auto)
  from the-mgu[OF id] have id: s · μ = map-vars-term T t · μ and σ: ?σ = μ ∘s
?σ
    unfolding the-mgu-def μ by auto
  have s · σ = s · (μ ∘s ?σ) unfolding ids using σ by simp
  also have ... = s · μ · ?σ by simp
  finally have ids: s · σ = s · μ · ?σ .
  {
    fix x
    have τ x = ?σ (T x) using ST unfolding the-inv-f-f[OF inj] by auto
    also have ... = (μ ∘s ?σ) (T x) using σ by simp
    also have ... = μ (T x) · ?σ unfolding subst-compose-def by simp
    finally have τ x = μ (T x) · ?σ .
  } note τ = this
  {
    fix t :: ('f,'v)term
    have t · τ = t · (λ x. μ (T x) · ?σ) unfolding τ[symmetric] ..
   also have ... = map-vars-term T t · μ · ?σ unfolding apply-subst-map-vars-term

        subst-subst by (rule term-subst-eq, simp add: subst-compose-def)
    finally have t · τ = map-vars-term T t · μ · ?σ .
  } note idt = this
  {
    fix x
    assume x ∈ S
    then have σ x = ?σ x by simp
    also have ... = (μ ∘s ?σ) x using σ by simp
    also have ... = μ x · ?σ unfolding subst-compose-def ..
    finally have σ x = μ x · ?σ .
  } note σ = this
  show ?thesis
    by (rule exI[of - μ], rule exI[of - ?σ], insert μ ids idt σ, auto)
qed

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)
∧
```

```
    (∀ x ∈ range x-var ∪ range z-var. σ x = μ x · δ)
proof −
  have inj: inj y-var unfolding inj-on-def by simp
  show ?thesis
    by (rule mgu-var-disjoint-right[OF s inj - id], auto)
qed

lemma not-elem-subst-of:
  assumes x ∉ set (map fst xs)
  shows (subst-of xs) x = Var x
  using assms proof (induct xs)
  case (Cons y xs)
  then show ?case unfolding subst-of-simps
   by (metis Term.term.simps(17) insert-iff list.simps(15) list.simps(9) singletonD
subst-compose subst-ident)
qed simp

lemma subst-of-id:
  assumes ⋀s. s ∈ (set ss) ⟶ (∃ x t. s = (x, t) ∧ t = Var x)
  shows subst-of ss = Var
  using assms proof(induct ss)
  case (Cons s ss)
  then obtain y t where s:s = (y, t) and t:t = Var y
    by (metis list.set-intros(1))
  from Cons have subst-of ss = Var
    by simp
  then show ?case
    unfolding subst-of-def foldr.simps o-apply s t by simp
qed simp


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
  using assms proof(induct ss)
  case (Cons a ss)
  show ?case proof(cases (x,t) ∈ set ss)
    case True
    from Cons(1)[OF True] Cons(3,4) have sub:subst-of ss x = t
      by (simp add: disjoint-iff)
    from Cons(2,4) have fst a ∉ vars-term t
      by fastforce
    then show ?thesis
      unfolding subst-of-simps subst-compose sub by simp
  next
    case False
    then have x ∉ set (map fst ss)
```

```
    using Cons(3) by auto
  then have sub:subst-of ss x = Var x
    by (meson not-elem-subst-of)
  from Cons(2) False have a = (x, t)
    by simp
  then show ?thesis
    unfolding subst-of-simps subst-compose sub by simp
 qed
qed simp

lemma unify-equation-same:
 assumes fst e = snd e
 shows unify (E1@e#E2) ys = unify (E1@E2) ys
using assms proof (induction E1@e#E2 ys arbitrary: E1 E2 e ys rule: unify.induct)
 case (2 f ss g ts E bs)
 show ?case proof(cases E1)
  case Nil
  with 2(3) show ?thesis
   by (simp add: unify-Cons-same)
 next
  case (Cons e1 es1)
  then have e1:e1 = (Fun f ss, Fun g ts)
   using 2(2) by simp
  show ?thesis proof(cases decompose (Fun f ss) (Fun g ts))
   case None
   then show ?thesis unfolding Cons e1 by simp
  next
   case (Some us)
   have us:us @ E = (us @ es1) @ e # E2
    using 2(2) Cons e1 by simp
   from 2(1)[OF Some us 2(3)] show ?thesis unfolding Cons e1 append-Cons
unify.simps Some by simp
  qed
 qed
next
 case (3 x t E bs)
 show ?case proof(cases E1)
  case Nil
  with 3(4) show ?thesis
   by (simp add: unify-Cons-same)
 next
  case (Cons e1 es1)
  with 3(3) have e1:e1 = (Var x, t)
   by simp
  with 3(3) Cons have E:E = es1 @ e # E2
   by simp
  show ?thesis proof(cases t = Var x)
   case True
   from 3(1)[OF True E 3(4)] show ?thesis
```

```
      unfolding Cons e1 True by simp
    next
      case False
      then show ?thesis proof(cases x ∉ vars-term t)
        case True
        let ?σ=(subst x t)
        have substs:subst-list ?σ E = (subst-list ?σ es1) @ (fst e · ?σ, snd e · ?σ)
# (subst-list ?σ E2)
          unfolding E by (simp add: subst-list-def)
        from 3(2)[OF False True substs] 3(4) show ?thesis
          unfolding Cons e1 append-Cons unify.simps using False True
          by (smt (verit, ccfv-SIG) E fst-eqD snd-eqD subst-list-append substs)
      next
        case False
        then show ?thesis
          unfolding Cons e1 append-Cons unify.simps using 3 Cons by auto
      qed
    qed
  qed
next
  case (4 f ts x E bs)
  show ?case proof(cases E1)
    case Nil
    with 4(3) show ?thesis
      by (simp add: unify-Cons-same)
  next
    case (Cons e1 es1)
    with 4(2) have e1:e1 = (Fun f ts, Var x)
      by simp
    with 4(2) Cons have E:E = es1 @ e # E2
      by simp
    show ?thesis proof(cases x ∉ vars-term (Fun f ts))
      case True
      let ?σ=(subst x (Fun f ts))
      have substs:subst-list ?σ E = (subst-list ?σ es1) @ (fst e · ?σ, snd e · ?σ) #
(subst-list ?σ E2)
        unfolding E by (simp add: subst-list-def)
      from 4(1)[OF True substs] 4(3) show ?thesis
        unfolding Cons e1 append-Cons unify.simps using True
        by (metis E fst-conv snd-conv subst-list-append substs)
    next
      case False
      then show ?thesis
        unfolding Cons e1 append-Cons unify.simps using 4 Cons by auto
    qed
  qed
qed simp

lemma unify-filter-same:
```

shows *unify* (*filter* (λ*e*. *fst e* ≠ *snd e*) *E*) *ys* = *unify E ys*
proof(*induction length E arbitrary*:*E rule*:*full-nat-induct*)
case *1*
show *?case* **proof**(*cases E*)
case (*Cons e es*)
then show *?thesis* **proof**(*cases filter* (λ*e*. *fst e* ≠ *snd e*) *E* = *E*)
case *False*
then obtain *E1 e E2* **where** *E*:*E* = *E1* @ *e* # *E2* **and** *eq*:*fst e* = *snd e*
by (*meson filter-True split-list*)
with *unify-equation-same* **have** *unify E ys* = *unify* (*E1* @ *E2*) *ys*
by *blast*
moreover from *1 E* **have** *unify* (*filter* (λ*e*. *fst e* ≠ *snd e*) (*E1* @ *E2*)) *ys* = *unify* (*E1* @ *E2*) *ys*
by (*metis* (*no-types*, *lifting*) *add-Suc-right length-append length-nth-simps*(*2*) *order-refl*)
moreover have *filter* (λ*e*. *fst e* ≠ *snd e*) *E* = *filter* (λ*e*. *fst e* ≠ *snd e*) (*E1* @ *E2*)
unfolding *E* **using** *eq* **by** *auto*
ultimately show *?thesis*
by *presburger*
qed *simp*
qed *simp*
qed

lemma *unify-ctxt-same*:
shows *unify* ((*C*⟨*s*⟩, *C*⟨*t*⟩)#*xs*) *ys* = *unify* ((*s*, *t*)#*xs*) *ys*
proof(*induct C*)
case (*More f ss1 C ss2*)
let *?us*=*zip* (*ss1* @ *C*⟨*s*⟩ # *ss2*) (*ss1* @ *C*⟨*t*⟩ # *ss2*)
have *decomp*:*decompose* (*Fun f* (*ss1* @ *C*⟨*s*⟩ # *ss2*)) (*Fun f* (*ss1* @ *C*⟨*t*⟩ # *ss2*)) = *Some ?us*
unfolding *decompose-def* **by** (*simp add*: *zip-option-zip-conv*)
have *unif*:*unify* (((*More f ss1 C ss2*)⟨*s*⟩, (*More f ss1 C ss2*)⟨*t*⟩) # *xs*) *ys* = *unify* (*?us* @ *xs*) *ys*
unfolding *intp-actxt.simps unify.simps decomp* **by** *simp*
have ∗:*?us* = (*zip ss1 ss1*) @ (*C*⟨*s*⟩, *C*⟨*t*⟩) # (*zip ss2 ss2*)
by *simp*
have *filter-us*:*filter* (λ*e*. *fst e* ≠ *snd e*) *?us* = *filter* (λ*e*. *fst e* ≠ *snd e*) [(*C*⟨*s*⟩, *C*⟨*t*⟩)]
unfolding ∗ *filter-append filter.simps* **by** (*smt* (*verit*, *ccfv-SIG*) *filter-False in-set-zip self-append-conv2*)
have *filter* (λ*e*. *fst e* ≠ *snd e*) (*?us*@*xs*) = *filter* (λ*e*. *fst e* ≠ *snd e*) ((*C*⟨*s*⟩, *C*⟨*t*⟩)#*xs*)
unfolding *filter-append filter-us filter.simps* **by** *simp*
with *More* **have** *unify* (*?us* @ *xs*) *ys* = *unify* ((*s*, *t*)#*xs*) *ys*
using *unify-filter-same* **by** (*smt* (*verit*, *ccfv-threshold*))
with *unif* **show** *?case* **by** *simp*
qed *simp*

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−
  from *assms* **obtain** *p* **where** *1*:(*x*, *p*) ∈ *set* (*zip* (*vars-term-list s*)(*var-poss-list s*)) **and** *2*:*p* ∈ *poss t t*|-*p* = *u*
   unfolding *left-substs-def Let-def* **using** *Pair-inject case-prodE filter-set in-set-idx length-map map-nth-eq-conv member-filter nth-mem old.prod.case* **by** *auto*
  from *1* **have** *p*:*p* ∈ *poss s*
    by (*metis set-zip-rightD var-poss-imp-poss var-poss-list-sound*)
  from *1* **obtain** *i* **where** *i* < *length* (*zip* (*vars-term-list s*)(*var-poss-list s*)) **and** (*vars-term-list s*)!*i* = *x* **and** (*var-poss-list s*)!*i* = *p*
    by (*smt* (*z3*) *Pair-inject length-zip mem-Collect-eq set-zip*)
  then have *s*|-*p* = *Var x*
    by (*metis length-zip min-less-iff-conj vars-term-list-var-poss-list*)
  with *2 p* **show** *?thesis*
    by *blast*
qed

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−
  from *assms* **obtain** *i* **where** (*var-poss-list s*)!*i* = *p* **and** (*vars-term-list s*)!*i* = *x*
   by (*metis in-set-conv-nth length-var-poss-list term.inject*(*1*) *var-poss-iff var-poss-list-sound vars-term-list-var-poss-list*)
  then have (*x*, *p*) ∈ *set* (*zip* (*vars-term-list s*)(*var-poss-list s*))
   by (*metis assms*(*1*) *assms*(*2*) *in-set-idx in-set-zip length-var-poss-list prod.sel*(*1*) *prod.sel*(*2*) *term.inject*(*1*) *var-poss-iff var-poss-list-sound vars-term-list-var-poss-list*)
  with *assms*(*3*) **have** (*x*, *p*) ∈ *set* (*filter* (λ(-, *p*). *p* ∈ *poss t*) (*zip* (*vars-term-list s*) (*var-poss-list s*)))
    by *simp*
  then show *?thesis* **unfolding** *left-substs-def Let-def assms*(*4*)[*symmetric*]
    by (*smt* (*z3*) *case-prod-conv in-set-conv-nth length-map map-nth-eq-conv*)
qed

lemma *right-substs-imp-props*:
assumes $(x, u) \in set\ (right\text{-}substs\ s\ t)$
shows $\exists q.\ q \in fun\text{-}poss\ s \wedge s|\text{-}q = u \wedge q \in poss\ t \wedge t|\text{-}q = Var\ x$
proof−
from *assms* **obtain** *q* **where** *1*:$(x, q) \in set\ (zip\ (vars\text{-}term\text{-}list\ t)(var\text{-}poss\text{-}list\ t))$ **and** *2*:$q \in fun\text{-}poss\ s\ s|\text{-}q = u$
unfolding *right-substs-def Let-def* **using** *Pair-inject case-prodE filter-set in-set-idx length-map map-nth-eq-conv member-filter nth-mem old.prod.case* **by** *auto*
from *1* **have** *q*:$q \in poss\ t$
by (*metis set-zip-rightD var-poss-imp-poss var-poss-list-sound*)
from *1* **obtain** *i* **where** $i < length\ (zip\ (vars\text{-}term\text{-}list\ t)(var\text{-}poss\text{-}list\ t))$ **and** $(vars\text{-}term\text{-}list\ t)!i = x$ **and** $(var\text{-}poss\text{-}list\ t)!i = q$
by (*smt* (*z3*) *Pair-inject length-zip mem-Collect-eq set-zip*)
then have $t|\text{-}q = Var\ x$
by (*metis length-zip min-less-iff-conj vars-term-list-var-poss-list*)
with *2 q* **show** *?thesis*
by *blast*
qed

lemma *props-imp-right-substs*:
assumes $q \in fun\text{-}poss\ s$ **and** $s|\text{-}q = u$ **and** $q \in poss\ t$ **and** $t|\text{-}q = Var\ x$
shows $(x, u) \in set\ (right\text{-}substs\ s\ t)$
proof−
from *assms* **obtain** *i* **where** $(var\text{-}poss\text{-}list\ t)!i = q$ **and** $(vars\text{-}term\text{-}list\ t)!i = x$
by (*metis in-set-conv-nth length-var-poss-list term.inject*(*1*) *var-poss-iff var-poss-list-sound vars-term-list-var-poss-list*)
then have $(x, q) \in set\ (zip\ (vars\text{-}term\text{-}list\ t)(var\text{-}poss\text{-}list\ t))$
by (*metis assms*(*3*) *assms*(*4*) *in-set-conv-nth in-set-zip length-var-poss-list prod.sel*(*1*) *prod.sel*(*2*) *term.inject*(*1*) *var-poss-iff var-poss-list-sound vars-term-list-var-poss-list*)
with *assms*(*1*) **have** $(x, q) \in set\ (filter\ (\lambda(\text{-}, p).\ p \in fun\text{-}poss\ s)\ (zip\ (vars\text{-}term\text{-}list\ t)\ (var\text{-}poss\text{-}list\ t)))$
by *simp*
then show *?thesis* **unfolding** *right-substs-def Let-def assms*(*2*)[*symmetric*]
by (*smt* (*z3*) *case-prod-conv in-set-conv-nth length-map map-nth-eq-conv*)
qed

lemma *map-fst-left-substs*:
$set\ (map\ fst\ (left\text{-}substs\ s\ t)) \subseteq vars\text{-}term\ s$
unfolding *left-substs-def* **using** *zip-fst* **by** *fastforce*

lemma *map-snd-left-substs*:
assumes $t' \in set\ (map\ snd\ (left\text{-}substs\ s\ t))$
shows $vars\text{-}term\ t' \subseteq vars\text{-}term\ t$
proof−
from *assms* **obtain** *x* **where** $(x, t') \in set\ (left\text{-}substs\ s\ t)$
by *force*
then show *?thesis*
using *left-substs-imp-props* **by** (*metis vars-term-subt-at*)

qed

lemma *map-fst-right-substs*:
set (*map fst* (*right-substs s t*)) ⊆ *vars-term t*
unfolding *right-substs-def* **using** *zip-fst* **by** *fastforce*

lemma *map-snd-right-substs*:
assumes $t' \in$ *set* (*map snd* (*right-substs s t*))
shows *vars-term* $t' \subseteq$ *vars-term s*
proof−
from *assms* **obtain** *x* **where** (*x*, t') ∈ *set* (*right-substs s t*)
by *force*
then show *?thesis*
using *right-substs-imp-props* **by** (*metis fun-poss-imp-poss vars-term-subt-at*)
qed

lemma *distinct-map-fst-left-substs*:
assumes *linear-term t*
shows *distinct* (*map fst* (*left-substs t s*))
proof−
from *linear-term-distinct-vars*[*OF assms*] **have** *dist*:*distinct* (*map fst* (*filter* (λ(*x*, *p*). *p* ∈ *poss s*) (*zip* (*vars-term-list t*) (*var-poss-list t*))))
by (*simp add*: *distinct-map-filter length-var-poss-list*)
have *map fst* (*left-substs t s*) = (*map fst* (*filter* (λ(*x*, *p*). *p* ∈ *poss s*) (*zip* (*vars-term-list t*) (*var-poss-list t*))))
unfolding *left-substs-def Let-def* **by** *auto*
with *dist* **show** *?thesis*
by *presburger*
qed

lemma *distinct-map-fst-right-substs*:
assumes *linear-term t*
shows *distinct* (*map fst* (*right-substs s t*))
proof−
from *linear-term-distinct-vars*[*OF assms*] **have** *dist*:*distinct* (*map fst* (*filter* (λ(*x*, *p*). *p* ∈ *fun-poss s*) (*zip* (*vars-term-list t*) (*var-poss-list t*))))
by (*simp add*: *distinct-map-filter length-var-poss-list*)
have *map fst* (*right-substs s t*) = (*map fst* (*filter* (λ(*x*, *p*). *p* ∈ *fun-poss s*) (*zip* (*vars-term-list t*) (*var-poss-list t*))))
unfolding *right-substs-def Let-def* **by** *auto*
with *dist* **show** *?thesis*
by *presburger*
qed

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−
{fix *i j* **assume** *j*:*j* < *length* (*left-substs t s*) **and** *i*:*i* < *j*

from *i j* **obtain** *x u* **where** *xu*:$(x, u) = (\mathit{left\text{-}substs}\ t\ s)!i$
by (*metis surj-pair*)
from *i j* **obtain** *y v* **where** *yv*:$(y, v) = (\mathit{left\text{-}substs}\ t\ s)!j$
by (*metis surj-pair*)
from *xu i j* **obtain** *p* **where** *p*:$p \in \mathit{poss}\ t\ \ t|\text{-}p = \mathit{Var}\ x\ \ p \in \mathit{poss}\ s\ \ s|\text{-}p = u$
using *left-substs-imp-props* **by** (*metis Suc-lessD less-trans-Suc nth-mem*)
from *yv i j* **obtain** *q* **where** *q*:$q \in \mathit{poss}\ t\ \ t|\text{-}q = \mathit{Var}\ y\ \ q \in \mathit{poss}\ s\ \ s|\text{-}q = v$
using *left-substs-imp-props* **by** (*metis nth-mem*)
from *assms*(*2*) **have** *distinct* (*map fst* (*left-substs t s*))
using *distinct-map-fst-left-substs* **by** *blast*
with *xu yv i j* **have** $x \neq y$
by (*metis* (*mono-tags*, *lifting*) *Suc-lessD distinct-map eq-key-imp-eq-value less-trans-Suc nat-neq-iff nth-eq-iff-index-eq nth-mem*)
with *p*(*1*,*2*) *q*(*1*,*2*) **have** $p \perp q$
by (*metis term.inject*(*1*) *var-poss-iff var-poss-parallel*)
with *assms*(*1*) *p*(*3*,*4*) *q*(*3*,*4*) **have** *vars-term* (*snd* ((*left-substs t s*)!*i*)) ∩ *vars-term* (*snd* ((*left-substs t s*)!*j*)) = {}
by (*metis linear-subterms-disjoint-vars snd-eqD xu yv*)
}
then show *?thesis* **unfolding** *is-partition-def map-map*[*symmetric*] **by** *auto*
qed

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−
{fix *i j* **assume** *j*:$j < \mathit{length}$ (*right-substs t s*) **and** *i*:$i < j$
from *i j* **obtain** *x u* **where** *xu*:$(x, u) = (\mathit{right\text{-}substs}\ t\ s)!i$
by (*metis surj-pair*)
from *i j* **obtain** *y v* **where** *yv*:$(y, v) = (\mathit{right\text{-}substs}\ t\ s)!j$
by (*metis surj-pair*)
from *xu i j* **obtain** *p* **where** *p*:$p \in \mathit{poss}\ s\ \ s|\text{-}p = \mathit{Var}\ x\ \ p \in \mathit{fun\text{-}poss}\ t\ \ t|\text{-}p = u$
using *right-substs-imp-props* **by** (*metis Suc-lessD less-trans-Suc nth-mem*)
from *yv i j* **obtain** *q* **where** *q*:$q \in \mathit{poss}\ s\ \ s|\text{-}q = \mathit{Var}\ y\ \ q \in \mathit{fun\text{-}poss}\ t\ \ t|\text{-}q = v$
using *right-substs-imp-props* **by** (*metis nth-mem*)
from *assms*(*1*) **have** *distinct* (*map fst* (*right-substs t s*))
using *distinct-map-fst-right-substs* **by** *blast*
with *xu yv i j* **have** $x \neq y$
by (*metis* (*mono-tags*, *lifting*) *Suc-lessD distinct-map eq-key-imp-eq-value less-trans-Suc nat-neq-iff nth-eq-iff-index-eq nth-mem*)
with *p*(*1*,*2*) *q*(*1*,*2*) **have** $p \perp q$
by (*metis term.inject*(*1*) *var-poss-iff var-poss-parallel*)
with *assms*(*2*) *p*(*3*,*4*) *q*(*3*,*4*) **have** *vars-term* (*snd* ((*right-substs t s*)!*i*)) ∩ *vars-term* (*snd* ((*right-substs t s*)!*j*)) = {}
by (*metis fun-poss-imp-poss linear-subterms-disjoint-vars snd-eqD xu yv*)
}
then show *?thesis* **unfolding** *is-partition-def map-map*[*symmetric*] **by** *auto*

qed

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−
{fix *x u* **assume** (*x*,*u*) ∈ *set* (*left-substs s t*)
then obtain *p* **where** *p*:*p* ∈ *poss s* *s*|-*p* = *Var x p* ∈ *poss t* *t*|-*p* = *u*
by (*meson left-substs-imp-props*)
{fix *y v* **assume** (*y*,*v*) ∈ *set* (*right-substs s t*)
then obtain *q* **where** *q*:*q* ∈ *poss t t*|-*q* = *Var y q* ∈ *fun-poss s* *s*|-*q* = *v*
by (*meson right-substs-imp-props*)
with *p* **have** *p* ⊥ *q*
by (*metis Term.term.simps*(*4*) *append.right-neutral fun-poss-fun-conv fun-poss-imp-poss parallel-pos prefix-pos-diff var-pos-maximal*)
with *assms p*(*1*,*2*) *q*(*3*,*4*) **have** *x* ∉ *vars-term v*
using *fun-poss-imp-poss linear-subterms-disjoint-vars* **by** *fastforce*
}
then have *x* ∉ ⋃ (*set* (*map* (*vars-term* ∘ *snd*) (*right-substs s t*)))
by *fastforce*
}
then show *?thesis* **by** *fastforce*
qed

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−
{fix *x u* **assume** (*x*,*u*) ∈ *set* (*right-substs s t*)
then obtain *p* **where** *p*:*p* ∈ *poss t t*|-*p* = *Var x p* ∈ *fun-poss s* *s*|-*p* = *u*
by (*meson right-substs-imp-props*)
{fix *y v* **assume** (*y*,*v*) ∈ *set* (*left-substs s t*)
then obtain *q* **where** *q*:*q* ∈ *poss s s*|-*q* = *Var y q* ∈ *poss t* *t*|-*q* = *v*
by (*meson left-substs-imp-props*)
with *p* **have** *p* ⊥ *q*
by (*metis Term.term.simps*(*4*) *append.right-neutral fun-poss-fun-conv fun-poss-imp-poss parallel-pos prefix-pos-diff var-pos-maximal*)
with *assms p*(*1*,*2*) *q*(*3*,*4*) **have** *x* ∉ *vars-term v*
using *fun-poss-imp-poss linear-subterms-disjoint-vars* **by** *fastforce*
}
then have *x* ∉ ⋃ (*set* (*map* (*vars-term* ∘ *snd*) (*left-substs s t*)))
by *fastforce*
}
then show *?thesis* **by** *fastforce*
qed

lemma *linear-unifier-same*:

```
  shows (linear-unifier t t) = Var
proof−
  let ?vars-left=filter (λ(-, p). p ∈ poss t) (zip (vars-term-list t)(var-poss-list t))
  have left:?vars-left = zip (vars-term-list t)(var-poss-list t)
   by (metis (no-types, lifting) filter-True split-beta var-poss-imp-poss var-poss-list-sound
zip-snd)
  let ?vars-right=filter (λ(-, q). q ∈ fun-poss t) (zip (vars-term-list t)(var-poss-list
t))
  have right:?vars-right = []
   by (metis (mono-tags, lifting) DiffE filter-False poss-simps(4) split-beta var-poss-list-sound
zip-snd)
  {fix i assume i:i < length (left-substs t t)
    let ?xi=vars-term-list t ! i
    from i have i < length (vars-term-list t)
      unfolding left-substs-def Let-def length-map left by simp
    then have left-substs t t ! i = (?xi, Var ?xi)
       unfolding left-substs-def left Let-def nth-map[OF i[unfolded left-substs-def
Let-def length-map left]]
      by (simp add: length-var-poss-list vars-term-list-var-poss-list)
  }note left-subst=this
  {fix x
    from left-subst have subst-of (left-substs t t) x = Var x
      using subst-of-id by (metis left-substs-imp-props prod.collapse)
  }
  then show ?thesis
    unfolding right-substs-def right left-substs-def left by auto
qed

lemma linear-unifier-var1:
  shows linear-unifier (Var x) t = subst x t
proof−
  have left-substs (Var x) t = [(x, t)]
     unfolding left-substs-def Let-def vars-term-list.simps var-poss-list.simps by
simp
  moreover have right-substs (Var x) t = []
    unfolding right-substs-def by simp
  ultimately show ?thesis
    by simp
qed

lemma linear-unifier-var2:
  shows linear-unifier (Fun f ts) (Var x) = subst x (Fun f ts)
proof−
  have left-substs (Fun f ts) (Var x) = []
    unfolding left-substs-def Let-def poss.simps
   by (metis (no-types, lifting) case-prodE filter-False map-is-Nil-conv set-zip-rightD
singletonD subt-at.simps(1) term.distinct(1) var-poss-iff var-poss-list-sound)
  moreover have right-substs (Fun f ts) (Var x) = [(x, Fun f ts)]
    unfolding right-substs-def by (simp add: vars-term-list.simps(1))
```

ultimately show *?thesis*
by *simp*
qed

lemma *linear-unifier-id*:
assumes $x \notin$ *vars-term s* **and** $x \notin$ *vars-term t*
shows (*linear-unifier s t*) *x* = *Var x*
using *assms* **by** (*metis* (*no-types*, *lifting*) *Set.basic-monos*(*7*) *eval-term.simps*(*1*) *map-fst-left-substs map-fst-right-substs not-elem-subst-of subst-compose subst-of-append*)

lemma *vars-subst-of*:
vars-subst (*subst-of ts*) ⊆ *set* (*map fst ts*) ∪ ⋃ (*set* (*map* (*vars-term* ∘ *snd*) *ts*))
proof(*induct ts*)
case *Nil*
show *?case* **unfolding** *subst-of-simps list.map vars-subst-def* **by** *simp*
next
case (*Cons t ts*)
have *vars-subst* (*subst* (*fst t*) (*snd t*)) ⊆ {*fst t*} ∪ (*vars-term* (*snd t*))
unfolding *vars-subst-def* **by** *auto*
with *Cons* **show** *?case* **unfolding** *subst-of-simps* **using** *vars-subst-compose*
by (*smt* (*verit*, *del-insts*) *Un-iff UnionI Union-mono comp-apply empty-iff insert-iff list.set-intros*(*1*) *list.simps*(*9*) *set-subset-Cons subset-iff*)
qed

lemma *vars-subst-linear-unifier*: *vars-subst* (*linear-unifier s t*) ⊆ *vars-term s* ∪ *vars-term t*
proof−
have *vars-subst* (*linear-unifier s t*) ⊆ (*vars-subst* (*subst-of* (*left-substs s t*))) ∪ (*vars-subst* (*subst-of* (*right-substs s t*)))
unfolding *subst-of-append* **using** *vars-subst-compose* **by** *force*
moreover have *vars-subst* (*subst-of* (*left-substs s t*)) ⊆ *vars-term s* ∪ *vars-term t*
proof−
{fix *i* **assume** *i* < *length* (*left-substs s t*)
then have *map* (*vars-term* ∘ *snd*) (*left-substs s t*) ! *i* ⊆ *vars-term t*
using *map-snd-left-substs nth-mem* **by** *fastforce*
}
then have ⋃ (*set* (*map* (*vars-term* ∘ *snd*) (*left-substs s t*))) ⊆ *vars-term t*
by (*metis Union-least in-set-conv-nth length-map*)
then show *?thesis*
using *vars-subst-of*[*of left-substs s t*] *map-fst-left-substs*
by (*metis* (*no-types*, *lifting*) *subset-trans sup.mono*)
qed
moreover have *vars-subst* (*subst-of* (*right-substs s t*)) ⊆ *vars-term s* ∪ *vars-term t*
proof−
{fix *i* **assume** *i* < *length* (*right-substs s t*)
then have *map* (*vars-term* ∘ *snd*) (*right-substs s t*) ! *i* ⊆ *vars-term s*

```
      using map-snd-right-substs nth-mem by fastforce
    }
    then have ⋃ (set (map (vars-term ∘ snd) (right-substs s t))) ⊆ vars-term s
      by (metis Union-least in-set-conv-nth length-map)
    then show ?thesis
      using vars-subst-of[of right-substs s t] map-fst-right-substs by fastforce
  qed
  ultimately show ?thesis by blast
qed

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−
  from assms have zip:ds = zip ss ts and l:length ss = length ts
    using decompose-Some by blast+
  {fix i j assume j:j < length ss and i:i < j
    from i j obtain si ti where s-t-i:(si, ti) = ds ! i ss ! i = si ts ! i = ti
      using l zip by force
    from j obtain sj tj where s-t-j:(sj, tj) = ds ! j ss ! j = sj ts ! j = tj
      using l zip by force
    have vars-term si ∩ vars-term tj = {}
      using i j s-t-i s-t-j disj l by fastforce
    moreover have vars-term si ∩ vars-term sj = {}
      using lin(1) s-t-i s-t-j i j var-in-linear-args by fastforce
    moreover have vars-term ti ∩ vars-term tj = {}
      using lin(2) s-t-i s-t-j i j l var-in-linear-args by fastforce
    moreover have vars-term ti ∩ vars-term sj = {}
      using i j s-t-i s-t-j disj l by fastforce
    ultimately have vars-subst (linear-unifier si ti) ∩ vars-subst (linear-unifier sj
tj) = {}
     using vars-subst-linear-unifier by (smt (verit, ccfv-threshold) Un-iff disjoint-iff
in-mono)
    then have vars-subst (map (λ(s,t). linear-unifier s t) ds ! i) ∩ vars-subst (map
(λ(s,t). linear-unifier s t) ds ! j) = {}
      using i j s-t-j s-t-i l zip by auto
  }
  then show ?thesis unfolding is-partition-def map-map[symmetric] length-map
zip using l by auto
qed

lemma compose-exists-subst:
  assumes compose σs x ≠ Var x
  shows ∃ i < length σs. (∀ j < i. (σs!j) x = Var x) ∧ (σs!i) x ≠ Var x
  using assms proof(induct σs)
  case (Cons σ σs)
  then show ?case proof(cases σ x = Var x)
```

case *True*
from *Cons*(*2*) **have** *compose* $\sigma s\ x \neq Var\ x$
unfolding *compose-simps subst-compose True* **by** *simp*
with *Cons*(*1*) **obtain** *i* **where** *i*:$i<length\ \sigma s\ \forall j<i.\ (\sigma s\ !\ j)\ x = Var\ x\ (\sigma s\ !\ i)\ x \neq Var\ x$ **by** *blast*
with *True* **have** $\forall j < Suc\ i.\ ((\sigma \# \sigma s)\ !\ j)\ x = Var\ x$
by (*metis less-Suc-eq-0-disj nth-Cons-0 nth-Cons-Suc*)
with *i* **show** *?thesis* **by** *auto*
qed *auto*
qed *simp*

lemma *subst-of-exists-binding*:
assumes *subst-of xs y* $\neq$ *Var y*
shows $\exists i < length\ xs.\ fst\ (xs!i) = y \wedge (\forall x \in set\ (drop\ (i+1)\ xs).\ fst\ x \neq y)$
using *assms* **proof**(*induct xs rule*:*rev-induct*)
case (*snoc x xs*)
then show *?case* **proof**(*cases fst x* = *y*)
case *False*
with *snoc*(*2*) **have** *subst-of xs y* $\neq$ *Var y*
unfolding *subst-of-append subst-compose*
by (*metis* (*no-types*, *lifting*) *empty-iff eval-term.simps*(*1*) *insert-iff subst-compose subst-ident subst-of-simps*(*1*,*3*) *term.set*(*3*))
with *snoc*(*1*) **obtain** *i* **where** *i*:$i < length\ xs\ fst\ (xs!i) = y\ \forall z \in set\ (drop\ (i+1)\ xs).\ fst\ z \neq y$ **by** *blast*
from *i*(*1*) **have** *drop* (*i*+*1*) (*xs*@[*x*]) = *drop* (*i*+*1*) *xs* @ [*x*] **by** *auto*
with *i*(*3*) *False* **have** $\forall z \in set\ (drop\ (i+1)\ (xs@[x])).\ fst\ z \neq y$ **by** *simp*
with *i*(*1*,*2*) **show** *?thesis*
by (*metis append-Cons-nth-left length-append-singleton less-Suc-eq-le less-imp-le-nat*)
qed *auto*
qed *simp*

lemma *linear-unifier-obtain-binding*:
assumes *disj*:*vars-term s* $\cap$ *vars-term t* = {} **and** *lin-s*:*linear-term s* **and** *lin-t*:*linear-term t*
and *u*:(*linear-unifier s t*) *x* = *u u* $\neq$ *Var x*
shows (*x* $\in$ *vars-term s* $\wedge$ (*x*,*u*) $\in$ *set* (*left-substs s t*)) $\vee$ (*x* $\in$ *vars-term t* $\wedge$ (*x*,*u*) $\in$ *set* (*right-substs s t*))
proof−
consider *x* $\in$ *vars-term s* | *x* $\in$ *vars-term t* | *x* $\notin$ *vars-term s* $\wedge$ *x* $\notin$ *vars-term t*
by *fastforce*
then show *?thesis* **proof**(*cases*)
case *1*
with *disj* **have** *x* $\notin$ *vars-term t* **by** *blast*
then have *right*:*subst-of* (*right-substs s t*) *x* = *Var x*
by (*meson in-mono map-fst-right-substs not-elem-subst-of*)
with *u* **have** *subst-of* (*left-substs s t*) *x* $\neq$ *Var x*
by (*simp add*: *subst-compose*)
then obtain *i u*′ **where** *i*:$i < length\ (left\text{-}substs\ s\ t)\ (left\text{-}substs\ s\ t)!i = (x, u')\ \forall subst \in set\ (drop\ (i+1)\ (left\text{-}substs\ s\ t)).\ fst\ subst \neq x$

using *subst-of-exists-binding* **by** (*metis* (*mono-tags*, *opaque-lifting*) *eq-fst-iff*)
then obtain *l1 l2* **where** *l1*:*l1* = *take i* (*left-substs s t*) **and** *l2*:*l2* = *drop* (*i*+*1*) (*left-substs s t*)
and *l1l2*:*left-substs s t* = *l1* @ [(*x*,*u*′)] @ *l2* **using** *id-take-nth-drop* **by** *fastforce*

from *i*(*3*) **have** *l2-subst*:*subst-of l2 x* = *Var x* **unfolding** *l2* **by** (*meson nth-mem subst-of-exists-binding*)
then have *1*:*subst-of* (*left-substs s t*) *x* = *u*′ · (*subst-of l1*)
unfolding *l1l2 subst-of-append subst-compose l2-subst eval-term.simps* **by** *simp*
from *i*(*1*,*2*) **obtain** *p* **where** *p*:*p* ∈ *poss t t*|-*p* = *u*′ **using** *left-substs-imp-props* **by** (*metis nth-mem*)
from *disj p* **have** *set* (*map fst* (*left-substs s t*)) ∩ (*vars-term u*′) = {}
by (*meson disjoint-iff map-fst-left-substs subsetD vars-term-subt-at*)
then have *2*:*u*′ · (*subst-of l1*) = *u*′
unfolding *l1* **by** (*smt* (*verit*, *best*) *disjoint-iff in-set-takeD not-elem-subst-of subst-apply-term-empty take-map term-subst-eq*)
then have *u*′:*subst-of* (*left-substs s t*) *x* = *u*′
using *1 2* **by** *simp*
from *i*(*1*,*2*) **have** *u*′-*elem*:(*x*, *u*′) ∈ *set* (*left-substs s t*) **by** (*metis nth-mem*)
with *u*′ *u* **show** *?thesis*
unfolding *subst-of-append subst-compose right eval-term.simps*
by (*meson map-fst-left-substs not-elem-subst-of subset-iff*)
next
case *2*
with *disj* **have** *x* ∉ *vars-term s* **by** *blast*
then have *subst-of* (*left-substs s t*) *x* = *Var x*
by (*meson in-mono map-fst-left-substs not-elem-subst-of*)
with *u* **have** *subst-of* (*right-substs s t*) *x* ≠ *Var x*
by (*metis subst-compose subst-monoid-mult.mult.left-neutral subst-of-append*)
then obtain *i u*′ **where** *i*:*i* < *length* (*right-substs s t*) (*right-substs s t*)!*i* = (*x*, *u*′) ∀ *subst* ∈ *set* (*drop* (*i*+*1*) (*right-substs s t*)). *fst subst* ≠ *x*
using *subst-of-exists-binding* **by** (*metis* (*mono-tags*, *opaque-lifting*) *eq-fst-iff*)
then obtain *l1 l2* **where** *l1*:*l1* = *take i* (*right-substs s t*) **and** *l2*:*l2* = *drop* (*i*+*1*) (*right-substs s t*)
and *l1l2*:*right-substs s t* = *l1* @ [(*x*,*u*′)] @ *l2* **using** *id-take-nth-drop* **by** *fastforce*
from *i*(*3*) **have** *l2-subst*:*subst-of l2 x* = *Var x* **unfolding** *l2* **by** (*meson nth-mem subst-of-exists-binding*)
then have *1*:*subst-of* (*right-substs s t*) *x* = *u*′ · (*subst-of l1*)
unfolding *l1l2 subst-of-append subst-compose l2-subst eval-term.simps* **by** *simp*
from *i*(*1*,*2*) **obtain** *p* **where** *p*:*p* ∈ *poss s s*|-*p* = *u*′
using *right-substs-imp-props* **by** (*metis fun-poss-imp-poss nth-mem*)
from *disj p* **have** *set* (*map fst* (*right-substs s t*)) ∩ (*vars-term u*′) = {}
by (*meson disjoint-iff map-fst-right-substs subsetD vars-term-subt-at*)
then have *2*:*u*′ · (*subst-of l1*) = *u*′
unfolding *l1* **by** (*smt* (*verit*, *best*) *disjoint-iff in-set-takeD not-elem-subst-of subst-apply-term-empty take-map term-subst-eq*)

then have u':*subst-of* (*right-substs s t*) $x = u'$
using *1 2* **by** *simp*
from *i*(*1*,*2*) **have** u'-*elem*:$(x, u') \in$ *set* (*right-substs s t*) **by** (*metis nth-mem*)
then have *set* (*map fst* (*left-substs s t*)) ∩ (*vars-term* u') = {}
using *distinct-fst-lsubsts-snd-rsubsts*[*OF lin-s*] **by** (*smt* (*verit*, *ccfv-SIG*) *Union-iff comp-apply disjoint-iff in-set-conv-nth length-map nth-map snd-conv*)
then have $u' \cdot$ (*subst-of* (*left-substs s t*)) = u'
by (*metis disjoint-iff not-elem-subst-of subst-apply-term-empty term-subst-eq*)

with *u* u'-*elem* **show** *?thesis*
unfolding *subst-of-append subst-compose* u' **by** (*metis map-fst-right-substs not-elem-subst-of subset-eq* u')
next
case *3*
then have $x \notin$ *set* (*map fst* ((*left-substs s t*) @ (*right-substs s t*)))
using *map-fst-left-substs map-fst-right-substs* **by** *fastforce*
then have (*linear-unifier s t*) x = *Var x*
by (*meson not-elem-subst-of*)
with *u* **show** *?thesis* **by** *simp*
qed
qed

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
from *assms* **have** *ds*:*ds* = *zip ss ts*
using *decompose-Some* **by** *auto*
show *?left* ⊆ *?right* **proof**
fix *x t* **assume** (*x*,*t*) ∈ *set* (*left-substs* (*Fun f ss*) (*Fun g ts*))
then obtain *p* **where** *1*:*p* ∈ *poss* (*Fun f ss*) **and** *2*:(*Fun f ss*)|-*p* = *Var x* **and** *3*:*p* ∈ *poss* (*Fun g ts*) **and** *4*:(*Fun g ts*)|-*p* = *t*
by (*meson left-substs-imp-props*)
from *1 2* **obtain** *j p*′ **where** *j1*:*j* < *length ss* **and** *p* = *j*#*p*′ **and** *p*′ ∈ *poss* (*ss*!*j*) **and** (*ss*!*j*)|-*p*′ = *Var x*
by *auto*
moreover with *3 4* **have** *j2*:*j* < *length ts* **and** *p*′ ∈ *poss* (*ts*!*j*) **and** (*ts*!*j*)|-*p*′ = *t*
by *auto*
ultimately have (*x*,*t*) ∈ *set* (*left-substs* (*ss*!*j*) (*ts*!*j*))
by (*meson props-imp-left-substs*)
moreover have ((*ss*!*j*),(*ts*!*j*)) ∈ *set ds*
unfolding *ds* **using** *j1 j2* **by** (*metis length-zip min-less-iff-conj nth-mem nth-zip*)
ultimately show (*x*,*t*) ∈ (⋃ *e*∈*set ds*. *set* (*left-substs* (*fst e*) (*snd e*)))
by *force*

qed
show *?right* ⊆ *?left* **proof**
fix *x t* **assume** (*x*,*t*) ∈ (⋃ *e*∈*set ds*. *set* (*left-substs* (*fst e*) (*snd e*)))
then obtain *j* **where** *j1*:*j* < *length ss* **and** *j2*:*j* < *length ts* **and** (*x*,*t*) ∈ *set* (*left-substs* (*ss*!*j*) (*ts*!*j*))
unfolding *ds* **by** (*metis* (*no-types*, *lifting*) *UN-E in-set-zip*)
then obtain *p* **where** *1*:*p* ∈ *poss* (*ss*!*j*) **and** *2*:(*ss*!*j*)|-*p* = *Var x* **and** *3*:*p* ∈ *poss* (*ts*!*j*) **and** *4*:(*ts*!*j*)|-*p* = *t*
by (*meson left-substs-imp-props*)
then have *j*#*p* ∈ *poss* (*Fun f ss*) **and** (*Fun f ss*)|-(*j*#*p*) = *Var x* **and** (*j*#*p*) ∈ *poss* (*Fun g ts*) **and** (*Fun g ts*)|-(*j*#*p*) = *t*
using *j1 j2* **by** *auto*
then show (*x*,*t*) ∈ *set* (*left-substs* (*Fun f ss*) (*Fun g ts*))
by (*meson props-imp-left-substs*)
qed
qed

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
from *assms* **have** *ds*:*ds* = *zip ss ts*
using *decompose-Some* **by** *auto*
show *?left* ⊆ *?right* **proof**
fix *x t* **assume** (*x*,*t*) ∈ *set* (*right-substs* (*Fun f ss*) (*Fun g ts*))
then obtain *q* **where** *1*:*q* ∈ *fun-poss* (*Fun f ss*) **and** *2*:(*Fun f ss*)|-*q* = *t* **and** *3*:*q* ∈ *poss* (*Fun g ts*) **and** *4*:(*Fun g ts*)|-*q* = *Var x*
by (*meson right-substs-imp-props*)
from *3 4* **obtain** *j q*′ **where** *j1*:*j* < *length ts* **and** *q* = *j*#*q*′ **and** *q*′ ∈ *poss* (*ts*!*j*) **and** (*ts*!*j*)|-*q*′ = *Var x*
by *auto*
moreover with *1 2* **have** *j2*:*j* < *length ss* **and** *q*′ ∈ *fun-poss* (*ss*!*j*) **and** (*ss*!*j*)|-*q*′ = *t*
by *auto*
ultimately have (*x*,*t*) ∈ *set* (*right-substs* (*ss*!*j*) (*ts*!*j*))
by (*meson props-imp-right-substs*)
moreover have ((*ss*!*j*),(*ts*!*j*)) ∈ *set ds*
unfolding *ds* **using** *j1 j2* **by** (*metis length-zip min-less-iff-conj nth-mem nth-zip*)
ultimately show (*x*,*t*) ∈ (⋃ *e*∈*set ds*. *set* (*right-substs* (*fst e*) (*snd e*)))
by *force*
qed
show *?right* ⊆ *?left* **proof**
fix *x t* **assume** (*x*,*t*) ∈ (⋃ *e*∈*set ds*. *set* (*right-substs* (*fst e*) (*snd e*)))
then obtain *j* **where** *j1*:*j* < *length ss* **and** *j2*:*j* < *length ts* **and** (*x*,*t*) ∈ *set* (*right-substs* (*ss*!*j*) (*ts*!*j*))
unfolding *ds* **by** (*metis* (*no-types*, *lifting*) *UN-E in-set-zip*)
then obtain *q* **where** *1*:*q* ∈ *fun-poss* (*ss*!*j*) **and** *2*:(*ss*!*j*)|-*q* = *t* **and** *3*:*q* ∈

poss (*ts*!*j*) **and** *4*:(*ts*!*j*)|-*q* = *Var x*
by (*meson right-substs-imp-props*)
then have *j*#*q* ∈ *fun-poss* (*Fun f ss*) **and** (*Fun f ss*)|-(*j*#*q*) = *t* **and** (*j*#*q*) ∈ *poss* (*Fun g ts*) **and** (*Fun g ts*)|-(*j*#*q*) = *Var x*
using *j1 j2* **by** *auto*
then show (*x*,*t*) ∈ *set* (*right-substs* (*Fun f ss*) (*Fun g ts*))
by (*meson props-imp-right-substs*)
qed
qed

lemma *subst-compose-id*:
assumes ⋀τ. τ ∈ *set* τs ⟹ *t* · τ = *t*
shows *t* · (*compose* τs) = *t*
using *assms* **by**(*induct* τs) *simp-all*

lemma *subst-compose-distinct-vars*:
assumes σ = *compose* τs **and** *part*:*is-partition* (*map vars-subst* τs)
and τ*i*:τ*i* ∈ *set* τs **and** *s*:τ*i x* = *s s* ≠ *Var x*
shows σ *x* = *s*
proof−
from τ*i* **obtain** *i* **where** *i*:*i* < *length* τs τs ! *i* = τ*i*
by (*metis in-set-idx*)
then have τs:τs = (*take i* τs) @ τ*i* # (*drop* (*Suc i*) τs)
using *id-take-nth-drop* **by** *blast*
from *s* **have** *x-vars-subst*:*x* ∈ *vars-subst* τ*i*
by (*metis fun-upd-same fun-upd-triv subst-apply-term-empty subst-compose vars-subst-compose-update*)
{fix *j* **assume** *j* < *i*
with *part i x-vars-subst* **have** *x* ∉ *vars-subst* (τs ! *j*)
unfolding *is-partition-alt is-partition-alt-def*
by (*metis* (*no-types*, *lifting*) *Int-iff dual-order.strict-trans equals0D is-partition-def length-map nth-map part*)
then have (τs ! *j*) *x* = *Var x*
unfolding *vars-subst-def* **by** (*meson UnI1 notin-subst-domain-imp-Var*)
}
then have *take-i-*τs:*compose* (*take i* τs) *x* = *Var x*
using *subst-compose-id*[*of take i* τs *Var x*] **using** *in-set-idx* **by** *force*
{fix *y* **assume** *y* ∈ *vars-term s*
with *s* **have** *y-vars-subst*:*y* ∈ *vars-subst* τ*i*
unfolding *vars-subst-def* **by** (*metis UnI2 Union-iff image-eqI notin-subst-domain-imp-Var subst-range.simps*)
{fix *j* **assume** *i* < *j j* < *length* τs
with *part i y-vars-subst* **have** *y* ∉ *vars-subst* (τs ! *j*)
unfolding *is-partition-alt is-partition-alt-def*
by (*metis* (*no-types*, *lifting*) *Int-iff equals0D is-partition-def length-map nth-map part*)
then have (τs ! *j*) *y* = *Var y*
unfolding *vars-subst-def* **by** (*meson UnI1 notin-subst-domain-imp-Var*)
}

then have *compose* (*drop* (*Suc i*) τs) *y* = *Var y*
using *subst-compose-id*[*of drop* (*Suc i*) τs *Var y*] **using** *in-set-idx* **by** *force*
}
then have $s \cdot$ (*compose* (*drop* (*Suc i*) τs)) = s
by (*simp add*: *term-subst-eq*)
with *take-i-*τs *s*(*1*) *i* **show** *?thesis*
by (*metis* τs *assms*(*1*) *compose-append compose-simps*(*3*) *eval-term.simps*(*1*) *subst-compose*)
qed

lemma *subst-id-compose*:
assumes σ = *compose* τs **and** *part*:*is-partition* (*map vars-subst* τs)
and $t \cdot \sigma = t$
and $\tau \in$ *set* τs
shows $t \cdot \tau = t$
using *assms subst-compose-distinct-vars* **by** (*metis* (*full-types*) *subst-apply-term-empty term-subst-eq-conv*)

lemma *compose-subst-of*:
assumes *set ss* = $\bigcup$ (*set* ‘ *set ss*′)
and *is-partition* (*map* (*vars-term* ∘ *snd*) *ss*) **and** *distinct* (*map fst ss*)
and *set* (*map fst ss*) ∩ $\bigcup$ (*set* (*map* (*vars-term* ∘ *snd*) *ss*)) = {}
and *is-partition* (*map vars-subst* (*map subst-of ss*′))
shows *subst-of ss* = *compose* (*map subst-of ss*′) (**is** *?*σ = *?*τ)
proof
fix *x*
show *?*σ *x* = *?*τ *x* **proof**(*cases x* ∈ *set* (*map fst ss*))
case *True*
then obtain *s* **where** *s*:(*x*, *s*) ∈ *set ss*
by *fastforce*
then have σ-*x*:*?*σ *x* = *s*
using *assms*(*3*) **by** (*smt* (*verit*) *UN-I assms*(*4*) *case-prodI2 disjoint-iff eq-key-imp-eq-value list.set-map o-apply prod.sel*(*2*) *subst-of-apply*)
from *s* **have** *s-x*:*s* ≠ *Var x*
using *assms*(*4*) **by** *fastforce*
from *s* **obtain** *ssi* **where** *ssi*:(*x*, *s*) ∈ *set ssi ssi* ∈ *set ss*′
using *assms*(*1*) **by** *auto*
then have *subst-of ssi x* = *s*
using *assms*(*1*,*3*,*4*) **by** (*smt* (*verit*, *ccfv-threshold*) *UN-I case-prodI2 disjoint-iff eq-key-imp-eq-value image-iff list.set-map o-apply snd-conv subst-of-apply*)
with *assms*(*5*) **have** *?*τ *x* = *s*
using *subst-compose-distinct-vars ssi*(*2*) *s-x* **by** (*smt* (*verit*, *del-insts*) *in-set-idx length-map nth-map nth-mem*)
with σ-*x* **show** *?thesis* **by** *simp*
next
case *False*
then have σ-*x*:*?*σ *x* = *Var x*
by (*simp add*: *not-elem-subst-of*)
{fix *ssi* **assume** *ssi* ∈ *set ss*′

```
    with False assms(1) have x ∉ set (map fst ssi)
      by auto
    then have (subst-of ssi) x = Var x
      by (simp add: not-elem-subst-of)
  }
  then have ?τ x = Var x
   using subst-compose-id by (smt (verit, ccfv-SIG) eval-term.simps(1) image-iff
list.set-map)
  with σ-x show ?thesis by simp
 qed
qed

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) ∩ vars-term (Fun g ts) = {}
  and decompose (Fun f ss) (Fun g ts) = Some ds
  and i:i < length ds and σ:σ = linear-unifier (fst (ds!i)) (snd (ds!i))
  and j:j < length ds j ≠ i
 shows fst (ds!j) · σ = fst (ds!j) ∧ snd (ds!j) · σ = snd (ds!j)
proof−
 from assms have zip:ds = zip ss ts and l:length ss = length ts
  using decompose-Some by blast+
 from i j obtain si ti where s-t-i:ds ! i = (si, ti) ss ! i = si ts ! i = ti
  using l zip by force
 from j obtain sj tj where s-t-j:ds ! j = (sj, tj) ss ! j = sj ts ! j = tj
  using l zip by force
 have vars-term sj ∩ vars-term ti = {}
  using i j s-t-i s-t-j disj l zip by fastforce
 moreover have vars-term sj ∩ vars-term si = {}
  using lin(1) s-t-i s-t-j i j var-in-linear-args
  by (metis Int-emptyI l length-map map-fst-zip zip)
 moreover have vars-term tj ∩ vars-term ti = {}
  using lin(2) s-t-i s-t-j i j l var-in-linear-args
  by (metis Int-emptyI l length-map map-fst-zip zip)
 moreover have vars-term tj ∩ vars-term si = {}
  using i j s-t-i s-t-j disj l zip by fastforce
 moreover from σ s-t-i have vars-subst σ ⊆ vars-term si ∪ vars-term ti
  by (metis fst-conv snd-conv vars-subst-linear-unifier)
 ultimately show ?thesis
  unfolding s-t-i s-t-j fst-conv snd-conv
  by (metis inf-sup-distrib1 subst-apply-term-ident sup.absorb-iff2 sup-bot.neutr-eq-iff
vars-subst-def)
qed

lemma linear-unifier-decompose:
 assumes 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 linear-unifier (Fun f ss) (Fun g ts) = compose (map (λ(s,t). linear-unifier
```

s t) *ds*)
proof−
let *?ls*=*left-substs* (*Fun f ss*) (*Fun g ts*) **and** *?rs*=*right-substs* (*Fun f ss*) (*Fun g ts*)
have *left*:*set ?ls* = (⋃(*s*, *t*)∈*set ds*. *set* (*left-substs s t*))
using *decompose-left-substs*[*OF ds*] **by** *auto*
have *right*:*set ?rs* = (⋃(*s*, *t*)∈*set ds*. *set* (*right-substs s t*))
using *decompose-right-substs*[*OF ds*] **by** *auto*
from *left right* **have** *sets*:*set* (*?ls* @ *?rs*) = ⋃ (*set* ‘ *set* (*map* (λ(*s*, *t*). *left-substs s t* @ *right-substs s t*) *ds*))
by *auto*
{fix *l* **assume** *l* ∈ (*set* (*map* (*vars-term* ∘ *snd*) *?ls*))
then obtain *t*′ **where** *t*′ ∈ *set* (*map snd ?ls*) **and** *vars-term t*′ = *l*
by *auto*
then have *l* ⊆ *vars-term* (*Fun g ts*)
using *map-snd-left-substs* **by** *blast*
}
then have *1*:⋃ (*set* (*map* (*vars-term* ∘ *snd*) *?ls*)) ⊆ *vars-term* (*Fun g ts*)
using *Union-least* **by** *blast*
{fix *r* **assume** *r* ∈ (*set* (*map* (*vars-term* ∘ *snd*) *?rs*))
then obtain *t*′ **where** *t*′ ∈ *set* (*map snd ?rs*) **and** *vars-term t*′ = *r*
by *auto*
then have *r* ⊆ *vars-term* (*Fun f ss*)
using *map-snd-right-substs* **by** *blast*
}
then have *2*:⋃ (*set* (*map* (*vars-term* ∘ *snd*) *?rs*)) ⊆ *vars-term* (*Fun f ss*)
using *Union-least* **by** *blast*
have *snd-disj*:⋃ (*set* (*map* (*vars-term* ∘ *snd*) *?ls*)) ∩ ⋃ (*set* (*map* (*vars-term* ∘ *snd*) *?rs*)) = {}
using *1 2 assms*(*3*) **by** *blast*
then have *part*:*is-partition* (*map* (*vars-term* ∘ *snd*) (*?ls* @ *?rs*))
using *is-partition-append*[*OF is-partition-map-snd-left-substs*[*OF assms*(*2*,*1*)] *is-partition-map-snd-right-substs*[*OF assms*(*2*,*1*)]]
unfolding *length-map map-append* **by** (*simp add*: *Union-disjoint*)
have *dist*:*distinct* (*map fst* (*?ls* @ *?rs*))
using *distinct-append distinct-map-fst-left-substs*[*OF assms*(*1*)] *distinct-map-fst-right-substs*[*OF assms*(*2*)] *map-fst-left-substs map-fst-right-substs*
by (*smt* (*verit*, *del-insts*) *disj inf*.*orderE inf-assoc inf-bot-right inf-left-commute map-append*)
have *set* (*map fst ?ls*) ∩ ⋃ (*set* (*map* (*vars-term* ∘ *snd*) *?ls*)) = {}
by (*meson 1 disj disjoint-iff map-fst-left-substs subsetD*)
moreover have *set* (*map fst ?ls*) ∩ ⋃ (*set* (*map* (*vars-term* ∘ *snd*) *?rs*)) = {}
using *assms*(*1*) *distinct-fst-lsubsts-snd-rsubsts* **by** *blast*
moreover have *set* (*map fst ?rs*) ∩ ⋃ (*set* (*map* (*vars-term* ∘ *snd*) *?rs*)) = {}
by (*meson 2 disj disjoint-iff map-fst-right-substs subsetD*)
moreover have *set* (*map fst ?rs*) ∩ ⋃ (*set* (*map* (*vars-term* ∘ *snd*) *?ls*)) = {}
using *assms*(*2*) *distinct-fst-rsubsts-snd-lsubsts* **by** *blast*
ultimately have *disj*:*set* (*map fst* (*?ls* @ *?rs*)) ∩ ⋃ (*set* (*map* (*vars-term* ∘ *snd*) (*?ls* @ *?rs*))) = {}

unfolding *map-append set-append* **by** (*simp add*: *boolean-algebra.conj-disj-distrib boolean-algebra.conj-disj-distrib2*)
have *part2*:*is-partition* (*map vars-subst* (*map subst-of* (*map* (λ(*s*, *t*). *left-substs s t* @ *right-substs s t*) *ds*)))
using *decompose-is-partition-vars-subst*[*OF assms*(*1*,*2*,*3*,*4*)]
by (*metis* (*mono-tags*, *lifting*) *case-prod-beta length-map map-nth-eq-conv*)
show *?thesis* **using** *compose-subst-of*[*OF sets part dist disj part2*]
by (*smt* (*verit*, *del-insts*) *case-prod-unfold length-map map-nth-eq-conv*)
qed

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
using *assms* **proof**(*induct arbitrary*: *res substs* τ *rule*:*unify.induct*)
case (*2 f ss g ts E*)
from *2*(*2*) **obtain** *ds* **where** *ds'*:*decompose* (*Fun f ss*) (*Fun g ts*) = *Some ds*
unfolding *unify.simps* **by** *fastforce*
then have *ds*:*ds* = *zip ss ts* **and** *l*:*length ss* = *length ts*
by *fastforce*+
with *2*(*4*) **have** ∀ *t* ∈ *set* (*map fst ds*). *linear-term t*
using *map-fst-zip* **by** (*metis* (*no-types*, *lifting*) *UnCI fst-conv linear-term.simps*(*2*) *list.set-intros*(*1*) *list.simps*(*9*))
moreover from *2*(*4*) *ds l* **have** ∀ *t* ∈ *set* (*map snd ds*). *linear-term t*
using *map-snd-zip* **by** (*metis* (*no-types*, *lifting*) *UnCI linear-term.simps*(*2*) *list.set-intros*(*1*) *list.simps*(*9*) *snd-conv*)
ultimately have *lin*:∀ *a*∈*set* (*map fst* (*ds* @ *E*)) ∪ *set* (*map snd* (*ds* @ *E*)). *linear-term a*
using *2*(*4*) **by** (*metis UnE UnI1 UnI2 list.set-intros*(*2*) *list.simps*(*9*) *map-append set-append*)
have *lin-f-g*:*linear-term* (*Fun f ss*) *linear-term* (*Fun g ts*)
using *2*(*4*) **by** *auto*
from *2*(*6*) **have** *vars*:*vars-term* (*Fun f ss*) ∩ *vars-term* (*Fun g ts*) = {}
by *fastforce*
from *ds'* *2*(*2*) **have** *unif*:*unify* (*ds* @ *E*) *substs* = *Some res*
by *auto*
have *compose* (*map* (λ*a*. *case a of* (*s*, *t*) ⇒ *linear-unifier s t*) *ds*) = *linear-unifier* (*Fun f ss*) (*Fun g ts*)
using *linear-unifier-decompose*[*OF lin-f-g vars ds'*] **by** *fastforce*
then have τ*2*:*compose* (*subst-of substs* # *map* (λ*a*. *case a of* (*s*, *t*) ⇒ *linear-unifier s t*) (*ds* @ *E*)) = τ

```
    using 2(3) compose-append by simp
  {fix i j σ assume i:i < j and j:j < length (ds @ E) and σ:σ = linear-unifier
(fst ((ds @ E) ! i)) (snd ((ds @ E) ! i))
    have fst ((ds @ E) ! j) · σ = fst ((ds @ E) ! j) ∧ snd ((ds @ E) ! j) · σ = snd
((ds @ E) ! j)
    proof(cases i < length ds)
      case True
      then have σ:σ = linear-unifier (fst (ds ! i)) (snd (ds ! i))
        by (simp add: σ dual-order.strict-trans i nth-append)
      show ?thesis proof(cases j < length ds)
        case True
        have lin:linear-term (Fun f ss) linear-term (Fun g ts)
          using 2(4) by simp+
        show ?thesis
          using linear-term-decompose-subst-id[OF lin vars ds' ‹i < length ds› σ
True] i True
          by (simp add: j nat-neq-iff nth-append)
      next
        case False
        let ?j'=j − length ds
        let ?τ=linear-unifier (Fun f ss) (Fun g ts)
        from False j have ?j' < length E
          by fastforce
        then have fst:fst (E ! ?j') · ?τ = fst (E ! ?j') and snd:snd (E ! ?j') · ?τ
= snd (E ! ?j')
          using 2(5) by force+
        have fst (E ! ?j') · σ = fst (E ! ?j')
          using subst-id-compose[OF linear-unifier-decompose[OF lin-f-g vars ds']
decompose-is-partition-vars-subst[OF lin-f-g vars ds']]
        by (smt (verit, best) True σ ds fst in-set-conv-nth l length-map map2-map-map
map-fst-zip map-snd-zip nth-map)
        moreover have snd (E ! ?j') · σ = snd (E ! ?j')
          using subst-id-compose[OF linear-unifier-decompose[OF lin-f-g vars ds']
decompose-is-partition-vars-subst[OF lin-f-g vars ds']]
        by (smt (verit, best) True σ ds snd in-set-conv-nth l length-map map2-map-map
map-fst-zip map-snd-zip nth-map)
        ultimately show ?thesis
          by (simp add: False nth-append)
      qed
    next
      case False
      let ?i'=i − length ds
      have i':?i' < length E
        using False i j by force
      from σ have σ':σ = linear-unifier (fst (E ! ?i')) (snd (E ! ?i'))
        by (simp add: False nth-append)
      let ?j'=j − length ds
      from False i j have ?i' < ?j'
        by simp
```

```
    moreover with j have ?j' < length E
      by fastforce
    ultimately show ?thesis
      using 2(5) i' σ' by (smt (verit, best) length-nth-simps(2) nat-diff-split
not-less-eq not-less-zero nth-Cons-Suc nth-append)
   qed
  }
  moreover
  { fix i assume i:i < length (ds @ E)
   have vars-term (fst ((ds @ E) ! i)) ∩ vars-term (snd ((ds @ E) ! i)) = {}
   proof(cases i < length ds)
    case True
    with ds have vars-term (fst (ds!i)) ⊆ vars-term (Fun f ss)
      using nth-mem by auto
    moreover from True ds have vars-term (snd (ds!i)) ⊆ vars-term (Fun g ts)
      using nth-mem by auto
    ultimately show ?thesis
     using 2(6) True by (metis Int-mono bot.extremum-uniqueI nth-append vars)
   next
    case False
    let ?i'=i − length ds
    have i':?i' < length E
      using False i by force
    with 2(6) have vars-term (fst (E ! ?i')) ∩ vars-term (snd (E ! ?i')) = {}
      by force
    then show ?thesis
      by (simp add: False nth-append)
   qed
  }
  ultimately show ?case
   using 2(1)[OF ds' unif τ2 lin] by blast
next
  case (3 x t E)
  show ?case proof(cases t = Var x)
   case True
   from 3(3) have unif:unify E substs = Some res
    unfolding True unify.simps by simp
   from 3(4) have τ2:compose (subst-of substs # map (λa. case a of (s, t) ⇒
linear-unifier s t) E) = τ
   unfolding True append-Cons list.map compose-simps using linear-unifier-same
by (metis Var-subst-compose old.prod.case)
   from 3(5) have lin:∀ a∈set (map fst E) ∪ set (map snd E). linear-term a
    by simp
   from 3(6) have ⋀i σ. i < length E ⟹ σ = linear-unifier (fst (E ! i)) (snd
(E ! i)) ⟹
        (∀ j<length E. i < j ⟶ fst (E ! j) · σ = fst (E ! j) ∧ snd (E ! j) · σ =
snd (E ! j))
    by (metis length-nth-simps(2) not-less-eq nth-Cons-Suc)
   moreover have (⋀i. i < length E ⟹ vars-term (fst (E ! i)) ∩ vars-term (snd
```

```
(E ! i)) = {})
      using 3(7) by fastforce
    ultimately show ?thesis using 3(1)[OF True unif τ2 lin] by simp
  next
    case False
    with 3(3) have x:x ∉ vars-term t
      by fastforce
    with 3(3) False have unif:unify (subst-list (subst x t) E) ((x, t) # substs) =
Some res
      by simp
    let ?σ=(subst x t)
    have σ:linear-unifier (Var x) t = ?σ
      using linear-unifier-var1 by simp
    from 3(7) have subst-list:subst-list (subst x t) E = E
    proof−
      {fix j assume j < length E
        then have j:Suc j < length ((Var x, t) # E)
          by simp
        with 3(6)[of 0 Suc j ?σ] σ have fst (E ! j) · ?σ = fst (E ! j) ∧ snd (E ! j)
· ?σ = snd (E ! j)
          by (metis fst-conv length-nth-simps(2) nth-Cons-0 nth-Cons-Suc snd-conv
zero-less-Suc)
      }
      then show ?thesis
        unfolding subst-list-def by (simp add: map-nth-eq-conv)
    qed
     have τ2:compose (subst-of ((x, t) # substs) # map (λa. case a of (s, t) ⇒
linear-unifier s t) (subst-list (subst x t) E)) = τ
      using 3(4) unfolding subst-list list.map prod.case σ subst-of-simps(3) com-
pose-append fst-conv snd-conv compose-simps(1,3)
      using subst-compose-assoc by blast
    from 3(5) have lin:∀ a∈set (map fst (subst-list (subst x t) E)) ∪ set (map snd
(subst-list (subst x t) E)). linear-term a
      unfolding subst-list by simp
    {fix i j σ assume i:i < j and j:j < length E and σ'':σ = linear-unifier (fst
(E ! i)) (snd (E ! i))
     with 3(6) have 1: fst (E ! j) · σ = fst (E ! j) ∧ snd (E ! j) · σ = snd (E ! j)
       by (metis length-nth-simps(2) not-less-eq nth-Cons-Suc)
    }
   moreover have (⋀i. i < length E ⟹ vars-term (fst (E ! i)) ∩ vars-term (snd
(E ! i)) = {})
      using 3(7) by fastforce
    ultimately show ?thesis
     using 3(2)[OF False x unif τ2 lin] 3(7) unfolding subst-list subst-of-simps(3)
by simp
  qed
next
  case (4 f ts x E)
  from 4(2) have x:x ∉ vars-term (Fun f ts)
```

```
    by fastforce
  with 4(2) have unif:unify (subst-list (subst x (Fun f ts)) E) ((x, Fun f ts) #
substs) = Some res
    by auto
  let ?σ=(subst x (Fun f ts))
  have σ:linear-unifier (Fun f ts) (Var x)  = ?σ
    using linear-unifier-var2 by simp
  have subst-list:subst-list (subst x (Fun f ts)) E = E
  proof−
    {fix j assume j < length E
      then have Suc j < length ((Fun f ts, Var x) # E)
        by simp
      with 4(5) σ have fst (E ! j) · ?σ = fst (E ! j) ∧ snd (E ! j) · ?σ = snd (E
! j)
        by (metis fst-conv length-nth-simps(2) nth-Cons-0 nth-Cons-Suc snd-conv
zero-less-Suc)
    }
    then show ?thesis
      unfolding subst-list-def by (simp add: map-nth-eq-conv)
  qed
  have τ2:compose (subst-of ((x, Fun f ts) # substs) # map (λa. case a of (s, t)
⇒ linear-unifier s t) (subst-list (subst x (Fun f ts)) E)) = τ
    using 4(3) unfolding subst-list list.map prod.case σ subst-of-simps(3) com-
pose-append fst-conv snd-conv compose-simps(1,3) by (simp add: subst-compose-assoc)
  from 4(4) have lin:∀ a∈set (map fst (subst-list (subst x (Fun f ts)) E)) ∪ set
(map snd (subst-list (subst x (Fun f ts)) E)). linear-term a
    unfolding subst-list by simp
  {fix i j σ assume i:i < j and j:j < length E and σ′:σ = linear-unifier (fst (E
! i)) (snd (E ! i))
    with 4(5) have 1:fst (E ! j) · σ = fst (E ! j) ∧ snd (E ! j) · σ = snd (E ! j)
      by (metis length-nth-simps(2) not-less-eq nth-Cons-Suc)
  }
  moreover have (⋀i. i < length E ⟹ vars-term (fst (E ! i)) ∩ vars-term (snd
(E ! i)) = {})
    using 4(6) by fastforce
  ultimately show ?case
    using 4(1)[OF x unif τ2 lin] 4(6) unfolding subst-list by simp
qed auto

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−
  let ?tau=linear-unifier s t
  from unif have mgu s t ≠ None
    by (meson mgu-complete)
  then obtain us where us:unify [(s, t)] [] = Some us
    unfolding mgu-def by fastforce
```

have *tau*:*compose* (*subst-of* [] # *map* (λ(*s*, *t*). *linear-unifier s t*) [(*s*, *t*)]) = *?tau*
by *simp*
have *lin*:∀ *t*∈*set* (*map fst* [(*s*, *t*)]) ∪ *set* (*map snd* [(*s*, *t*)]). *linear-term t*
using *distinct distinct-vars-linear-term* **by** *auto*
have *vars-term s* ∩ *vars-term t* = {}
using *distinct* **by** *simp*
then have *subst-of us* = *?tau*
using *unify-linear-terms*[*OF us tau lin*] **by** *simp*
then show *?thesis*
using *us* **by** (*simp add*: *mgu-def*)
qed

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*
using *assms* **by** (*fastforce simp*: *is-mgu-def unifiers-def*)

lemma *subst-set-insert* [*simp*]:
subst-set σ (*insert e E*) = *insert* (*fst e* · σ, *snd e* · σ) (*subst-set* σ *E*)
by (*auto simp*: *subst-set-def*)

lemma *unifiable-UnD* [*dest*]:
unifiable (*M* ∪ *N*) ⟹ *unifiable M* ∧ *unifiable N*
by (*auto simp*: *unifiable-def*)

lemma *supt-imp-not-unifiable*:
assumes *s* ⊳ *t*
shows ¬ *unifiable* {(*t*, *s*)}
proof
assume *unifiable* {(*t*, *s*)}
then obtain σ **where** σ ∈ *unifiers* {(*t*, *s*)}
by (*auto simp*: *unifiable-def*)
then have *t* · σ = *s* · σ **by** (*auto*)
moreover have *s* · σ ⊳ *t* · σ
using *assms* **by** (*metis instance-no-supt-imp-no-supt*)
ultimately show *False* **by** *auto*

qed

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

lemma *unifiers-Int1* [*simp*]:
(*s*, *t*) ∈ *E* ⟹ *unifiers* {(*s*, *t*)} ∩ *unifiers E* = *unifiers E*
by (*auto simp*: *unifiers-def*)

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
using *assms*
proof (*induct p arbitrary*: *q l2*)
case (*Cons i p*)
from *this*(*3*) **obtain** *f ls* **where**
l2[*simp*]: *l2* = *Fun f ls* **and**
i: *i* < *length ls* **and**
p: *p* ∈ *poss* (*ls* ! *i*)
by (*cases l2*) (*auto*)
then have *l2i*: *l2* |- ((*i* # *p*)) = *ls* ! *i* |- *p* **by** *auto*
have *linear-term* (*ls* ! *i*) **using** *Cons*(*4*) *l2 i* **by** *simp*
moreover have *vars-term l1* ∩ *vars-term* (*ls* ! *i*) = {} **using** *Cons*(*5*) *l2 i* **by** *force*
ultimately have *IH*: ⋀*q*. *q* ∈ *poss* (*ls* ! *i*) ⟹ *p* ⊥ *q* ⟹ *ls* ! *i* |- *q* = *ls* ! *i* |- *q* · σ
using *Cons*(*1*)[*OF Cons*(*2*)[*unfolded l2i*] *p*] **by** *blast*
from *Cons*(*7*) **obtain** *j q*′ **where** *q*: *q* = *j* # *q*′ **by** (*cases q*) *auto*
show *?case*
proof (*cases j* = *i*)
case *True* **with** *Cons*(*6*,*7*) *IH q* **show** *?thesis* **by** *simp*
next
case *False*
from *Cons*(*6*) *q* **have** *j*: *j* < *length ls* **by** *simp*
{ fix *y*
assume *y*: *y* ∈ *vars-term* (*l2* |- *q*)
let *?τ* = λ*x*. *if x* = *y then Var y else* σ *x*
from *y Cons*(*6*) *q j* **have** *yj*:*y* ∈ *vars-term* (*ls* ! *j*)
by *simp* (*meson subt-at-imp-supteq subteq-Var-imp-in-vars-term supteq-Var supteq-trans*)
{ fix *i j*
assume *j*:*j* < *length ls* **and** *i*:*i* < *length ls* **and** *neq*: *i* ≠ *j*
from *j Cons*(*4*) **have** ∀ *i* < *j*. *vars-term* (*ls* ! *i*) ∩ *vars-term* (*ls* ! *j*) = {}

by (*auto simp* : *is-partition-def*)
moreover from *i Cons*(*4*) **have** ∀ *j* < *i*. *vars-term* (*ls* ! *i*) ∩ *vars-term* (*ls* ! *j*) = {}
by (*auto simp* : *is-partition-def*)
ultimately have *vars-term* (*ls* ! *i*) ∩ *vars-term* (*ls* ! *j*) = {}
using *neq* **by** (*cases i* < *j*) *auto*
}
from *this*[*OF i j False*] **have** *y* ∉ *vars-term* (*ls* ! *i*) **using** *yj* **by** *auto*
then have *y* ∉ *vars-term* (*l2* |- ((*i* # *p*)))
by (*metis l2i p subt-at-imp-supteq subteq-Var-imp-in-vars-term supteq-Var supteq-trans*)
then have ∀ *x* ∈ *vars-term* (*l2* |- ((*i* # *p*))). *?τ x* = *σ x* **by** *auto*
then have *l2τσ*: *l2* |- ((*i* # *p*)) · *?τ* = *l2* |- ((*i* # *p*)) · *σ* **using** *term-subst-eq*[*of - σ ?τ*] **by** *simp*
from *Cons*(*5*) **have** *y* ∉ *vars-term l1* **using** *y Cons*(*6*) *vars-term-subt-at* **by** *fastforce*
then have ∀ *x* ∈ *vars-term l1*. *?τ x* = *σ x* **by** *auto*
then have *l1τσ*:*l1* · *?τ* = *l1* · *σ* **using** *term-subst-eq*[*of - σ ?τ*] **by** *simp*
have *l1* · *σ* = *l2* |- (*i* # *p*) · *σ* **using** *Cons*(*2*) **unfolding** *is-imgu-def* **by** *auto*
then have *l1* · *?τ* = *l2* |- (*i* # *p*) · *?τ* **using** *l1τσ l2τσ* **by** *simp*
then have *?τ* ∈ *unifiers* {(*l2* |- (*i* # *p*), *l1*)} **unfolding** *unifiers-def* **by** *simp*
with *Cons*(*2*) **have** *τσ*:*?τ* = *σ* ∘$_s$ *?τ* **unfolding** *is-imgu-def* **by** *blast*
have *Var y* = *Var y* · *σ*
proof (*rule ccontr*)
let *?x* = *Var y* · *σ*
assume ∗:*Var y* ≠ *?x*
have *Var y* = *Var y* · *?τ* **by** *auto*
also have ... = (*Var y* · *σ*) · *?τ* **using** *τσ subst-subst* **by** *metis*
finally have *xy*:*?x* · *σ* = *Var y* **using** ∗ **by** (*cases σ y*) *auto*
have *σ* ∘$_s$ *σ* = *σ* **using** *Cons*(*2*) **unfolding** *is-imgu-def* **by** *auto*
then have *?x* · (*σ* ∘$_s$ *σ*) = *Var y* **using** *xy* **by** *auto*
moreover have *?x* · *σ* · *σ* = *?x* **using** *xy* **by** *auto*
ultimately show *False* **using** ∗ **by** *auto*
qed
}
then show *?thesis* **by** (*simp add*: *term-subst-eq*)
qed
qed *auto*

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 (*induct rule*: *compute-NF.raw-induct*[*OF - res*, *of* λ *g a b*. *g* = *f* ⟶ (*a*,*b*) ∈ *r*^*, *THEN mp*[*OF - refl*]])
case (*1 cnf g a b*)
show *?case*
proof
assume *g* = *f*
note *1* = *1*[*unfolded this*]
show (*a*,*b*) ∈ *r*^*
proof (*cases f a*)
case *None*
with *1*(*2*) **show** *?thesis* **by** *simp*
next
case (*Some c*)
from *1*(*2*)[*unfolded this*] **have** *cnf f c* = *Some b* **by** *simp*
from *1*(*1*)[*OF this*] **have** (*c*,*b*) ∈ *r*^* **by** *auto*
with *f-sound*[*OF Some*] **show** *?thesis* **by** *auto*
qed
qed
qed

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 (*induct rule*: *compute-NF.raw-induct*[*OF - res*, *of* λ *g a b*. *g* = *f* ⟶ *b* ∈ *NF r*, *THEN mp*[*OF - refl*]])
case (*1 cnf g a b*)
show *?case*
proof
assume *g* = *f*
note *1* = *1*[*unfolded this*]
show *b* ∈ *NF r*
proof (*cases f a*)
case *None*
with *f-complete*[*OF None*] *1*(*2*)
show *?thesis* **by** *simp*
next
case (*Some c*)
from *1*(*2*)[*unfolded this*] **have** *cnf f c* = *Some b* **by** *simp*
from *1*(*1*)[*OF this*] **show** *?thesis* **by** *simp*
qed
qed
qed

lemma *compute-NF-SN*: **assumes** *SN*: *SN r*
and *f-sound*: $\bigwedge$ *a b*. *f a* = *Some b* $\Longrightarrow$ (*a*,*b*) $\in$ *r*
shows $\exists$ *b*. *compute-NF f a* = *Some b* (**is** *?P a*)
proof −
let *?r* = {(*a*,*b*). *f a* = *Some b*}
have *?r* $\subseteq$ *r* **using** *f-sound* **by** *auto*
from *SN-subset*[*OF SN this*] **have** *SNr*: *SN ?r* **.**
show *?thesis*
proof (*induct rule*: *SN-induct*[*OF SNr*, *of* λ *a*. *?P a*])
case (*1 a*)
show *?case*
proof (*cases f a*)
case *None* **then show** *?thesis* **by** *auto*
next
case (*Some b*)
then have (*a*,*b*) $\in$ *?r* **by** *simp*
from *1*[*OF this*] *f-sound*[*OF Some*] **show** *?thesis*
using *Some* **by** *auto*
qed
qed
qed

definition *compute-trancl A R* = *R*^+ `` *A*
lemma *compute-trancl-rtrancl*[*code-unfold*]: {*b*. (*a*,*b*) $\in$ *R*^*} = *insert a* (*compute-trancl* {*a*} *R*)
proof −
have *id*: *R*^* = (*Id* $\cup$ *R*^+) **by** *regexp*
show *?thesis* **unfolding** *id compute-trancl-def* **by** *auto*
qed

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 −
have *R*: *R*^+ = *R O R*^* **by** *regexp*
define *B* **where** *B* = *R* `` *A*
define *R'* **where** *R'* = {*ab* $\in$ *R*. *fst ab* $\notin$ *A* $\wedge$ *snd ab* $\notin$ *B*}
note *d* = *compute-trancl-def*
show *?thesis* **unfolding** *Let-def B-def*[*symmetric*] *R'-def*[*symmetric*] *d*
proof (*cases B* $\subseteq$ {})
case *True*
then show *R*^+ `` *A* = (*if B* $\subseteq$ {} *then* {} *else B* $\cup$ *R'*^+ `` *B*) **unfolding** *B-def R* **by** *auto*
next
case *False*
have *R'* $\subseteq$ *R* **unfolding** *R'-def* **by** *auto*
then have *R'*^+ $\subseteq$ *R*^+ **by** (*rule trancl-mono-set*)
also have ... $\subseteq$ *R*^* **by** *auto*

```
    finally have mono: R'^+ ⊆ R^* .
    have B ∪ R'^+ `` B = R^+ `` A
    proof
      show B ∪ R'^+ `` B ⊆ R^+ `` A unfolding B-def R using mono
        by blast
    next
      show R^+ `` A ⊆ B ∪ R'^+ `` B
      proof
        fix x
        assume x ∈ R^+ `` A
        then obtain a where a: a ∈ A and ax: (a,x) ∈ R^+ by auto
        from ax a show x ∈ B ∪ R'^+ `` B
        proof (induct)
          case base
          then show ?case unfolding B-def by auto
        next
          case (step x y)
          from step(3)[OF step(4)] have x: x ∈ B ∪ R'^+ `` B .
          show ?case
          proof (cases y ∈ B)
            case False note y = this
            show ?thesis
            proof (cases x ∈ A)
              case True
              with y step(2) show ?thesis unfolding B-def by auto
            next
              case False
              with y step(2) have (x,y) ∈ R' unfolding R'-def by auto
              with x have y ∈ (R' ∪ (R'^+ O R')) `` B by blast
              also have R' ∪ (R'^+ O R') = R'^+ by regexp
              finally show ?thesis by blast
            qed
          qed auto
        qed
      qed
    qed
    with False show R^+ `` A = (if B ⊆ {} then {} else B ∪ R'^+ `` B) by auto
  qed
qed

lemma trancl-image-code[code-unfold]: R^+ `` A = compute-trancl A R unfolding
compute-trancl-def by auto
lemma compute-rtrancl[code-unfold]: R^* `` A = A ∪ compute-trancl A R
proof −
  have id: R^* = (Id ∪ R^+) by regexp
  show ?thesis unfolding id compute-trancl-def by auto
qed
lemma trancl-image-code'[code-unfold]: (a,b) ∈ R^+ ⟷ b ∈ compute-trancl {a}
R unfolding compute-trancl-def by auto
```

lemma *rtrancl-image-code*[*code-unfold*]: $(a,b) \in R\hat{}* \longleftrightarrow b = a \vee b \in$ *compute-trancl* $\{a\}$ R
using *compute-rtrancl*[*of* R $\{a\}$] **by** *auto*

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 **by** *auto*

lemma *closure-mono*: $r \subseteq s \Longrightarrow$ *closure* $r \subseteq$ *closure* s **by** (*auto elim*: *closure.cases*)

lemma *subset-closure*: $r \subseteq$ *closure* r
using *closure.intros* [**where** a = *nil*] **by** (*auto simp*: *cop-nil*)

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

lemma *closure-subset*: *closed* $r \Longrightarrow$ *closure* $r \subseteq r$
by (*auto simp*: *closed-def*)

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
by (*auto simp*: *closed-def elim*: *closure.cases*)

lemma *closedD* [*dest*]: *closed* $r \Longrightarrow (x, y) \in r \Longrightarrow$ (*cop* a x, *cop* a y) $\in r$
by (*auto simp*: *closed-def*)

lemma *closed-closure* [*intro*]: *closed* (*closure* r)
using *closure.intros* [**where** a = *add* a b **for** a b]
by (*auto simp*: *closed-def cop-add elim*!: *closure.cases*)

lemma *subset-closure-Un*:

closure $r \subseteq$ *closure* $(r \cup s)$
closure $s \subseteq$ *closure* $(r \cup s)$
by (*auto elim*!: *closure.cases*)

lemma *closure-Un*: *closure* $(r \cup s) =$ *closure* $r \cup$ *closure* s
using *subset-closure-Un* **by** (*auto elim*: *closure.cases*)

lemma *closure-id* [*simp*]: *closed* $r \Longrightarrow$ *closure* $r = r$
using *subset-closure* **and** *closure-subset* **by** *blast*

lemma *closed-Un* [*intro*]: *closed* $r \Longrightarrow$ *closed* $s \Longrightarrow$ *closed* $(r \cup s)$ **by** *blast*

lemma *closed-Inr* [*intro*]: *closed* $r \Longrightarrow$ *closed* $s \Longrightarrow$ *closed* $(r \cap s)$ **by** *blast*

lemma *closed-rtrancl* [*intro*]: *closed* $r \Longrightarrow$ *closed* (r^*)
by (*best intro*: *rtrancl-into-rtrancl elim*: *rtrancl.induct*)

lemma *closed-trancl* [*intro*]: *closed* $r \Longrightarrow$ *closed* (r^+)
by (*best intro*: *trancl-into-trancl elim*: *trancl.induct*)

lemma *closed-converse* [*intro*]: *closed* $r \Longrightarrow$ *closed* (r^{-1}) **by** *blast*

lemma *closed-comp* [*intro*]: *closed* $r \Longrightarrow$ *closed* $s \Longrightarrow$ *closed* $(r\ O\ s)$ **by** *blast*

lemma *closed-relpow* [*intro*]: *closed* $r \Longrightarrow$ *closed* $(r \hat{\ } n)$
by (*auto intro*: *relpow-image* [*OF closedD*])

lemma *closed-conversion* [*intro*]: *closed* $r \Longrightarrow$ *closed* $(r^{\leftrightarrow *})$
by (*auto simp*: *conversion-def*)

lemma *closed-relto* [*intro*]: *closed* $r \Longrightarrow$ *closed* $s \Longrightarrow$ *closed* $(relto\ r\ s)$ **by** *blast*

lemma *closure-diff-subset*: *closure* $r -$ *closure* $s \subseteq$ *closure* $(r - s)$ **by** (*auto elim*: *closure.cases*)

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 $P\ s\ t$
using *assms* **by** (*induct*) *simp*

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 $P\ s\ t$
using *assms* **by** (*induct*) *auto*

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
by (*force simp flip*: *ctxt-ctxt-compose*)

lemma *rstep-rule* [*intro*]: $(l, r) \in R \Longrightarrow (l, r) \in$ *rstep* R
using *rstep.rstep* [**where** $C = \Box$ **and** $\sigma =$ *Var* **and** $R = R$] **by** *simp*

lemma *rstep-subst* [*intro*]: $(s, t) \in$ *rstep* $R \Longrightarrow (s \cdot \sigma, t \cdot \sigma) \in$ *rstep* R
by (*force simp flip*: *subst-subst-compose*)

lemma *rstep-empty* [*simp*]: *rstep* $\{\} = \{\}$
by *auto*

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

lemma *rstep-union*: *rstep* $(R \cup S) =$ *rstep* $R \cup$ *rstep* S
by *auto*

lemma *rstep-converse* [*simp*]: *rstep* $(R^{-1}) = (\text{rstep}\ R)^{-1}$
by *auto*

interpretation *subst*: *rel-closure* $\lambda\sigma\ t.\ t \cdot \sigma$ *Var* $\lambda x\ y.\ y \circ_s x$ **by** (*standard*) *auto*
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* $\Box$ $(\circ_c)$ **by** (*standard*) *auto*
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*)
by (*force elim*: *ctxt.closure.cases subst.closure.cases*)

lemma *ctxt-closed-rstep* [*intro*]: *ctxt.closed* (*rstep R*)
by (*simp add*: *rstep-eq-closure ctxt.closed-closure*)

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*
using *ctxt.closedD* [*of r s t More f ss* $\Box$ *ts*] **by** *auto*

4.1 Well-formed TRSs

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

lemma *wf-trs-imp-lhs-Fun*:
wf-trs R $\Longrightarrow$ (*l*,*r*) $\in$ *R* $\Longrightarrow$ $\exists$ *f ts*. *l* = *Fun f ts*
unfolding *wf-trs-def* **by** *blast*

lemma *rstep-imp-Fun*:
assumes *wf-trs R*
shows (*s*, *t*) $\in$ *rstep R* $\Longrightarrow$ $\exists$ *f ss*. *s* = *Fun f ss*
proof −
assume (*s*, *t*) $\in$ *rstep R*
then obtain *C l r* σ **where** *lr*: (*l*,*r*) $\in$ *R* **and** *s*: *s* = *C* $\langle$ *l* $\cdot$ σ $\rangle$ **by** *auto*
with *wf-trs-imp-lhs-Fun*[*OF assms lr*] **show** *?thesis* **by** (*cases C*, *auto*)
qed

lemma *SN-Var*:
assumes *wf-trs R* **shows** *SN-on* (*rstep R*) {*Var x*}
proof (*rule ccontr*)
assume ¬ *?thesis*
then obtain *S* **where** [*symmetric*]: *S 0* = *Var x*
and *chain*: *chain* (*rstep R*) *S* **by** *auto*
then have (*Var x*, *S* (*Suc 0*)) $\in$ *rstep R* **by** *force*
then obtain *C l r* σ **where** (*l*, *r*) $\in$ *R* **and** *Var x* = *C*$\langle$*l* $\cdot$ $\sigma$$\rangle$ **by** *best*
then have *Var x* = *l* $\cdot$ σ **by** (*induct C*) *simp-all*

then obtain y **where** $l = Var\ y$ **by** (*induct l*) *simp-all*
with *assms* **and** ‹$(l, r) \in R$› **show** *False* **unfolding** *wf-trs-def* **by** *blast*
qed

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*)
by (*auto simp*: *vars-rule-def*)

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*
unfolding *vars-trs-def* **by** *auto*

lemma *finite-trs-has-finite-vars*:
assumes *finite R* **shows** *finite* (*vars-trs R*)
using *assms* **unfolding** *vars-trs-def vars-rule-def* [*abs-def*] **by** *simp*

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*
using *funs-term-funas-term* **unfolding** *funs-rule-def funas-rule-def image-Un* **by** *metis*

lemma *funs-trs-funas-trs*:*funs-trs R* = *fst* ‘ *funas-trs R*
unfolding *funs-trs-def funas-trs-def image-UN* **using** *funs-rule-funas-rule* **by** *metis*

lemma *finite-funas-rule*: *finite* (*funas-rule lr*)

unfolding *funas-rule-def*
using *finite-funas-term* **by** *auto*

lemma *finite-funas-trs*:
assumes *finite R*
shows *finite* (*funas-trs R*)
unfolding *funas-trs-def*
using *assms finite-funas-rule* **by** *auto*

lemma *funas-empty*[*simp*]: *funas-trs* {} = {} **unfolding** *funas-trs-def* **by** *simp*

lemma *funas-trs-union*[*simp*]: *funas-trs* (*R* ∪ *S*) = *funas-trs R* ∪ *funas-trs S*
unfolding *funas-trs-def* **by** *blast*

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 = (⋃ *r*∈*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 = (⋃ *r*∈*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* × *nat*) ⇒ *bool*
where
defined R fn ⟷ (∃ *l r*. (*l*, *r*) ∈ *R* ∧ *root l* = *Some fn*)

lemma *defined-funas-trs*: **assumes** *d*: *defined R fn* **shows** *fn* ∈ *funas-trs R*
proof −
from *d* [*unfolded defined-def*] **obtain** *l r*
where (*l*, *r*) ∈ *R* **and** *root l* = *Some fn* **by** *auto*
then show *?thesis*
unfolding *funas-trs-def funas-rule-def* [*abs-def*] **by** (*cases l*) *force*+
qed

fun *root-list* :: ('*f*, '*v*) *term* ⇒ ('*f* × *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*
by (*simp add*: *vars-rule-list-def vars-rule-def*)

lemma *set-funs-rule-list* [*simp*]:
set (*funs-rule-list r*) = *funs-rule r*
by (*simp add*: *funs-rule-list-def funs-rule-def*)

lemma *set-funas-rule-list* [*simp*]:
set (*funas-rule-list r*) = *funas-rule r*
by (*simp add*: *funas-rule-list-def funas-rule-def*)

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

by (*cases fst r snd r rule*: *term.exhaust* [*case-product term.exhaust*])
(*auto simp*: *roots-rule-list-def roots-rule-def ac-simps*)

lemma *set-funas-args-rule-list* [*simp*]:
set (*funas-args-rule-list r*) = *funas-args-rule r*
by (*simp add*: *funas-args-rule-list-def funas-args-rule-def*)

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*)
by (*simp add*: *vars-trs-def vars-trs-list-def*)

lemma *set-funs-trs-list* [*simp*]:
set (*funs-trs-list R*) = *funs-trs* (*set R*)
by (*simp add*: *funs-trs-def funs-trs-list-def*)

lemma *set-funas-trs-list* [*simp*]:
set (*funas-trs-list R*) = *funas-trs* (*set R*)
by (*simp add*: *funas-trs-def funas-trs-list-def*)

lemma *set-roots-trs-list* [*simp*]:
set (*roots-trs-list R*) = *roots-trs* (*set R*)
by (*simp add*: *roots-trs-def roots-trs-list-def*)

lemma *set-funas-args-trs-list* [*simp*]:
set (*funas-args-trs-list R*) = *funas-args-trs* (*set R*)
by (*simp add*: *funas-args-trs-def funas-args-trs-list-def*)

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 [] = [] **unfolding** *vars-trs-list-def* **by** *simp*

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

lemma *funas-term-subst-rhs*:
assumes *funas-trs* $R \subseteq F$ **and** $(l, r) \in R$ **and** *funas-term* $(l \cdot \sigma) \subseteq F$
shows *funas-term* $(r \cdot \sigma) \subseteq F$
proof −
have *vars-term* $r \subseteq$ *vars-term* l **using** ‹*wf-trs R*› **and** ‹$(l, r) \in R$› **by** (*auto simp*: *wf-trs-def*)
moreover have *funas-term* $l \subseteq F$ **and** *funas-term* $r \subseteq F$
using ‹*funas-trs* $R \subseteq F$› **and** ‹$(l, r) \in R$› **by** (*auto simp*: *funas-defs*) *force*+
ultimately show *?thesis*
using ‹*funas-term* $(l \cdot \sigma) \subseteq F$› **by** (*force simp*: *funas-term-subst*)
qed

lemma *vars-rule-lhs*:
$r \in R \Longrightarrow$ *vars-rule* r = *vars-term* (*fst* r)
using ‹*wf-trs R*› **by** (*cases r*) (*auto simp*: *wf-trs-def vars-rule-def*)

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 −
from ‹$s \rhd t$› **obtain** C **where** $C \neq \Box$ **and** $C\langle t\rangle = s$ **by** *auto*
from ‹*ctxt.closed R*› **and** ‹$(t,u) \in R$›
have *RCtCu*: $(C\langle t\rangle, C\langle u\rangle) \in R$ **by** (*rule ctxt.closedD*)
from ‹$C \neq \Box$› **have** $C\langle u\rangle \rhd u$ **by** *auto*
from *RCtCu* **have** $(s, C\langle u\rangle) \in R$ **unfolding** ‹$C\langle t\rangle = s$› **.**
from *this* **and** ‹$C\langle u\rangle \rhd u$› **show** *?thesis* **by** *auto*
qed

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

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* $(R \cup \{\triangleright\})$
proof −
from ‹*ctxt.closed R*› **have** *quasi-commute R* $\{\triangleright\}$
unfolding *quasi-commute-def* **by** (*rule ctxt-closed-imp-qc-supt*)
have *SN* $\{\triangleright\}$ **by** (*rule SN-supt*)
from ‹*SN R*› **and** ‹*SN* $\{\triangleright\}$› **and** ‹*quasi-commute R* $\{\triangleright\}$›
show *?thesis* **by** (*rule quasi-commute-imp-SN*)
qed

lemma *stable-loop-imp-not-SN*:
assumes *stable*: *subst.closed r* **and** *steps*: $(s,\ s \cdot \sigma) \in r^+$
shows ¬ *SN-on r* $\{s\}$
proof −
let *?f* = $\lambda\ i.\ s \cdot$ (*power.power Var* $(\circ_s)\ \sigma\ i$)
have *main*: $\bigwedge\ i.\ (?f\ i,\ ?f\ (Suc\ i)) \in r^+$
proof −
fix *i*
show $(?f\ i,\ ?f\ (Suc\ i)) \in r^+$
proof (*induct i*)
case (*Suc i*)
from *Suc subst.closed-trancl*[*OF stable*] **have** *step*: $(?f\ i \cdot \sigma,\ ?f\ (Suc\ i) \cdot \sigma)$
$\in r^+$ **by** *auto*
let $?\sigma g$ = *power.power Var* $(\circ_s)\ \sigma\ i$
let $?\sigma gs$ = *power.power Var* $(\circ_s)\ \sigma$ (*Suc i*)
have *idi*: $?\sigma g \circ_s \sigma = \sigma \circ_s ?\sigma g$ **by** (*rule subst-monoid-mult.power-commutes*)
have *idsi*: $?\sigma gs \circ_s \sigma = \sigma \circ_s ?\sigma gs$ **by** (*rule subst-monoid-mult.power-commutes*)
have $?f\ i \cdot \sigma = s \cdot ?\sigma g\ \circ_s \sigma$ **by** *simp*
also have $\ldots = ?f\ (Suc\ i)$ **unfolding** *idi* **by** *simp*
finally have *one*: $?f\ i \cdot \sigma = ?f\ (Suc\ i)$ **.**
have $?f\ (Suc\ i) \cdot \sigma = s \cdot ?\sigma gs \circ_s \sigma$ **by** *simp*
also have $\ldots = ?f\ (Suc\ (Suc\ i))$ **unfolding** *idsi* **by** *simp*
finally have *two*: $?f\ (Suc\ i) \cdot \sigma = ?f\ (Suc\ (Suc\ i))$ **by** *simp*
show *?case* **using** *one two step* **by** *simp*
qed (*auto simp*: *steps*)
qed
then have ¬ *SN-on* (r^+) $\{?f\ 0\}$ **unfolding** *SN-on-def* **by** *best*
then show *?thesis* **using** *SN-on-trancl* **by** *force*
qed

lemma *subst-closed-supteq*: *subst.closed* $\{\trianglerighteq\}$ **by** *blast*

lemma *subst-closed-supt*: *subst.closed* $\{\triangleright\}$ **by** *blast*

lemma *ctxt-closed-supt-subset*: *ctxt.closed R* $\Longrightarrow$ $\{\triangleright\}\ O\ R \subseteq R\ O\ \{\triangleright\}$ **by** *blast*

4.4 Properties of Rewrite Steps

lemma *rstep-relcomp-idemp1* [*simp*]:

rstep (*rstep R O rstep S*) = *rstep R O rstep S*
proof −
{ fix *s t*
assume (*s*, *t*) ∈ *rstep* (*rstep R O rstep S*)
then have (*s*, *t*) ∈ *rstep R O rstep S*
by (*induct*) *blast*+ **}**
then show *?thesis* **by** *auto*
qed

lemma *rstep-relcomp-idemp2* [*simp*]:
rstep (*rstep R O rstep S O rstep T*) = *rstep R O rstep S O rstep T*
proof −
{ fix *s t*
assume (*s*, *t*) ∈ *rstep* (*rstep R O rstep S O rstep T*)
then have (*s*, *t*) ∈ *rstep R O rstep S O rstep T*
by (*induct*) *blast*+ **}**
then show *?thesis* **by** *auto*
qed

lemma *ctxt-closed-rsteps* [*intro*]: *ctxt.closed* ((*rstep R*)*) **by** *blast*

lemma *subset-rstep*: *R* ⊆ *rstep R* **by** *auto*

lemma *subst-closure-rstep-subset*: *subst.closure* (*rstep R*) ⊆ *rstep R*
by (*auto elim*: *subst.closure.cases*)

lemma *subst-closed-rstep* [*intro*]: *subst.closed* (*rstep R*) **by** *blast*

lemma *subst-closed-rsteps*: *subst.closed* ((*rstep R*)*) **by** *blast*

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*)
using *supt-rsteps-subset* [*of R*] **by** (*auto simp*: *supt-supteq-set-conv*)

lemma *quasi-commute-rsteps-supt*:
quasi-commute ((*rstep R*)*) {⊳}
unfolding *quasi-commute-def* **using** *supt-rsteps-subset* [*of R*] **by** *auto*

lemma *rstep-UN*:
rstep (⋃ *i*∈*A*. *R i*) = (⋃ *i*∈*A*. *rstep* (*R i*))
by (*force*)

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*) ∧

$(C\langle snd\ r \cdot \sigma\rangle = t)\}$

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 −
{ fix *s t*
have ((*s*,*t*) ∈ *?l*) = ((*s*,*t*) ∈ *?r*)
unfolding *rstep-r-p-s-def Let-def*
using *ctxt-supt-id* [*of p s*] **and** *subt-at-ctxt-of-pos-term* [*of p s fst r* · *σ*] **by** *auto* **}**
then show *?thesis* **by** *auto*
qed

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*)⟨$r_1 \cdot \sigma_1$⟩) ∈ *rstep-r-p-s* R_2 (l_2, r_2) p_2 σ_2 ∧
(*u*, (*ctxt-of-pos-term* p_1 *u*)⟨$r_1 \cdot \sigma_1$⟩) ∈ *rstep-r-p-s* R_1 (l_1, r_1) p_1 σ_1
proof −
have *p1*: p_1 ∈ *poss s* **and** *lr1*: (l_1, r_1) ∈ R_1 **and** *σ1*: *s* |- $p_1 = l_1 \cdot \sigma_1$
and *t*: *t* = *replace-at s* p_1 ($r_1 \cdot \sigma_1$)
and *p2*: p_2 ∈ *poss s* **and** *lr2*: (l_2, r_2) ∈ R_2 **and** *σ2*: *s* |- $p_2 = l_2 \cdot \sigma_2$
and *u*: *u* = *replace-at s* p_2 ($r_2 \cdot \sigma_2$) **using** *assms* **by** (*auto simp*: *rstep-r-p-s-def'*)

have *replace-at t* p_2 ($r_2 \cdot \sigma_2$) = *replace-at u* p_1 ($r_1 \cdot \sigma_1$)
using *t* **and** *u* **and** *parallel-replace-at* [*OF* ‹$p_1 \perp p_2$› *p1 p2*] **by** *auto*
moreover
have (*t*, (*ctxt-of-pos-term* p_2 *t*)⟨$r_2 \cdot \sigma_2$⟩) ∈ *rstep-r-p-s* R_2 (l_2, r_2) p_2 σ_2
proof −
have *t* |- $p_2 = l_2 \cdot \sigma_2$ **using** *σ2* **and** *parallel-replace-at-subt-at* [*OF par p1 p2*] **and** *t* **by** *auto*
moreover have p_2 ∈ *poss t* **using** *parallel-poss-replace-at* [*OF par p1*] **and** *t* **and** *p2* **by** *auto*
ultimately show *?thesis* **using** *lr2* **and** *ctxt-supt-id* [*of* p_2 *t*] **by** (*simp add*: *rstep-r-p-s-def*)
qed
moreover
have (*u*, (*ctxt-of-pos-term* p_1 *u*)⟨$r_1 \cdot \sigma_1$⟩) ∈ *rstep-r-p-s* R_1 (l_1, r_1) p_1 σ_1
proof −
have *par'*: $p_2 \perp p_1$ **using** *parallel-pos-sym* [*OF par*] **.**
have *u* |- $p_1 = l_1 \cdot \sigma_1$ **using** *σ1* **and** *parallel-replace-at-subt-at* [*OF par' p2 p1*] **and** *u* **by** *auto*
moreover have p_1 ∈ *poss u* **using** *parallel-poss-replace-at* [*OF par' p2*] **and** *u* **and** *p1* **by** *auto*
ultimately show *?thesis* **using** *lr1* **and** *ctxt-supt-id* [*of* p_1 *u*] **by** (*simp add*:

rstep-r-p-s-def)
qed
ultimately show *?thesis* **by** *auto*
qed

lemma *rstep-iff-rstep-r-p-s*:
$(s, t) \in$ *rstep* $R \longleftrightarrow (\exists\, l\ r\ p\ \sigma.\ (s, t) \in$ *rstep-r-p-s* $R\ (l, r)\ p\ \sigma)$ (**is** *?lhs* = *?rhs*)
proof
assume $(s, t) \in$ *rstep* R
then obtain $C\ \sigma\ l\ r$ **where** s: $s = C\langle l \cdot \sigma\rangle$ **and** t: $t = C\langle r \cdot \sigma\rangle$ **and** $(l, r) \in R$ **by** *auto*
let *?p* = *hole-pos* C
let *?C* = *ctxt-of-pos-term ?p s*
have C: *ctxt-of-pos-term ?p s* = C **unfolding** s **by** (*induct* C) *simp-all*
have *?p* $\in$ *poss s* **unfolding** s **by** *simp*
moreover have $(l, r) \in R$ **by** *fact*
moreover have *?C*$\langle l \cdot \sigma\rangle = s$ **unfolding** C **by** (*simp add*: s)
moreover have *?C*$\langle r \cdot \sigma\rangle = t$ **unfolding** C **by** (*simp add*: t)
ultimately show *?rhs* **unfolding** *rstep-r-p-s-def Let-def* **by** *auto*
next
assume *?rhs*
then obtain $l\ r\ p\ \sigma$ **where** $p \in$ *poss* s
and $(l, r) \in R$
and s[*symmetric*]: (*ctxt-of-pos-term* $p\ s$)$\langle l \cdot \sigma\rangle = s$
and t[*symmetric*]: (*ctxt-of-pos-term* $p\ s$)$\langle r \cdot \sigma\rangle = t$
unfolding *rstep-r-p-s-def Let-def* **by** *auto*
then show *?lhs* **by** *auto*
qed

lemma *rstep-r-p-s-imp-rstep*:
assumes $(s, t) \in$ *rstep-r-p-s* $R\ r\ p\ \sigma$
shows $(s, t) \in$ *rstep* R
using *assms* **by** (*cases* r) (*auto simp*: *rstep-iff-rstep-r-p-s*)

Rewriting steps below the root position.

definition
nrrstep :: $('f, 'v)$ *trs* $\Rightarrow$ $('f, 'v)$ *trs*
where
nrrstep $R = \{(s,t).\ \exists\, r\ i\ ps\ \sigma.\ (s,t) \in$ *rstep-r-p-s* $R\ r\ (i\#ps)\ \sigma\}$

An alternative characterisation of non-root rewrite steps.

lemma *nrrstep-def'*:
nrrstep $R = \{(s, t).\ \exists\, l\ r\ C\ \sigma.\ (l, r) \in R \wedge C \neq \Box \wedge s = C\langle l{\cdot}\sigma\rangle \wedge t = C\langle r{\cdot}\sigma\rangle\}$
(**is** *?lhs* = *?rhs*)
proof
show *?lhs* $\subseteq$ *?rhs*
proof (*rule subrelI*)
fix $s\ t$ **assume** $(s, t) \in$ *nrrstep* R
then obtain $l\ r\ i\ ps\ \sigma$ **where** *step*: $(s, t) \in$ *rstep-r-p-s* $R\ (l, r)\ (i\ \#\ ps)\ \sigma$

unfolding *nrrstep-def* **by** *best*
let *?C* = *ctxt-of-pos-term* (*i* # *ps*) *s*
from *step* **have***i* # *ps* ∈ *poss s* **and** (*l*, *r*) ∈ *R* **and** *s* = *?C*⟨*l*·*σ*⟩ **and** *t* = *?C*⟨*r*·*σ*⟩
unfolding *rstep-r-p-s-def Let-def* **by** *auto*
moreover from ‹*i* # *ps* ∈ *poss s*› **have** *?C* ≠ □ **by** (*induct s*) *auto*
ultimately show (*s*, *t*) ∈ *?rhs* **by** *auto*
qed
next
show *?rhs* ⊆ *?lhs*
proof (*rule subrelI*)
fix *s t* **assume** (*s*, *t*) ∈ *?rhs*
then obtain *l r C σ* **where** *in-R*: (*l*, *r*) ∈ *R* **and** *C* ≠ □
and *s*: *s* = *C*⟨*l*·*σ*⟩ **and** *t*: *t* = *C*⟨*r*·*σ*⟩ **by** *auto*
from ‹*C* ≠ □› **obtain** *i p* **where** *ip*: *hole-pos C* = *i* # *p* **by** (*induct C*) *auto*
have *i* # *p* ∈ *poss s* **using** *hole-pos-poss*[*of C*] **unfolding** *s ip* **by** *simp*
then have *C*: *C* = *ctxt-of-pos-term* (*i* # *p*) *s*
unfolding *s ip*[*symmetric*] **by** *simp*
from ‹*i* # *p* ∈ *poss s*› *in-R s t* **have** (*s*, *t*) ∈ *rstep-r-p-s R* (*l*, *r*) (*i* # *p*) *σ*
unfolding *rstep-r-p-s-def C* **by** *simp*
then show (*s*, *t*) ∈ *nrrstep R* **unfolding** *nrrstep-def* **by** *best*
qed
qed

lemma *nrrstepI*: (*l*,*r*) ∈ *R* ⟹ *s* = *C*⟨*l*·*σ*⟩ ⟹ *t* = *C*⟨*r* · *σ*⟩ ⟹ *C* ≠ □ ⟹ (*s*,*t*) ∈ *nrrstep R* **unfolding** *nrrstep-def′* **by** *auto*

lemma *nrrstep-union*: *nrrstep* (*R* ∪ *S*) = *nrrstep R* ∪ *nrrstep S*
unfolding *nrrstep-def′* **by** *blast*

lemma *nrrstep-empty*[*simp*]: *nrrstep* {} = {} **unfolding** *nrrstep-def′* **by** *blast*

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*)
by (*auto simp*: *rrstep-def rstep-r-p-s-def*)

lemma *rules-subset-rrstep* [*simp*]: *R* ⊆ *rrstep R*
by (*force simp*: *rrstep-def′ intro*: *exI* [*of* - *Var*])

lemma *rrstep-union*: *rrstep* (*R* ∪ *S*) = *rrstep R* ∪ *rrstep S* **unfolding** *rrstep-def′* **by** *blast*

lemma *rrstep-empty*[*simp*]: *rrstep* {} = {}
unfolding *rrstep-def'* **by** *auto*

lemma *subst-closed-rrstep*: *subst.closed* (*rrstep R*)
unfolding *subst.closed-def*
proof
fix *ss ts*
assume (*ss*,*ts*) ∈ *subst.closure* (*rrstep R*)
then show (*ss*,*ts*) ∈ *rrstep R*
proof
fix *s t σ*
assume *ss*: *ss* = *s* · *σ* **and** *ts*: *ts* = *t* · *σ* **and** *step*: (*s*,*t*) ∈ *rrstep R*
from *step* **obtain** *l r δ* **where** *lr*: (*l*,*r*) ∈ *R* **and** *s*: *s* = *l* · *δ* **and** *t*: *t* = *r* · *δ*
unfolding *rrstep-def'* **by** *auto*
obtain *sig* **where** *sig* = *δ* ∘$_s$ *σ* **by** *auto*
with *ss s ts t* **have** *ss* = *l* · *sig* **and** *ts* = *r* · *sig* **by** *simp*+
with *lr* **show** (*ss*,*ts*) ∈ *rrstep R* **unfolding** *rrstep-def'* **by** (*auto simp*: *Let-def*)
qed
qed

lemma *rstep-iff-rrstep-or-nrrstep*: *rstep R* = (*rrstep R* ∪ *nrrstep R*)
proof
show *rstep R* ⊆ *rrstep R* ∪ *nrrstep R*
proof (*rule subrelI*)
fix *s t* **assume** (*s*,*t*) ∈ *rstep R*
then obtain *l r p σ* **where** *rstep-rps*: (*s*,*t*) ∈ *rstep-r-p-s R* (*l*,*r*) *p σ*
by (*auto simp*: *rstep-iff-rstep-r-p-s*)
then show (*s*,*t*) ∈ *rrstep R* ∪ *nrrstep R* **unfolding** *rrstep-def nrrstep-def* **by**
(*cases p*) *auto*
qed
next
show *rrstep R* ∪ *nrrstep R* ⊆ *rstep R*
proof (*rule subrelI*)
fix *s t* **assume** (*s*,*t*) ∈ *rrstep R* ∪ *nrrstep R*
then show (*s*,*t*) ∈ *rstep R* **by** (*auto simp*: *rrstep-def nrrstep-def rstep-iff-rstep-r-p-s*)
qed
qed

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 −
from *nrrstep* **obtain** *C* **where** *C*:*C* = *ctxt-of-pos-term* (*i*#*ps*) (*Fun f ss*)
and *pos*: (*i*#*ps*) ∈ *poss* (*Fun f ss*)
and *Rlr*: (*l*,*r*) ∈ *R*
and *Fun*: *C*⟨*l*·*σ*⟩ = *Fun f ss*
and *t*: *C*⟨*r*·*σ*⟩ = *t* **unfolding** *rstep-r-p-s-def Let-def* **by** *auto*
then obtain *D* **where** *C'*:*C* = *More f* (*take i ss*) *D* (*drop* (*Suc i*) *ss*) **by** *auto*
then have *CFun*: *C*⟨*l*·*σ*⟩ = *Fun f* (*take i ss* @ (*D*⟨*l*·*σ*⟩) # *drop* (*Suc i*) *ss*) **by**

auto
from *pos* **have** *len*: $i < length\ ss$ **by** *auto*
from *len*
have (*take i ss* @ ($D\langle l{\cdot}\sigma\rangle$) # *drop* (*Suc i*) *ss*)!$i = D\langle l{\cdot}\sigma\rangle$ **by** (*auto simp*: *nth-append*)
with *C Fun CFun* **have** *ssi*: $ss!i = D\langle l{\cdot}\sigma\rangle$ **by** *auto*
from C' *t* **have** t': $t = Fun\ f$ (*take i ss* @ ($D\langle r{\cdot}\sigma\rangle$) # *drop* (*Suc i*) *ss*) **by** *auto*
from *len*
have (*take i ss* @ ($D\langle r{\cdot}\sigma\rangle$) # *drop* (*Suc i*) *ss*)!$i = D\langle r{\cdot}\sigma\rangle$ **by** (*auto simp*: *nth-append*)
with t' **have** $t|\text{-}[i] = (D\langle r{\cdot}\sigma\rangle)|\text{-}[]$ **by** *auto*
then have *subt-at*: $t|\text{-}[i] = D\langle r{\cdot}\sigma\rangle$ **by** *simp*
from *C* C' **have** $D = ctxt\text{-}of\text{-}pos\text{-}term\ ps\ (ss!i)$ **by** *auto*
with *pos Rlr ssi*[*symmetric*] *subt-at*[*symmetric*]
show *?thesis* **unfolding** *rstep-r-p-s-def Let-def* **by** *auto*
qed

lemma *ctxt-closure-rstep-eq* [*simp*]: *ctxt.closure* (*rstep R*) = *rstep R*
by (*rule ctxt.closure-id*) *blast*

lemma *subst-closure-rstep-eq* [*simp*]: *subst.closure* (*rstep R*) = *rstep R*
by (*rule subst.closure-id*) *blast*

lemma *supt-rstep-subset*:
$\{\rhd\}\ O\ rstep\ R \subseteq rstep\ R\ O\ \{\rhd\}$
proof (*rule subrelI*)
fix *s t* **assume** (*s*,*t*) ∈ $\{\rhd\}\ O\ rstep\ R$ **then show** (*s*,*t*) ∈ *rstep R O* $\{\rhd\}$
proof (*induct s*)
case (*Var x*)
then have $\exists u.\ Var\ x \rhd u \wedge (u,t) \in rstep\ R$ **by** *auto*
then obtain *u* **where** $Var\ x \rhd u$ **and** (*u*,*t*) ∈ *rstep R* **by** *auto*
from ‹$Var\ x \rhd u$› **show** *?case* **by** (*cases rule*: *supt.cases*)
next
case (*Fun f ss*)
then obtain *u* **where** $Fun\ f\ ss \rhd u$ **and** (*u*,*t*) ∈ *rstep R* **by** *auto*
from ‹$Fun\ f\ ss \rhd u$› **obtain** *C*
where $C \neq \Box$ **and** $C\langle u\rangle = Fun\ f\ ss$ **by** *auto*
from ‹$C \neq \Box$› **have** $C\langle t\rangle \rhd t$ **by** (*rule nectxt-imp-supt-ctxt*)
from ‹(*u*,*t*) ∈ *rstep R*› **have** ($C\langle u\rangle$,$C\langle t\rangle$) ∈ *rstep R* **..**
then have ($Fun\ f\ ss$,$C\langle t\rangle$) ∈ *rstep R* **unfolding** ‹$C\langle u\rangle = Fun\ f\ ss$› **.**
with ‹$C\langle t\rangle \rhd t$› **show** *?case* **by** *auto*
qed
qed

lemma *ne-rstep-seq-imp-list-of-terms*:
assumes (*s*,*t*) ∈ (*rstep R*)$^+$
shows $\exists ts.\ length\ ts > 1 \wedge ts!0 = s \wedge ts!(length\ ts - 1) = t \wedge$
$(\forall i{<}length\ ts - 1.\ (ts!i,ts!(Suc\ i)) \in (rstep\ R))$ (**is** $\exists ts.$ - ∧ - ∧ - ∧ *?P ts*)
using *assms*

```
proof (induct rule: trancl.induct)
  case (r-into-trancl x y)
  let ?ts = [x,y]
  have length ?ts > 1 by simp
  moreover have ?ts!0 = x by simp
  moreover have ?ts!(length ?ts − 1) = y by simp
  moreover from r-into-rtrancl r-into-trancl have ?P ?ts by auto
  ultimately show ?case by fast
next
  case (trancl-into-trancl x y z)
  then obtain ts where length ts > 1 and ts!0 = x
    and last1: ts!(length ts − 1) = y and ?P ts by auto
  let ?ts = ts@[z]
  have len: length ?ts = length ts + 1 by simp
  from ‹length ts > 1› have length ?ts > 1 by auto
  moreover with ‹ts!0 = x› have ?ts!0 = x by (induct ts) auto
  moreover have last2: ?ts!(length ?ts − 1) = z by simp
  moreover have ?P ?ts
  proof (intro allI impI)
    fix i assume A: i < length ?ts − 1
    show (?ts!i,?ts!(Suc i)) ∈ rstep R
    proof (cases i < length ts − 1)
     case True with ‹?P ts› A show ?thesis unfolding len unfolding nth-append
by auto
    next
      case False with A have i = length ts − 1 by simp
      with last1 ‹length ts > 1› have ?ts!i = y unfolding nth-append by auto
       have Suc i = length ?ts − 1 using ‹i = length ts − 1› using ‹length ts >
1› by auto
      with last2 have ?ts!(Suc i) = z by auto
     from ‹(y,z) ∈ rstep R› show ?thesis unfolding ‹?ts!(Suc i) = z› ‹?ts!i = y›
.
    qed
  qed
  ultimately show ?case by fast
qed


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 $(t,u) \in$ *restrict-SN R R*
shows $(\exists\, t.\ (s,t) \in$ *restrict-SN R R* $\wedge\ (t,u) \in$ *restrict-SN* $(E\ O\ \{\triangleright\})\ R)$ (**is** $\exists\ t.$ - $\wedge\ (t,u) \in$ *?snSub*)
proof −
from ‹$(s,t) \in$ *?snSub*› **obtain** v **where** *SN-on R* $\{s\}$ **and** *ac*: $(s,v) \in E$ **and** $v \triangleright t$ **unfolding** *restrict-SN-def supt-def* **by** *auto*
from ‹$v \triangleright t$› **obtain** C **where** $C \neq$ *Hole* **and** v: $C\langle t\rangle = v$ **by** *best*
from ‹$(t,u) \in$ *restrict-SN R R*› **have** $(t,u) \in R$ **unfolding** *restrict-SN-def* **by** *auto*
from ‹*ctxt.closed R*› **and** *this* **have** *RCtCu*: $(C\langle t\rangle, C\langle u\rangle) \in R$ **by** (*rule ctxt.closedD*)
with *v ac* **have** $(s, C\langle u\rangle) \in E\ O\ R$ **by** *auto*
then have *sCu*: $(s, C\langle u\rangle) \in R$ **using** *E* **by** *simp*
with ‹*SN-on R* $\{s\}$› **have** *one*: *SN-on R* $\{C\langle u\rangle\}$
using *step-preserves-SN-on*[*of s* $C\langle u\rangle$ *R*] **by** *blast*
from ‹$C \neq \Box$› **have** $C\langle u\rangle \triangleright u$ **by** *auto*
with *one E* **have** $(C\langle u\rangle, u) \in$ *?snSub* **unfolding** *restrict-SN-def supt-def* **by** *auto*
from *sCu* **and** ‹*SN-on R* $\{s\}$› **and** ‹$(C\langle u\rangle, u) \in$ *?snSub*› **show** *?thesis* **unfolding** *restrict-SN-def* **by** *auto*
qed

lemma *ctxt-closed-R-imp-restrict-SN-qc-E-supt*:
assumes *ctxt*: *ctxt.closed R*
shows *quasi-commute* (*restrict-SN R R*) (*restrict-SN* $(E\ O\ \{\triangleright\}\ O\ E)\ R$) (**is** *quasi-commute ?r ?s*)
proof −
have *?s O ?r* $\subseteq$ *?r O* $(?r \cup ?s)^*$
proof (*rule subrelI*)
fix *x y*
assume $(x,y) \in$ *?s O ?r*
from *this* **obtain** *z* **where** $(x,z) \in$ *?s* **and** $(z,y) \in$ *?r* **by** *best*
then obtain *v w* **where** *ac*: $(x,v) \in E$ **and** *vw*: $v \triangleright w$ **and** *wz*: $(w,z) \in E$ **and** *zy*: $(z,y) \in R$ **and** *SN-on R* $\{x\}$ **unfolding** *restrict-SN-def supt-def*
using *E*(*2*) **by** *auto*
from *wz zy* **have** $(w,y) \in E\ O\ R$ **by** *auto*
with *E* **have** *wy*: $(w,y) \in R$ **by** *auto*
from *ctxt-closed-R-imp-supt-R-distr*[*OF ctxt vw wy*] **obtain** *w* **where** $(v,w) \in R$ **and** $w \triangleright y$ **using** *ctxt-closed-R-imp-supt-R-distr*[**where** $R = R$ **and** $s = v$ **and** $t = z$ **and** $u = y$] **by** *auto*
with *ac E* **have** $(x,w) \in R$ **and** $w \triangleright y$ **by** *auto*
from *this* **and** ‹*SN-on R* $\{x\}$› **have** *SN-on R* $\{w\}$ **using** *step-preserves-SN-on*
unfolding *supt-supteq-conv* **by** *auto*
with ‹$w \triangleright y$› *E* **have** $(w,y) \in$ *?s* **unfolding** *restrict-SN-def supt-def* **by** *force*
with ‹$(x,w) \in R$› ‹*SN-on R* $\{x\}$› **show** $(x,y) \in$ *?r O* $(?r \cup ?s)^*$ **unfolding** *restrict-SN-def* **by** *auto*
qed
then show *?thesis* **unfolding** *quasi-commute-def* **.**
qed

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 −
let *?r* = *restrict-SN R R*
let *?s* = *restrict-SN* (*E O* {▷} *O E*) *R*
from ‹*ctxt.closed R*› **have** *quasi-commute ?r ?s*
by (*rule ctxt-closed-R-imp-restrict-SN-qc-E-supt*)
from *SN* **have** *SN ?s* **by** (*rule SN-subset*, *auto simp*: *restrict-SN-def*)
have *SN ?r* **by** (*rule SN-restrict-SN-idemp*)
from ‹*SN ?r*› **and** ‹*SN ?s*› **and** ‹*quasi-commute ?r ?s*›
show *?thesis* **unfolding** *restrict-SN-supt-E-def* **by** (*rule quasi-commute-imp-SN*)
qed
end

lemma *E-compatible-Id*: *E-compatible R Id*
by *standard auto*

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 (*rule ccontr*)
assume ¬ *SN-on R* {*t*}
then obtain *A* **where** *A 0* = *t* **and** ∀ *i*. (*A i*,*A* (*Suc i*)) ∈ *R*
unfolding *SN-on-def* **by** *best*
from ‹*s* ⊵ *t*› **obtain** *C* **where** *s* = *C*⟨*t*⟩ **by** *auto*
let *?B* = λ*i*. *C*⟨*A i*⟩
have ∀ *i*. (*?B i*,*?B*(*Suc i*)) ∈ *R*
proof
fix *i*
from ‹∀ *i*. (*A i*,*A*(*Suc i*)) ∈ *R*› **have** (*?B i*,*?B*(*Suc i*)) ∈ *ctxt.closure*(*R*) **by** *fast*
then show (*?B i*,*?B*(*Suc i*)) ∈ *R* **using** ‹*ctxt.closed R*› **by** *auto*
qed
with ‹*A 0* = *t*› **have** *?B 0* = *s* ∧ (∀ *i*. (*?B i*,*?B*(*Suc i*)) ∈ *R*) **unfolding** ‹*s* = *C*⟨*t*⟩› **by** *auto*
then have ¬ *SN-on R* {*s*} **unfolding** *SN-on-def* **by** *auto*
with *assms* **show** *False* **by** *simp*
qed

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*)

using *E-compatible.ctxt-closed-R-imp-supt-restrict-SN-E-distr*[*OF E-compatible-Id R - tu, of s*]
st **by** *auto*

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*)
using *E-compatible.ctxt-closed-R-imp-restrict-SN-qc-E-supt*[*OF E-compatible-Id assms*] **by** *auto*

lemma *ctxt-closed-imp-SN-restrict-SN-supt*:
assumes *ctxt.closed R*
shows *SN* (*restrict-SN-supt R*)
using *E-compatible.ctxt-closed-imp-SN-restrict-SN-E-supt*[*OF E-compatible-Id assms*]
unfolding *E-compatible.restrict-SN-supt-E-def*[*OF E-compatible-Id*] *restrict-SN-supt-def*
using *SN-supt* **by** *auto*

lemma *SN-restrict-SN-supt-rstep*:
shows *SN* (*restrict-SN-supt* (*rstep R*))
proof −
have *ctxt.closed* (*rstep R*) **by** (*rule ctxt-closed-rstep*)
then show *?thesis* **by** (*rule ctxt-closed-imp-SN-restrict-SN-supt*)
qed

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 −
assume (*Fun f ss*,*t*) ∈ *nrrstep R*
then obtain *l r i ps σ* **where** *rstep-rps*:(*Fun f ss*,*t*) ∈ *rstep-r-p-s R* (*l*,*r*) (*i*#*ps*) *σ*
unfolding *nrrstep-def* **by** *auto*
then obtain *C*
where (*l*,*r*) ∈ *R*
and *pos*: (*i*#*ps*) ∈ *poss* (*Fun f ss*)
and *C*: *C* = *ctxt-of-pos-term* (*i*#*ps*) (*Fun f ss*)
and *C*⟨*l*·*σ*⟩ = *Fun f ss*
and *t*: *C*⟨*r*·*σ*⟩ = *t*
unfolding *rstep-r-p-s-def Let-def* **by** *auto*
then obtain *D* **where** *C* = *More f* (*take i ss*) *D* (*drop* (*Suc i*) *ss*) **by** *auto*
with *t* **have** *t*′: *t* = *Fun f* (*take i ss* @ (*D*⟨*r*·*σ*⟩) # *drop* (*Suc i*) *ss*) **by** *auto*
from *rstep-rps* **have** (*ss*!*i*,*t*|-[*i*]) ∈ *rstep-r-p-s R* (*l*,*r*) *ps σ*
by (*rule rstep-i-pos-imp-rstep-arg-i-pos*)
then have *rstep*:(*ss*!*i*,*t*|-[*i*]) ∈ *rstep R* **by** (*auto simp*: *rstep-iff-rstep-r-p-s*)
then have (*C*⟨*ss*!*i*⟩,*C*⟨*t*|-[*i*]⟩) ∈ *rstep R* **..**
from *pos* **have** *len*: *i* < *length ss* **by** *auto*
from *len*
have (*take i ss* @ (*D*⟨*r*·*σ*⟩) # *drop* (*Suc i*) *ss*)!*i* = *D*⟨*r*·*σ*⟩ **by** (*auto simp*:

nth-append)
with $C\ t'$ **have** $t|\text{-}[i] = D\langle r{\cdot}\sigma\rangle$ **by** *auto*
with t' **have** $t = Fun\ f\ (take\ i\ ss\ @\ (t|\text{-}[i])\ \#\ drop\ (Suc\ i)\ ss)$ **by** *auto*
with *len* **have** $t = Fun\ f\ (ss[i{:=}t|\text{-}[i]])$ **by** (*auto simp*: *upd-conv-take-nth-drop*)
with *rstep len* **show** $\exists\, i\ s.\ t = Fun\ f\ (ss[i{:=}s]) \wedge (ss!i,s) \in rstep\ R \wedge i < length\ ss$ **by** *auto*
qed

lemma *rstep-cases*[*consumes 1*, *case-names root nonroot*]:
$[\![(s,t) \in rstep\ R;\ (s,t) \in rrstep\ R \Longrightarrow P;\ (s,t) \in nrrstep\ R \Longrightarrow P]\!] \Longrightarrow P$
by (*auto simp*: *rstep-iff-rrstep-or-nrrstep*)

lemma *nrrstep-imp-rstep*: $(s,t) \in nrrstep\ R \Longrightarrow (s,t) \in rstep\ R$
by (*auto simp*: *rstep-iff-rrstep-or-nrrstep*)

lemma *nrrstep-imp-Fun*: $(s,t) \in nrrstep\ R \Longrightarrow \exists f\ ss.\ s = Fun\ f\ ss$
proof −
assume $(s,t) \in nrrstep\ R$
then obtain *i ps* **where** $i\#ps \in poss\ s$
unfolding *nrrstep-def rstep-r-p-s-def Let-def* **by** *auto*
then show $\exists f\ ss.\ s = Fun\ f\ ss$ **by** (*cases s*) *auto*
qed

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 −
from ‹$(s,t) \in nrrstep\ R$› **obtain** *f ss* **where** $s = Fun\ f\ ss$ **using** *nrrstep-imp-Fun* **by** *blast*
with ‹$(s,t) \in nrrstep\ R$› **have** $(Fun\ f\ ss,t) \in nrrstep\ R$ **by** *simp*
then obtain *i u* **where** $t = Fun\ f\ (ss[i := u])$ **and** $(ss!i,u) \in rstep\ R$ **and** $i < length\ ss$
using *nrrstep-imp-pos-term* **by** *best*
from ‹$s = Fun\ f\ ss$› **and** ‹$t = Fun\ f\ (ss[i := u])$› **have** $num\text{-}args\ s = num\text{-}args\ t$ **by** *auto*
from ‹$i < length\ ss$› **and** ‹$s = Fun\ f\ ss$› **have** $i < num\text{-}args\ s$ **by** *auto*
from ‹$s = Fun\ f\ ss$› **have** $s|\text{-}[i] = (ss!i)$ **by** *auto*
from ‹$t = Fun\ f\ (ss[i := u])$› **and** ‹$i < length\ ss$› **have** $t|\text{-}[i] = u$ **by** *auto*
from ‹$s = Fun\ f\ ss$› **and** ‹$t = Fun\ f\ (ss[i := u])$›
have $\forall j.\ j \neq i \longrightarrow s|\text{-}[j] = t|\text{-}[j]$ **by** *auto*
with ‹$(ss!i,u) \in rstep\ R$›
and ‹$i < num\text{-}args\ s$›
and ‹$num\text{-}args\ s = num\text{-}args\ t$›
and ‹$s|\text{-}[i] = (ss!i)$›[*symmetric*] **and** ‹$t|\text{-}[i] = u$›[*symmetric*]
show *?thesis* **by** (*auto*)
qed

lemma *nrrstep-subt*: **assumes** $(s,\ t) \in nrrstep\ R$ **shows** $\exists\, u \lhd s.\ \exists\, v \lhd t.\ (u,\ v) \in rstep\ R$
proof −
from *assms* **obtain** $l\ r\ C\ \sigma$ **where** $(l,\ r) \in R$ **and** $C \neq \Box$
and *s*: $s = C\langle l \cdot \sigma \rangle$ **and** *t*: $t = C\langle r \cdot \sigma \rangle$ **unfolding** *nrrstep-def′* **by** *best*
from ‹$C \neq \Box$› s **have** $s \rhd l \cdot \sigma$ **by** *auto*
moreover from ‹$C \neq \Box$› t **have** $t \rhd r \cdot \sigma$ **by** *auto*
moreover from ‹$(l,\ r) \in R$› **have** $(l \cdot \sigma,\ r \cdot \sigma) \in rstep\ R$ **by** *auto*
ultimately show *?thesis* **by** *auto*
qed

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 −
from *assms* **obtain** $l\ r\ C\ \sigma$ **where** $(l,\ r) \in R$ **and** $C \neq \Box$
and *s*: $s = C\langle l \cdot \sigma \rangle$ **and** *t*: $t = C\langle r \cdot \sigma \rangle$ **unfolding** *nrrstep-def′* **by** *best*
from ‹$C \neq \Box$› **obtain** *f ss1 D ss2* **where** *C*: $C = More\ f\ ss1\ D\ ss2$ **by** (*induct C*) *auto*
have $s = Fun\ f\ (ss1\ @\ D\langle l \cdot \sigma \rangle\ \#\ ss2)$ (**is** - $= Fun\ f\ ?ss$) **by** (*simp add*: *s C*)
moreover have $t = Fun\ f\ (ss1\ @\ D\langle r \cdot \sigma \rangle\ \#\ ss2)$ (**is** - $= Fun\ f\ ?ts$) **by** (*simp add*: *t C*)
moreover have $length\ ?ss = length\ ?ts$ **by** *simp*
moreover have $\exists j{<}length\ ?ss.$
$(?ss!j,\ ?ts!j) \in rstep\ R \wedge (\forall\, i{<}length\ ?ss.\ i \neq j \longrightarrow ?ss!i = ?ts!i)$
proof −
let $?j = length\ ss1$
have $?j < length\ ?ss$ **by** *simp*
moreover have $(?ss!?j,\ ?ts!?j) \in rstep\ R$
proof −
from ‹$(l,\ r) \in R$› **have** $(D\langle l \cdot \sigma \rangle,\ D\langle r \cdot \sigma \rangle) \in rstep\ R$ **by** *auto*
then show *?thesis* **by** *auto*
qed
moreover have $\forall\, i{<}length\ ?ss.\ i \neq ?j \longrightarrow ?ss!i = ?ts!i$ (**is** $\forall\, i{<}length\ ?ss.$ - $\longrightarrow ?P\ i$)
proof (*intro allI impI*)
fix i **assume** $i < length\ ?ss$ **and** $i \neq ?j$
then have $i < length\ ss1 \vee length\ ss1 < i$ **by** *auto*
then show $?P\ i$
proof
assume $i < length\ ss1$ **then show** $?P\ i$ **by** (*auto simp*: *nth-append*)
next
assume $i > length\ ss1$ **then show** $?P\ i$
using ‹$i < length\ ?ss$› **by** (*auto simp*: *nth-Cons′ nth-append*)
qed
qed
ultimately show *?thesis* **by** *best*
qed

ultimately show *?thesis* **by** *auto*
qed

lemma *nrrstep-iff-arg-rstep*:
(*s*,*t*) ∈ *nrrstep R* ⟷
(∃*f ss i t′*. *s* = *Fun f ss* ∧ *i* < *length ss* ∧ *t* = *Fun f* (*ss*[*i*:=*t′*]) ∧ (*ss*!*i*,*t′*) ∈ *rstep R*)
(**is** *?L* ⟷ *?R*)
proof
assume *L*: *?L*
from *nrrstep-args*[*OF this*]
obtain *f ss ts* **where** *s* = *Fun f ss t* = *Fun f ts* **by** *auto*
with *nrrstep-imp-pos-term*[*OF L*[*unfolded this*]]
show *?R* **by** *auto*
next assume *R*: *?R*
then obtain *f i ss t′*
where *s*: *s* = *Fun f ss* **and** *t*: *t* = *Fun f* (*ss*[*i*:=*t′*])
and *i*: *i* < *length ss* **and** *st′*: (*ss* ! *i*, *t′*) ∈ *rstep R* **by** *auto*
from *st′* **obtain** *C l r σ* **where** *lr*: (*l*, *r*) ∈ *R* **and** *s′*: *ss*!*i* = *C*⟨*l* · *σ*⟩ **and** *t′*: *t′* = *C*⟨*r* · *σ*⟩ **by** *auto*
let *?D* = *More f* (*take i ss*) *C* (*drop* (*Suc i*) *ss*)
have *s* = *?D*⟨*l* · *σ*⟩ *t* = *?D*⟨*r* · *σ*⟩ **unfolding** *s t*
using *id-take-nth-drop*[*OF i*] *upd-conv-take-nth-drop*[*OF i*] *s′ t′* **by** *auto*
with *lr* **show** *?L* **apply**(*rule nrrstepI*) **using** *t′* **by** *auto*
qed

lemma *subterms-NF-imp-SN-on-nrrstep*:
assumes ∀ *s*◁*t*. *s* ∈ *NF* (*rstep R*) **shows** *SN-on* (*nrrstep R*) {*t*}
proof
fix *S* **assume** *S* 0 ∈ {*t*} **and** ∀ *i*. (*S i*, *S* (*Suc i*)) ∈ *nrrstep R*
then have (*t*, *S* (*Suc* 0)) ∈ *nrrstep R* **by** *auto*
then obtain *l r C σ* **where** (*l*, *r*) ∈ *R* **and** *C* ≠ □ **and** *t*: *t* = *C*⟨*l*·*σ*⟩ **unfolding** *nrrstep-def′* **by** *auto*
then obtain *f ss1 D ss2* **where** *C*: *C* = *More f ss1 D ss2* **by** (*induct C*) *auto*
have *t* ▷ *D*⟨*l*·*σ*⟩ **unfolding** *t C* **by** *auto*
moreover have *D*⟨*l*·*σ*⟩ ∉ *NF* (*rstep R*)
proof −
from ‹(*l*, *r*) ∈ *R*› **have** (*D*⟨*l*·*σ*⟩, *D*⟨*r*·*σ*⟩) ∈ *rstep R* **by** *auto*
then show *?thesis* **by** *auto*
qed
ultimately show *False* **using** *assms* **by** *simp*
qed

lemma *args-NF-imp-SN-on-nrrstep*:
assumes ∀ *t*∈*set ts*. *t* ∈ *NF* (*rstep R*) **shows** *SN-on* (*nrrstep R*) {*Fun f ts*}
proof
fix *S* **assume** *S* 0 ∈ {*Fun f ts*} **and** ∀ *i*. (*S i*, *S* (*Suc i*)) ∈ *nrrstep R*
then have (*Fun f ts*, *S* (*Suc* 0)) ∈ *nrrstep R*

```
    unfolding singletonD[OF ‹S 0 ∈ {Fun f ts}›, symmetric] by simp
  then obtain l r C σ where (l, r) ∈ R and C ≠ □ and Fun f ts = C⟨l·σ⟩
    unfolding nrrstep-def' by auto
  then obtain ss1 D ss2 where C: C = More f ss1 D ss2 by (induct C) auto
  with ‹Fun f ts = C⟨l·σ⟩› have D⟨l·σ⟩ ∈ set ts by auto
  moreover have D⟨l·σ⟩ ∉ NF (rstep R)
  proof −
    from ‹(l, r) ∈ R› have (D⟨l·σ⟩, D⟨r·σ⟩) ∈ rstep R by auto
    then show ?thesis by auto
  qed
  ultimately show False using assms by simp
qed

lemma rrstep-imp-rule-subst:
  assumes (s,t) ∈ rrstep R
  shows ∃ l r σ. (l,r) ∈ R ∧ (l·σ) = s ∧ (r·σ) = t
proof −
  have ctxt-of-pos-term [] s = Hole by auto
   obtain l r σ where (s,t) ∈ rstep-r-p-s R (l,r) [] σ using assms unfolding
rrstep-def by best
  then have let C = ctxt-of-pos-term [] s in [] ∈ poss s ∧ (l,r) ∈ R ∧ C⟨l·σ⟩ = s
∧ C⟨r·σ⟩ = t unfolding rstep-r-p-s-def by simp
   with ‹ctxt-of-pos-term [] s = Hole› have (l,r) ∈ R and l·σ = s and r·σ = t
unfolding Let-def by auto
  then show ?thesis by auto
qed

lemma nrrstep-preserves-root:
  assumes (Fun f ss,t) ∈ nrrstep R (is (?s,t) ∈ nrrstep R) shows ∃ ts. t = (Fun
f ts)
  using assms unfolding nrrstep-def rstep-r-p-s-def Let-def by auto

lemma nrrstep-equiv-root: assumes (s,t) ∈ nrrstep R shows ∃ f ss ts. s = Fun f
ss ∧ t = Fun f ts
proof −
  from assms obtain f ss where s = Fun f ss using nrrstep-imp-Fun by best
  with assms obtain ts where t = Fun f ts using nrrstep-preserves-root by best
  from ‹s = Fun f ss› and ‹t = Fun f ts› show ?thesis by best
qed

lemma nrrstep-reflects-root:
  assumes (s,Fun g ts) ∈ nrrstep R (is (s,?t) ∈ nrrstep R)
  shows ∃ ss. s = (Fun g ss)
proof −
  from assms obtain f ss ts' where s = Fun f ss and Fun g ts = Fun f ts' using
nrrstep-equiv-root by best
  then have f = g by simp
  with ‹s = Fun f ss› have s = Fun g ss by simp
  then show ?thesis by auto
```

qed

lemma *nrrsteps-preserve-root*:
assumes (*Fun f ss*,*t*) ∈ (*nrrstep R*)*
shows ∃ *ts*. *t* = (*Fun f ts*)
using *assms* **by** *induct* (*auto simp*: *nrrstep-preserves-root*)

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 −
from *assms* **have** [*i*] ∈ *poss ?s* **using** *empty-pos-in-poss* **by** *simp*
from ‹(*?s*,*?t*) ∈ *nrrstep R*›
obtain *l r j ps σ*
where *A*: *let C* = *ctxt-of-pos-term* (*j*#*ps*) *?s in* (*j*#*ps*) ∈ *poss ?s* ∧ (*l*,*r*) ∈ *R* ∧ *C*⟨*l*·*σ*⟩ = *?s* ∧ *C*⟨*r*·*σ*⟩ = *?t* **unfolding** *nrrstep-def rstep-r-p-s-def* **by** *force*
let *?C* = *ctxt-of-pos-term* (*j*#*ps*) *?s*
from *A* **have** (*j*#*ps*) ∈ *poss ?s* **and** (*l*,*r*) ∈ *R* **and** *?C*⟨*l*·*σ*⟩ = *?s* **and** *?C*⟨*r*·*σ*⟩ = *?t* **using** *Let-def* **by** *auto*
have *C*: *?C* = *More f* (*take j ss*) (*ctxt-of-pos-term ps* (*ss*!*j*)) (*drop* (*Suc j*) *ss*) (**is** *?C* = *More f ?ss1 ?D ?ss2*) **by** *auto*
from ‹*?C*⟨*l*·*σ*⟩ = *?s*› **have** *?D*⟨*l*·*σ*⟩ = (*ss*!*j*) **by** (*auto simp*: *take-drop-imp-nth*)
from ‹(*l*,*r*) ∈ *R*› **have** (*l*·*σ*,*r*·*σ*) ∈ (*subst.closure R*) **by** *auto*
then have (*?D*⟨*l*·*σ*⟩,*?D*⟨*r*·*σ*⟩) ∈ (*ctxt.closure*(*subst.closure R*))
and (*?C*⟨*l*·*σ*⟩,*?C*⟨*r*·*σ*⟩) ∈ (*ctxt.closure*(*subst.closure R*)) **by** (*auto simp only*: *ctxt.closure.intros*)
then have *D-rstep*: (*?D*⟨*l*·*σ*⟩,*?D*⟨*r*·*σ*⟩) ∈ *rstep R* **and** (*?C*⟨*l*·*σ*⟩,*?C*⟨*r*·*σ*⟩) ∈ *rstep R*
unfolding *rstep-eq-closure* **by** *auto*
from ‹*?C*⟨*r*·*σ*⟩ = *?t*› **and** *C* **have** *?t* = *Fun f* (*take j ss* @ *?D*⟨*r*·*σ*⟩ # *drop* (*Suc j*) *ss*) **by** *auto*
then have *ts*: *ts* = (*take j ss* @ *?D*⟨*r*·*σ*⟩ # *drop* (*Suc j*) *ss*) **by** *auto*
have *j* = *i* ∨ *j* ≠ *i* **by** *simp*
from ‹*j*#*ps* ∈ *poss ?s*› **have** *j* < *length ss* **by** *simp*
then have (*take j ss* @ *?D*⟨*r*·*σ*⟩ # *drop* (*Suc j*) *ss*) ! *j* = *?D*⟨*r*·*σ*⟩ **by** (*auto simp*: *nth-append-take*)
with *ts* **have** *ts*!*j* = *?D*⟨*r*·*σ*⟩ **by** *auto*
have *j* = *i* ∨ *j* ≠ *i* **by** *simp*
then show (*ss*!*i*,*ts*!*i*) ∈ (*rstep R*)*
proof
assume *j* = *i*
with ‹*ts*!*j* = *?D*⟨*r*·*σ*⟩› **and** ‹*?D*⟨*l*·*σ*⟩ = *ss*!*j*› **and** *D-rstep* **show** *?thesis* **by** *auto*
next
assume *j* ≠ *i*
with ‹*i* < *length ss*› **and** ‹*j* < *length ss*› **have** (*take j ss* @ *?D*⟨*r*·*σ*⟩ # *drop* (*Suc j*) *ss*) ! *i* = *ss*!*i* **by** (*auto simp*: *nth-append-take-drop-is-nth-conv*)
with *ts* **have** *ts*!*i* = *ss*!*i* **by** *auto*
then show *?thesis* **by** *auto*

qed
qed

lemma *nrrstep-imp-arg-rsteps*:
assumes $(s,t) \in$ *nrrstep R* **and** $i <$ *num-args s* **shows** $($*args s*$!i$,*args t*$!i) \in ($*rstep R*$)^*$
proof (*cases s*)
fix *x* **assume** $s =$ *Var x* **then show** *?thesis* **using** *assms* **by** *auto*
next
fix *f ss* **assume** $s =$ *Fun f ss*
then have $($*Fun f ss*$,t) \in$ *nrrstep R* **using** *assms* **by** *simp*
then obtain *ts* **where** $t =$ *Fun f ts* **using** *nrrstep-preserves-root* **by** *best*
with ‹$($*s*$,t) \in$ *nrrstep R*› **and** ‹$s =$ *Fun f ss*› **have** $($*Fun f ss*$,$*Fun f ts*$) \in$ *nrrstep R* **by** *simp*
from ‹$s =$ *Fun f ss*› **and** ‹$i <$ *num-args s*› **have** $i <$ *length ss* **by** *simp*
with ‹$($*Fun f ss*$,$*Fun f ts*$) \in$ *nrrstep R*›
have $($*ss*$!i,$*ts*$!i) \in ($*rstep R*$)^*$ **by** (*rule nrrstep-Fun-imp-arg-rsteps*)
with ‹$s =$ *Fun f ss*› **and** ‹$t =$ *Fun f ts*› **show** *?thesis* **by** *simp*
qed

lemma *nrrsteps-imp-rsteps*: $(s,t) \in ($*nrrstep R*$)^* \Longrightarrow (s,t) \in ($*rstep R*$)^*$
proof (*induct rule*: *rtrancl.induct*)
case (*rtrancl-refl a*) **then show** *?case* **by** *simp*
next
case (*rtrancl-into-rtrancl a b c*)
then have *IH*: $(a,b) \in ($*rstep R*$)^*$ **and** *nrrstep*: $(b,c) \in$ *nrrstep R* **by** *auto*
from *nrrstep* **have** $(b,c) \in$ *rstep R* **using** *nrrstep-imp-rstep* **by** *auto*
with *IH* **show** *?case* **by** *auto*
qed

lemma *nrrstep-Fun-preserves-num-args*:
assumes $($*Fun f ss*$,$*Fun f ts*$) \in$ *nrrstep R* (**is** $($*?s*$,$*?t*$) \in$ *nrrstep R*$)$
shows *length ss* $=$ *length ts*
proof −
from *assms* **obtain** *l r i ps* σ
where *let C* $=$ *ctxt-of-pos-term* $(i\#ps)$ *?s in* $(i\#ps) \in$ *poss ?s* $\wedge$ $(l,r) \in R \wedge C\langle l{\cdot}\sigma\rangle =$ *?s* $\wedge$ $C\langle r{\cdot}\sigma\rangle =$ *?t* (**is** *let C* $=$ *?C in* -)
unfolding *nrrstep-def rstep-r-p-s-def* **by** *force*
then have $(l,r) \in R$ **and** *Cl*: $?C\langle l{\cdot}\sigma\rangle =$ *?s* **and** *Cr*: $?C\langle r{\cdot}\sigma\rangle =$ *?t* **unfolding** *Let-def* **by** *auto*
have *C*: *?C* $=$ *More f* (*take i ss*) (*ctxt-of-pos-term ps* (*ss*$!i$)) (*drop* (*Suc i*) *ss*) (**is** *?C* $=$ *More f ?ss1 ?D ?ss2*) **by** *simp*
from *C* **and** *Cl* **have** *s*: *?s* $=$ *Fun f* (*take i ss* @ $?D\langle l{\cdot}\sigma\rangle$ # *drop* (*Suc i*) *ss*) (**is** *?s* $=$ *Fun f ?ss*) **by** *simp*
from *C* **and** *Cr* **have** *t*: *?t* $=$ *Fun f* (*take i ss* @ $?D\langle r{\cdot}\sigma\rangle$ # *drop* (*Suc i*) *ss*) (**is** *?t* $=$ *Fun f ?ts*) **by** *simp*
from *s* **and** *t* **have** *ss*: *ss* $=$ *?ss* **and** *ts*: *ts* $=$ *?ts* **by** *auto*
have *length ?ss* $=$ *length ?ts* **by** *auto*
with *ss* **and** *ts* **show** *?thesis* **by** *simp*

qed

lemma *nrrstep-equiv-num-args*:
assumes $(s,t) \in$ *nrrstep R* **shows** *num-args s* = *num-args t*
proof −
from *assms* **obtain** *f ss ts* **where** *s*:*s* = *Fun f ss* **and** *t*:*t* = *Fun f ts* **using** *nrrstep-equiv-root* **by** *best*
with *assms* **have** (*Fun f ss*,*Fun f ts*) ∈ *nrrstep R* **by** *simp*
then have *length ss* = *length ts* **by** (*rule nrrstep-Fun-preserves-num-args*)
with *s* **and** *t* **show** *?thesis* **by** *auto*
qed

lemma *nrrsteps-equiv-num-args*:
assumes $(s,t) \in (nrrstep\ R)^*$ **shows** *num-args s* = *num-args t*
using *assms* **by** (*induct*, *auto simp*: *nrrstep-equiv-num-args*)

lemma *nrrstep-preserves-num-args*:
assumes $(s,t) \in$ *nrrstep R* **and** *i* < *num-args s* **shows** *i* < *num-args t*
proof (*cases s*)
fix *x* **assume** *s* = *Var x* **then show** *?thesis* **using** *assms* **by** *auto*
next
fix *f ss* **assume** *s* = *Fun f ss*
with *assms* **obtain** *ts* **where** *t* = *Fun f ts* **using** *nrrstep-preserves-root* **by** *best*
from ‹(*s*,*t*) ∈ *nrrstep R*› **have** *length ss* = *length ts* **unfolding** ‹*s* = *Fun f ss*› **and** ‹*t* = *Fun f ts*› **by** (*rule nrrstep-Fun-preserves-num-args*)
with *assms* **and** ‹*s* = *Fun f ss*› **and** ‹*t* = *Fun f ts*› **show** *?thesis* **by** *auto*
qed

lemma *nrrstep-reflects-num-args*:
assumes $(s,t) \in$ *nrrstep R* **and** *i* < *num-args t* **shows** *i* < *num-args s*
proof (*cases t*)
fix *x* **assume** *t* = *Var x* **then show** *?thesis* **using** *assms* **by** *auto*
next
fix *g ts* **assume** *t* = *Fun g ts*
with *assms* **obtain** *ss* **where** *s* = *Fun g ss* **using** *nrrstep-reflects-root* **by** *best*
from ‹(*s*,*t*) ∈ *nrrstep R*› **have** *length ss* = *length ts* **unfolding** ‹*s* = *Fun g ss*› **and** ‹*t* = *Fun g ts*› **by** (*rule nrrstep-Fun-preserves-num-args*)
with *assms* **and** ‹*s* = *Fun g ss*› **and** ‹*t* = *Fun g ts*› **show** *?thesis* **by** *auto*
qed

lemma *nrrsteps-imp-arg-rsteps*:
assumes $(s,t) \in (nrrstep\ R)^*$ **and** *i* < *num-args s*
shows $(args\ s!i, args\ t!i) \in (rstep\ R)^*$
using *assms*
proof (*induct rule*: *rtrancl.induct*)
case (*rtrancl-refl a*) **then show** *?case* **by** *auto*
next
case (*rtrancl-into-rtrancl a b c*)
then have *IH*: $(args\ a!i, args\ b!i) \in (rstep\ R)^*$ **by** *auto*

from ‹$(a,b) \in (nrrstep\ R)^*$› **and** ‹$i < num\text{-}args\ a$› **have** $i < num\text{-}args\ b$ **by** *induct* (*auto simp*: *nrrstep-preserves-num-args*)
with ‹$(b,c) \in nrrstep\ R$›
have (*args b*!*i*,*args c*!*i*) ∈ (*rstep R*)* **by** (*rule nrrstep-imp-arg-rsteps*)
with *IH* **show** *?case* **by** *simp*
qed

lemma *nrrsteps-imp-eq-root-arg-rsteps*:
assumes *steps*: (*s*,*t*) ∈ (*nrrstep R*)*
shows *root s* = *root t* ∧ (∀ *i*<*num-args s*. (*s* |- [*i*], *t* |- [*i*]) ∈ (*rstep R*)*)
proof (*cases s*)
case (*Var x*)
have *s* = *t* **using** *steps* **unfolding** *Var*
proof (*induct*)
case (*step y z*)
from *nrrstep-imp-Fun*[*OF step*(*2*)] *step*(*3*) **have** *False* **by** *auto*
then show *?case* **..**
qed *simp*
then show *?thesis* **by** *auto*
next
case (*Fun f ss*)
from *nrrsteps-equiv-num-args*[*OF steps*]
nrrsteps-imp-arg-rsteps[*OF steps*]
nrrsteps-preserve-root[*OF steps*[*unfolded Fun*]]
show *?thesis* **unfolding** *Fun* **by** *auto*
qed

lemma *SN-on-imp-SN-on-subt*:
assumes *SN-on* (*rstep R*) {*t*} **shows** ∀ *s*◁*t*. *SN-on* (*rstep R*) {*s*}
proof (*rule ccontr*)
assume ¬(∀ *s*◁*t*. *SN-on* (*rstep R*) {*s*})
then obtain *s* **where** *t* ▷ *s* **and** ¬ *SN-on* (*rstep R*) {*s*} **by** *auto*
then obtain *S* **where** *S 0* = *s* **and** *chain*: *chain* (*rstep R*) *S* **by** *auto*
from ‹*t* ▷ *s*› **obtain** *C* **where** *t*: *t* = *C*⟨*s*⟩ **by** *auto*
let *?S* = λ*i*. *C*⟨*S i*⟩
from ‹*S 0* = *s*› **have** *?S 0* = *t* **by** (*simp add*: *t*)
moreover from *chain* **have** *chain* (*rstep R*) *?S* **by** *blast*
ultimately have ¬ *SN-on* (*rstep R*) {*t*} **by** *best*
with *assms* **show** *False* **by** *simp*
qed

lemma *not-SN-on-subt-imp-not-SN-on*:
assumes ¬ *SN-on* (*rstep R*) {*t*} **and** *s* ▷ *t*
shows ¬ *SN-on* (*rstep R*) {*s*}
using *assms SN-on-imp-SN-on-subt* **by** *blast*

lemma *SN-on-instance-imp-SN-on-var*:
assumes *SN-on* (*rstep R*) {*t* · σ} **and** *x* ∈ *vars-term t*
shows *SN-on* (*rstep R*) {*Var x* · σ}

proof −
from *assms* **have** $t \trianglerighteq Var\ x$ **by** *auto*
then have $t{\cdot}\sigma \trianglerighteq (Var\ x){\cdot}\sigma$ **by** (*rule supteq-subst*)
with *SN-on-imp-SN-on-subt* **and** *assms* **show** *?thesis* **by** *best*
qed

lemma *var-imp-var-of-arg*:
assumes $x \in vars\text{-}term\ (Fun\ f\ ss)$ (**is** $x \in vars\text{-}term\ ?s$)
shows $\exists\, i < num\text{-}args\ (Fun\ f\ ss).\ x \in vars\text{-}term\ (ss!i)$
proof −
from *assms* **have** $x \in \bigcup(set\ (map\ vars\text{-}term\ ss))$ **by** *simp*
then have $x \in (\bigcup i{<}length\ ss.\ vars\text{-}term(ss!i))$ **unfolding** *set-conv-nth* **by** *auto*
then have $\exists\, i{<}length\ ss.\ x \in vars\text{-}term(ss!i)$ **using** *UN-iff* **by** *best*
then obtain i **where** $i < length\ ss$ **and** $x \in vars\text{-}term(ss!i)$ **by** *auto*
then have $i < num\text{-}args\ ?s$ **by** *simp*
with ‹$x \in vars\text{-}term(ss!i)$› **show** *?thesis* **by** *auto*
qed

lemma *subt-instance-and-not-subst-imp-subt*:
$s{\cdot}\sigma \trianglerighteq t \Longrightarrow \forall\, x \in vars\text{-}term\ s.\ \neg((Var\ x){\cdot}\sigma \trianglerighteq t) \Longrightarrow \exists\, u.\ s \trianglerighteq u \wedge t = u{\cdot}\sigma$
proof (*induct s arbitrary*: *t rule*: *term.induct*)
case (*Var x*) **then show** *?case* **by** *auto*
next
case (*Fun f ss*)
from ‹$Fun\ f\ ss{\cdot}\sigma \trianglerighteq t$› **have** $(Fun\ f\ ss{\cdot}\sigma = t) \vee (Fun\ f\ ss{\cdot}\sigma \rhd t)$ **by** *auto*
then show *?case*
proof
assume $Fun\ f\ ss{\cdot}\sigma = t$ **with** *Fun* **show** *?thesis* **by** *auto*
next
assume $Fun\ f\ ss{\cdot}\sigma \rhd t$
then have $Fun\ f\ (map\ (\lambda t.\ t{\cdot}\sigma)\ ss) \rhd t$ **by** *simp*
then have $\exists\, s \in set\ (map\ (\lambda t.\ t{\cdot}\sigma)\ ss).\ s \trianglerighteq t$ (**is** $\exists\, s \in set\ ?ss'.\ s \trianglerighteq t$) **by** (*rule supt-Fun-imp-arg-supteq*)
then obtain s' **where** $s' \in set\ ?ss'$ **and** $s' \trianglerighteq t$ **by** *best*
then have $\exists\, i < length\ ?ss'.\ ?ss'!i = s'$ **using** *in-set-conv-nth*[**where** $?x = s'$] **by** *best*
then obtain i **where** $i < length\ ?ss'$ **and** $?ss'!i = s'$ **by** *best*
then have $?ss'!i = (ss!i){\cdot}\sigma$ **by** *auto*
from ‹$?ss'!i = s'$› **have** $s' = (ss!i){\cdot}\sigma$ **unfolding** ‹$?ss'!i = (ss!i){\cdot}\sigma$› **by** *simp*
from ‹$s' \trianglerighteq t$› **have** $(ss!i){\cdot}\sigma \trianglerighteq t$ **unfolding** ‹$s' = (ss!i){\cdot}\sigma$› **by** *simp*
with ‹$i < length\ ?ss'$› **have** $(ss!i) \in set\ ss$ **by** *auto*
with ‹$(ss!i){\cdot}\sigma \trianglerighteq t$› **have** $\exists\, s \in set\ ss.\ s{\cdot}\sigma \trianglerighteq t$ **by** *best*
then obtain s **where** $s \in set\ ss$ **and** $s{\cdot}\sigma \trianglerighteq t$ **by** *best*
with *Fun* **have** $\forall\, x \in vars\text{-}term\ s.\ \neg((Var\ x){\cdot}\sigma \trianglerighteq t)$ **by** *force*
from *Fun*
have *IH*: $s \in set\ ss \longrightarrow (\forall\, v.\ s{\cdot}\sigma \trianglerighteq v \longrightarrow (\forall\, x \in vars\text{-}term\ s.\ \neg\ Var\ x{\cdot}\sigma \trianglerighteq v) \longrightarrow (\exists\, u.\ s \trianglerighteq u \wedge v = u{\cdot}\sigma))$
by *auto*
with ‹$s \in set\ ss$› **have** $!!v.\ s{\cdot}\sigma \trianglerighteq v \Longrightarrow (\forall\, x \in vars\text{-}term\ s.\ \neg\ Var\ x{\cdot}\sigma \trianglerighteq v)$

$\longrightarrow$ ($\exists\, u.\ s \trianglerighteq u \wedge v = u{\cdot}\sigma$)
by *simp*
with ‹$s{\cdot}\sigma \trianglerighteq t$› **have** ($\forall\, x \in$ *vars-term s.* $\neg$ *Var* $x{\cdot}\sigma \trianglerighteq t$) $\longrightarrow$ ($\exists\, u.\ s \trianglerighteq u \wedge t = u{\cdot}\sigma$) **by** *simp*
with ‹$\forall\, x \in$ *vars-term s.* $\neg$ *Var* $x{\cdot}\sigma \trianglerighteq t$› **obtain** u **where** $s \trianglerighteq u$ **and** $t = u{\cdot}\sigma$ **by** *best*
with ‹$s \in$ *set ss*› **have** *Fun f ss* $\trianglerighteq u$ **by** *auto*
with ‹$t = u{\cdot}\sigma$› **show** *?thesis* **by** *best*
qed
qed

lemma *SN-imp-SN-subt*:
SN-on (*rstep R*) {s} $\Longrightarrow$ $s \trianglerighteq t$ $\Longrightarrow$ *SN-on* (*rstep R*) {t}
by (*rule ctxt-closed-SN-on-subt*[*OF ctxt-closed-rstep*])

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 −
from *supt* **have** $t \trianglerighteq s$ **by** *auto*
then show *?thesis* **using** *ctxt-closed-SN-on-subt*[*OF ctxt SN*, *of s*] **by** *simp*
qed

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}
using *assms E*(*1*) **unfolding** *SN-on-all-reducts-SN-on-conv*[*of - t*] *SN-on-all-reducts-SN-on-conv*[*of - s*]
by *blast*

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 (*cases* (s,t) $\in$ *R*)
case *True*
from *step-preserves-SN-on*[*OF this SN*] **show** *?thesis* **.**
next
case *False*
with *st* **obtain** u v **where** *su*: (s,u) $\in$ E **and** *uv*: $u \rhd v$ **and** *vt*: (v,t) $\in$ E **by** *auto*
have *u*: *SN-on R* {u} **by** (*rule SN-on-step-E-imp-SN-on*[*OF SN su*])
with *uv R* **have** *SN-on R* {v} **by** (*metis subterm-preserves-SN-gen*)
then show *?thesis* **by** (*rule SN-on-step-E-imp-SN-on*[*OF - vt*])

qed

lemma *restrict-SN-supt-E-I*:
ctxt.closed R $\Longrightarrow$ *SN-on R* $\{s\}$ $\Longrightarrow$ $(s,t) \in R \cup E\ O\ \{\rhd\}\ O\ E \Longrightarrow (s,t) \in$ *restrict-SN-supt-E*
unfolding *restrict-SN-supt-E-def restrict-SN-def*
using *SN-on-step-REs-imp-SN-on*[*of s t*] *E*(*2*) **by** *auto*

lemma *ctxt-closed-imp-SN-on-E-supt*:
assumes *R*: *ctxt.closed R*
and *SN*: *SN* (*E O* $\{\rhd\}$ *O E*)
shows *SN-on* (*R* $\cup$ *E O* $\{\rhd\}$ *O E*) $\{t.$ *SN-on R* $\{t\}\}$
proof −
{
fix *f*
assume *f0*: *SN-on R* $\{f\ 0\}$ **and** *f*: $\bigwedge$ *i*. (*f i*, *f* (*Suc i*)) $\in R \cup E\ O\ \{\rhd\}\ O\ E$
from *ctxt-closed-imp-SN-restrict-SN-E-supt*[*OF assms*]
have *SN*: *SN restrict-SN-supt-E* **.**
{
fix *i*
have *SN-on R* $\{f\ i\}$
by (*induct i*, *rule f0*, *rule SN-on-step-REs-imp-SN-on*[*OF R f*])
} **note** *fi* = *this*
{
fix *i*
from *f*[*of i*] *fi*[*of i*]
have (*f i*, *f* (*Suc i*)) $\in$ *restrict-SN-supt-E* **by** (*metis restrict-SN-supt-E-I*[*OF R*])
}
with *SN* **have** *False* **by** *auto*
}
then show *?thesis* **unfolding** *SN-on-def* **by** *blast*
qed
end

lemma *subterm-preserves-SN*:
SN-on (*rstep R*) $\{t\} \Longrightarrow t \rhd s \Longrightarrow$ *SN-on* (*rstep R*) $\{s\}$
by (*rule subterm-preserves-SN-gen*[*OF ctxt-closed-rstep*])

lemma *SN-on-r-imp-SN-on-supt-union-r*:
assumes *ctxt*: *ctxt.closed R*
and *SN-on R T*
shows *SN-on* (*supt* $\cup$ *R*) *T* (**is** *SN-on ?S T*)
proof (*rule ccontr*)
assume ¬ *SN-on ?S T*
then obtain *s* **where** *ini*: *s 0* : *T* **and** *chain*: *chain ?S s*
unfolding *SN-on-def* **by** *auto*
have *SN*: $\forall$ *i*. *SN-on R* $\{s\ i\}$
proof

```
    fix i show SN-on R {s i}
    proof (induct i)
      case 0 show ?case using assms using ‹s 0 ∈ T› and SN-on-subset2[of {s
0} T R] by simp
    next
      case (Suc i)
      from chain have (s i, s (Suc i)) ∈ ?S by simp
      then show ?case
      proof
        assume (s i, s (Suc i)) ∈ supt
        from subterm-preserves-SN-gen[OF ctxt Suc this] show ?thesis .
      next
        assume (s i, s (Suc i)) ∈ R
        from step-preserves-SN-on[OF this Suc] show ?thesis .
      qed
    qed
  qed
  have ¬ (∃ S. ∀ i. (S i, S (Suc i)) ∈ restrict-SN-supt R)
   using ctxt-closed-imp-SN-restrict-SN-supt[OF ctxt] unfolding SN-defs by auto
  moreover have ∀ i. (s i, s (Suc i)) ∈ restrict-SN-supt R
  proof
    fix i
    from SN have SN: SN-on R {s i} by simp
    from chain have (s i, s (Suc i)) ∈ supt ∪ R by simp
    then show (s i, s (Suc i)) ∈ restrict-SN-supt R
      unfolding restrict-SN-supt-def restrict-SN-def using SN by auto
  qed
  ultimately show False by auto
qed

lemma SN-on-rstep-imp-SN-on-supt-union-rstep:
  SN-on (rstep R) T ⟹ SN-on (supt ∪ rstep R) T
  by (rule SN-on-r-imp-SN-on-supt-union-r[OF ctxt-closed-rstep])

lemma SN-supt-r-trancl:
  assumes ctxt: ctxt.closed R
    and a: SN R
  shows SN ((supt ∪ R)⁺)
proof −
  have SN (supt ∪ R)
    using SN-on-r-imp-SN-on-supt-union-r[OF ctxt, of UNIV]
      and a by force
  then show SN ((supt ∪ R)⁺) by (rule SN-imp-SN-trancl)
qed

lemma SN-supt-rstep-trancl:
  SN (rstep R) ⟹ SN ((supt ∪ rstep R)⁺)
  by (rule SN-supt-r-trancl[OF ctxt-closed-rstep])
```

lemma *SN-imp-SN-arg-gen*:
assumes *ctxt*: *ctxt.closed R*
and *SN*: *SN-on R* {*Fun f ts*} **and** *arg*: $t \in$ *set ts* **shows** *SN-on R* {t}
proof −
from *arg* **have** *Fun f ts* $\trianglerighteq$ t **by** *auto*
with *SN* **show** *?thesis* **by** (*rule ctxt-closed-SN-on-subt*[*OF ctxt*])
qed

lemma *SN-imp-SN-arg*:
SN-on (*rstep R*) {*Fun f ts*} $\Longrightarrow$ $t \in$ *set ts* $\Longrightarrow$ *SN-on* (*rstep R*) {t}
by (*rule SN-imp-SN-arg-gen*[*OF ctxt-closed-rstep*])

lemma *SNinstance-imp-SN*:
assumes *SN-on* (*rstep R*) {$t \cdot \sigma$}
shows *SN-on* (*rstep R*) {t}
proof
fix f
assume *prem*: $f\ 0 \in \{t\}$ $\forall\, i.\ (f\ i, f\ (Suc\ i)) \in$ *rstep R*
let $?g = \lambda\ i.\ (f\ i) \cdot \sigma$
from *prem* **have** $?g\ 0 = t \cdot \sigma \wedge (\forall\ i.\ (?g\ i,\ ?g\ (Suc\ i)) \in$ *rstep R*) **using** *subst-closed-rstep*
by *auto*
then have ¬ *SN-on* (*rstep R*) {$t \cdot \sigma$} **by** *auto*
with *assms* **show** *False* **by** *blast*
qed

lemma *rstep-imp-C-s-r*:
assumes (s,t) $\in$ *rstep R*
shows $\exists\, C\ \sigma\ l\ r.\ (l,r) \in R \wedge s = C\langle l \cdot \sigma\rangle \wedge t = C\langle r \cdot \sigma\rangle$
proof −
from *assms* **obtain** $l\ r\ p\ \sigma$ **where** *step*:(s,t) $\in$ *rstep-r-p-s R* (l,r) $p\ \sigma$
unfolding *rstep-iff-rstep-r-p-s* **by** *best*
let $?C$ = *ctxt-of-pos-term p s*
from *step* **have** $p \in$ *poss s* **and** (l,r) $\in R$ **and** $?C\langle l \cdot \sigma\rangle = s$ **and** $?C\langle r \cdot \sigma\rangle = t$
unfolding *rstep-r-p-s-def Let-def* **by** *auto*
then have (l,r) $\in R \wedge s = ?C\langle l \cdot \sigma\rangle \wedge t = ?C\langle r \cdot \sigma\rangle$ **by** *auto*
then show *?thesis* **by** *force*
qed

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* ($R \cup S$) = *map-funs-trs fg R* ∪ *map-funs-trs fg S*

unfolding *map-funs-trs.simps* **by** *auto*

lemma *rstep-map-funs-term*: **assumes** *R*: $\bigwedge f.\ f \in$ *funs-trs R* $\Longrightarrow$ *h f* = *f* **and** *step*: (*s*,*t*) $\in$ *rstep R*
shows (*map-funs-term h s*, *map-funs-term h t*) $\in$ *rstep R*
proof −
from *step* **obtain** *C l r* σ **where** *s*: *s* = *C*$\langle$*l* $\cdot$ $\sigma\rangle$ **and** *t*: *t* = *C*$\langle$*r* $\cdot$ $\sigma\rangle$ **and** *rule*: (*l*,*r*) $\in$ *R* **by** *auto*
let *?*σ = *map-funs-subst h* σ
let *?h* = *map-funs-term h*
note *funs-defs* = *funs-rule-def*[*abs-def*] *funs-trs-def*
from *rule* **have** *lr*: *funs-term l* $\cup$ *funs-term r* $\subseteq$ *funs-trs R* **unfolding** *funs-defs*
by *auto*
have *hl*: *?h l* = *l*
by (*rule funs-term-map-funs-term-id*[*OF R*], *insert lr*, *auto*)
have *hr*: *?h r* = *r*
by (*rule funs-term-map-funs-term-id*[*OF R*], *insert lr*, *auto*)
show *?thesis* **unfolding** *s t*
unfolding *map-funs-subst-distrib map-funs-term-ctxt-distrib hl hr*
by (*rule rstepI*[*OF rule refl refl*])
qed

lemma *wf-trs-map-funs-trs*[*simp*]: *wf-trs* (*map-funs-trs f R*) = *wf-trs R*
unfolding *wf-trs-def*
proof (*rule iffI*, *intro allI impI*)
fix *l r*
assume $\forall$ *l r*. (*l*, *r*) $\in$ *map-funs-trs f R* $\longrightarrow$ ($\exists$ *f ts*. *l* = *Fun f ts*) $\wedge$ *vars-term r* $\subseteq$ *vars-term l* **and** (*l*, *r*) $\in$ *R*
then show ($\exists$ *f ts*. *l* = *Fun f ts*) $\wedge$ *vars-term r* $\subseteq$ *vars-term l* **by** (*cases l*, *force*+)
next
assume *ass*: $\forall$ *l r*. (*l*, *r*) $\in$ *R* $\longrightarrow$ ($\exists$ *f ts*. *l* = *Fun f ts*) $\wedge$ *vars-term r* $\subseteq$ *vars-term l*
show $\forall$ *l r*. (*l*, *r*) $\in$ *map-funs-trs f R* $\longrightarrow$ ($\exists$ *f ts*. *l* = *Fun f ts*) $\wedge$ *vars-term r* $\subseteq$ *vars-term l*
proof (*intro allI impI*)
fix *l r*
assume (*l*, *r*) $\in$ *map-funs-trs f R*
with *ass*
show ($\exists$ *f ts*. *l* = *Fun f ts*) $\wedge$ *vars-term r* $\subseteq$ *vars-term l*
by (*cases l*, *force*+)
qed
qed

lemma *map-funs-trs-comp*: *map-funs-trs fg* (*map-funs-trs gh R*) = *map-funs-trs* (*fg o gh*) *R*
proof −
have *mr*: *map-funs-rule* (*fg o gh*) = *map-funs-rule fg o map-funs-rule gh*
by (*rule ext*, *auto simp*: *map-funs-term-comp*)
then show *?thesis*

by (*auto simp*: *map-funs-term-comp image-comp mr*)
qed

lemma *map-funs-trs-mono*: **assumes** $R \subseteq R'$ **shows** *map-funs-trs fg R* $\subseteq$ *map-funs-trs fg R'*
using *assms* **by** *auto*

lemma *map-funs-trs-power-mono*:
fixes *R R'* :: ('*f*,'*v*)*trs* **and** *fg* :: '*f* $\Rightarrow$ '*f*
assumes $R \subseteq R'$ **shows** ((*map-funs-trs fg*)^^*n*) *R* $\subseteq$ ((*map-funs-trs fg*)^^*n*) *R'*
using *assms* **by** (*induct n*, *simp*, *auto*)

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

lemma *rstep-imp-map-rstep*:
assumes (*s*, *t*) $\in$ *rstep R*
shows (*map-funs-term fg s*, *map-funs-term fg t*) $\in$ *rstep* (*map-funs-trs fg R*)
using *assms*
proof (*induct*)
case (*IH C* σ *l r*)
then have (*map-funs-term fg l*, *map-funs-term fg r*) $\in$ *map-funs-trs fg R* (**is** (*?l*, *?r*) $\in$ *?R*)
unfolding *map-funs-trs.simps* **by** *force*
then have (*?l*, *?r*) $\in$ *rstep ?R* **..**
then have (*?l* $\cdot$ *map-funs-subst fg* σ, *?r* $\cdot$ *map-funs-subst fg* σ) $\in$ *rstep ?R* **..**
then show *?case* **by** *auto*
qed

lemma *rsteps-imp-map-rsteps*: **assumes** (*s*,*t*) $\in$ (*rstep R*)*
shows (*map-funs-term fg s*, *map-funs-term fg t*) $\in$ (*rstep* (*map-funs-trs fg R*))*
using *assms*
proof (*induct*, *clarify*)
case (*step y z*)
then have (*map-funs-term fg y*, *map-funs-term fg z*) $\in$ *rstep* (*map-funs-trs fg R*)
using *rstep-imp-map-rstep*
by (*auto simp*: *map-funs-trs.simps*)
with *step* **show** *?case* **by** *auto*
qed

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 (*rule ccontr*)
assume $\neg$ *SN-on* (*rstep R*) {*t*}
from *this* **obtain** *f* **where** *cond*: *f 0* = *t* $\wedge$ ($\forall$ *i*. (*f i*, *f* (*Suc i*)) $\in$ *rstep R*)
unfolding *SN-on-def* **by** *auto*
obtain *g* **where** *g*: *g* = (λ *i*. *map-funs-term fg* (*f i*)) **by** *auto*
with *cond* **have** *cond2*: *g 0* = *map-funs-term fg t* $\wedge$ ($\forall$ *i*. (*g i*, *g* (*Suc i*)) $\in$ *rstep* (*map-funs-trs fg R*))

using *rstep-imp-map-rstep*[**where** *fg* = *fg*] **by** *blast*
from *SN*
have ¬ (∃ *g*. (*g* 0 = *map-funs-term fg t* ∧ (∀ *i*. (*g i*, *g* (*Suc i*)) ∈ *rstep* (*map-funs-trs fg R*))))
unfolding *SN-on-def* **by** *auto*
with *cond2* **show** *False* **by** *auto*
qed

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
assume (*s*,*t*) ∈ *rstep R*
then show (*map-funs-term fg s*, *map-funs-term fg t*) ∈ *rstep*(*map-funs-trs fg R*)
by (*rule rstep-imp-map-rstep*)
next
assume (*map-funs-term fg s*, *map-funs-term fg t*) ∈ *rstep* (*map-funs-trs fg R*)
then have (*map-funs-term fg s*, *map-funs-term fg t*) ∈ *ctxt.closure*(*subst.closure*(*map-funs-trs fg R*))
by (*simp add*: *rstep-eq-closure*)
then obtain *C u v* **where** (*u*,*v*) ∈ *subst.closure*(*map-funs-trs fg R*) **and** *C*⟨*u*⟩ = *map-funs-term fg s*
and *C*⟨*v*⟩ = *map-funs-term fg t*
by (*cases rule*: *ctxt.closure.cases*) *force*
then obtain *σ w x* **where** (*w*,*x*) ∈ *map-funs-trs fg R* **and** *w*·*σ* = *u* **and** *x*·*σ* = *v*
by (*cases rule*: *subst.closure.cases*) *force*
then obtain *l r* **where** *w* = *map-funs-term fg l* **and** *x* = *map-funs-term fg r*
and (*l*,*r*) ∈ *R* **by** (*auto simp*: *map-funs-trs.simps*)
have *ps*: *C*⟨*map-funs-term fg l* · *σ*⟩ = *map-funs-term fg s* **and** *pt*: *C*⟨*map-funs-term fg r* · *σ*⟩ = *map-funs-term fg t*
unfolding ‹*w* = *map-funs-term fg l*›[*symmetric*] ‹*x* = *map-funs-term fg r*›[*symmetric*] ‹*w*·*σ* = *u*› ‹*x*·*σ* = *v*›
‹*C*⟨*u*⟩ = *map-funs-term fg s*› ‹*C*⟨*v*⟩ = *map-funs-term fg t*› **by** *auto*
let *?gf* = *the-inv fg*
let *?C* = *map-funs-ctxt ?gf C*
let *?σ* = *map-funs-subst ?gf σ*
have *gffg*: *?gf* ∘ *fg* = *id* **using** *the-inv-f-f*[*OF assms*] **by** (*intro ext*, *auto*)
from *ps* **and** *pt* **have** *s* = *map-funs-term ?gf* (*C*⟨*map-funs-term fg l* · *σ*⟩)
and *t* = *map-funs-term ?gf* (*C*⟨*map-funs-term fg r* · *σ*⟩) **by** (*auto simp*: *map-funs-term-comp gffg*)
then have *s*: *s* = *?C*⟨*map-funs-term ?gf* (*map-funs-term fg l* · *σ*)⟩
and *t*: *t* = *?C*⟨*map-funs-term ?gf* (*map-funs-term fg r* · *σ*)⟩ **using** *map-funs-term-ctxt-distrib* **by** *auto*
from *s* **have** *s* = *?C*⟨*l*·*?σ*⟩ **by** (*simp add*: *map-funs-term-comp gffg*)
from *t* **have** *t* = *?C*⟨*r*·*?σ*⟩ **by** (*simp add*: *map-funs-term-comp gffg*)
from ‹(*l*, *r*) ∈ *R*› **have** (*l*·*?σ*, *r*·*?σ*) ∈ *subst.closure R* **by** *blast*
then have (*?C*⟨*l*·*?σ*⟩,*?C*⟨*r*·*?σ*⟩) ∈ *ctxt.closure*(*subst.closure R*) **by** *blast*

then show $(s,t) \in rstep\ R$ **unfolding** ‹$s = ?C\langle l \cdot ?\sigma\rangle$› ‹$t = ?C\langle r \cdot ?\sigma\rangle$› **by** (*simp add*: *rstep-eq-closure*)
qed

lemma *rstep-map-funs-trs-power-mono*:
fixes $R\ R' :: ('f,'v)trs$ **and** $fg :: 'f \Rightarrow 'f$
assumes *subset*: $R \subseteq R'$ **shows** *rstep* $(((map\text{-}funs\text{-}trs\ fg)\ ^{\frown}n)\ R) \subseteq rstep\ (((map\text{-}funs\text{-}trs\ fg)\ ^{\frown}n)\ R')$
by (*rule rstep-mono*, *rule map-funs-trs-power-mono*, *rule subset*)

lemma *subsetI3*: $(\bigwedge x\ y\ z.\ (x,\ y,\ z) \in A \Longrightarrow (x,\ y,\ z) \in B) \Longrightarrow A \subseteq B$ **by** *auto*

lemma *aux*: $(\bigcup a{\in}P.\ \{(x,y,z).\ x = fst\ a \wedge y = snd\ a \wedge Q\ a\ z\}) = \{(x,y,z).\ (x,y) \in P \wedge Q\ (x,y)\ z\}$ (**is** *?P* = *?Q*)
proof
show *?P* $\subseteq$ *?Q*
proof
fix *x* **assume** $x \in ?P$
then obtain *a* **where** $a \in P$ **and** $x \in \{(x,y,z).\ x = fst\ a \wedge y = snd\ a \wedge Q\ a\ z\}$ **by** *auto*
then obtain *b* **where** $Q\ (fst\ x,\ fst\ (snd\ x))\ (snd\ (snd\ x))$ **and** $(fst\ x,\ fst\ (snd\ x)) = a$ **and** $snd\ (snd\ x) = b$ **by** *force*
from ‹$a \in P$› **have** $(fst\ a,snd\ a) \in P$ **unfolding** *split-def* **by** *simp*
from ‹$Q\ (fst\ x,\ fst\ (snd\ x))\ (snd\ (snd\ x))$› **have** $Q\ a\ b$ **unfolding** ‹$(fst\ x,\ fst\ (snd\ x)) = a$› ‹$snd\ (snd\ x) = b$› **.**
from ‹$(fst\ a,snd\ a) \in P$› **and** ‹$Q\ a\ b$› **show** $x \in ?Q$ **unfolding** *split-def* ‹$(fst\ x,\ fst\ (snd\ x)) = a$›[*symmetric*] ‹$snd\ (snd\ x) = b$›[*symmetric*] **by** *simp*
qed
next
show *?Q* $\subseteq$ *?P*
proof (*rule subsetI3*)
fix *x y z* **assume** $(x,y,z) \in ?Q$
then have $(x,y) \in P$ **and** $Q\ (x,y)\ z$ **by** *auto*
then have $x = fst(x,y) \wedge y = snd(x,y) \wedge Q\ (x,y)\ z$ **by** *auto*
then have $(x,y,z) \in \{(x,y,z).\ x = fst(x,y) \wedge y = snd(x,y) \wedge Q\ (x,y)\ z\}$ **by** *auto*
then have $\exists\, p{\in}P.\ (x,y,z) \in \{(x,y,z).\ x = fst\ p \wedge y = snd\ p \wedge Q\ p\ z\}$ **using** ‹$(x,y) \in P$› **by** *blast*
then show $(x,y,z) \in ?P$ **unfolding** *UN-iff*[*symmetric*] **by** *simp*
qed
qed

lemma *finite-imp-finite-DP-on'*:
assumes *finite R*
shows *finite* $\{(l,\ r,\ u).$
$\exists\, h\ us.\ u = Fun\ h\ us \wedge (l,\ r) \in R \wedge r \trianglerighteq u \wedge (h,\ length\ us) \in F \wedge \neg\ (l \triangleright u)\}$
proof −
have $\bigwedge l\ r.\ (l,\ r) \in R \Longrightarrow finite\ \{u.\ r \trianglerighteq u\}$
proof −

fix l r
assume $(l, r) \in R$
show *finite* $\{u.\ r \trianglerighteq u\}$ **by** (*rule finite-subterms*)
qed
with ‹*finite R*› **have** *finite*($\bigcup (l, r) \in R.\ \{u.\ r \trianglerighteq u\}$) **by** *auto*
have *finite*($\bigcup lr \in R.\ \{(l, r, u).\ l = fst\ lr \wedge r = snd\ lr \wedge snd\ lr \trianglerighteq u\}$)
proof (*rule finite-UN-I*)
show *finite R* **by** (*rule* ‹*finite R*›)
next
fix *lr*
assume $lr \in R$
have *finite* $\{u.\ snd\ lr \trianglerighteq u\}$ **by** (*rule finite-subterms*)
then show *finite* $\{(l,r,u).\ l = fst\ lr \wedge r = snd\ lr \wedge snd\ lr \trianglerighteq u\}$ **by** *auto*
qed
then have *finite* $\{(l,r,u).\ (l,r) \in R \wedge r \trianglerighteq u\}$ **unfolding** *aux* **by** *auto*
have $\{(l,r,u).\ (l,r) \in R \wedge r \trianglerighteq u\} \supseteq \{(l,r,u).\ (\exists h\ us.\ u = Fun\ h\ us \wedge (l,r) \in R \wedge r \trianglerighteq u \wedge (h, length\ us) \in F \wedge \neg(l \rhd u))\}$ **by** *auto*
with ‹*finite* $\{(l,r,u).\ (l,r) \in R \wedge r \trianglerighteq u\}$› **show** *?thesis* **using** *finite-subset* **by** *fast*
qed

lemma *card-image-le'*:
assumes *finite S*
shows *card* $(\bigcup y \in S.\{x.\ x = f\ y\}) \leq card\ S$
proof −
have *A*:$(\bigcup y \in S.\ \{x.\ x = f\ y\}) = f\ `\ S$ **by** *auto*
from *assms* **show** *?thesis* **unfolding** *A* **using** *card-image-le* **by** *auto*
qed

lemma *subteq-of-map-imp-map*: *map-funs-term g s* $\trianglerighteq t \Longrightarrow \exists u.\ t = map\text{-}funs\text{-}term\ g\ u$
proof (*induct s arbitrary*: *t*)
case (*Var x*)
then have *map-funs-term g* (*Var x*) $\rhd t \vee map\text{-}funs\text{-}term\ g\ (Var\ x) = t$ **by** *auto*
then show *?case*
proof
assume *map-funs-term g* (*Var x*) $\rhd t$ **then show** *?thesis* **by** (*cases rule*: *supt.cases*) *auto*
next
assume *map-funs-term g* (*Var x*) $= t$ **then show** *?thesis* **by** *best*
qed
next
case (*Fun f ss*)
then have *map-funs-term g* (*Fun f ss*) $\rhd t \vee map\text{-}funs\text{-}term\ g\ (Fun\ f\ ss) = t$ **by** *auto*
then show *?case*
proof
assume *map-funs-term g* (*Fun f ss*) $\rhd t$
then show *?case* **using** *Fun* **by** (*cases rule*: *supt.cases*) (*auto simp*: *supt-supteq-conv*)

```
  next
    assume map-funs-term g (Fun f ss) = t then show ?thesis by best
  qed
qed

lemma map-funs-term-inj:
  assumes inj (fg :: ('f ⇒ 'g))
  shows inj (map-funs-term fg)
proof −
  {
    fix s t :: ('f,'v)term
    assume map-funs-term fg s = map-funs-term fg t
    then have s = t
    proof (induct s arbitrary: t)
      case (Var x) with assms show ?case by (induct t) auto
    next
      case (Fun f ss) then show ?case
      proof (induct t)
        case (Var y) then show ?case by auto
      next
        case (Fun g ts)
        then have A: map (map-funs-term fg) ss = map (map-funs-term fg) ts by
simp
        then have len-eq:length ss = length ts by (rule map-eq-imp-length-eq)
        from A have !!i. i < length ss ⟹ (map (map-funs-term fg) ss)!i = (map
(map-funs-term fg) ts)!i by auto
          with len-eq have eq: !!i. i < length ss ⟹ map-funs-term fg  (ss!i) =
map-funs-term fg (ts!i) using nth-map by auto
        have in-set: !!i. i < length ss ⟹ (ss!i) ∈ set ss by auto
        from Fun have ∀ i < length ss. (ss!i) = (ts!i) using in-set eq by auto
        with len-eq have ss = ts using nth-equalityI[where xs = ss and ys = ts]
by simp
        have f = g using Fun ‹inj fg› unfolding inj-on-def by auto
        with ‹ss = ts› show ?case by simp
      qed
    qed
  }
  then show ?thesis unfolding inj-on-def by auto
qed

lemma rsteps-closed-ctxt:
  assumes (s, t) ∈ (rstep R)*
  shows (C⟨s⟩, C⟨t⟩) ∈ (rstep R)*
proof −
  from assms obtain n where (s,t) ∈ (rstep R)^^n
    using rtrancl-is-UN-relpow by auto
  then show ?thesis
  proof (induct n arbitrary: s)
    case 0 then show ?case by auto
```

next
case (*Suc n*)
from *relpow-Suc-D2*[*OF* ‹(*s*,*t*) ∈ (*rstep R*)^^*Suc n*›] **obtain** *u*
where (*s*,*u*) ∈ (*rstep R*) **and** (*u*,*t*) ∈ (*rstep R*)^^*n* **by** *auto*
from ‹(*s*,*u*) ∈ (*rstep R*)› **have** (*C*⟨*s*⟩,*C*⟨*u*⟩) ∈ (*rstep R*) **..**
from *Suc* **and** ‹(*u*,*t*) ∈ (*rstep R*)^^*n*› **have** (*C*⟨*u*⟩,*C*⟨*t*⟩) ∈ (*rstep R*)* **by** *simp*
with ‹(*C*⟨*s*⟩,*C*⟨*u*⟩) ∈ (*rstep R*)› **show** *?case* **by** *auto*
qed
qed

lemma *one-imp-ctxt-closed*: **assumes** *one*: ⋀ *f bef s t aft*. (*s*,*t*) ∈ *r* ⟹ (*Fun f* (*bef* @ *s* # *aft*), *Fun f* (*bef* @ *t* # *aft*)) ∈ *r*
shows *ctxt.closed r*
proof
fix *s t C*
assume *st*: (*s*,*t*) ∈ *r*
show (*C*⟨*s*⟩, *C*⟨*t*⟩) ∈ *r*
proof (*induct C*)
case (*More f bef C aft*)
from *one*[*OF More*] **show** *?case* **by** *auto*
qed (*insert st*, *auto*)
qed

lemma *ctxt-closed-nrrstep* [*intro*]: *ctxt.closed* (*nrrstep R*)
proof (*rule one-imp-ctxt-closed*)
fix *f bef s t aft*
assume (*s*,*t*) ∈ *nrrstep R*
from *this*[*unfolded nrrstep-def'*] **obtain** *l r C* σ
where *lr*: (*l*,*r*) ∈ *R* **and** *C*: *C* ≠ □
and *s*: *s* = *C*⟨*l*·σ⟩ **and** *t*: *t* = *C*⟨*r* · σ⟩ **by** *auto*
show (*Fun f* (*bef* @ *s* # *aft*), *Fun f* (*bef* @ *t* # *aft*)) ∈ *nrrstep R*
proof (*rule nrrstepI*[*OF lr*])
show *More f bef C aft* ≠ □ **by** *simp*
qed (*insert s t*, *auto*)
qed

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*
unfolding *all-ctxt-closed-def* **by** *auto*

lemma *all-ctxt-closed-sig-reflE*: **assumes** *all*: *all-ctxt-closed F r*
shows *funas-term t* $\subseteq$ *F* $\Longrightarrow$ (*t*,*t*) $\in$ *r*
proof (*induct t*)
case (*Var x*)
from *all*[*unfolded all-ctxt-closed-def*] **show** *?case* **by** *auto*
next
case (*Fun f ts*)
show *?case*
by (*rule all-ctxt-closedD*[*OF all - - Fun*(*1*)], *insert Fun*(*2*), *force*+)
qed

lemma *all-ctxt-closed-reflE*: **assumes** *all*: *all-ctxt-closed UNIV r*
shows (*t*,*t*) $\in$ *r*
by (*rule all-ctxt-closed-sig-reflE*[*OF all*], *auto*)

lemma *all-ctxt-closed-relcomp*: **assumes** *all-ctxt-closed UNIV R all-ctxt-closed UNIV S*
shows *all-ctxt-closed UNIV* (*R O S*)
unfolding *all-ctxt-closed-def*
proof (*intro allI impI conjI*)
show (*Var x*, *Var x*) $\in$ *R O S* **for** *x* **using** *assms* **unfolding** *all-ctxt-closed-def* **by** *auto*
fix *f ts ss*
assume *len*: *length ts* = *length ss*
and *steps*: $\forall$ *i*<*length ts*. (*ts* ! *i*, *ss* ! *i*) $\in$ *R O S*
hence $\forall$ *i*. $\exists$ *us*. *i* < *length ts* $\longrightarrow$ (*ts* ! *i*, *us*) $\in$ *R* $\wedge$ (*us*, *ss* ! *i*) $\in$ *S* **by** *blast*
from *choice*[*OF this*] **obtain** *us* **where** *steps*: $\bigwedge$ *i*. *i*<*length ts* $\Longrightarrow$ (*ts* ! *i*, *us i*) $\in$ *R* $\wedge$ (*us i*, *ss* ! *i*) $\in$ *S*
by *blast*
let *?us* = *map us* [*0*..<*length ss*]
from *all-ctxt-closedD*[*OF assms*(*2*)] *steps len* **have** *us*: (*Fun f ?us*, *Fun f ss*) $\in$ *S* **by** *auto*
from *all-ctxt-closedD*[*OF assms*(*1*)] *steps len* **have** *tu*: (*Fun f ts*, *Fun f ?us*) $\in$ *R* **by** *force*
from *tu us*
show (*Fun f ts*, *Fun f ss*) $\in$ *R O S* **by** *auto*
qed

lemma *all-ctxt-closed-relpow*:
assumes *acc*:*all-ctxt-closed UNIV Q*
shows *all-ctxt-closed UNIV* (*Q* $\frown$ *n*)
proof (*induct n*)
case *0*
thus *?case* **by** (*auto simp*: *all-ctxt-closed-def nth-equalityI*)
next
case (*Suc n*)
from *all-ctxt-closed-relcomp*[*OF this acc*]
show *?case* **by** *simp*

qed

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* $\subseteq$ *F*
and *steps*: $\bigwedge x.\ x \in$ *vars-term t* $\Longrightarrow (\sigma\ x, \tau\ x) \in r$
and *sig-subst*: $\bigwedge x.\ x \in$ *vars-term t* $\Longrightarrow$ *funas-term* $(\sigma\ x) \cup$ *funas-term* $(\tau\ x) \subseteq F$
shows $(t \cdot \sigma, t \cdot \tau) \in r$
using *sig steps sig-subst*
proof (*induct t*)
case (*Var x*)
then show *?case* **by** *auto*
next
case (*Fun f ts*)
{
fix *t*
assume *t*: $t \in$ *set ts*
with *Fun*(2−3) **have** *funas-term t* $\subseteq F \bigwedge x.\ x \in$ *vars-term t* $\Longrightarrow (\sigma\ x, \tau\ x) \in r$ **by** *auto*
from *Fun*(1)[*OF t this Fun*(4)] *t* **have** *step*: $(t \cdot \sigma, t \cdot \tau) \in r$ **by** *auto*
from *Fun*(4) *t* **have** $\bigwedge x.\ x \in$ *vars-term t* $\Longrightarrow$ *funas-term* $(\sigma\ x) \cup$ *funas-term* $(\tau\ x) \subseteq F$ **by** *auto*
with ‹*funas-term t* $\subseteq F$› **have** *funas-term* $(t \cdot \sigma) \cup$ *funas-term* $(t \cdot \tau) \subseteq F$ **unfolding** *funas-term-subst* **by** *auto*
note *step this*
}
then have *steps*: $\bigwedge i.\ i <$ *length ts* $\Longrightarrow (ts\ !\ i \cdot \sigma, ts\ !\ i \cdot \tau) \in r \wedge$ *funas-term* $(ts\ !\ i \cdot \sigma) \cup$ *funas-term* $(ts\ !\ i \cdot \tau) \subseteq F$ **unfolding** *set-conv-nth* **by** *auto*
with *all-ctxt-closedD*[*OF all*, *of f map* $(\lambda\ t.\ t \cdot \tau)$ *ts map* $(\lambda\ t.\ t \cdot \sigma)$ *ts*] *Fun*(2)
show *?case* **by** *auto*
qed

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-term t* $\Longrightarrow (\sigma\ x, \tau\ x) \in r$
shows $(t \cdot \sigma, t \cdot \tau) \in r$
by (*rule all-ctxt-closed-subst-step-sig*[*OF all - steps*], *auto*)

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(*induct C*)
case *Hole*

```
  from step show ?case by auto
next
  case (More f bef C aft)
  let ?n = length bef
  let ?m = Suc (?n + length aft)
  show ?case unfolding intp-actxt.simps
  proof (rule all-ctxt-closedD[OF all])
    fix i
    let ?t = λ s. (bef @ C ⟨ s ⟩ # aft) ! i
    assume i < length (bef @ C⟨s⟩ # aft)
    then have i: i < ?m by auto
   then have mem: ⋀ s. ?t s ∈ set (bef @ C ⟨ s ⟩ # aft) unfolding set-conv-nth
by auto
    from mem[of s] More Fs show funas-term (?t s) ⊆ F by auto
    from mem[of t] More Ft show funas-term (?t t) ⊆ F by auto
    from More have step: (C ⟨ s ⟩, C ⟨ t ⟩) ∈ R by auto
    {
      fix s
      assume s ∈ set bef ∪ set aft
      with More have funas-term s ⊆ F by auto
      from all-ctxt-closed-sig-reflE[OF all this] have (s,s) ∈ R by auto
    } note steps = this
    show (?t s, ?t t) ∈ R
    proof (cases i = ?n)
      case True
      then show ?thesis using step by auto
    next
      case False
      show ?thesis
      proof (cases i < ?n)
        case True
        then show ?thesis unfolding append-Cons-nth-left[OF True] using steps
by auto
      next
        case False
        with ‹i ≠ ?n› i have ∃ k. k < length aft ∧ i = Suc ?n + k by presburger
        then obtain k where k: k < length aft and i: i = Suc ?n + k by auto
        from k show ?thesis using steps unfolding i by (auto simp: nth-append)
      qed
    qed
  qed (insert More, auto)
qed

lemma trans-ctxt-sig-imp-all-ctxt-closed: assumes tran: trans r
  and refl: ⋀ t. funas-term t ⊆ F ⟹ (t,t) ∈ r
  and ctxt: ⋀ C s t. funas-ctxt C ⊆ F ⟹ funas-term s ⊆ F ⟹ funas-term t ⊆
F ⟹ (s,t) ∈ r ⟹ (C ⟨ s ⟩, C ⟨ t ⟩) ∈ r
shows all-ctxt-closed F r
  unfolding all-ctxt-closed-def
```

proof (*intro conjI*, *intro allI impI*)
fix *f ts ss*
assume *f*: (*f*,*length ss*) ∈ *F* **and**
l: *length ts* = *length ss* **and**
steps: ∀ *i* < *length ts*. (*ts* ! *i*, *ss* ! *i*) ∈ *r* **and**
sig: ∀ *i* < *length ts*. *funas-term* (*ts* ! *i*) ∪ *funas-term* (*ss* ! *i*) ⊆ *F*
from *sig* **have** *sig-ts*: ⋀ *t*. *t* ∈ *set ts* ⟹ *funas-term t* ⊆ *F* **unfolding** *set-conv-nth*
by *auto*
let *?p* = λ *ss*. (*Fun f ts*, *Fun f ss*) ∈ *r* ∧ *funas-term* (*Fun f ss*) ⊆ *F*
let *?r* = λ *xsi ysi*. (*xsi*, *ysi*) ∈ *r* ∧ *funas-term ysi* ⊆ *F*
have *init*: *?p ts* **by** (*rule conjI*[*OF refl*], *insert f sig-ts l*, *auto*)
have *?p ss*
proof (*rule parallel-list-update*[**where** *p* = *?p* **and** *r* = *?r*, *OF - HOL.refl init*
l[*symmetric*]])
fix *xs i y*
assume *len*: *length xs* = *length ts*
and *i*: *i* < *length ts*
and *r*: *?r* (*xs* ! *i*) *y*
and *p*: *?p xs*
let *?C* = *More f* (*take i xs*) *Hole* (*drop* (*Suc i*) *xs*)
have *id1*: *Fun f xs* = *?C* ⟨ *xs* ! *i*⟩ **using** *id-take-nth-drop*[*OF i*[*folded len*]] **by**
simp
have *id2*: *Fun f* (*xs*[*i* := *y*]) = *?C* ⟨ *y* ⟩ **using** *upd-conv-take-nth-drop*[*OF*
i[*folded len*]] **by** *simp*
from *p*[*unfolded id1*] **have** *C*: *funas-ctxt ?C* ⊆ *F* **and** *xi*: *funas-term* (*xs* ! *i*)
⊆ *F* **by** *auto*
from *r* **have** *funas-term y* ⊆ *F* (*xs* ! *i*, *y*) ∈ *r* **by** *auto*
with *ctxt*[*OF C xi this*] *C* **have** *r*: (*Fun f xs*, *Fun f* (*xs*[*i* := *y*])) ∈ *r*
and *f*: *funas-term* (*Fun f* (*xs*[*i* := *y*])) ⊆ *F* **unfolding** *id1 id2* **by** *auto*
from *p r tran* **have** (*Fun f ts*, *Fun f* (*xs*[*i* := *y*])) ∈ *r* **unfolding** *trans-def* **by**
auto
with *f*
show *?p* (*xs*[*i* := *y*]) **by** *auto*
qed (*insert sig steps*, *auto*)
then show (*Fun f ts*, *Fun f ss*) ∈ *r* **..**
qed (*insert refl*, *auto*)

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*
by (*rule trans-ctxt-sig-imp-all-ctxt-closed*[*OF tran - ctxt.closedD*[*OF ctxt*]], *insert*
refl[*unfolded refl-on-def*], *auto*)

lemma *all-ctxt-closed-rsteps*[*intro*]: *all-ctxt-closed F* ((*rstep r*)*)
by (*blast intro*: *trans-ctxt-imp-all-ctxt-closed trans-rtrancl refl-rtrancl*)

lemma *subst-rsteps-imp-rsteps*:
fixes σ :: ('*f*, '*v*) *subst*

assumes $\bigwedge x.\ x \in vars\text{-}term\ t \Longrightarrow (\sigma\ x,\ \tau\ x) \in (rstep\ R)^*$
shows $(t \cdot \sigma,\ t \cdot \tau) \in (rstep\ R)^*$
by (*rule all-ctxt-closed-subst-step*)
(*insert assms*, *auto*)

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 −
from *ctxt* **have** *ctxt-rt*: *ctxt.closed* (R^*) **by** *blast*
from *ctxt* **have** *ctxt-t*: *ctxt.closed* (R^+) **by** *blast*
from *id-take-nth-drop*[*OF i*] **have** *ts*: *ts* = *take i ts* @ *ts* ! *i* # *drop* (*Suc i*) *ts* (**is** - = *?ts*) **by** *auto*
from *id-take-nth-drop*[*OF i*[*simplified len*]] **have** *ss*: *ss* = *take i ss* @ *ss* ! *i* # *drop* (*Suc i*) *ss* (**is** - = *?ss*) **by** *auto*
let *?mid* = *take i ss* @ *ts* ! *i* # *drop* (*Suc i*) *ss*
from *trans-ctxt-imp-all-ctxt-closed*[*OF trans-rtrancl refl-rtrancl ctxt-rt*] **have** *all*: *all-ctxt-closed UNIV* (R^*) **.**
from *ctxt-closed-one*[*OF ctxt-t step*] **have** $(Fun\ f\ ?mid,\ Fun\ f\ ?ss) \in R^+$ **.**
then have *part1*: $(Fun\ f\ ?mid,\ Fun\ f\ ss) \in R^+$ **unfolding** *ss*[*symmetric*] **.**
from *ts* **have** *lents*: *length ts* = *length ?ts* **by** *simp*
have $(Fun\ f\ ts,\ Fun\ f\ ?mid) \in R^*$
proof (*rule all-ctxt-closedD*[*OF all*])
fix *j*
assume *jts*: $j < length\ ts$
from *i len* **have** *i*: $i < length\ ss$ **by** *auto*
show $(ts\ !\ j,\ ?mid\ !\ j) \in R^*$
proof (*cases* $j < i$)
case *True*
with *i* **have** *j*: $j < length\ ss$ **by** *auto*
with *True* **have** *id*: *?mid* ! *j* = *ss* ! *j* **by** (*simp add*: *nth-append*)
from *steps len j* **have** $(ts\ !\ j,\ ss\ !\ j) \in R^*$ **by** *auto*
then show *?thesis* **using** *id* **by** *simp*
next
case *False*
show *?thesis*
proof (*cases* $j = i$)
case *True*
then have *?mid* ! *j* = *ts* ! *j* **using** *i* **by** (*simp add*: *nth-append*)
then show *?thesis* **by** *simp*
next
case *False*
from *i* **have** *min*: *min* (*length ss*) *i* = *i* **by** *simp*
from *False* ‹¬ *j* < *i*› **have** *j* > *i* **by** *arith*
then obtain *k* **where** *k*: *j* − *i* = *Suc k* **by** (*cases j* − *i*, *auto*)

```
      then have j: j = Suc (i+k) by auto
      with jts len have ss: Suc i + k ≤ length ss and jlen: j < length ts by auto
      then have ?mid ! j = ss ! j using j i by (simp add: nth-append min ‹¬ j
< i› nth-drop[OF ss])
      with steps jlen show ?thesis by auto
    qed
  qed
 qed (insert lents[symmetric] len, auto)
 with part1 show ?thesis by auto
qed

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 −
 {
  fix u
  assume u ∈ set (ctxt-to-term-list C)
  from ctxt-to-term-list-supt[OF this, of t] have C⟨t⟩ ⊵ u
   by (rule supt-imp-supteq)
  from ctxt-closed-SN-on-subt[OF ctxt, OF SN this]
  have SN-on r {u} by auto
 }
 then show ?thesis
  unfolding SN-on-def by auto
qed

lemma rstep-subset: ctxt.closed R' ⟹ subst.closed R' ⟹ R ⊆ R' ⟹ rstep R ⊆
R' by fast

lemma trancl-rstep-ctxt:
 (s,t) ∈ (rstep R)+ ⟹ (C⟨s⟩, C⟨t⟩) ∈ (rstep R)+
 by (rule ctxt.closedD, blast)

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 −
 let ?tss = λi. take i ts @ drop i ss
 {
  fix bef :: ('f,'v)term list and s t and  aft :: ('f,'v)term list
  assume (s,t) ∈ (r (length bef))* and len: length ts = Suc (length bef + length
aft)
  then have (Fun f (bef @ s # aft), Fun f (bef @ t # aft)) ∈ R*
```

```
    proof (induct)
      case (step t u)
      from step(3)[OF len] ctxt[OF step(2) len] show ?case by auto
    qed simp
  }
  note one = this
  have a:∀ i ≤ length ts. (Fun f ss,Fun f (?tss i)) ∈ R*
  proof (intro allI impI)
    fix i assume i ≤ length ts then show (Fun f ss,Fun f (?tss i)) ∈ R*
    proof (induct i)
      case 0
      then show ?case by simp
    next
      case (Suc i)
      then have IH: (Fun f ss,Fun f (?tss i)) ∈ R*
        and i: i < length ts by auto
      have si: ?tss (Suc i) = take i ts @ ts ! i # drop (Suc i) ss
        unfolding take-Suc-conv-app-nth[OF i] by simp
      have ii: ?tss i = take i ts @ ss ! i # drop (Suc i) ss
        unfolding Cons-nth-drop-Suc[OF i[unfolded len[symmetric]]] ..
      from i have i': length (take i ts) < length ts  and len': length (take i ts) = i
by auto
      from len i have len'': length ts = Suc (length (take i ts) + length (drop (Suc
i) ss)) by simp
      from one[OF args[OF i'] len''] IH
      show ?case unfolding si ii len' by auto
    qed
  qed
  from this[THEN spec, THEN mp[OF - le-refl]]
  show ?thesis using len by auto
qed

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 (rule args-steps-imp-steps-gen[OF - len])
  fix i
  assume i < length ts then show (ss ! i, ts ! i) ∈ R* using args len by auto
qed (insert ctxt-closed-one[OF ctxt], auto)

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
  using assms(4 −)
proof (induction t arbitrary: p)
  case (Var x)
  then show ?case using refl by (simp add: refl-on-def)
next
  case (Fun f ts)
  then obtain i q where [simp]: p = i # q and i: i < length ts
    and q: q ∈ poss (ts ! i) and [simp]: ts ! i |- q = Var x by (cases p) auto
  let ?C = ctxt-of-pos-term q (ts ! i · σ)
  let ?ts = map (λt. t · τ) ts
  let ?ss = take i (map (λt. t · σ) ts) @ ?C⟨τ x⟩ # drop (Suc i) (map (λt. t · σ)
ts)
  have ∀ j<length ts. (?ss ! j, ?ts ! j) ∈ r
  proof (intro allI impI)
    fix j
    assume j: j < length ts
    moreover
    { assume [simp]: j = i
      have ?ss ! j = ?C⟨τ x⟩ using i by (simp add: nth-append-take)
      with Fun.IH [of ts ! i q]
      have (?ss ! j, ?ts ! j) ∈ r using q and i by simp }
    moreover
    { assume j < i
      with i have ?ss ! j = ts ! j · σ
        and ?ts ! j = ts ! j · τ by (simp-all add: nth-append-take-is-nth-conv)
     then have (?ss ! j, ?ts ! j) ∈ r by (auto simp: ∗ all-ctxt-closed-subst-step [OF
acc]) }
    moreover
    { assume j > i
      with i and j have ?ss ! j = ts ! j · σ
        and ?ts ! j = ts ! j · τ by (simp-all add: nth-append-drop-is-nth-conv)
     then have (?ss ! j, ?ts ! j) ∈ r by (auto simp: ∗ all-ctxt-closed-subst-step [OF
acc]) }
    ultimately show (?ss ! j, ?ts ! j) ∈ r by arith
  qed
  moreover have i < length ts by fact
  ultimately show ?case
    by (auto intro: all-ctxt-closedD [OF acc])
qed

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)∗
  by (intro replace-at-subst-steps[OF - - assms], auto simp: refl-on-def)
```

lemma *substs-rsteps*:
assumes $\bigwedge x.\ (\sigma\ x,\ \tau\ x) \in (rstep\ R)^*$
shows $(t \cdot \sigma,\ t \cdot \tau) \in (rstep\ R)^*$
proof (*induct t*)
case (*Var y*)
show *?case* **using** *assms* **by** *simp-all*
next
case (*Fun f ts*)
then have $\forall i{<}length\ (map\ (\lambda t.\ t \cdot \sigma)\ ts).$
$(map\ (\lambda t.\ t \cdot \sigma\ )\ ts\ !\ i,\ map\ (\lambda t.\ t \cdot \tau)\ ts\ !\ i) \in (rstep\ R)^*$ **by** *auto*
from *args-rsteps-imp-rsteps* [*OF - this*] **show** *?case* **by** *simp*
qed

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 −
from ‹(*?s*,*?t*) ∈ *nrrstep R*›
obtain *l r j ps σ* **where** *A*: *let C* = *ctxt-of-pos-term* (*j*#*ps*) *?s in* (*j*#*ps*) ∈ *poss ?s* ∧ (*l*,*r*) ∈ *R* ∧ *C*⟨*l*·*σ*⟩ = *?s* ∧ *C*⟨*r*·*σ*⟩ = *?t* **unfolding** *nrrstep-def rstep-r-p-s-def* **by** *force*
let *?C* = *ctxt-of-pos-term* (*j*#*ps*) *?s*
from *A* **have** (*j*#*ps*) ∈ *poss ?s* **and** (*l*,*r*) ∈ *R* **and** *?C*⟨*l*·*σ*⟩ = *?s* **and** *?C*⟨*r*·*σ*⟩ = *?t* **using** *Let-def* **by** *auto*
have *C*: *?C* = *More f* (*take j ss*) (*ctxt-of-pos-term ps* (*ss*!*j*)) (*drop* (*Suc j*) *ss*) (**is** *?C* = *More f ?ss1 ?D ?ss2*) **by** *auto*
from ‹*?C*⟨*l*·*σ*⟩ = *?s*› **have** *?D*⟨*l*·*σ*⟩ = (*ss*!*j*) **by** (*auto simp*: *take-drop-imp-nth*)
from ‹(*l*,*r*) ∈ *R*› **have** (*l*·*σ*,*r*·*σ*) ∈ (*subst.closure R*) **by** *auto*
then have (*?D*⟨*l*·*σ*⟩,*?D*⟨*r*·*σ*⟩) ∈ (*ctxt.closure*(*subst.closure R*)) **and** (*?C*⟨*l*·*σ*⟩,*?C*⟨*r*·*σ*⟩) ∈ (*ctxt.closure*(*subst.closure R*)) **by** (*auto simp only*: *ctxt.closure.intros*)
then have *D-rstep*: (*?D*⟨*l*·*σ*⟩,*?D*⟨*r*·*σ*⟩) ∈ *rstep R* **and** (*?C*⟨*l*·*σ*⟩,*?C*⟨*r*·*σ*⟩) ∈ *rstep R*
by (*auto simp*: *rstep-eq-closure*)
from ‹*?C*⟨*r*·*σ*⟩ = *?t*› **and** *C* **have** *?t* = *Fun f* (*take j ss* @ *?D*⟨*r*·*σ*⟩ # *drop* (*Suc j*) *ss*) **by** *auto*
then have *ts*: *ts* = (*take j ss* @ *?D*⟨*r*·*σ*⟩ # *drop* (*Suc j*) *ss*) **by** *auto*
from ‹*j*#*ps* ∈ *poss ?s*› **have** *r0*: *j* < *length ss* **by** *simp*
then have (*take j ss* @ *?D*⟨*r*·*σ*⟩ # *drop* (*Suc j*) *ss*) ! *j* = *?D*⟨*r*·*σ*⟩ **by** (*auto simp*: *nth-append-take*)
with *ts* **have** *ts*!*j* = *?D*⟨*r*·*σ*⟩ **by** *auto*
let *?C'* = *More f* (*take j ss*) □ (*drop* (*Suc j*) *ss*)
from *D-rstep* **have** *r1*: (*ss*!*j*,*ts*!*j*) ∈ *rstep R* **unfolding** ‹*ts*!*j* = *?D*⟨*r*·*σ*⟩› ‹*?D*⟨*l*·*σ*⟩ = *ss*!*j*› **by** *simp*
have *?s* = *?C*⟨*l*·*σ*⟩ **unfolding** ‹*?C*⟨*l*·*σ*⟩ = *?s*› **by** *simp*
also have ... = *?C'*⟨*?D*⟨*l*·*σ*⟩⟩ **unfolding** *C* **by** *simp*
finally have *r2*:*?C'*⟨*ss*!*j*⟩ = *?s* **unfolding** ‹*?D*⟨*l*·*σ*⟩ = *ss*!*j*› **by** *simp*

have *?t = ?C⟨r·σ⟩* **unfolding** ‹*?C⟨r·σ⟩ = ?t*› **by** *simp*
also have *... = ?C'⟨?D⟨r·σ⟩⟩* **unfolding** *C* **by** *simp*
finally have *r3*:*?C'⟨ts!j⟩ = ?t* **unfolding** ‹*ts!j = ?D⟨r·σ⟩*› **by** *simp*
from *r0 r1 r2 r3* **show** *?thesis* **by** *best*
qed

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 (*rule ext*)
fix *ab* :: *'c ∗ 'b*
obtain *a b* **where** *ab* = (*a*,*b*) **by** *force*
have ((λ(*x*,*y*). (*x*,*f y*))∘(λ(*x*,*y*). (*x*,*g y*))) (*a*,*b*) = (λ(*x*,*y*). (*x*,(*f*∘*g*) *y*)) (*a*,*b*) **by** *simp*
then show *?f ab = ?g ab* **unfolding** ‹*ab* = (*a*,*b*)› **by** *simp*
qed

lemma *restrict-singleton*:
assumes *x* ∈ *subst-domain σ* **shows** ∃ *t*. *σ* |*s* {*x*} = (λ*y*. *if y = x then t else Var y*)
proof −
have *σ* |*s* {*x*} = (λ*y*. *if y = x then σ y else Var y*) **by** (*simp add*: *subst-restrict-def*)
then have *σ* |*s* {*x*} = (λ*y*. *if y = x then σ x else Var y*) **by** (*simp cong*: *if-cong*)
then show *?thesis* **by** (*rule exI*[*of* - *σ x*])
qed

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
assume *?left*
then obtain *l r p σ* **where** *step*: (*s*,*t*) ∈ *rstep-r-p-s R* (*l*,*r*) *p σ*
unfolding *rstep-iff-rstep-r-p-s* **by** *blast*
obtain *D* **where** *D*: *D = ctxt-of-pos-term p s* **by** *auto*
with *step* **have** *Rrule*: (*l*,*r*) ∈ *R* **and** *s*: *s = D*⟨*l* · *σ*⟩ **and** *t*: *t = D*⟨*r* · *σ*⟩
unfolding *rstep-r-p-s-def* **by** (*force simp*: *Let-def*)+
then show *?right* **unfolding** *rstep-r-c-s-def* **by** *auto*
next
assume *?right*
from *this* **obtain** *l r C σ* **where** (*l*,*r*) ∈ *R* ∧ (*s*,*t*) ∈ *rstep-r-c-s* (*l*,*r*) *C σ* **by** *auto*
then have *rule*: (*l*,*r*) ∈ *R* **and** *s*: *s = C*⟨*l* · *σ*⟩ **and** *t*: *t = C*⟨*r* · *σ*⟩
unfolding *rstep-r-c-s-def* **by** *auto*
show *?left* **unfolding** *rstep-eq-closure* **by** (*auto simp*: *s t intro*: *rule*)
qed

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
assume *?right*
show *?left*
proof
fix *s t*
assume (*s*,*t*) ∈ *rstep R*
then obtain *l r C σ* **where** *step*: (*l*,*r*) ∈ *R* ∧ (*s*,*t*) ∈ *rstep-r-c-s* (*l*,*r*) *C σ*
unfolding *rstep-iff-rstep-r-c-s* **by** *best*
then have *Rrule*: (*l*,*r*) ∈ *R* **and** *s*: *s* = *C*⟨*l* · *σ*⟩ **and** *t*: *t* = *C*⟨*r* · *σ*⟩
unfolding *rstep-r-c-s-def* **by** (*force*)+
from *Rrule* ‹*?right*› **obtain** *l′ r′ C′ σ′* **where** *Srule*: (*l′*,*r′*) ∈ *S* **and** *l*: *l* = *C′*⟨*l′* · *σ′*⟩ **and** *r*: *r* = *C′*⟨*r′* · *σ′*⟩
by (*force simp*: *Let-def*)+
let *?D* = *C* ∘*c* (*C′* ·*c* *σ*)
let *?sig* = *σ′* ∘*s* *σ*
have *s2*: *s* = *?D*⟨*l′* · *?sig*⟩ **by** (*simp add*: *s l*)
have *t2*: *t* = *?D*⟨*r′* · *?sig*⟩ **by** (*simp add*: *t r*)
from *s2 t2* **have** *sStep*: (*s*,*t*) ∈ *rstep-r-c-s* (*l′*,*r′*) *?D ?sig* **unfolding** *rstep-r-c-s-def* **by** *force*
with *Srule* **show** (*s*,*t*) ∈ *rstep S* **by** (*simp only*: *rstep-iff-rstep-r-c-s*, *blast*)
qed
next
assume *?left*
show *?right*
proof (*rule ccontr*)
assume ¬ *?right*
from *this* **obtain** *l r* **where** (*l*,*r*) ∈ *R* **and** *cond*: ∀ *l′ r′ C σ*. (*l′*,*r′*) ∈ *S* ⟶ (*l* ≠ *C*⟨*l′* · *σ*⟩ ∨ *r* ≠ *C*⟨*r′* · *σ*⟩) **by** *blast*
then have (*l*,*r*) ∈ *rstep R* **by** *blast*
with ‹*?left*› **have** (*l*,*r*) ∈ *rstep S* **by** *auto*
with *cond* **show** *False* **by** (*simp only*: *rstep-iff-rstep-r-c-s*, *unfold rstep-r-c-s-def*, *force*)
qed
qed

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 −
from ‹(*s*,*t*) ∈ *rstep R*› **obtain** *l r C σ* **where** *R*: (*l*,*r*) ∈ *R*
and *s*: *s* = *C*⟨*l*·*σ*⟩ **and** *t*: *t* = *C*⟨*r*·*σ*⟩ **by** *auto*
from ‹*funas-trs R* ⊆ *F*› **and** *R* **have** *funas-term r* ⊆ *F*
unfolding *funas-defs* [*abs-def*] **by** *force*
with *wf*[*OF R*] ‹*funas-term s* ⊆ *F*› **show** *?thesis* **unfolding** *s t* **by** (*force simp*: *funas-term-subst*)

qed

lemma *rstep-preserves-funas-terms*:
assumes *funas-trs R* $\subseteq$ *F* **and** *funas-term s* $\subseteq$ *F* **and** (*s*,*t*) $\in$ *rstep R*
and *wf*: *wf-trs R*
shows *funas-term t* $\subseteq$ *F*
by (*rule rstep-preserves-funas-terms-var-cond*[*OF assms*(*1*−*3*)], *insert wf*[*unfolded wf-trs-def*], *auto*)

lemma *rsteps-preserve-funas-terms-var-cond*:
assumes *F*: *funas-trs R* $\subseteq$ *F* **and** *s*: *funas-term s* $\subseteq$ *F* **and** *steps*: (*s*,*t*) $\in$ (*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*
using *steps*
proof (*induct*)
case *base* **then show** *?case* **by** (*rule s*)
next
case (*step t u*)
show *?case* **by** (*rule rstep-preserves-funas-terms-var-cond*[*OF F step*(*3*) *step*(*2*) *wf*])
qed

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*
by (*rule rsteps-preserve-funas-terms-var-cond*[*OF assms*(*1*−*3*)], *insert wf*[*unfolded wf-trs-def*], *auto*)

lemma *no-Var-rstep* [*simp*]:
assumes *wf-trs R* **and** (*Var x*, *t*) $\in$ *rstep R* **shows** *False*
using *rstep-imp-Fun*[*OF assms*] **by** *auto*

lemma *lhs-wf*:
assumes *R*: (*l*, *r*) $\in$ *R* **and** *funas-trs R* $\subseteq$ *F*
shows *funas-term l* $\subseteq$ *F*
using *assms* **by** (*force simp*: *funas-trs-def funas-rule-def*)

lemma *rhs-wf*:
assumes *R*: (*l*, *r*) $\in$ *R* **and** *funas-trs R* $\subseteq$ *F*
shows *funas-term r* $\subseteq$ *F*
using *assms* **by** (*force simp*: *funas-trs-def funas-rule-def*)

lemma *supt-map-funs-term* [*intro*]:
assumes *t* $\rhd$ *s*
shows *map-funs-term fg t* $\rhd$ *map-funs-term fg s*
using *assms*
proof (*induct*)

case (*arg s ss f*)
then have *map-funs-term fg s* ∈ *set*(*map* (*map-funs-term fg*) *ss*) **by** *simp*
then show *?case* **unfolding** *term.map* **by** (*rule supt.arg*)
next
case (*subt s ss u f*)
then have *map-funs-term fg s* ∈ *set*(*map* (*map-funs-term fg*) *ss*) **by** *simp*
with ‹*map-funs-term fg s* ▷ *map-funs-term fg u*› **show** *?case*
unfolding *term.map* **by** (*metis supt.subt*)
qed

lemma *nondef-root-imp-arg-step*:
assumes (*Fun f ss*, *t*) ∈ *rstep R*
and *wf*: ∀ (*l*, *r*)∈*R*. *is-Fun l*
and *ndef*: ¬ *defined R* (*f*, *length ss*)
shows ∃ *i*<*length ss*. (*ss* ! *i*, *t* |- [*i*]) ∈ *rstep R*
∧ *t* = *Fun f* (*take i ss* @ (*t* |- [*i*]) # *drop* (*Suc i*) *ss*)
proof −
from *assms* **obtain** *l r p* σ
where *rstep-r-p-s*: (*Fun f ss*, *t*) ∈ *rstep-r-p-s R* (*l*, *r*) *p* σ
unfolding *rstep-iff-rstep-r-p-s* **by** *auto*
let *?C* = *ctxt-of-pos-term p* (*Fun f ss*)
from *rstep-r-p-s* **have** *p* ∈ *poss* (*Fun f ss*) **and** (*l*, *r*) ∈ *R*
and *?C*⟨*l* · σ⟩ = *Fun f ss* **and** *?C*⟨*r* · σ⟩ = *t* **unfolding** *rstep-r-p-s-def Let-def*
by *auto*
have ∃ *i q*. *p* = *i*#*q*
proof (*cases p*)
case *Cons* **then show** *?thesis* **by** *auto*
next
case *Nil*
have *?C* = □ **unfolding** *Nil* **by** *simp*
with ‹*?C*⟨*l*·σ⟩ = *Fun f ss*› **have** *l*·σ = *Fun f ss* **by** *simp*
have ∀ (*l*,*r*)∈*R*. ∃ *f ss*. *l* = *Fun f ss*
proof (*intro ballI impI*)
fix *lr* **assume** *lr* ∈ *R*
with *wf* **have** ∀ *x*. *fst lr* ≠ *Var x* **by** *auto*
then have ∃ *f ss*. (*fst lr*) = *Fun f ss* **by** (*cases fst lr*) *auto*
then show (λ(*l*,*r*). ∃ *f ss*. *l* = *Fun f ss*) *lr* **by** *auto*
qed
with ‹(*l*,*r*) ∈ *R*› **obtain** *g ts* **where** *l* = *Fun g ts* **unfolding** *wf-trs-def* **by** *best*
with ‹*l*·σ = *Fun f ss*› ‹*l* = *Fun g ts*› **and** ‹(*l*, *r*) ∈ *R*› *ndef*
show *?thesis* **unfolding** *defined-def* **by** *auto*
qed
then obtain *i q* **where** *p* = *i*#*q* **by** *auto*
from ‹*p* ∈ *poss*(*Fun f ss*)› **have** *i* < *length ss* **and** *q* ∈ *poss*(*ss*!*i*) **unfolding** ‹*p*
= *i*#*q*› **by** *auto*
let *?D* = *ctxt-of-pos-term q* (*ss*!*i*)
have *C*: *?C* = *More f* (*take i ss*) *?D* (*drop* (*Suc i*) *ss*) **unfolding** ‹*p* = *i*#*q*› **by**
auto
from ‹*?C*⟨*l*·σ⟩ = *Fun f ss*› **have** *take i ss*@*?D*⟨*l*·σ⟩#*drop* (*Suc i*) *ss* = *ss* **un-**

folding *C* **by** *auto*
then have (*take i ss@?D⟨l·σ⟩#drop* (*Suc i*) *ss*)!*i* = *ss*!*i* **by** *simp*
with ‹*i* < *length ss*› **have** *?D⟨l·σ⟩* = *ss*!*i* **using** *nth-append-take*[**where** *xs* = *ss* **and** *i* = *i*] **by** *auto*
have *t*: *t* = *Fun f* (*take i ss@?D⟨r·σ⟩#drop* (*Suc i*) *ss*) **unfolding** ‹*?C⟨r·σ⟩* = *t*›[*symmetric*] *C* **by** *simp*
from ‹*i* < *length ss*› **have** *t*|-[*i*] = *?D⟨r·σ⟩* **unfolding** *t* **unfolding** *subt-at.simps* **using** *nth-append-take*[**where** *xs* = *ss* **and** *i* = *i*] **by** *auto*
from ‹*q* ∈ *poss*(*ss*!*i*)› **and** ‹(*l*,*r*) ∈ *R*›
and ‹*?D⟨l·σ⟩* = *ss*!*i*› **and** ‹*t*|-[*i*] = *?D⟨r·σ⟩*›[*symmetric*]
have (*ss*!*i*,*t*|-[*i*]) ∈ *rstep-r-p-s R* (*l*,*r*) *q σ* **unfolding** *rstep-r-p-s-def Let-def* **by** *auto*
then have (*ss*!*i*,*t*|-[*i*]) ∈ *rstep R* **unfolding** *rstep-iff-rstep-r-p-s* **by** *auto*
from ‹*i* < *length ss*› **and** *this* **and** *t* **show** *?thesis* **unfolding** ‹*t*|-[*i*] = *?D⟨r·σ⟩*›[*symmetric*] **by** *auto*
qed

lemma *nondef-root-imp-arg-steps*:
assumes (*Fun f ss*,*t*) ∈ (*rstep R*)*
and *wf*: ∀ (*l*, *r*)∈*R*. *is-Fun l*
and ¬ *defined R* (*f*,*length ss*)
shows ∃ *ts*. *length ts* = *length ss* ∧ *t* = *Fun f ts* ∧ (∀ *i*<*length ss*. (*ss*!*i*,*ts*!*i*) ∈ (*rstep R*)*)
proof −
from *assms* **obtain** *n* **where** (*Fun f ss*,*t*) ∈ (*rstep R*)^^*n*
using *rtrancl-imp-relpow* **by** *best*
then show *?thesis*
proof (*induct n arbitrary*: *t*)
case *0* **then show** *?case* **by** *auto*
next
case (*Suc n*)
then obtain *u* **where** (*Fun f ss*,*u*) ∈ (*rstep R*)^^*n* **and** (*u*,*t*) ∈ *rstep R* **by** *auto*
with *Suc* **obtain** *ts* **where** *IH1*: *length ts* = *length ss* **and** *IH2*: *u* = *Fun f ts* **and** *IH3*: ∀ *i*<*length ss*. (*ss*!*i*,*ts*!*i*) ∈ (*rstep R*)* **by** *auto*
from ‹(*u*,*t*) ∈ *rstep R*› **have** (*Fun f ts*,*t*) ∈ *rstep R* **unfolding** ‹*u* = *Fun f ts*› **.**
from *nondef-root-imp-arg-step*[*OF this wf* ‹¬ *defined R* (*f*,*length ss*)›[*simplified IH1*[*symmetric*]]]
obtain *j* **where** *j* < *length ts*
and (*ts*!*j*,*t*|-[*j*]) ∈ *rstep R*
and *B*: *t* = *Fun f* (*take j ts*@(*t*|-[*j*])#*drop* (*Suc j*) *ts*) (**is** *t* = *Fun f ?ts*) **by** *auto*
from ‹*j* < *length ts*› **have** *length ?ts* = *length ts* **by** *auto*
then have *A*: *length ?ts* = *length ss* **unfolding** ‹*length ts* = *length ss*› **.**
have *C*: ∀ *i*<*length ss*. (*ss*!*i*,*?ts*!*i*) ∈ (*rstep R*)*
proof (*intro allI*, *intro impI*)
fix *i* **assume** *i* < *length ss*
from ‹*i* < *length ss*› **and** *IH3* **have** (*ss*!*i*,*ts*!*i*) ∈ (*rstep R*)* **by** *auto*
have *i* = *j* ∨ *i* ≠ *j* **by** *auto*

```
      then show (ss!i,?ts!i) ∈ (rstep R)*
      proof
        assume i = j
        from ‹j < length ts› have j ≤ length ts by simp
        from nth-append-take[OF this] have ?ts!j = t|-[j] by simp
       from ‹(ts!j,t|-[j]) ∈ rstep R› have (ts!i,t|-[i]) ∈ rstep R unfolding ‹i = j› .
        with ‹(ss!i,ts!i) ∈ (rstep R)*› show ?thesis unfolding ‹i = j› unfolding
‹?ts!j = t|-[j]› by auto
      next
        assume i ≠ j
        from ‹i < length ss› have i ≤ length ts unfolding ‹length ts = length ss›
by simp
        from ‹j < length ts› have j ≤ length ts by simp
        from nth-append-take-drop-is-nth-conv[OF ‹i ≤ length ts› ‹j ≤ length ts› ‹i
≠ j›]
        have ?ts!i = ts!i by simp
        with ‹(ss!i,ts!i) ∈ (rstep R)*› show ?thesis by auto
      qed
    qed
    from A and B and C show ?case by blast
  qed
qed

lemma rstep-imp-nrrstep:
  assumes is-Fun s and ¬ defined R (the (root s)) and ∀(l,r)∈R. is-Fun l
    and (s, t) ∈ rstep R
  shows (s, t) ∈ nrrstep R
proof −
  from ‹is-Fun s› obtain f ss where s: s = Fun f ss by (cases s) auto
  with assms have undef: ¬ defined R (f, length ss) by simp
  from assms have non-var: ∀(l, r)∈R. is-Fun l by auto
  from nondef-root-imp-arg-step[OF ‹(s, t) ∈ rstep R›[unfolded s] non-var undef]
  obtain i where i < length ss and step: (ss ! i, t |- [i]) ∈ rstep R
    and t: t = Fun f (take i ss @ (t |- [i]) # drop (Suc i) ss) by auto
  from step obtain C l r σ where (l, r) ∈ R and lhs: ss ! i = C⟨l · σ⟩
    and rhs: t |- [i] = C⟨r · σ⟩ by auto
  let ?C = More f (take i ss) C (drop (Suc i) ss)
  have (l, r) ∈ R by fact
  moreover have ?C ≠ □ by simp
  moreover have s = ?C⟨l · σ⟩
  proof −
    have s = Fun f (take i ss @ ss!i # drop (Suc i) ss)
      using id-take-nth-drop[OF ‹i < length ss›] unfolding s by simp
    also have ... = ?C⟨l · σ⟩ by (simp add: lhs)
    finally show ?thesis .
  qed
  moreover have t = ?C⟨r · σ⟩
  proof −
    have t = Fun f (take i ss @ t |- [i] # drop (Suc i) ss) by fact
```

also have ... = *Fun f* (*take i ss* @ $C\langle r \cdot \sigma \rangle$ # *drop* (*Suc i*) *ss*) **by** (*simp add*: *rhs*)
finally show *?thesis* **by** *simp*
qed
ultimately show (*s*, *t*) ∈ *nrrstep R* **unfolding** *nrrstep-def'* **by** *blast*
qed

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*)*
using ‹(*s*, *t*) ∈ (*rstep R*)*›
proof (*induct*)
case *base* **show** *?case* **by** *simp*
next
case (*step u v*)
from *assms* **obtain** *f ss* **where** *s*: *s* = *Fun f ss* **by** (*induct s*) *auto*
from *nrrsteps-preserve-root*[*OF* ‹(*s*, *u*) ∈ (*nrrstep R*)*›[*unfolded s*]]
obtain *ts* **where** *u*: *u* = *Fun f ts* **by** *auto*
from *nrrsteps-equiv-num-args*[*OF* ‹(*s*, *u*) ∈ (*nrrstep R*)*›[*unfolded s*]]
have *len*: *length ss* = *length ts* **unfolding** *u* **by** *simp*
have *is-Fun u* **by** (*simp add*: *u*)
have *undef*: ¬ *defined R* (*the* (*root u*))
using ‹¬ *defined R* (*the* (*root s*))›
unfolding *s u* **by** (*simp add*: *len*)
have (*u*, *v*) ∈ *nrrstep R*
using *rstep-imp-nrrstep*[*OF* ‹*is-Fun u*› *undef no-vars*] *step*
by *simp*
with *step* **show** *?case* **by** *auto*
qed

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 (*rule steps-imp-not-SN-on*)
fix *t* :: ('*f*,'*v*)*term*
let *?yt* = *subst y t*
show (*t*, *r* · *?yt*) ∈ *rstep R*
by (*rule rstepI*[*OF assms*, **where** *C* = □ **and** σ = *?yt*], *auto simp*: *subst-def*)
qed

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*}
using *ctxt-closed-SN-on-subt*[*OF ctxt - sub*] *SN*
by *auto*

lemma *root-Some*:
assumes *root t* = *Some fn*
obtains *ss* **where** *length ss* = *snd fn* **and** *t* = *Fun* (*fst fn*) *ss*
using *assms* **by** (*induct t*) *auto*

lemma *map-funs-rule-power*:
fixes *f* :: *'f* ⇒ *'f*
shows ((*map-funs-rule f*) ^^ *n*) = *map-funs-rule* (*f* ^^ *n*)
proof (*rule sym*, *intro ext*, *clarify*)
fix *l r* :: (*'f*,*'v*)*term*
show *map-funs-rule* (*f* ^^ *n*) (*l*,*r*) = (*map-funs-rule f* ^^ *n*) (*l*,*r*)
proof (*induct n*)
case *0*
show *?case* **by** (*simp add*: *term.map-ident*)
next
case (*Suc n*)
have *map-funs-rule* (*f* ^^ *Suc n*) (*l*,*r*) = *map-funs-rule f* (*map-funs-rule* (*f* ^^ *n*) (*l*,*r*))
by (*simp add*: *map-funs-term-comp*)
also have ... = *map-funs-rule f* ((*map-funs-rule f* ^^ *n*) (*l*,*r*)) **unfolding** *Suc* **..**
also have ... = (*map-funs-rule f* ^^ (*Suc n*)) (*l*,*r*) **by** *simp*
finally show *?case* **.**
qed
qed

lemma *map-funs-trs-power*:
fixes *f* :: *'f* ⇒ *'f*
shows *map-funs-trs f* ^^ *n* = *map-funs-trs* (*f* ^^ *n*)
proof
fix *R* :: (*'f*, *'v*) *trs*
have *map-funs-rule* (*f* ^^ *n*) ‘ *R* = (*map-funs-rule f* ^^ *n*) ‘ *R* **unfolding** *map-funs-rule-power* **..**
also have ... = ((λ *R*. *map-funs-trs f R*) ^^ *n*) *R* **unfolding** *map-funs-trs.simps*
apply (*induct n*)
apply *simp*
by (*metis comp-apply funpow.simps*(*2*) *image-comp*)
finally have *map-funs-rule* (*f* ^^ *n*) ‘ *R* = (*map-funs-trs f* ^^ *n*) *R* **.**
then show (*map-funs-trs f* ^^ *n*) *R* = *map-funs-trs* (*f* ^^ *n*) *R*
by (*simp add*: *map-funs-trs.simps*)
qed

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 −
let *?S* = {*t* | *t*. *s* ⊵ *t* ∧ ¬ *SN-on R* {*t*}}
from *assms* **have** *s*: *s* ∈ *?S* **by** *auto*
from *mp*[*OF spec*[*OF spec*[*OF SN-imp-minimal*[*OF SN-supt*]]] *s*]
obtain *t* **where** *st*: *s* ⊵ *t* **and** *nSN*: ¬ *SN-on R* {*t*}
and *min*: ∀ *u*. (*t*,*u*) ∈ *supt* ⟶ *u* ∉ *?S* **by** *auto*
have *t* ∈ *Tinf R* **unfolding** *Tinf-def*
proof (*intro CollectI allI impI conjI nSN*)
fix *u*
assume *u*: *t* ▷ *u*
from *u st* **have** *s* ▷ *u* **using** *supteq-supt-trans* **by** *auto*
with *min u* **show** *SN-on R* {*u*} **by** *auto*
qed
with *st* **show** *?thesis* **by** *auto*
qed

lemma *not-SN-imp-Tinf*:
assumes ¬ *SN R* **shows** ∃ *t*. *t* ∈ *Tinf R*
using *assms not-SN-imp-subt-Tinf* **unfolding** *SN-on-def* **by** *blast*

lemma *ctxt-of-pos-term-map-funs-term-conv* [*iff*]:
assumes *p* ∈ *poss s*
shows *map-funs-ctxt fg* (*ctxt-of-pos-term p s*) = (*ctxt-of-pos-term p* (*map-funs-term fg s*))
using *assms*
proof (*induct s arbitrary*: *p*)
case (*Var x*) **then show** *?case* **by** *simp*
next
case (*Fun f ss*) **then show** *?case*
proof (*cases p*)
case *Nil* **then show** *?thesis* **by** *simp*
next
case (*Cons i q*)
with ‹*p* ∈ *poss*(*Fun f ss*)› **have** *i* < *length ss* **and** *q* ∈ *poss*(*ss*!*i*) **unfolding** *Cons poss.simps* **by** *auto*
then have *ss*!*i* ∈ *set ss* **by** *simp*
with *Fun* **and** ‹*q* ∈ *poss*(*ss*!*i*)›
have *IH*: *map-funs-ctxt fg*(*ctxt-of-pos-term q* (*ss*!*i*)) = *ctxt-of-pos-term q* (*map-funs-term fg* (*ss*!*i*)) **by** *simp*
have *map-funs-ctxt fg*(*ctxt-of-pos-term p* (*Fun f ss*)) = *map-funs-ctxt fg*(*ctxt-of-pos-term* (*i*#*q*) (*Fun f ss*)) **unfolding** *Cons* **by** *simp*
also have ... = *map-funs-ctxt fg*(*More f* (*take i ss*) (*ctxt-of-pos-term q* (*ss*!*i*)) (*drop* (*Suc i*) *ss*)) **by** *simp*
also have ... = *More* (*fg f*) (*map* (*map-funs-term fg*) (*take i ss*)) (*map-funs-ctxt fg*(*ctxt-of-pos-term q* (*ss*!*i*))) (*map* (*map-funs-term fg*) (*drop* (*Suc i*) *ss*)) **by** *simp*
also have ... = *More* (*fg f*) (*map* (*map-funs-term fg*) (*take i ss*)) (*ctxt-of-pos-term q* (*map-funs-term fg* (*ss*!*i*))) (*map* (*map-funs-term fg*) (*drop* (*Suc i*) *ss*)) **unfolding** *IH* **by** *simp*
also have ... = *More* (*fg f*) (*take i* (*map* (*map-funs-term fg*) *ss*)) (*ctxt-of-pos-term*

q (*map* (*map-funs-term fg*) *ss*!*i*)) (*drop* (*Suc i*) (*map* (*map-funs-term fg*) *ss*)) **unfolding** *nth-map*[*OF* ‹*i* < *length ss*›,*symmetric*] *take-map drop-map nth-map* **by** *simp*
finally show *?thesis* **unfolding** *Cons* **by** *simp*
qed
qed

lemma *var-rewrite-imp-not-SN*:
assumes *sn*: *SN-on* (*rstep R*) {*u*} **and** *step*: (*t*, *s*) ∈ *rstep R*
shows *is-Fun t*
using *assms*
proof (*cases t*)
case (*Fun f ts*) **then show** *?thesis* **by** *simp*
next
case (*Var x*)
from *step* **obtain** *l r p σ* **where** (*Var x*, *s*) ∈ *rstep-r-p-s R* (*l*,*r*) *p σ* **unfolding** *Var rstep-iff-rstep-r-p-s* **by** *best*
then have *l* · *σ* = *Var x* **and** *rule*: (*l*,*r*) ∈ *R* **unfolding** *rstep-r-p-s-def* **by** (*auto simp*: *Let-def*)
from *this* **obtain** *y* **where** *l* = *Var y* (**is** - = *?y*) **by** (*cases l*, *auto*)
with *rule* **have** (*?y*, *r*) ∈ *R* **by** *auto*
then have ¬ (*SN-on* (*rstep R*) {*u*}) **by** (*rule left-var-imp-not-SN*)
with *sn* **show** *?thesis* **by** *blast*
qed

lemma *rstep-id*: *rstep Id* = *Id* **by** *auto*

lemma *map-funs-rule-id* [*simp*]: *map-funs-rule id* = *id*
by (*intro ext*, *auto*)

lemma *map-funs-trs-id* [*simp*]: *map-funs-trs id* = *id*
by (*intro ext*, *auto simp*: *map-funs-trs.simps*)

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*
unfolding *sig-step-def* **by** *auto*

lemma *sig-step-UNIV*: *sig-step UNIV R* = *R* **unfolding** *sig-step-def* **by** *simp*

lemma *sig-stepI*[*intro*]: (*a*,*b*) ∈ *R* ⟹ *funas-term a* ⊆ *F* ⟹ *funas-term b* ⊆ *F* ⟹ (*a*,*b*) ∈ *sig-step F R* **unfolding** *sig-step-def* **by** *auto*

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* **unfolding** *sig-step-def* **by** *auto*

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 (*rule trans-ctxt-sig-imp-all-ctxt-closed*)
fix $C :: ('f,'v)ctxt$ **and** $s\ t :: ('f,'v)term$
assume *C*: *funas-ctxt C* $\subseteq$ *F*
and *s*: *funas-term s* $\subseteq$ *F*
and *t*: *funas-term t* $\subseteq$ *F*
and *steps*: $(s,t) \in ?R^*$
from *steps*
show $(C\langle s \rangle, C\langle t \rangle) \in ?R^*$
proof (*induct*)
case (*step t u*)
from *step*(*2*) **have** *tu*: $(t,u) \in$ *rstep R* **and** *t*: *funas-term t* $\subseteq$ *F* **and** *u*: *funas-term u* $\subseteq$ *F* **by** *auto*
have $(C\langle t \rangle, C\langle u \rangle) \in ?R$ **by** (*rule sig-stepI*[*OF rstep-ctxt*[*OF tu*]], *insert C t u*, *auto*)
with *step*(*3*) **show** *?case* **by** *auto*
qed *auto*
qed (*auto intro*: *trans-rtrancl*)

lemma *wf-loop-imp-sig-ctxt-rel-not-SN*:
assumes *R*: $(l,C\langle l\rangle) \in R$ **and** *wf-l*: *funas-term l* $\subseteq$ *F*
and *wf-C*: *funas-ctxt C* $\subseteq$ *F*
and *ctxt*: *ctxt.closed R*
shows $\neg$ *SN-on* (*sig-step F R*) $\{l\}$
proof −
let $?t = \lambda\ i.\ (C\widehat{\ }i)\langle l\rangle$
have $\forall\, i.$ *funas-term* (*?t i*) $\subseteq$ *F*
proof
fix *i* **show** *funas-term* (*?t i*) $\subseteq$ *F* **unfolding** *funas-term-ctxt-apply*
by (*rule Un-least*[*OF - wf-l*], *induct i*, *insert wf-C*, *auto*)
qed
moreover have $\forall\, i.$ (*?t i*,*?t*(*Suc i*)) $\in R$
proof
fix *i*
show (*?t i*, *?t* (*Suc i*)) $\in R$
proof (*induct i*)
case *0* **with** *R* **show** *?case* **by** *auto*
next
case (*Suc i*)
from *ctxt.closedD*[*OF ctxt Suc*, *of C*]
show *?case* **by** *simp*
qed
qed
ultimately have *steps*: $\forall\, i.$ (*?t i*,*?t*(*Suc i*)) $\in$ *sig-step F R* **unfolding** *sig-step-def* **by** *blast*
show *?thesis* **unfolding** *SN-defs*
by (*simp*, *intro exI*[*of - ?t*], *simp only*: *steps*, *simp*)
qed

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 −
let *?σ* = (λ*x*. *r*)
let *?t* = λ*i*. (*?σ* ^^ *i*) *x*
obtain *t* **where** *t*: *t* = *?t* **by** *auto*
from *rhs-wf*[*OF x F*] **have** *wf-r*: *funas-term r* ⊆ *F* **.**
{
fix *i*
have *funas-term* (*?t i*) ⊆ *F*
proof (*induct i*)
case *0* **show** *?case* **using** *wf-r* **by** *auto*
next
case (*Suc i*)
have *?t* (*Suc i*) = *?t i* · *?σ* **unfolding** *subst-power-Suc subst-compose-def* **by** *simp*
also have *funas-term* ... ⊆ *F* **unfolding** *funas-term-subst*[*of ?t i*]
using *Suc wf-r* **by** *auto*
finally show *?case* **.**
qed
} note *wf-t* = *this*
{
fix *i*
have (*t i*, *t* (*Suc i*)) ∈ (*sig-step F* (*rstep R*)) **unfolding** *t*
by (*rule sig-stepI*[*OF rstepI*[*OF x*, *of* - □ *?σ* ^^ *i*] *wf-t wf-t*], *auto simp*: *subst-compose-def*)
} note *steps* = *this*
have *x*: *t 0* = *Var x* **unfolding** *t* **by** *simp*
with *steps* **show** *?thesis* **unfolding** *SN-defs not-not*
by (*intro exI*[*of* - *t*], *auto*)
qed

lemma *rhs-free-vars-imp-sig-step-not-SN*:
assumes *R*: (*l*,*r*) ∈ *R* **and** *free*: ¬ *vars-term r* ⊆ *vars-term l*
and *F*: *funas-trs R* ⊆ *F*
shows ¬ *SN-on* (*sig-step F* (*rstep R*)) {*l*}
proof −
from *free* **obtain** *x* **where** *x*: *x* ∈ *vars-term r* − *vars-term l* **by** *auto*
then have *x* ∈ *vars-term r* **by** *simp*
from *supteq-Var*[*OF this*] **have** *r* ⊵ *Var x* **.**
then obtain *C* **where** *r*: *C*⟨*Var x*⟩ = *r* **by** *auto*
let *?σ* = λ*y*. *if y* = *x then l else Var y*
let *?t* = λ*i*. ((*C* ·$_c$ *?σ*)^*i*)⟨*l*⟩
from *rhs-wf*[*OF R*] *F* **have** *wf-r*: *funas-term r* ⊆ *F* **by** *fast*
from *lhs-wf*[*OF R*] *F* **have** *wf-l*: *funas-term l* ⊆ *F* **by** *fast*
from *wf-r*[*unfolded r*[*symmetric*]]
have *wf-C*: *funas-ctxt C* ⊆ *F* **by** *simp*

from x **have** *neq*: $\forall\, y{\in}$*vars-term* $l.\ y \neq x$ **by** *auto*
have $l{\cdot}?\sigma = l \cdot$ *Var*
by (*rule term-subst-eq*, *insert neq*, *auto*)
then have l: $l \cdot ?\sigma = l$ **by** *simp*
have *wf-C*: *funas-ctxt* $(C \cdot_c ?\sigma) \subseteq F$ **using** *wf-C wf-l*
by *simp*
have *rsigma*: $r{\cdot}?\sigma = (C \cdot_c ?\sigma)\langle l\rangle$ **unfolding** r[*symmetric*] **by** *simp*
from R **have** *lr*: $(l \cdot ?\sigma,\ r \cdot ?\sigma) \in$ *rstep* R **by** *auto*
then have *lr*: $(l,(C \cdot_c ?\sigma)\langle l\rangle) \in$ *rstep* R **unfolding** l **unfolding** *rsigma* **.**
show *?thesis*
by (*rule wf-loop-imp-sig-ctxt-rel-not-SN*[*OF lr wf-l wf-C ctxt-closed-rstep*])
qed

lemma *lhs-var-imp-rstep-not-SN*: **assumes** $(Var\ x,r) \in R$ **shows** $\neg$ *SN*(*rstep* R)
using *lhs-var-imp-sig-step-not-SN-on*[*OF assms subset-refl*] **unfolding** *sig-step-def SN-defs* **by** *blast*

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) $\{l\}$
using *rhs-free-vars-imp-sig-step-not-SN*[*OF assms subset-refl*] **unfolding** *sig-step-def SN-defs* **by** *blast*

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
assume *SN*: *SN-on* (*rstep* R) $\{t\}$
let *?C* = *ctxt-of-pos-term* p t
from *step* **have** *left*: $?C\langle l \cdot \sigma\rangle = t$ (**is** $?t = t$) **and** *right*: $?C\langle r \cdot \sigma\rangle = s$ **and** *pos*: $p \in$ *poss* t
and *rule*: $(l,r) \in R$
unfolding *rstep-r-p-s-def* **by** (*auto simp*: *Let-def*)
from *rhs-free-vars-imp-rstep-not-SN*[*OF rule vars*] **have** *nSN*:$\neg$ *SN-on* (*rstep* R) $\{l\}$ **by** *simp*
from *SN-imp-SN-subt*[*OF SN ctxt-imp-supteq*[*of ?C* $l \cdot \sigma$, *simplified left*]]
have *SN*: *SN-on* (*rstep* R) $\{l \cdot \sigma\}$ **.**
from *SNinstance-imp-SN*[*OF SN*] *nSN* **show** *False* **by** *simp*
qed

lemma *not-SN-on-rstep-subst-apply-term*[*intro*]:
assumes $\neg$ *SN-on* (*rstep* R) $\{t\}$ **shows** $\neg$ *SN-on* (*rstep* R) $\{t \cdot \sigma\}$
using *assms* **unfolding** *SN-on-def* **by** *best*

lemma *SN-rstep-imp-wf-trs*: **assumes** *SN* (*rstep* R) **shows** *wf-trs* R
proof (*rule ccontr*)
assume $\neg$ *wf-trs* R
then obtain l r **where** R: $(l,r) \in R$

and *not-wf*: $(\forall f\ ts.\ l \neq Fun\ f\ ts) \vee \neg(vars\text{-}term\ r \subseteq vars\text{-}term\ l)$ **unfolding** *wf-trs-def*
by *auto*
from *not-wf* **have** $\neg\ SN\ (rstep\ R)$
proof
assume *free*: $\neg\ vars\text{-}term\ r \subseteq vars\text{-}term\ l$
from *rhs-free-vars-imp-rstep-not-SN*[*OF R free*] **show** *?thesis* **unfolding** *SN-defs* **by** *auto*
next
assume $\forall f\ ts.\ l \neq Fun\ f\ ts$
then obtain x **where** $l{:}l = Var\ x$ **by** (*cases l*) *auto*
with R **have** $(Var\ x,r) \in R$ **unfolding** l **by** *simp*
from *lhs-var-imp-rstep-not-SN*[*OF this*] **show** *?thesis* **by** *simp*
qed
with *assms* **show** *False* **by** *blast*
qed

lemma *SN-sig-step-imp-wf-trs*: **assumes** *SN*: $SN\ (sig\text{-}step\ F\ (rstep\ R))$ **and** *F*: $funas\text{-}trs\ R \subseteq F$ **shows** *wf-trs R*
proof (*rule ccontr*)
assume $\neg$ *wf-trs R*
then obtain $l\ r$ **where** *R*: $(l,r) \in R$
and *not-wf*: $(\forall f\ ts.\ l \neq Fun\ f\ ts) \vee \neg(vars\text{-}term\ r \subseteq vars\text{-}term\ l)$ **unfolding** *wf-trs-def*
by *auto*
from *not-wf* **have** $\neg\ SN\ (sig\text{-}step\ F\ (rstep\ R))$
proof
assume *free*: $\neg\ vars\text{-}term\ r \subseteq vars\text{-}term\ l$
from *rhs-free-vars-imp-sig-step-not-SN*[*OF R free F*] **show** *?thesis* **unfolding** *SN-on-def* **by** *auto*
next
assume $\forall f\ ts.\ l \neq Fun\ f\ ts$
then obtain x **where** $l{:}l = Var\ x$ **by** (*cases l*) *auto*
with R **have** $(Var\ x,r) \in R$ **unfolding** l **by** *simp*
from *lhs-var-imp-sig-step-not-SN-on*[*OF this F*] **show** *?thesis*
unfolding *SN-on-def* **by** *auto*
qed
with *assms* **show** *False* **by** *blast*
qed

lemma *rhs-free-vars-imp-rstep-not-SN'*:
assumes $(l,\ r) \in R$ **and** $\neg\ vars\text{-}term\ r \subseteq vars\text{-}term\ l$
shows $\neg\ SN\ (rstep\ R)$
using *rhs-free-vars-imp-rstep-not-SN* [*OF assms*] **by** (*auto simp*: *SN-defs*)

lemma *SN-imp-variable-condition*:
assumes $SN\ (rstep\ R)$
shows $\forall (l,\ r) \in R.\ vars\text{-}term\ r \subseteq vars\text{-}term\ l$
using *assms* **and** *rhs-free-vars-imp-rstep-not-SN'* [*of - - R*] **by** *blast*

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 −
from *rstep-imp-C-s-r*[*OF rstep*] **obtain** $C\ \sigma\ l\ r$
where R: $(l,r) \in R$ **and** s: $C\langle l \cdot \sigma\rangle = s$ **and** t: $C\langle r \cdot \sigma\rangle = t$ **by** *fast*
show *?thesis* **proof** (*cases C*)
case *Hole*
from $s\ t$ **have** $l \cdot \sigma = s$ **and** $r \cdot \sigma = t$ **by** (*auto simp*: *Hole*)
with R **show** *?thesis* **by** (*rule root*)
next
case (*More f ss1 D ss2*)
let *?u* = $D\langle l \cdot \sigma\rangle$
let *?v* = $D\langle r \cdot \sigma\rangle$
have $s = Fun\ f\ (ss1\ @\ ?u\ \#\ ss2)$ **by** (*simp add*: *More s*[*symmetric*])
moreover have $t = Fun\ f\ (ss1\ @\ ?v\ \#\ ss2)$ **by** (*simp add*: *More t*[*symmetric*])
moreover have (*?u*,*?v*) ∈ *rstep R* **using** R **by** *auto*
ultimately show *?thesis* **by** (*rule nonroot*)
qed
qed

lemma *NF-Var*: **assumes** *wf*: *wf-trs R* **shows** $(Var\ x, t) \notin rstep\ R$
proof
assume $(Var\ x, t) \in rstep\ R$
from *rstep-imp-C-s-r*[*OF this*] **obtain** $C\ l\ r\ \sigma$
where R: $(l,r) \in R$ **and** *lhs*: $Var\ x = C\langle l \cdot \sigma\rangle$ **by** *fast*
from *lhs* **have** $Var\ x = l \cdot \sigma$ **by** (*induct C*) *auto*
then obtain y **where** l: $l = Var\ y$ **by** (*induct l*) *auto*
from *wf R* **obtain** *f ss* **where** $l = Fun\ f\ ss$ **unfolding** *wf-trs-def* **by** *best*
with l **show** *False* **by** *simp*
qed

lemma *rstep-cases-Fun'*[*consumes 2*, *case-names root nonroot*]:
assumes *wf*: *wf-trs R*
and *rstep*: $(Fun\ f\ ss,t) \in rstep\ R$
and *root'*: $\bigwedge ls\ r\ \sigma.\ (Fun\ f\ ls,r) \in R \Longrightarrow map\ (\lambda t.\ t\cdot\sigma)\ ls = ss \Longrightarrow r\cdot\sigma = t \Longrightarrow P$
and *nonroot'*: $\bigwedge i\ u.\ i < length\ ss \Longrightarrow t = Fun\ f\ (take\ i\ ss@u\#drop\ (Suc\ i)\ ss) \Longrightarrow (ss!i,u) \in rstep\ R \Longrightarrow P$
shows P
using *rstep* **proof** (*cases rule*: *rstep-cases'*)
case (*root l r σ*)
with *wf* **obtain** *g ls* **where** l: $l = Fun\ g\ ls$ **unfolding** *wf-trs-def* **by** *best*
from *root* **have** [*simp*]: $g = f$ **unfolding** l **by** *simp*
from *root* **have** $(Fun\ f\ ls,r) \in R$ **and** $map\ (\lambda t.\ t\cdot\sigma)\ ls = ss$ **and** $r\cdot\sigma = t$ **unfolding**

```
l by auto
  then show ?thesis by (rule root')
next
  case (nonroot g ss1 u ss2 v)
  then have [simp]: g = f and args: ss = ss1 @ u # ss2 by auto
  let ?i = length ss1
  from args have ss1: take ?i ss = ss1 by simp
  from args have drop ?i ss = u # ss2 by simp
  then have drop (Suc 0) (drop ?i ss) = ss2 by simp
  then have ss2: drop (Suc ?i) ss = ss2 by simp
  from args have len: ?i < length ss by simp
  from id-take-nth-drop[OF len] have ss = take ?i ss @ ss!?i # drop (Suc ?i) ss
by simp
 then have u: ss!?i = u unfolding args unfolding ss1[unfolded args] ss2[unfolded
args] by simp
  from nonroot have t = Fun f (take ?i ss@v#drop (Suc ?i) ss) unfolding ss1
ss2 by simp
  moreover from nonroot have (ss!?i,v) ∈ rstep R unfolding u by simp
  ultimately show ?thesis by (rule nonroot'[OF len])
qed

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 −
  from ‹wf-trs R› and ‹(Fun f ss, t) ∈ rstep R› show ?thesis
  proof (cases rule: rstep-cases-Fun')
    case (root ls r σ)
    then have defined R (f, length ss) by (auto simp: defined-def)
    with ‹¬ defined R (f, length ss)› show ?thesis by simp
  next
    case (nonroot i u) then show ?thesis by simp
  qed
qed

lemma rstep-ctxt-imp-nrrstep: assumes step: (s,t) ∈ rstep R and C: C ≠ □
shows (C⟨s⟩,C⟨t⟩) ∈ nrrstep R
proof −
  from step obtain l r D σ where (l,r) ∈ R s = D⟨l · σ⟩ t = D⟨r · σ⟩ by auto
  thus ?thesis unfolding nrrstep-def' using C
    by (intro CollectI, unfold split, intro exI[of - C ∘c D] exI conjI, auto) (cases
C, auto)
qed

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

from *rstep-ctxt-imp-nrrstep*[*OF step*(*2*) *C*] *step*(*3*) **show** *?case* **by** *simp*
qed *simp*

lemma *nrrstep-mono*:
assumes $R \subseteq R'$
shows *nrrstep* $R \subseteq$ *nrrstep* R'
using *assms* **by** (*force simp*: *nrrstep-def rstep-r-p-s-def Let-def*)

lemma *rrstepE*:
assumes $(s, t) \in$ *rrstep* R
obtains l **and** r **and** σ **where** $(l, r) \in R$ **and** $s = l \cdot \sigma$ **and** $t = r \cdot \sigma$
using *assms* **by** (*auto simp*: *rrstep-def rstep-r-p-s-def*)

lemma *nrrstepE*:
assumes $(s, t) \in$ *nrrstep* R
obtains C **and** l **and** r **and** σ **where** $C \neq \Box$ **and** $(l, r) \in R$
and $s = C\langle l \cdot \sigma \rangle$ **and** $t = C\langle r \cdot \sigma \rangle$
using *assms* **by** (*auto simp*: *nrrstep-def rstep-r-p-s-def Let-def*)
(*metis ctxt.cop-nil list.discI poss-Cons-poss replace-at-subt-at subt-at-id-imp-eps*)

lemma *singleton-subst-restrict* [*simp*]:
subst x s $|s$ $\{x\}$ = *subst x s*
unfolding *subst-def subst-restrict-def* **by** (*rule ext*) *simp*

lemma *singleton-subst-map* [*simp*]:
$f \circ$ *subst x s* = $(f \circ Var)(x := f\ s)$ **by** (*intro ext*, *auto simp*: *subst-def*)

lemma *subst-restrict-vars* [*simp*]:
$(\lambda z.$ *if* $z \in V$ *then* $f\ z$ *else* $g\ z)$ $|s$ $V = f$ $|s$ V
unfolding *subst-restrict-def*
proof (*intro ext*)
fix x
show (*if* $x \in V$ *then if* $x \in V$ *then* $f\ x$ *else* $g\ x$ *else* $Var\ x$)
= (*if* $x \in V$ *then* $f\ x$ *else* $Var\ x$) **by** *simp*
qed

lemma *subst-restrict-restrict* [*simp*]:
assumes $V \cap W = \{\}$
shows $((\lambda z.$ *if* $z \in V$ *then* $f\ z$ *else* $g\ z)$ $|s$ $W) = g$ $|s$ W
unfolding *subst-restrict-def*
proof (*intro ext*)
fix x
show (*if* $x \in W$ *then if* $x \in V$ *then* $f\ x$ *else* $g\ x$ *else* $Var\ x$)
= (*if* $x \in W$ *then* $g\ x$ *else* $Var\ x$) **using** *assms* **by** *auto*
qed

lemma *rstep-rstep*: *rstep* (*rstep* R) = *rstep* R
proof −

have *ctxt.closure* (*subst.closure* (*rstep R*)) = *rstep R* **by** (*simp only*: *subst-closure-rstep-eq ctxt-closure-rstep-eq*)
then show *?thesis* **unfolding** *rstep-eq-closure* **.**
qed

lemma *rstep-trancl-distrib*: *rstep* (R^+) ⊆ (*rstep R*)$^+$
proof
fix *s t*
assume (*s*,*t*) ∈ *rstep* (R^+)
then show (*s*,*t*) ∈ (*rstep R*)$^+$
proof
fix *l r C σ*
presume *lr*: (*l*,*r*) ∈ R^+ **and** *s*: *s* = $C\langle l \cdot \sigma\rangle$ **and** *t*: *t* = $C\langle r \cdot \sigma\rangle$
from *lr* **have** ($C\langle l \cdot \sigma\rangle$, $C\langle r \cdot \sigma\rangle$) ∈ (*rstep R*)$^+$
proof(*induct*)
case (*base r*)
then show *?case* **by** *auto*
next
case (*step r rr*)
from *step*(*2*) **have** ($C\langle r \cdot \sigma\rangle$, $C\langle rr \cdot \sigma\rangle$) ∈ (*rstep R*) **by** *auto*
with *step*(*3*) **show** *?case* **by** *auto*
qed
then show (*s*,*t*) ∈ (*rstep R*)$^+$ **unfolding** *s t* **.**
qed *auto*
qed

lemma *rsteps-closed-subst*:
assumes (*s*, *t*) ∈ (*rstep R*)*
shows (*s* · *σ*, *t* · *σ*) ∈ (*rstep R*)*
using *assms* **and** *subst.closed-rtrancl* [*OF subst-closed-rstep*] **by** (*auto simp*: *subst.closed-def*)

lemma *join-subst*:
subst.closed r ⟹ (*s*, *t*) ∈ $r^\downarrow$ ⟹ (*s* · *σ*, *t* · *σ*) ∈ $r^\downarrow$
by (*simp add*: *join-def subst.closedD subst.closed-comp subst.closed-converse subst.closed-rtrancl*)

lemma *join-subst-rstep* [*intro*]:
(*s*, *t*) ∈ (*rstep R*)$^\downarrow$ ⟹ (*s* · *σ*, *t* · *σ*) ∈ (*rstep R*)$^\downarrow$
by (*intro join-subst*, *auto*)

lemma *join-ctxt* [*intro*]:
assumes (*s*, *t*) ∈ (*rstep R*)$^\downarrow$
shows ($C\langle s\rangle$, $C\langle t\rangle$) ∈ (*rstep R*)$^\downarrow$
proof −
from *assms* **obtain** *u* **where** (*s*, *u*) ∈ (*rstep R*)* **and** (*t*, *u*) ∈ (*rstep R*)* **by** *auto*
then have ($C\langle s\rangle$, $C\langle u\rangle$) ∈ (*rstep R*)* **and** ($C\langle t\rangle$, $C\langle u\rangle$) ∈ (*rstep R*)* **by** (*auto intro*: *rsteps-closed-ctxt*)

then show *?thesis* **by** *blast*
qed

lemma *rstep-simps*:
rstep $(R^{=}) = (rstep\ R)^{=}$
rstep $(rstep\ R) = rstep\ R$
rstep $(R \cup S) = rstep\ R \cup rstep\ S$
rstep Id = *Id*
rstep $(R^{\leftrightarrow}) = (rstep\ R)^{\leftrightarrow}$
by *auto*

lemma *rstep-rtrancl-idemp* [*simp*]:
rstep $((rstep\ R)^*) = (rstep\ R)^*$
proof −
{ fix *s t*
assume $(s, t) \in rstep\ ((rstep\ R)^*)$
then have $(s, t) \in (rstep\ R)^*$
by (*induct*) (*metis rsteps-closed-ctxt rsteps-closed-subst*) **}**
then show *?thesis* **by** *auto*
qed

lemma *all-ctxt-closed-rstep-conversion*:
all-ctxt-closed UNIV $((rstep\ R)^{\leftrightarrow *})$
unfolding *conversion-def rstep-simps*(*5*)[*symmetric*] **by** *blast*

definition *instance-rule* :: $('f, 'v)$ *rule* ⇒ $('f, 'w)$ *rule* ⇒ *bool* **where**
[*code del*]: *instance-rule lr st* ⟷ $(\exists\ \sigma.\ fst\ lr = fst\ st \cdot \sigma \wedge snd\ lr = snd\ st \cdot \sigma)$

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 −
from *eq*[*unfolded instance-rule-def*]
obtain τ **where** *s*: $s = l \cdot \tau$ **and** *t*: $t = r \cdot \tau$ **by** *auto*
show *?thesis*
proof
fix *x*
assume *x* ∈ *vars-term t*
from *this*[*unfolded t*] **have** *x* ∈ *vars-term* $(l \cdot \tau)$ **using** *vars* **unfolding** *vars-term-subst* **by** *auto*
then show *x* ∈ *vars-term s* **unfolding** *s* **by** *auto*
qed

qed

lemma *instance-rule-rstep*: **assumes** *step*: $(s,t) \in rstep\ \{lr\}$
and *bex*: *Bex R* (*instance-rule lr*)
shows $(s,t) \in rstep\ R$
proof −
from *bex* **obtain** lr' **where** *inst*: *instance-rule lr lr'* **and** *R*: $lr' \in R$ **by** *auto*
obtain *l r* **where** *lr*: $lr = (l,r)$ **by** *force*
obtain $l'\ r'$ **where** *lr'*: $lr' = (l',r')$ **by** *force*
note *inst* = *inst*[*unfolded lr lr'*]
note *R* = *R*[*unfolded lr'*]
from *inst*[*unfolded instance-rule-def*] **obtain** σ **where** *l*: $l = l' \cdot \sigma$ **and** *r*: $r = r' \cdot \sigma$ **by** *auto*
from *step*[*unfolded lr*] **obtain** *C* τ **where** $s = C\ \langle l \cdot \tau \rangle\ t = C\ \langle r \cdot \tau \rangle$ **by** *auto*
with *l r* **have** *s*: $s = C\langle l' \cdot (\sigma \circ_s \tau)\rangle$ **and** *t*: $t = C\langle r' \cdot (\sigma \circ_s \tau)\rangle$ **by** *auto*
from *rstepI*[*OF R s t*] **show** *?thesis* **.**
qed

lemma *eq-rule-mod-vars-var-cond*: **assumes** *eq*: $(l,r) =_v (s,t)$
and *vars*: *vars-term r* $\subseteq$ *vars-term l*
shows *vars-term t* $\subseteq$ *vars-term s*
by (*rule instance-rule-var-cond*[*OF - vars*], *insert eq*[*unfolded eq-rule-mod-vars-def*], *auto*)

lemma *eq-rule-mod-varsE*[*elim*]: **fixes** $l :: ('f,'v)term$
assumes $(l,r) =_v (s,t)$
shows $\exists\ \sigma\ \tau.\ l = s \cdot \sigma \wedge r = t \cdot \sigma \wedge s = l \cdot \tau \wedge t = r \cdot \tau \wedge range\ \sigma \subseteq range\ Var \wedge range\ \tau \subseteq range\ Var$
proof −
from *assms*[*unfolded eq-rule-mod-vars-def instance-rule-def fst-conv snd-conv*]
obtain $\sigma\ \tau$ **where** *l*: $l = s \cdot \sigma$ **and** *r*: $r = t \cdot \sigma$ **and** *s*: $s = l \cdot \tau$ **and** *t*: $t = r \cdot \tau$ **by** *blast+*
obtain $f :: 'f$ **where** *True* **by** *auto*
let *?vst* = *vars-term* (*Fun f* [*s*,*t*])
let *?vlr* = *vars-term* (*Fun f* [*l*,*r*])
define σ' **where** $\sigma' \equiv \lambda\ x.\ if\ x \in ?vst\ then\ \sigma\ x\ else\ Var\ x$
define τ' **where** $\tau' \equiv \lambda\ x.\ if\ x \in ?vlr\ then\ \tau\ x\ else\ Var\ x$
show *?thesis*
proof (*intro exI conjI*)
show *l*: $l = s \cdot \sigma'$ **unfolding** *l* σ'*-def*
by (*rule term-subst-eq*, *auto*)
show *r*: $r = t \cdot \sigma'$ **unfolding** *r* σ'*-def*
by (*rule term-subst-eq*, *auto*)
show *s*: $s = l \cdot \tau'$ **unfolding** *s* τ'*-def*
by (*rule term-subst-eq*, *auto*)
show *t*: $t = r \cdot \tau'$ **unfolding** *t* τ'*-def*
by (*rule term-subst-eq*, *auto*)
have *Fun f* [*s*,*t*] $\cdot$ *Var* = *Fun f* [*l*, *r*] $\cdot\ \tau'$ **unfolding** *s t* **by** *simp*
also have $\ldots$ = *Fun f* [*s*,*t*] $\cdot\ (\sigma' \circ_s \tau')$ **unfolding** *l r* **by** *simp*

finally have *Fun f* [*s*,*t*] · (σ′ ∘$_s$ τ′) = *Fun f* [*s*,*t*] · *Var* **by** *simp*
from *term-subst-eq-rev*[*OF this*] **have** *vst*: ⋀ *x*. *x* ∈ *?vst* ⟹ σ′ *x* · τ′ = *Var* *x* **unfolding** *subst-compose-def* **by** *auto*
have *Fun f* [*l*,*r*] · *Var* = *Fun f* [*s*, *t*] · σ′ **unfolding** *l r* **by** *simp*
also have ... = *Fun f* [*l*,*r*] · (τ′ ∘$_s$ σ′) **unfolding** *s t* **by** *simp*
finally have *Fun f* [*l*,*r*] · (τ′ ∘$_s$ σ′) = *Fun f* [*l*,*r*] · *Var* **by** *simp*
from *term-subst-eq-rev*[*OF this*] **have** *vlr*: ⋀ *x*. *x* ∈ *?vlr* ⟹ τ′ *x* · σ′ = *Var x* **unfolding** *subst-compose-def* **by** *auto*
{
fix *x*
have σ′ *x* ∈ *range Var*
proof (*cases x* ∈ *?vst*)
case *True*
from *vst*[*OF this*] **show** *?thesis* **by** (*cases* σ′ *x*, *auto*)
next
case *False*
then show *?thesis* **unfolding** σ′*-def* **by** *auto*
qed
}
then show *range* σ′ ⊆ *range Var* **by** *auto*
{
fix *x*
have τ′ *x* ∈ *range Var*
proof (*cases x* ∈ *?vlr*)
case *True*
from *vlr*[*OF this*] **show** *?thesis* **by** (*cases* τ′ *x*, *auto*)
next
case *False*
then show *?thesis* **unfolding** τ′*-def* **by** *auto*
qed
}
then show *range* τ′ ⊆ *range Var* **by** *auto*
qed
qed

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*
unfolding *linear-trs-def* **by** *auto*

lemma *linear-trsI*[*intro*]: ⟦ ⋀ *l r*. (*l*,*r*) ∈ *R* ⟹ *linear-term l* ∧ *linear-term r*⟧ ⟹ *linear-trs R*
unfolding *linear-trs-def* **by** *auto*

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*)
unfolding *left-linear-trs-def* **by** *auto*

lemma *left-linear-mono*: **assumes** *left-linear-trs S* **and** $R \subseteq S$ **shows** *left-linear-trs R*
using *assms* **unfolding** *left-linear-trs-def* **by** *auto*

lemma *left-linear-map-funs-trs*[*simp*]: *left-linear-trs* (*map-funs-trs f R*) = *left-linear-trs R*
unfolding *left-linear-trs-def* **by** (*auto simp*: *map-funs-trs.simps*)

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*
using *weak-match*
proof (*induct rule*: *rstep-induct-rule* [*OF rstep*])
case (*1 C sig l r*)
from *1*(*2*) **show** *?case*
proof (*induct C arbitrary*: *s*)
case (*More f bef C aft s*)
let *?n* = *Suc* (*length bef* + *length aft*)
let *?m* = *length bef*
from *More*(*2*) **obtain** *ss* **where** *s*: *s* = *Fun f ss* **and** *lss*: *?n* = *length ss* **and** *wm*: $(\forall i<$*length ss. weak-match* (*ss* ! *i*) ((*bef* @ *C*$\langle l \cdot sig \rangle$ # *aft*) ! *i*)) **by** (*cases s*, *auto*)
from *lss wm*[*THEN spec*, *of ?m*] **have** *weak-match* (*ss* ! *?m*) *C*$\langle l \cdot sig \rangle$ **by** *auto*
from *More*(*1*)[*OF this*] **obtain** *t* **where** *wmt*: *weak-match t C*$\langle r \cdot sig \rangle$ **and** *step*: (*ss* ! *?m*,*t*) $\in$ *rstep R* **by** *auto*
from *lss* **have** *mss*: *?m* < *length ss* **by** *simp*
let *?tsi* = λ *t. take ?m ss* @ *t* # *drop* (*Suc ?m*) *ss*
let *?ts* = *?tsi t*
let *?ss* = *?tsi* (*ss* ! *?m*)
from *id-take-nth-drop*[*OF mss*]
have *lts*: *length ?ts* = *?n* **using** *lss* **by** *auto*
show *?case*
proof (*rule exI*[*of* - *Fun f ?ts*], *intro conjI*)
have *weak-match* (*Fun f ?ts*) (*More f bef C aft*)$\langle r \cdot sig \rangle$ =
weak-match (*Fun f ?ts*) (*Fun f* (*bef* @ *C*$\langle r \cdot sig \rangle$ # *aft*)) **by** *simp*
also have ... **proof** (*unfold weak-match.simps lts*, *intro conjI refl allI impI*)
fix *i*

assume *i*: $i < ?n$
show *weak-match* (*?ts* ! *i*) ((*bef* @ $C\langle r \cdot sig\rangle$ # *aft*) ! *i*)
proof (*cases* $i = ?m$)
case *True*
have *weak-match* (*?ts* ! *i*) ((*bef* @ $C\langle r \cdot sig\rangle$ # *aft*) ! *i*) = *weak-match t* $C\langle r \cdot sig\rangle$
using *True mss* **by** (*simp add*: *nth-append*)
then show *?thesis* **using** *wmt* **by** *simp*
next
case *False*
have *eq*: *?ts* ! *i* = *ss* ! *i* ∧ (*bef* @ $C\langle r \cdot sig\rangle$ # *aft*) ! *i* = (*bef* @ $C\langle l \cdot sig\rangle$ # *aft*) ! *i*
proof (*cases* $i < ?m$)
case *True*
then show *?thesis* **by** (*simp add*: *nth-append lss*[*symmetric*])
next
case *False*
with ‹$i \neq ?m$› *i* **have** ∃ *j*. $i = Suc\ (?m + j) \wedge j < length\ aft$ **by** *presburger*
then obtain *j* **where** *i*: $i = Suc\ (?m + j)$ **and** *j*: $j < length\ aft$ **by** *auto*
then have *id*: $(Suc\ (length\ bef + j) - min\ (Suc\ (length\ bef + length\ aft))\ (length\ bef)) = Suc\ j$ **by** *simp*
from *j* **show** *?thesis* **by** (*simp add*: *nth-append i id lss*[*symmetric*])
qed
then show *?thesis* **using** *wm*[*THEN spec*, *of i*] *i*[*unfolded lss*] **by** (*simp*)
qed
qed *simp*
finally show *weak-match* (*Fun f ?ts*) (*More f bef C aft*)$\langle r \cdot sig\rangle$ **by** *simp*
next
have $s = Fun\ f\ ?ss$ **unfolding** *s* **using** *id-take-nth-drop*[*OF mss*, *symmetric*] **by** *simp*
also have ... = (*More f* (*take ?m ss*) □ (*drop* (*Suc ?m*) *ss*))$\langle$(*ss* ! *?m*)$\rangle$ (**is** - = *?C*$\langle$-$\rangle$) **by** *simp*
finally have *s*: $s = ?C\langle ss\ !\ ?m\rangle$ **.**
have *t*: $Fun\ f\ ?ts = ?C\langle t\rangle$ **by** *simp*
from *rstep-ctxt*[*OF step*]
show (*s*, *Fun f ?ts*) ∈ *rstep R*
unfolding *s t* **.**
qed
next
case (*Hole s*)
from *ll 1*(*1*) **have** *linear-term l* **unfolding** *left-linear-trs-def* **by** *auto*
from *linear-weak-match*[*OF this Hole*[*simplified*] *refl*] **obtain** τ **where**
$s = l \cdot \tau$ **and** (∀ $x \in vars\text{-}term\ l$. *weak-match* ($Var\ x \cdot \tau$) ($Var\ x \cdot sig$))
by *auto*
then obtain *tau* **where** *s*: $s = l \cdot tau$ **and** *wm*: (∀ $x \in vars\text{-}term\ l$. *weak-match* (*tau x*) ($Var\ x \cdot sig$))
by (*auto*)
let *?delta* = (λ *x*. *if* $x \in vars\text{-}term\ l$ *then tau x else* $Var\ x \cdot sig$)

```
    show ?case
    proof (rule exI[of - r · ?delta], rule conjI)
      have s = l · (tau |s (vars-term l)) unfolding s by (rule coincidence-lemma)
      also have ... = l · (?delta  |s (vars-term l)) by simp
      also have ... = l · ?delta by (rule coincidence-lemma[symmetric])
      finally have s: s = l · ?delta .
      from 1(1) have step: (l · ?delta, r · ?delta) ∈ rstep R by auto
      then show (s, r · ?delta) ∈ rstep R unfolding s .
    next
      have weak-match (r · ?delta) (r · sig)
      proof (induct r)
        case (Fun f ss)
        from this[unfolded set-conv-nth]
        show ?case by (force)
      next
        case (Var x)
        show ?case
        proof (cases x ∈ vars-term l)
          case True
          with wm Var show ?thesis by simp
        next
          case False
          show ?thesis by (simp add: Var False weak-match-refl)
        qed
      qed
      then show weak-match (r · ?delta) (□ ⟨(r · sig)⟩) by simp
    qed
  qed
qed

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
  fix f
  assume t0: f 0 ∈ {t} and chain: chain (rstep R) f
  let ?s = S R s f
  let ?P = λi u. (?s i, u) ∈ rstep R ∧ weak-match u (f (Suc i))
  have ∀ i. (?s i, ?s (Suc i)) ∈ rstep R ∧ weak-match (?s (Suc i)) (f (Suc i))
  proof
```

```
    fix i show (?s i, ?s (Suc i)) ∈ rstep R ∧ weak-match (?s (Suc i)) (f (Suc i))
    proof (induct i)
      case 0
      from chain have ini: (f 0, f (Suc 0)) ∈ rstep R by simp
     then have (t, f (Suc 0)) ∈ rstep R unfolding singletonD[OF t0, symmetric]
.
      from someI-ex[OF left-linear-weak-match-rstep[OF this wm ll]]
      show ?case by simp
    next
      case (Suc i)
      then have IH1: (?s i, ?s (Suc i)) ∈ rstep R
        and IH2: weak-match (?s (Suc i)) (f (Suc i)) by auto
      from chain have nxt: (f (Suc i), f (Suc (Suc i))) ∈ rstep R by simp
      from someI-ex[OF left-linear-weak-match-rstep[OF this IH2 ll]]
      have ∃ u. ?P (Suc i) u by auto
      from someI-ex[OF this]
      show ?case by simp
    qed
  qed
  moreover have ?s 0 = s by simp
  ultimately have ¬ SN-on (rstep R) {s} by best
  with SN show False by simp
qed
end

lemma lhs-notin-NF-rstep: (l, r) ∈ R ⟹ l ∉ NF (rstep R) by auto

lemma NF-instance:
  assumes (t · σ) ∈ NF (rstep R) shows t ∈ NF (rstep R)
  using assms by auto

lemma NF-subterm:
  assumes t ∈ NF (rstep R) and t ⊵ s
  shows s ∈ NF (rstep R)
proof (rule ccontr)
  assume ¬ ?thesis
  then obtain u where (s, u) ∈ rstep R by auto
  from ‹t ⊵ s› obtain C where t = C⟨s⟩ by auto
  with ‹(s, u) ∈ rstep R› have (t, C⟨u⟩) ∈ rstep R by auto
  then have t ∉ NF (rstep R) by auto
  with assms show False by simp
qed

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*
unfolding *map-funs-trs-wa-def* **by** *auto*

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*
by (*induct t*, *auto*)

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 −
have *map-funs-trs-wa ?fgh R* = {(*map-funs-term-wa ?fgh l*, *map-funs-term-wa ?fgh r*)| *l r*. (*l*,*r*) ∈ *R*} **unfolding** *map-funs-trs-wa-def* **by** *auto*
also have ... = {(*map-funs-term-wa gh* (*map-funs-term-wa fg l*), *map-funs-term-wa gh* (*map-funs-term-wa fg r*)) | *l r*. (*l*,*r*) ∈ *R*} **unfolding** *map-funs-term-wa-compose*
..
finally show *?thesis* **unfolding** *map-funs-trs-wa-def* **by** *force*
qed

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 −
{
fix *l r*
assume (*l*,*r*) ∈ *R*
with *R* **have** *l*: *funas-term l* ⊆ *F* **and** *r*: *funas-term r* ⊆ *F* **unfolding** *funas-trs-def*
by (*force simp*: *funas-rule-def*)+
from *map-funs-term-wa-funas-term-id*[*OF l id*] *map-funs-term-wa-funas-term-id*[*OF r id*]
have *map-funs-term-wa f l* = *l map-funs-term-wa f r* = *r* **by** *auto*
} **note** *main* = *this*
have *map-funs-trs-wa f R* = {(*map-funs-term-wa f l*, *map-funs-term-wa f r*) | *l r*. (*l*,*r*) ∈ *R*}
unfolding *map-funs-trs-wa-def* **by** *force*
also have ... = *R* **using** *main* **by** *force*
finally show *?thesis* .
qed

lemma *map-funs-trs-wa-rstep*: **assumes** *step*:$(s,t) \in$ *rstep R*
shows (*map-funs-term-wa fg s*,*map-funs-term-wa fg t*) $\in$ *rstep* (*map-funs-trs-wa fg R*)
using *step*
proof (*induct*)
case (*IH C* σ *l r*)
show *?case* **unfolding** *map-funs-trs-wa-def*
by (*rule rstepI*[**where** *l* = *map-funs-term-wa fg l* **and** *r* = *map-funs-term-wa fg r* **and** *C* = *map-funs-ctxt-wa fg C*], *auto simp*: *IH*)
qed

lemma *map-funs-trs-wa-rsteps*: **assumes** *step*:$(s,t) \in$ (*rstep R*)*
shows (*map-funs-term-wa fg s*,*map-funs-term-wa fg t*) $\in$ (*rstep* (*map-funs-trs-wa fg R*))*
using *step*
proof (*induct*)
case (*step a b*)
from *map-funs-trs-wa-rstep*[*OF step*(*2*), *of fg*] *step*(*3*) **show** *?case* **by** *auto*
qed *auto*

lemma *rstep-ground*:
assumes *wf-trs*: $\bigwedge l\ r.\ (l,\ r) \in R \Longrightarrow$ *vars-term r* $\subseteq$ *vars-term l*
and *ground*: *ground s*
and *step*: $(s,\ t) \in$ *rstep R*
shows *ground t*
using *step ground*
proof (*induct*)
case (*IH C* σ *l r*)
from *wf-trs*[*OF IH*(*1*)] *IH*(*2*)
show *?case* **by** *auto*
qed

lemma *rsteps-ground*:
assumes *wf-trs*: $\bigwedge l\ r.\ (l,\ r) \in R \Longrightarrow$ *vars-term r* $\subseteq$ *vars-term l*
and *ground*: *ground s*
and *steps*: $(s,\ t) \in$ (*rstep R*)*
shows *ground t*
using *steps ground*
by (*induct*, *insert rstep-ground*[*OF wf-trs*], *auto*)

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

definition *non-collapsing R* $\longleftrightarrow$ ($\forall$ *lr* $\in$ *R*. *is-Fun* (*snd lr*))

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

assume $s \rhd t$ **show** *?thesis*
proof (*rule UnI1*)
from ‹$s \rhd t$› **show** $s \cdot \sigma \rhd t \cdot \sigma$ **by** (*rule supt-subst*)
qed
next
assume $(s, t) \in$ *rstep* R **show** *?thesis*
proof (*rule UnI2*)
from ‹$(s, t) \in$ *rstep* R› **show** $(s \cdot \sigma, t \cdot \sigma) \in$ *rstep* R **..**
qed
qed

lemma *supt-rstep-trancl-stable*:
assumes $(s, t) \in (\{\rhd\} \cup$ *rstep* $R)^+$
shows $(s \cdot \sigma, t \cdot \sigma) \in (\{\rhd\} \cup$ *rstep* $R)^+$
using *assms* **proof** (*induct*)
case (*base u*)
then have $(s \cdot \sigma, u \cdot \sigma) \in \{\rhd\} \cup$ *rstep* R **by** (*rule supt-rstep-stable*)
then show *?case* **..**
next
case (*step u v*)
from ‹$(s \cdot \sigma, u \cdot \sigma) \in (\{\rhd\} \cup$ *rstep* $R)^+$›
and *supt-rstep-stable*[*OF* ‹$(u, v) \in \{\rhd\} \cup$ *rstep* R›, *of* σ]
show *?case* **..**
qed

lemma *supt-rsteps-stable*:
assumes $(s, t) \in (\{\rhd\} \cup$ *rstep* $R)^*$
shows $(s \cdot \sigma, t \cdot \sigma) \in (\{\rhd\} \cup$ *rstep* $R)^*$
using *assms*
proof (*induct*)
case *base* **then show** *?case* **..**
next
case (*step u v*)
from ‹$(s, u) \in (\{\rhd\} \cup$ *rstep* $R)^*$› **and** ‹$(u, v) \in \{\rhd\} \cup$ *rstep* R›
have $(s, v) \in (\{\rhd\} \cup$ *rstep* $R)^+$ **by** (*rule rtrancl-into-trancl1*)
from *trancl-into-rtrancl*[*OF supt-rstep-trancl-stable*[*OF this*]]
show *?case* **.**
qed

lemma *eq-rule-mod-vars-refl*[*simp*]: $r =_v r$
proof (*cases r*)
case (*Pair l r*)
{
have *fst* $(l, r) =$ *fst* $(l, r) \cdot$ *Var* $\wedge$ *snd* $(l, r) =$ *snd* $(l, r) \cdot$ *Var* **by** *auto*
}
then show *?thesis* **unfolding** *Pair eq-rule-mod-vars-def instance-rule-def* **by** *best*
qed

lemma *instance-rule-refl*[*simp*]: *instance-rule r r*
using *eq-rule-mod-vars-refl*[*of r*] **unfolding** *eq-rule-mod-vars-def* **by** *simp*

lemma *is-Fun-Fun-conv*: *is-Fun t* = (∃*f ts*. *t* = *Fun f ts*) **by** *auto*

lemma *wf-trs-def'*:
wf-trs R = (∀ (*l*, *r*)∈*R*. *is-Fun l* ∧ *vars-term r* ⊆ *vars-term l*)
by (*rule iffI*) (*auto simp*: *wf-trs-def is-Fun-Fun-conv*)

definition *wf-rule* :: ('*f*, '*v*) *rule* ⇒ *bool* **where**
wf-rule r ⟷ *is-Fun* (*fst r*) ∧ *vars-term* (*snd r*) ⊆ *vars-term* (*fst r*)

definition *wf-rules* :: ('*f*, '*v*) *trs* ⇒ ('*f*, '*v*) *trs* **where**
wf-rules R = {*r*. *r* ∈ *R* ∧ *wf-rule r*}

lemma *wf-trs-wf-rules*[*simp*]: *wf-trs* (*wf-rules R*)
unfolding *wf-trs-def' wf-rules-def wf-rule-def split-def* **by** *simp*

lemma *wf-rules-subset*[*simp*]: *wf-rules R* ⊆ *R*
unfolding *wf-rules-def* **by** *auto*

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 (*rule ccontr*)
assume ¬ *?thesis*
then obtain *l r* **where** ¬ *wf-trs R* ∨ *R* ≠ {} ∧ (*l*,*r*) ∈ *S* ∧ ¬ *vars-term r* ⊆ *vars-term l* (**is** - ∨ *?two*) **by** *auto*
then show *False*
proof
assume ¬ *wf-trs R*
with *SN-rstep-imp-wf-trs*[*OF SN-rel-imp-SN*[*OF assms*]]
show *False* **by** *simp*
next
assume *?two*
then obtain *ll rr x* **where** *lr*: (*l*,*r*) ∈ *S* **and** *llrr*: (*ll*,*rr*) ∈ *R* **and** *x*: *x* ∈ *vars-term r* **and** *nx*: *x* ∉ *vars-term l* **by** *auto*
obtain *f* **and** *σ*
where *sigma*: *σ* = (*λy*. *if x* = *y then Fun f* [*ll*,*l*] *else Var y*) **by** *auto*
have *id*: *σ* |*s* (*vars-term l*) = *Var* **unfolding** *sigma*
by (*simp add*: *subst-restrict-def*, *rule ext*, *auto simp*: *nx*)
have *l*: *l* = *l* · *σ* **by** (*simp add*: *coincidence-lemma*[*of l σ*] *id*)
have (*l* · *σ*, *r* · *σ*) ∈ *rstep S* **using** *lr* **by** *auto*
with *l* **have** *sstep*: (*l*, *r* · *σ*) ∈ *rstep S* **by** *simp*
from *supteq-subst*[*OF supteq-Var*[*OF x*], *of σ*] **have**

```
    r · σ ⊵ Fun f [ll,l] unfolding sigma by auto
  then obtain C where C⟨Fun f [ll, l]⟩ = r · σ by auto
  with sstep have sstep: (l,C⟨Fun f [ll, l]⟩) ∈ rstep S by simp
  obtain r where r: r = relto (rstep R) (rstep S) ∪ {▷} by auto
  have (C⟨Fun f [ll,l]⟩, C⟨Fun f [rr,l]⟩) ∈ rstep R
    by (intro rstepI[OF llrr, of - C ∘c More f [] □ [l] Var], auto)
  with sstep have relto: (l,C⟨Fun f [rr,l]⟩) ∈ r unfolding r by auto
  have C⟨Fun f [rr,l]⟩ ⊵ Fun f [rr,l] using ctxt-imp-supteq by auto
  also have Fun f [rr,l] ▷ l by auto
  finally have supt: C⟨Fun f [rr,l]⟩ ▷ l unfolding supt-def by simp
  then have (C⟨Fun f [rr,l]⟩, l) ∈ r unfolding r by auto
  with relto have loop: (l, l) ∈ r+ by auto
  have SN r unfolding r
    by (rule SN-imp-SN-union-supt[OF SN-rel[unfolded SN-rel-defs]], blast)
  then have SN (r+) by (rule SN-imp-SN-trancl)
  with loop show False unfolding SN-on-def by auto
 qed
qed

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
  using assms
proof
  fix ml mr C σ
  presume mem: (ml,mr) ∈ ?R and s: s = C⟨ml · σ⟩ and t: t = C⟨mr · σ⟩
  let ?m = map-vars-term f
  from mem obtain l r where mem: (l,r) ∈ R and id: ml = ?m l mr = ?m r
    unfolding map-vars-trs-def by auto
  have id: s · τ = (C ·c τ)⟨?m l · σ ∘s τ⟩ t · τ = (C ·c τ)⟨?m r · σ ∘s τ⟩ by (auto
simp: s t id)
  then show (s · τ, t · τ) ∈ rstep R
    unfolding id apply-subst-map-vars-term
    using mem by auto
qed auto

lemma map-vars-rsteps:
  assumes (s,t) ∈ (rstep (map-vars-trs f R))* (is - ∈ (rstep ?R)*)
  shows (s · τ, t · τ) ∈ (rstep R)*
  using assms
proof (induct)
  case base then show ?case by simp
next
  case (step t u)
```

from *map-vars-trs-rstep*[*OF step*(*2*), *of* τ] *step*(*3*) **show** *?case* **by** *auto*
qed

lemma *rsteps-subst-closed*: $(s,t) \in (rstep\ R)^+ \Longrightarrow (s \cdot \sigma,\ t \cdot \sigma) \in (rstep\ R)^+$
proof −
let *?R* = *rstep R*
assume *steps*: $(s,t) \in ?R^+$
have *subst*: *subst.closed* (*?R*$^+$) **by** (*rule subst.closed-trancl*[*OF subst-closed-rstep*])
from *this*[*unfolded subst.closed-def*] *steps* **show** *?thesis* **by** *auto*
qed

lemma *supteq-rtrancl-supt*:
$(R^+\ O\ \{\trianglerighteq\}) \subseteq (\{\triangleright\} \cup R)^+$ (**is** *?l* ⊆ *?r*)
proof
fix *x z*
assume $(x,z) \in ?l$
then obtain *y* **where** *xy*: $(x,y) \in R^+$ **and** *yz*: $y \trianglerighteq z$ **by** *auto*
from *xy* **have** *xy*: $(x,y) \in ?r$ **by** (*rule trancl-mono*, *simp*)
show $(x,z) \in ?r$
proof (*cases* $y = z$)
case *True*
with *xy* **show** *?thesis* **by** *simp*
next
case *False*
with *yz* **have** *yz*: $(y,z) \in \{\triangleright\} \cup R$ **by** *auto*
with *xy* **have** *xz*: $(x,z) \in ?r\ O\ (\{\triangleright\} \cup R)$ **by** *auto*
then show *?thesis* **by** (*metis UnCI trancl-unfold*)
qed
qed

lemma *rrstepI*[*intro*]: $(l,\ r) \in R \Longrightarrow s = l \cdot \sigma \Longrightarrow t = r \cdot \sigma \Longrightarrow (s,\ t) \in rrstep\ R$
unfolding *rrstep-def′* **by** *auto*

lemma *CS-rrstep-conv*: *subst.closure* = *rrstep*
apply (*intro ext*)
apply (*unfold rrstep-def′*)
apply (*intro subset-antisym*)
by (*insert subst.closure.cases*, *blast*, *auto*)

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*]:$(l,\ r) \in R \Longrightarrow p \in poss\ s \Longrightarrow s\ |\text{-}\ p = l \cdot \sigma \Longrightarrow$
$(s,\ replace\text{-}at\ s\ p\ (r \cdot \sigma)) \in rstep\text{-}pos\ R\ p$

lemma *rstep-pos-subst*:
assumes $(s,\ t) \in rstep\text{-}pos\ R\ p$
shows $(s \cdot \sigma,\ t \cdot \sigma) \in rstep\text{-}pos\ R\ p$
using *assms*

proof (*cases*)
case (*rule l r σ′*)
with *rstep-pos.intros* [*OF this*(*2*), *of p s* · *σ σ′* ∘$_s$ *σ*]
show *?thesis* **by** (*auto simp*: *ctxt-of-pos-term-subst*)
qed

lemma *rstep-pos-rule*:
assumes (*l*, *r*) ∈ *R*
shows (*l*, *r*) ∈ *rstep-pos R* []
using *rstep-pos.intros* [*OF assms*, *of* [] *l Var*] **by** *simp*

lemma *rstep-pos-rstep-r-p-s-conv*:
rstep-pos R p = {(*s*, *t*) | *s t r σ*. (*s*, *t*) ∈ *rstep-r-p-s R r p σ*}
by (*auto simp*: *rstep-r-p-s-def Let-def subt-at-ctxt-of-pos-term*
intro: *replace-at-ident*
elim!: *rstep-pos.cases*)

lemma *rstep-rstep-pos-conv*:
rstep R = {(*s*, *t*) | *s t p*. (*s*, *t*) ∈ *rstep-pos R p*}
by (*force simp*: *rstep-pos-rstep-r-p-s-conv rstep-iff-rstep-r-p-s*)

lemma *rstep-pos-supt*:
assumes (*s*, *t*) ∈ *rstep-pos R p*
and *q*: *q* ∈ *poss u* **and** *u*: *u* |- *q* = *s*
shows (*u*, (*ctxt-of-pos-term q u*)⟨*t*⟩) ∈ *rstep-pos R* (*q* @ *p*)
using *assms*
proof (*cases*)
case (*rule l r σ*)
with *q* **and** *u* **have** (*q* @ *p*) ∈ *poss u* **and** *u* |- (*q* @ *p*) = *l* · *σ* **by** *auto*
with *rstep-pos.rule* [*OF rule*(*2*) *this*] **show** *?thesis*
unfolding *rule* **by** (*auto simp*: *ctxt-of-pos-term-append u*)
qed

lemma *rrstep-rstep-pos-conv*:
rrstep R = *rstep-pos R* []
by (*auto simp*: *rrstep-def rstep-pos-rstep-r-p-s-conv*)

lemma *rrstep-imp-rstep*:
assumes (*s*, *t*) ∈ *rrstep R*
shows (*s*, *t*) ∈ *rstep R*
using *assms* **by** (*auto simp*: *rrstep-def rstep-iff-rstep-r-p-s*)

lemma *not-NF-rstep-imp-subteq-not-NF-rrstep*:
assumes *s* ∉ *NF* (*rstep R*)
shows ∃ *t* ◁ *s*. *t* ∉ *NF* (*rrstep R*)
proof −
from *assms* **obtain** *u* **where** (*s*, *u*) ∈ *rstep R* **by** *auto*
then obtain *l r C σ* **where** (*l*, *r*) ∈ *R* **and** *s*: *s* = *C*⟨*l* · *σ*⟩ **and** *u*: *u* = *C*⟨*r* ·
σ⟩ **by** *auto*

then have $(l \cdot \sigma,\ r \cdot \sigma) \in rrstep\ R$ **and** $l \cdot \sigma \trianglelefteq s$ **by** *auto*
then show *?thesis* **by** *blast*
qed

lemma *all-subt-NF-rrstep-iff-all-subt-NF-rstep*:
$(\forall\, s \lhd t.\ s \in NF\ (rrstep\ R)) \longleftrightarrow (\forall\, s \lhd t.\ s \in NF\ (rstep\ R))$
by (*auto dest*: *rrstep-imp-rstep supt-supteq-trans not-NF-rstep-imp-subteq-not-NF-rrstep*)

lemma *not-in-poss-imp-NF-rstep-pos* [*simp*]:
assumes $p \notin poss\ s$
shows $s \in NF\ (rstep\text{-}pos\ R\ p)$
using *assms* **by** (*auto simp*: *NF-def elim*: *rstep-pos.cases*)

lemma *Var-rstep-imp-rstep-pos-Empty*:
assumes $(Var\ x,\ t) \in rstep\ R$
shows $(Var\ x,\ t) \in rstep\text{-}pos\ R\ []$
using *assms* **by** (*metis Var-supt nrrstep-subt rrstep-rstep-pos-conv rstep-cases*)

lemma *rstep-args-NF-imp-rrstep*:
assumes $(s,\ t) \in rstep\ R$
and $\forall\, u \lhd s.\ u \in NF\ (rstep\ R)$
shows $(s,\ t) \in rrstep\ R$
using *assms* **by** (*metis NF-iff-no-step nrrstep-subt rstep-cases*)

lemma *rstep-pos-imp-rstep-pos-Empty*:
assumes $(s,\ t) \in rstep\text{-}pos\ R\ p$
shows $(s\ |\text{-}\ p,\ t\ |\text{-}\ p) \in rstep\text{-}pos\ R\ []$
using *assms* **by** (*cases*) (*auto simp*: *replace-at-subt-at intro*: *rstep-pos-rule rstep-pos-subst*)

lemma *rstep-pos-arg*:
assumes $(s,\ t) \in rstep\text{-}pos\ R\ p$
and $i < length\ ss$ **and** $ss\ !\ i = s$
shows $(Fun\ f\ ss,\ (ctxt\text{-}of\text{-}pos\text{-}term\ [i]\ (Fun\ f\ ss))\langle t\rangle) \in rstep\text{-}pos\ R\ (i\ \#\ p)$
using *assms*
by *cases* (*auto simp*: *rstep-pos.simps*)

lemma *rstep-imp-max-pos*:
assumes $(s,\ t) \in rstep\ R$
shows $\exists\, u.\ \exists\, p{\in}poss\ s.\ (s,\ u) \in rstep\text{-}pos\ R\ p \wedge (\forall\, v \lhd s\ |\text{-}\ p.\ v \in NF\ (rstep\ R))$
using *assms*
proof (*induction s arbitrary*: *t*)
case (*Var x*)
from *Var-rstep-imp-rstep-pos-Empty* [*OF this*] **show** *?case* **by** *auto*
next
case (*Fun f ss*)
show *?case*
proof (*cases* $\forall\, v \lhd Fun\ f\ ss\ |\text{-}\ []\ .\ v \in NF\ (rstep\ R)$)
case *True*
moreover with *Fun.prems*

have (*Fun f ss*, *t*) ∈ *rstep-pos R* []
by (*auto dest*: *rstep-args-NF-imp-rrstep simp*: *rrstep-rstep-pos-conv*)
ultimately show *?thesis* **by** *auto*
next
case *False*
then obtain *v* **where** *v* ◁ *Fun f ss* **and** *v* ∉ *NF* (*rstep R*) **by** *auto*
then obtain *s* **and** *w* **where** *s* ∈ *set ss* **and** *s* ⊵ *v* **and** (*s*, *w*) ∈ *rstep R*
by (*auto simp*: *NF-def*) (*metis NF-iff-no-step NF-subterm supt-Fun-imp-arg-supteq*)
from *Fun.IH* [*OF this*(*1*, *3*)] **obtain** *u* **and** *p*
where *p* ∈ *poss s* **and** ∗: (*s*, *u*) ∈ *rstep-pos R p*
and ∗∗: ∀ *v* ◁ *s* |- *p*. *v* ∈ *NF* (*rstep R*) **by** *blast*
from ‹*s* ∈ *set ss*› **obtain** *i*
where *i* < *length ss* **and** [*simp*]:*ss* ! *i* = *s* **by** (*auto simp*: *in-set-conv-nth*)
with ‹*p* ∈ *poss s*› **have** *i* # *p* ∈ *poss* (*Fun f ss*) **by** *auto*
moreover with ∗∗ **have** ∀ *v* ◁ *Fun f ss* |- (*i* # *p*). *v* ∈ *NF* (*rstep R*) **by** *auto*
moreover from *rstep-pos-arg* [*OF* ∗ ‹*i* < *length ss*› ‹*ss* ! *i* = *s*›]
have (*Fun f ss*, (*ctxt-of-pos-term* [*i*] (*Fun f ss*))⟨*u*⟩) ∈ *rstep-pos R* (*i* # *p*) **.**
ultimately show *?thesis* **by** *blast*
qed
qed

4.6 Normal Forms

abbreviation *NF-trs* :: ('*f*, '*v*) *trs* ⇒ ('*f*, '*v*) *terms* **where**
NF-trs R ≡ *NF* (*rstep R*)

lemma *NF-trs-mono*: *r* ⊆ *s* ⟹ *NF-trs s* ⊆ *NF-trs r*
by (*rule NF-anti-mono*[*OF rstep-mono*])

lemma *NF-trs-union*: *NF-trs* (*R* ∪ *S*) = *NF-trs R* ∩ *NF-trs S*
unfolding *rstep-union* **using** *NF-anti-mono*[*of - rstep R* ∪ *rstep S*] **by** *auto*

abbreviation *NF-terms* :: ('*f*, '*v*) *terms* ⇒ ('*f*, '*v*) *terms* **where**
NF-terms Q ≡ *NF* (*rstep* (*Id-on Q*))

lemma *NF-terms-anti-mono*:
Q ⊆ *Q'* ⟹ *NF-terms Q'* ⊆ *NF-terms Q*
by (*rule NF-trs-mono*, *auto*)

lemma *lhs-var-not-NF*:
assumes *l* ∈ *T* **and** *is-Var l* **shows** *t* ∉ *NF-terms T*
proof −
from *assms* **obtain** *x* **where** *l*: *l* = *Var x* **by** (*cases l*, *auto*)
let *?σ* = *subst x t*
from *assms* **have** *l* ∉ *NF-terms T* **by** *auto*
with *NF-instance*[*of l ?σ Id-on T*]
have *l* · *?σ* ∉ *NF-terms T* **by** *auto*
then show *?thesis* **by** (*simp add*: *l subst-def*)
qed

lemma *not-NF-termsE*[*elim*]:
assumes $s \notin$ *NF-terms* Q
obtains $l\ C\ \sigma$ **where** $l \in Q$ **and** $s = C\langle l \cdot \sigma \rangle$
proof −
from *assms* **obtain** t **where** $(s, t) \in$ *rstep* (*Id-on* Q) **by** *auto*
with ‹$\bigwedge l\ C\ \sigma.\ [\![l \in Q; s = C\langle l \cdot \sigma\rangle]\!] \Longrightarrow$ *thesis*› **show** *?thesis* **by** *auto*
qed

lemma *notin-NF-E* [*elim*]:
fixes $R :: ('f, 'v)$ *trs*
assumes $t \notin$ *NF-trs* R
obtains $C\ l$ **and** $\sigma :: ('f, 'v)$ *subst* **where** $l \in$ *lhss* R **and** $t = C\langle l \cdot \sigma\rangle$
proof −
assume *1*: $\bigwedge l\ C\ (\sigma::('f, 'v)\ subst).\ l \in$ *lhss* $R \Longrightarrow t = C\langle l \cdot \sigma\rangle \Longrightarrow$ *thesis*
from *assms* **obtain** u **where** $(t, u) \in$ *rstep* R **by** (*auto simp*: *NF-def*)
then obtain $C\ \sigma\ l\ r$ **where** $(l, r) \in R$ **and** $t = C\langle l \cdot \sigma\rangle$ **by** *blast*
with *1* **show** *?thesis* **by** *force*
qed

lemma *NF-ctxt-subst*: *NF-terms* $Q = \{t.\ \neg\ (\exists\ C\ q\ \sigma.\ t = C\langle q{\cdot}\sigma\rangle \wedge q \in Q)\}$ (**is** - = *?R*)
proof −
{
fix t
assume $t \notin$ *?R*
then obtain $C\ q\ \sigma$ **where** *t*: $t = C\langle q{\cdot}\sigma\rangle$ **and** *q*: $q \in Q$ **by** *auto*
have $(t,t) \in$ *rstep* (*Id-on* Q)
unfolding t **using** q **by** *auto*
then have $t \notin$ *NF-terms* Q **by** *auto*
}
moreover
{
fix t
assume $t \notin$ *NF-terms* Q
then obtain $C\ q\ \sigma$ **where** *t*: $t = C\langle q{\cdot}\sigma\rangle$ **and** *q*: $q \in Q$ **by** *auto*
then have $t \notin$ *?R* **by** *auto*
}
ultimately show *?thesis* **by** *auto*
qed

lemma *some-NF-imp-no-Var*:
assumes $t \in$ *NF-terms* Q
shows *Var* $x \notin Q$
proof
assume *Var* $x \in Q$
with *assms*[*unfolded NF-ctxt-subst*] **have** $\bigwedge\ \sigma\ C.\ t \neq C\ \langle\ \sigma\ x\ \rangle$ **by** *force*
from *this*[*of Hole* λ -. t] **show** *False* **by** *simp*
qed

lemma *NF-map-vars-term-inj*:
assumes *inj*: $\bigwedge x.\ n\ (m\ x) = x$ **and** *NF*: $t \in$ *NF-terms* Q
shows (*map-vars-term* $m\ t$) $\in$ *NF-terms* (*map-vars-term* m ‘ Q)
proof (*rule ccontr*)
assume ¬ *?thesis*
then obtain u **where** (*map-vars-term* $m\ t$, u) $\in$ *rstep* (*Id-on* (*map-vars-term* m ‘ Q)) **by** *blast*
then obtain *ml mr* $C\ \sigma$ **where** *in-mR*: (*ml*, *mr*) $\in$ *Id-on* (*map-vars-term* m ‘ Q)
and *mt*: *map-vars-term* $m\ t = C\langle ml \cdot \sigma\rangle$ **by** *best*
let *?m* = n
from *in-mR* **obtain** $l\ r$ **where** (l, r) $\in$ *Id-on* Q **and** *ml*: *ml* = *map-vars-term* $m\ l$ **by** *auto*
have t = *map-vars-term ?m* (*map-vars-term* $m\ t$) **by** (*simp add*: *map-vars-term-inj-compose*[*of* n m, *OF inj*])
also have ... = *map-vars-term ?m* ($C\langle ml \cdot \sigma\rangle$) **by** (*simp add*: *mt*)
also have ... = (*map-vars-ctxt ?m* C)$\langle$*map-vars-term ?m* (*map-vars-term* $m\ l \cdot \sigma$)$\rangle$
by (*simp add*: *map-vars-term-ctxt-commute ml*)
also have ... = (*map-vars-ctxt ?m* C)$\langle l \cdot$ (*map-vars-subst-ran ?m* ($\sigma \circ m$))$\rangle$
by (*simp add*: *apply-subst-map-vars-term map-vars-subst-ran*)
finally show *False* **using** *NF* **and** ‹(l, r) $\in$ *Id-on* Q› **by** *auto*
qed

lemma *notin-NF-terms*: $t \in Q \Longrightarrow t \notin$ *NF-terms* Q
using *lhs-notin-NF-rstep*[*of t t Id-on Q*] **by** (*simp add*: *Id-on-iff*)

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-terms* Q
by (*rule ccontr*, *rule not-NF-termsE* [*OF - NF*])

lemma *NF-args-imp-NF*:
assumes *ss*: $\bigwedge s.\ s \in$ *set ss* $\Longrightarrow s \in$ *NF-terms* Q
and *someNF*: $t \in$ *NF-terms* Q
and *root*: *Some* (*f*,*length ss*) $\notin$ *root* ‘ Q
shows (*Fun f ss*) $\in$ *NF-terms* Q
proof
fix $C\ l\ \sigma$
assume *id*: *Fun f ss* = $C\ \langle\ l \cdot \sigma\ \rangle$ **and** *l*: $l \in Q$
show *False*
proof (*cases* C)
case *Hole*
with *id* **have** *id*: *Fun f ss* = $l \cdot \sigma$ **by** *simp*
show *False*
proof (*cases* l)
case (*Fun g ls*)
with *id* **have** *fg*: $f = g$ **and** *ss*: *ss* = *map* ($\lambda\ s.\ s \cdot \sigma$) *ls* **by** *auto*

```
    from arg-cong[OF ss, of length] have len: length ss = length ls by simp
    from l[unfolded Fun] root[unfolded fg len] show False by force
   next
    case (Var x)
    from some-NF-imp-no-Var[OF someNF] Var l show False by auto
   qed
  next
   case (More g bef D aft)
   note id = id[unfolded More]
   from id have NF: ss ! length bef = D ⟨l · σ⟩ by auto
   from id have mem: ss ! length bef ∈ set ss by auto
   from ss[OF mem, unfolded NF-ctxt-subst NF] l show False by auto
  qed
qed

lemma NF-Var-is-Fun:
  assumes Q: Ball Q is-Fun
  shows Var x ∈ NF-terms Q
proof
  fix C l σ
  assume x: Var x = C ⟨ l · σ ⟩ and l: l ∈ Q
  from l Q obtain f ls where l: l = Fun f ls by (cases l, auto)
  then show False using x by (cases C, auto)
qed

lemma NF-terms-lhss [simp]: NF-terms (lhss R) = NF (rstep R)
proof
  show NF (rstep R) ⊆ NF-terms (lhss R) by force
next
  show NF-terms (lhss R) ⊆ NF (rstep R)
  proof
    fix s assume NF: s ∈ NF-terms (lhss R)
    show s ∈ NF (rstep R)
    proof (rule ccontr)
      assume s ∉ NF (rstep R)
      then obtain t where (s, t) ∈ rstep R by auto
      then obtain l r C σ where (l, r) ∈ R and s: s = C⟨l · σ⟩ by auto
      then have (l, l) ∈ Id-on (lhss R) by force
      then have (s, s) ∈ rstep (Id-on (lhss R)) unfolding s by auto
      with NF show False by auto
    qed
  qed
qed
```

4.7 Relative Rewrite Steps

```
abbreviation relstep R E ≡ relto (rstep R) (rstep E)

lemma args-SN-on-relstep-nrrstep-imp-args-SN-on:
```

assumes *SN*: $\bigwedge u.\ s \rhd u \Longrightarrow$ *SN-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 −
from *nrrstepE*[*OF st*] **obtain** $C\ l\ r\ \sigma$ **where** $C \neq \Box$ **and** *lr*: $(l,r) \in R \cup E$
and *s*: $s = C\langle l \cdot \sigma\rangle$ **and** *t*: $t = C\langle r \cdot \sigma\rangle$ **by** *blast*
then obtain *f bef C aft* **where** *s*: $s =$ *Fun f* (*bef* @ $C\langle l \cdot \sigma\rangle$ # *aft*) **and** *t*: $t =$ *Fun f* (*bef* @ $C\langle r \cdot \sigma\rangle$ # *aft*)
by (*cases C*, *auto*)
let *?ts* = *bef* @ $C\langle r \cdot \sigma\rangle$ # *aft*
let *?ss* = *bef* @ $C\langle l \cdot \sigma\rangle$ # *aft*
from *supt* **obtain** *D* **where** $t = D\langle u\rangle$ **and** $D \neq \Box$ **by** *auto*
then obtain *bef′ aft′ D* **where** *t′*: $t =$ *Fun f* (*bef′* @ $D\langle u\rangle$ # *aft′*) **unfolding** *t* **by** (*cases D*, *auto*)
have $D\langle u\rangle \trianglerighteq u$ **by** *auto*
then have *supt*: $\bigwedge s.\ s \rhd D\langle u\rangle \Longrightarrow s \rhd u$ **by** (*metis supt-supteq-trans*)
show *SN-on* (*relstep R E*) $\{u\}$
proof (*cases* $D\langle u\rangle \in$ *set ?ss*)
case *True*
then have $s \rhd D\langle u\rangle$ **unfolding** *s* **by** *auto*
then have $s \rhd u$ **by** (*rule supt*)
with *SN* **show** *?thesis* **by** *auto*
next
case *False*
have $D\langle u\rangle \in$ *set ?ts* **using** *arg-cong*[*OF t′*[*unfolded t*], *of args*] **by** *auto*
with *False* **have** *Du*: $D\langle u\rangle = C\langle r \cdot \sigma\rangle$ **by** *auto*
have $s \rhd C\langle l \cdot \sigma\rangle$ **unfolding** *s* **by** *auto*
with *SN* **have** *SN-on* (*relstep R E*) $\{C\langle l \cdot \sigma\rangle\}$ **by** *auto*
from *step-preserves-SN-on-relto*[*OF - this*, *of* $C\langle r \cdot \sigma\rangle$] *lr*
have *SN*: *SN-on* (*relstep R E*) $\{D\langle u\rangle\}$ **using** *Du* **by** *auto*
show *?thesis*
by (*rule ctxt-closed-SN-on-subt*[*OF ctxt.closed-relto SN*], *auto*)
qed
qed

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*)
unfolding *Tinf-def*
by (*intro CollectI conjI*[*OF t*] *allI impI*)
(*rule args-SN-on-relstep-nrrstep-imp-args-SN-on*[*OF - st*],
insert tinf[*unfolded Tinf-def*], *auto*)

lemma *subterm-preserves-SN-on-relstep*:
SN-on (*relstep R E*) $\{s\} \Longrightarrow s \trianglerighteq t \Longrightarrow$ *SN-on* (*relstep R E*) $\{t\}$
using *SN-imp-SN-subt* [*of rstep* (*rstep* ((*rstep E*)*) *O rstep R O rstep* ((*rstep E*)*))]

by (*simp only*: *rstep-relcomp-idemp2*) (*simp only*: *rstep-rtrancl-idemp*)

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

lemma *rstep-ruleI* [*intro*]:
$s = C\langle l \cdot \sigma\rangle \Longrightarrow t = C\langle r \cdot \sigma\rangle \Longrightarrow (s, t) \in rstep\text{-}rule\ (l, r)$
by (*auto simp*: *rstep-rule.simps*)

lemma *rstep-rule-ctxt*:
$(s, t) \in rstep\text{-}rule\ \varrho \Longrightarrow (C\langle s\rangle, C\langle t\rangle) \in rstep\text{-}rule\ \varrho$
using *rstep-rule.rule* [*of* $C\langle s\rangle$ $C \circ_c D$ ϱ - $C\langle t\rangle$ **for** *D*]
by (*auto elim*: *rstep-rule.cases simp*: *ctxt-of-pos-term-append*)

lemma *rstep-rule-subst*:
assumes $(s, t) \in rstep\text{-}rule\ \varrho$
shows $(s \cdot \sigma, t \cdot \sigma) \in rstep\text{-}rule\ \varrho$
using *assms*
proof (*cases*)
case (*rule C* τ)
then show *?thesis*
using *rstep-rule.rule* [*of* $s \cdot \sigma$ - ϱ τ $\circ_s$ σ]
by (*auto elim*!: *rstep-rule.cases simp*: *ctxt-of-pos-term-subst*)
qed

lemma *rstep-rule-imp-rstep*:
$\varrho \in R \Longrightarrow (s, t) \in rstep\text{-}rule\ \varrho \Longrightarrow (s, t) \in rstep\ R$
by (*force elim*: *rstep-rule.cases*)

lemma *rstep-imp-rstep-rule*:
assumes $(s, t) \in rstep\ R$
obtains *l r* **where** $(l, r) \in R$ **and** $(s, t) \in rstep\text{-}rule\ (l, r)$
using *assms* **by** *blast*

lemma *term-subst-rstep*:
assumes $\bigwedge x.\ x \in vars\text{-}term\ t \Longrightarrow (\sigma\ x, \tau\ x) \in rstep\ R$
shows $(t \cdot \sigma, t \cdot \tau) \in (rstep\ R)^*$
using *assms*
proof (*induct t*)
case (*Fun f ts*)
{ fix t_i
assume t_i: $t_i \in set\ ts$
with *Fun*(*2*) **have** $\bigwedge x.\ x \in vars\text{-}term\ t_i \Longrightarrow (\sigma\ x, \tau\ x) \in rstep\ R$ **by** *auto*
from *Fun*(*1*) [*OF* t_i *this*] **have** $(t_i \cdot \sigma, t_i \cdot \tau) \in (rstep\ R)^*$ **by** *blast*
}
then show *?case* **by** (*simp add*: *args-rsteps-imp-rsteps*)
qed (*auto*)

lemma *term-subst-rsteps*:
assumes $\bigwedge x.\ x \in \mathit{vars\text{-}term}\ t \Longrightarrow (\sigma\ x, \tau\ x) \in (\mathit{rstep}\ R)^*$
shows $(t \cdot \sigma, t \cdot \tau) \in (\mathit{rstep}\ R)^*$
by (*metis assms rstep-rtrancl-idemp rtrancl-idemp term-subst-rstep*)

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 $(u \cdot \sigma_1, u \cdot \sigma_2) \in (\mathit{rstep}\ R)^{\downarrow}$
using *assms*
proof −
{ **fix** x
assume $x \in \mathit{vars\text{-}term}\ u$
from *assms* [*OF this*] **have** $\exists \sigma.\ (\sigma_1\ x, \sigma\ x) \in (\mathit{rstep}\ R)^* \wedge (\sigma_2\ x, \sigma\ x) \in (\mathit{rstep}\ R)^*$ **by** *auto*
}
then have $\forall x \in \mathit{vars\text{-}term}\ u.\ \exists \sigma.\ (\sigma_1\ x, \sigma\ x) \in (\mathit{rstep}\ R)^* \wedge (\sigma_2\ x, \sigma\ x) \in (\mathit{rstep}\ R)^*$ **by** *blast*
then obtain s **where** $\forall x \in \mathit{vars\text{-}term}\ u.\ (\sigma_1\ x, (s\ x)\ x) \in (\mathit{rstep}\ R)^* \wedge (\sigma_2\ x, (s\ x)\ x) \in (\mathit{rstep}\ R)^*$ **by** *metis*
then obtain σ **where** $\forall x \in \mathit{vars\text{-}term}\ u.\ (\sigma_1\ x, \sigma\ x) \in (\mathit{rstep}\ R)^* \wedge (\sigma_2\ x, \sigma\ x) \in (\mathit{rstep}\ R)^*$ **by** *fast*
then have $(u \cdot \sigma_1, u \cdot \sigma) \in (\mathit{rstep}\ R)^* \wedge (u \cdot \sigma_2, u \cdot \sigma) \in (\mathit{rstep}\ R)^*$ **using** *term-subst-rsteps* **by** *metis*
then show *?thesis* **by** *blast*
qed

lemma *funas-trs-converse* [*simp*]: *funas-trs* $(R^{-1}) = \mathit{funas\text{-}trs}\ R$
by (*auto simp*: *funas-defs*)

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−
from *assms* **obtain** σ **where** *step*:$t = (\mathit{ctxt\text{-}of\text{-}pos\text{-}term}\ p\ s)\langle r \cdot \sigma\rangle\ p \in \mathit{poss}\ s\ s\ |\text{-}\ p = l \cdot \sigma$
unfolding *rstep-pos.simps* **by** *auto*
with *replace-at-below-poss*[*of p s p*] **have** *pt*:$p \in \mathit{poss}\ t$ **by** *auto*
with *step ctxt-supt-id*[*OF step*(*2*)] **have** $s = (\mathit{ctxt\text{-}of\text{-}pos\text{-}term}\ p\ t)\langle l \cdot \sigma\rangle$
by (*simp add*: *ctxt-of-pos-term-replace-at-below*)
with *step ctxt-supt-id*[*OF pt*] **show** *?thesis* **unfolding** *rstep-pos.simps*
by (*metis pt replace-at-subt-at singletonI*)
qed

lemma *conversion-ctxt-closed*: $(s, t) \in (\mathit{rstep}\ R)^{\leftrightarrow *} \Longrightarrow (C\langle s\rangle, C\langle t\rangle) \in (\mathit{rstep}\ R)^{\leftrightarrow *}$
using *rsteps-closed-ctxt* **unfolding** *conversion-def*
by (*simp only*: *rstep-simps*(*5*)[*symmetric*])

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

using *rsteps-closed-subst* **unfolding** *conversion-def*
by (*simp only*: *rstep-simps*(*5*)[*symmetric*])

lemma *rstep-simulate-conv*:
assumes $\bigwedge l\ r.\ (l,\ r) \in S \Longrightarrow (l,\ r) \in (rstep\ R)^{\leftrightarrow *}$
shows $(rstep\ S) \subseteq (rstep\ R)^{\leftrightarrow *}$
proof
fix *s t*
assume $(s,\ t) \in rstep\ S$
then obtain *l r C* σ **where** *s*: $s = C\langle l \cdot \sigma\rangle$ **and** *t*:$t = C\langle r \cdot \sigma\rangle$ **and** *lr*: $(l,\ r) \in S$
unfolding *rstep-iff-rstep-r-c-s rstep-r-c-s-def* **by** *auto*
with *assms* **have** $(l,\ r) \in (rstep\ R)^{\leftrightarrow *}$ **by** *auto*
then show $(s,\ t) \in (rstep\ R)^{\leftrightarrow *}$ **using** *conversion-ctxt-closed conversion-subst-closed s t* **by** *metis*
qed

lemma *symcl-simulate-conv*:
assumes $\bigwedge l\ r.\ (l,\ r) \in S \Longrightarrow (l,\ r) \in (rstep\ R)^{\leftrightarrow *}$
shows $(rstep\ S)^{\leftrightarrow} \subseteq (rstep\ R)^{\leftrightarrow *}$
using *rstep-simulate-conv*[*OF assms*]
by *auto* (*metis conversion-inv subset-iff*)

lemma *conv-union-simulate*:
assumes $\bigwedge l\ r.\ (l,\ r) \in S \Longrightarrow (l,\ r) \in (rstep\ R)^{\leftrightarrow *}$
shows $(rstep\ (R \cup S))^{\leftrightarrow *} = (rstep\ R)^{\leftrightarrow *}$
proof
show $(rstep\ (R \cup S))^{\leftrightarrow *} \subseteq (rstep\ R)^{\leftrightarrow *}$
unfolding *conversion-def*
proof
fix *s t*
assume $(s,\ t) \in ((rstep\ (R \cup S))^{\leftrightarrow})^*$
then show $(s,\ t) \in ((rstep\ R)^{\leftrightarrow})^*$
proof (*induct rule*: *rtrancl-induct*)
case (*step u t*)
then have $(u,\ t) \in (rstep\ R)^{\leftrightarrow} \vee (u,\ t) \in (rstep\ S)^{\leftrightarrow}$ **by** *auto*
then show *?case*
proof
assume $(u,\ t) \in (rstep\ R)^{\leftrightarrow}$
with *step* **show** *?thesis* **using** *rtrancl-into-rtrancl* **by** *metis*
next
assume $(u,\ t) \in (rstep\ S)^{\leftrightarrow}$
with *symcl-simulate-conv*[*OF assms*] **have** $(u,\ t) \in (rstep\ R)^{\leftrightarrow *}$ **by** *auto*
with *step* **show** *?thesis* **by** *auto*
qed
qed *simp*
qed
next
show $(rstep\ R)^{\leftrightarrow *} \subseteq (rstep\ (R \cup S))^{\leftrightarrow *}$

unfolding *conversion-def*
using *rstep-union rtrancl-mono sup.cobounded1 symcl-Un*
by *metis*
qed

definition *suptrel R* = (*relto* {▷} (*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*
using *assms* **unfolding** *critical-pairs-def* **by** *blast*

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*$_2$
unfolding *critical-pairs-def* **using** *assms* **by** *blast*

lemma *critical-Peaks-main*:
fixes P R :: $('f, 'v)$ *trs*
assumes *tu*: $(t, u) \in$ *rrstep* P **and** *ts*: $(t, s) \in$ *rstep* R
shows $(s, u) \in (\mathit{rstep}\ R)\widehat{\ }* \ O\ \mathit{rrstep}\ P\ O\ ((\mathit{rstep}\ R)\widehat{\ }*)\widehat{\ }{-}1 \vee$
$(\exists\ C\ l\ m\ r\ \sigma.\ s = C\langle l \cdot \sigma\rangle \wedge t = C\langle m \cdot \sigma\rangle \wedge u = C\langle r \cdot \sigma\rangle \wedge$
$(l, m, r) \in$ *critical-Peaks* $P\ R$)
proof −
let *?R* = *rstep* R
let *?CP* = *critical-Peaks* $P\ R$
from *rrstepE*[*OF tu*] **obtain** *l1 r1* $\sigma 1$ **where** *lr1*: $(l1, r1) \in P$ **and** *t1*: $t = l1 \cdot \sigma 1$ **and** *u*: $u = r1 \cdot \sigma 1$ **by** *auto*
from *ts* **obtain** *C l2 r2* $\sigma 2$ **where** *lr2*: $(l2, r2) \in R$ **and** *t2*: $t = C\langle l2 \cdot \sigma 2\rangle$ **and** *s*: $s = C\langle r2 \cdot \sigma 2\rangle$ **by** *auto*
from *t1 t2* **have** *id*: $l1 \cdot \sigma 1 = C\langle l2 \cdot \sigma 2\rangle$ **by** *auto*
let *?p* = *hole-pos C*
show *?thesis*
proof (*cases* $?p \in$ *poss l1* $\wedge$ *is-Fun* (*l1* |- *?p*))
case *True*
then have *p*: $?p \in$ *poss l1* **by** *auto*
from *ctxt-supt-id* [*OF p*] **obtain** *D* **where** *Dl1*: $D\langle l1 \text{ |- } ?p\rangle = l1$
and *D*: $D =$ *ctxt-of-pos-term* (*hole-pos C*) *l1* **by** *blast*
from *arg-cong* [*OF Dl1*, *of* $\lambda\ t.\ t \cdot \sigma 1$]
have $(D \cdot_c \sigma 1)\langle(l1 \text{ |- } ?p) \cdot \sigma 1\rangle = C\langle l2 \cdot \sigma 2\rangle$ **unfolding** *id* **by** *simp*
from *arg-cong* [*OF this*, *of* $\lambda\ t.\ t$ |- *?p*]
have $l2 \cdot \sigma 2 = (D \cdot_c \sigma 1)\langle(l1 \text{ |- } ?p) \cdot \sigma 1\rangle$ |- *?p* **by** *simp*
also have ... $= (D \cdot_c \sigma 1)\langle(l1 \text{ |- } ?p) \cdot \sigma 1\rangle$ |- (*hole-pos* $(D \cdot_c \sigma 1)$)
using *hole-pos-ctxt-of-pos-term* [*OF p*] **unfolding** *D* **by** *simp*
also have ... $= (l1 \text{ |- } ?p) \cdot \sigma 1$ **by** (*rule subt-at-hole-pos*)
finally have *ident*: $l2 \cdot \sigma 2 = l1 \text{ |- } ?p \cdot \sigma 1$ **by** *auto*
from *mgu-vd-complete* [*OF ident* [*symmetric*]]
obtain $\mu 1\ \mu 2\ \varrho$ **where** *mgu*: *mgu-vd ren* $(l1 \text{ |- } ?p)\ l2 =$ *Some* $(\mu 1, \mu 2)$ **and**
$\mu 1$: $\sigma 1 = \mu 1 \circ_s \varrho$
and $\mu 2$: $\sigma 2 = \mu 2 \circ_s \varrho$
and *ident'*: $l1 \text{ |- } ?p \cdot \mu 1 = l2 \cdot \mu 2$ **by** *blast*
have *in-cp*: $((D \cdot_c \mu 1)\langle r2 \cdot \mu 2\rangle, l1 \cdot \mu 1, r1 \cdot \mu 1) \in ?CP$
unfolding *critical-Peaks-def*
apply *clarify*
apply (*intro exI conjI*)
apply (*rule refl*)
apply (*rule lr1*)
apply (*rule lr2*)
apply (*rule Dl1*[*symmetric*])
apply (*rule True*[*THEN conjunct2*])
apply (*rule mgu*)
done
from *hole-pos-ctxt-of-pos-term* [*OF p*] *D* **have** *pD*: *?p* = *hole-pos D* **by** *simp*
from *id* **have** *C*: *C* = *ctxt-of-pos-term ?p* $(l1 \cdot \sigma 1)$ **by** *simp*
have $C\langle r2 \cdot \sigma 2\rangle = (\mathit{ctxt\text{-}of\text{-}pos\text{-}term}\ ?p\ (l1 \cdot \sigma 1))\langle r2 \cdot \sigma 2\rangle$ **using** *C* **by** *simp*
also have ... $= (\mathit{ctxt\text{-}of\text{-}pos\text{-}term}\ ?p\ l1 \cdot_c \sigma 1)\langle r2 \cdot \sigma 2\rangle$ **unfolding** *ctxt-of-pos-term-subst*

```
[OF p] ..
  also have ... = (D ·c σ1)⟨r2 · σ2⟩ unfolding D ..
  finally have Crσ: C⟨r2 · σ2⟩ = (D ·c σ1)⟨r2 · σ2⟩ .
  show ?thesis unfolding Crσ s u t1 unfolding μ1 μ2
  proof (rule disjI2, intro exI, intro conjI)
    show r1 · μ1 ∘s ϱ = □⟨r1 · μ1 · ϱ⟩ by simp
  qed (insert in-cp, auto)
 next
  case False
  from pos-into-subst [OF id - False]
  obtain q q' x where p: ?p = q @ q' and q: q ∈ poss l1 and l1q: l1 |- q = Var
x by auto
  have l2 · σ2 = C⟨l2 · σ2⟩ |- (q @ q') unfolding p [symmetric] by simp
  also have ... = l1 · σ1 |- (q @ q') unfolding id ..
  also have ... = l1 |- q · σ1 |- q' using q by simp
  also have ... = σ1 x |- q' unfolding l1q by simp
  finally have l2x: l2 · σ2 = σ1 x |- q' by simp
  have pp: ?p ∈ poss (l1 · σ1) unfolding id by simp
  then have q @ q' ∈ poss (l1 · σ1) unfolding p .
  then have q' ∈ poss (l1 · σ1 |- q) unfolding poss-append-poss ..
  with q have q' ∈ poss (l1 |- q · σ1) by auto
  then have q'x: q' ∈ poss (σ1 x) unfolding l1q by simp
  from ctxt-supt-id [OF q'x] obtain E where σ1x: E⟨l2 · σ2⟩ = σ1 x
    and E: E = ctxt-of-pos-term q' (σ1 x)
    unfolding l2x by blast
  let ?e = E⟨r2 · σ2⟩
  from hole-pos-ctxt-of-pos-term [OF q'x] E have q': q' = hole-pos E by simp
  from σ1x have σ1x': σ1 x = E⟨l2 · σ2⟩ by simp
  let ?σ = λ y. if y = x then ?e else σ1 y
  have (u, r1 · ?σ) ∈ (rstep R)^* unfolding u
  proof (rule subst-rsteps-imp-rsteps)
    fix y
    show (σ1 y, ?σ y) ∈ (rstep R)^*
    proof (cases y = x)
      case True
      show ?thesis unfolding True σ1x' using lr2 by auto
    qed simp
  qed
  hence r1u: (r1 · ?σ, u) ∈ ((rstep R)^*)^-1 by auto
  show ?thesis
  proof (rule disjI1, intro relcompI)
    show (r1 · ?σ, u) ∈ ((rstep R)^*)^-1 by fact
    show (l1 · ?σ, r1 · ?σ) ∈ rrstep P using lr1 by auto
    from q have ql1: q ∈ poss (l1 · σ1) by simp
    have s = replace-at (C⟨l2 · σ2⟩) ?p (r2 · σ2) unfolding s by simp
    also have ... = replace-at (l1 · σ1) ?p (r2 · σ2) unfolding id ..
    also have ... = replace-at (l1 · σ1) q ?e
    proof -
      have E = ctxt-of-pos-term q' (l1 · σ1 |- q)
```

```
      unfolding subt-at-subst [OF q] l1q E by simp
    then show ?thesis
      unfolding p
      unfolding ctxt-of-pos-term-append [OF ql1]
      by simp
  qed
  finally have s: s = replace-at (l1 · σ1) q ?e .
  from q l1q have (replace-at (l1 · σ1) q ?e, l1 · ?σ) ∈ ?R^*
  proof (induct l1 arbitrary: q)
    case (Fun f ls)
    from Fun(2, 3) obtain i p where q: q = i # p and i: i < length ls and
p: p ∈ poss (ls ! i) and px: ls ! i |- p = Var x by (cases q, auto)
    from i have ls ! i ∈ set ls by auto
    from Fun(1) [OF this p px] have rec: (replace-at (ls ! i · σ1) p ?e, ls ! i ·
?σ) ∈ ?R^* .
    let ?lsσ = map (λ t. t · σ1) ls
    let ?lsσ' = map (λ t. t · ?σ) ls
    have id: replace-at (Fun f ls · σ1) q ?e = Fun f (take i ?lsσ @ replace-at
(ls ! i · σ1) p ?e # drop (Suc i) ?lsσ) (is - = Fun f ?r)
      unfolding q using i by simp
    show ?case unfolding id unfolding eval-term.simps
    proof (rule all-ctxt-closedD [of UNIV])
      fix j
      assume j: j < length ?r
      show (?r ! j, ?lsσ' ! j) ∈ ?R^*
      proof (cases j = i)
        case True
        show ?thesis using i True using rec by (auto simp: nth-append)
      next
        case False
        have ?r ! j = ?lsσ ! j
          by (rule nth-append-take-drop-is-nth-conv, insert False i j, auto)
        also have ... = ls ! j · σ1 using j i by auto
        finally have idr: ?r ! j = ls ! j · σ1 .
        from j i have idl: ?lsσ' ! j = ls ! j · ?σ by auto
        show ?thesis unfolding idr idl
        proof (rule subst-rsteps-imp-rsteps)
          fix y
          show (σ1 y, ?σ y) ∈ ?R^*
          proof (cases y = x)
            case True then show ?thesis using σ1x' lr2 by auto
          qed simp
        qed
      qed
    qed (insert i, auto)
  qed simp
  then show (s, l1 · ?σ) ∈ ?R^* unfolding s .
 qed
qed
```

qed

lemma *critical-Peaks-main-rrstep*:
fixes $R :: ('f, 'v)\ trs$
assumes *tu*: $(t, u) \in rrstep\ R$ **and** *ts*: $(t, s) \in rstep\ R$
shows $(s, u) \in join\ (rstep\ R) \vee$
$(\exists\ C\ l\ m\ r\ \sigma.\ s = C\langle l \cdot \sigma\rangle \wedge t = C\langle m \cdot \sigma\rangle \wedge u = C\langle r \cdot \sigma\rangle \wedge$
$(l, m, r) \in critical\text{-}Peaks\ R\ R)$
using *critical-Peaks-main*[*OF assms*]
proof
assume $(s, u) \in (rstep\ R)^*\ O\ rrstep\ R\ O\ ((rstep\ R)^*)^{-1}$
also have $\ldots \subseteq (rstep\ R)^*\ O\ ((rstep\ R)^*)^{-1}$
unfolding *rstep-iff-rrstep-or-nrrstep* **by** *regexp*
finally have $(s, u) \in join\ (rstep\ R)$ **by** *blast*
thus *?thesis* **by** *auto*
qed *auto*

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 −
show *?thesis* **unfolding** *rstep-iff-rstep-r-p-s*
proof (*intro exI*)
show $(?one, ?two) \in rstep\text{-}r\text{-}p\text{-}s\ R\ (l2,r2)\ p2\ \sigma 2$
unfolding *rstep-r-p-s-def Let-def*
apply (*intro CollectI*, *unfold split fst-conv snd-conv*)
using *p1 p12 p2 step2*
by (*metis ctxt-supt-id parallel-poss-replace-at parallel-replace-at-subt-at*)
qed
qed

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 −
let *?R* = *rstep R*
let *?CP* = *critical-Peaks R R*
from *tu* **obtain** *C1 l1 r1 σ1* **where** *lr1*: $(l1, r1) \in R$ **and** *t1*: $t = C1\langle l1 \cdot \sigma 1\rangle$
and *u*: $u = C1\langle r1 \cdot \sigma 1\rangle$ **by** *auto*
from *ts* **obtain** *C2 l2 r2 σ2* **where** *lr2*: $(l2, r2) \in R$ **and** *t2*: $t = C2\langle l2 \cdot \sigma 2\rangle$
and *s*: $s = C2\langle r2 \cdot \sigma 2\rangle$ **by** *auto*

```
  define n where n = size C1 + size C2
  from t1 t2 u s n-def show ?thesis
  proof (induct n arbitrary: C1 C2 s t u rule: less-induct)
    case (less n C1 C2 s t u)
    show ?case
    proof (cases C1)
      case Hole
      with less(2,4) lr1 have tu: (t, u) ∈ rrstep R by auto
      from less(3,5) lr2 have ts: (t, s) ∈ rstep R by auto
      from critical-Peaks-main-rrstep[OF tu ts] show ?thesis by auto
    next
      case (More f1 bef1 D1 aft1) note C1 = this
      show ?thesis
      proof (cases C2)
        case Hole
        with less(3,5) lr2 have ts: (t, s) ∈ rrstep R by auto
        from less(2,4) lr1 have tu: (t, u) ∈ rstep R by auto
        from critical-Peaks-main-rrstep[OF ts tu] show ?thesis by auto
      next
        case (More f2 bef2 D2 aft2) note C2 = this
        from less(2−3) C1 C2
        have id: (More f1 bef1 D1 aft1)⟨l1 · σ1⟩ = (More f2 bef2 D2 aft2)⟨l2 · σ2⟩
by auto
        let ?n1 = length bef1
        let ?n2 = length bef2
        from id have f: f1 = f2 by simp
        show ?thesis
        proof (cases ?n1 = ?n2)
          case True
          with id have idb: bef1 = bef2 and ida: aft1 = aft2
            and idD: D1⟨l1 · σ1⟩ = D2⟨l2 · σ2⟩ by auto
          let ?C = More f2 bef2 □ aft2
          have id1: C1 = ?C ∘c D1 unfolding C1 f ida idb by simp
          have id2: C2 = ?C ∘c D2 unfolding C2 by simp
          define m where m = size D1 + size D2
          have mn: m < n unfolding less m-def C1 C2 by auto
          note IH = less(1)[OF mn refl idD refl refl m-def]
          then show ?thesis
          proof
            assume ( D2⟨r2 · σ2⟩, D1⟨r1 · σ1⟩) ∈ join ?R
            then obtain s' where seq1: (D1⟨r1 · σ1⟩, s') ∈ ?R^*
              and seq2: (D2⟨r2 · σ2⟩, s') ∈ ?R^* by auto
            from rsteps-closed-ctxt [OF seq1, of ?C]
            have seq1: (C1⟨r1 · σ1⟩, ?C⟨s'⟩) ∈ ?R^* using id1 by auto
            from rsteps-closed-ctxt [OF seq2, of ?C]
            have seq2: (C2⟨r2 · σ2⟩, ?C⟨s'⟩) ∈ ?R^* using id2 by auto
            from seq1 seq2 show ?thesis using less by auto
          next
            assume ∃ C l m r σ. D2⟨r2 · σ2⟩ = C⟨l · σ⟩ ∧ D1⟨l1 · σ1⟩ = C⟨m ·
```

```
σ⟩ ∧ D1⟨r1 · σ1⟩ = C⟨r · σ⟩ ∧ ((l, m, r) ∈ ?CP ∨ (r, m, l) ∈ ?CP)
          then obtain C l m r σ where idD: D2⟨r2 · σ2⟩ = C⟨l · σ⟩ D1⟨l1 ·
σ1⟩ = C⟨m · σ⟩ D1⟨r1 · σ1⟩ = C⟨r · σ⟩ and mem: ((l, m, r) ∈ ?CP ∨ (r, m, l)
∈ ?CP) by blast
         show ?thesis
           apply (intro disjI2)
           apply (unfold less id1 id2)
           apply (intro exI [of - ?C ∘c C] exI)
           by (rule conjI [OF - conjI [OF - conjI[OF - mem]]], insert idD, auto)
        qed
       next
        case False
        let ?p1 = ?n1 # hole-pos D1
        let ?p2 = ?n2 # hole-pos D2
        have l2: C1⟨l1 · σ1⟩ |- ?p2 = l2 · σ2 unfolding C1 id by simp
        have p12: ?p1 ⊥ ?p2 using False by simp
        have p1: ?p1 ∈ poss (C1⟨l1 · σ1⟩) unfolding C1 by simp
       have p2: ?p2 ∈ poss (C1⟨l1 · σ1⟩) unfolding C1 unfolding id by simp
        let ?one = replace-at (C1⟨l1 · σ1⟩) ?p1 (r1 · σ1)
        have one: C1⟨r1 · σ1⟩ = ?one unfolding C1 by simp
        from parallel-rstep [OF p12 p1 p2 l2 lr2, of r1 · σ1]
        have (?one, replace-at ?one ?p2 (r2 · σ2)) ∈ rstep R by auto
         then have one: (C1⟨r1 · σ1⟩, replace-at ?one ?p2 (r2 · σ2)) ∈ (rstep
R)^* unfolding one by simp
         have l1: C2⟨l2 · σ2⟩ |- ?p1 = l1 · σ1 unfolding C2 id [symmetric] by
simp
         have p21: ?p2 ⊥ ?p1 using False by simp
       have p1': ?p1 ∈ poss (C2⟨l2 · σ2⟩) unfolding C2 id [symmetric] by simp
        have p2': ?p2 ∈ poss (C2⟨l2 · σ2⟩) unfolding C2 by simp
        let ?two = replace-at (C2⟨l2 · σ2⟩) ?p2 (r2 · σ2)
        have two: C2⟨r2 · σ2⟩ = ?two unfolding C2 by simp
        from parallel-rstep [OF p21 p2' p1' l1 lr1, of r2 · σ2]
        have (?two, replace-at ?two ?p1 (r1 · σ1)) ∈ rstep R by auto
         then have two: (C2⟨r2 · σ2⟩, replace-at ?two ?p1 (r1 · σ1)) ∈ (rstep
R)^* unfolding two by simp
       have replace-at ?one ?p2 (r2 · σ2) = replace-at (replace-at (C1⟨l1 · σ1⟩)
?p2 (r2 · σ2)) ?p1 (r1 · σ1)
         by (rule parallel-replace-at [OF p12 p1 p2])
        also have ... = replace-at ?two ?p1 (r1 · σ1) unfolding C1 C2 id ..
       finally have one-two: replace-at ?one ?p2 (r2 · σ2) = replace-at ?two ?p1
(r1 · σ1) .
        show ?thesis unfolding less
         by (rule disjI1, insert one one-two two, auto)
      qed
     qed
    qed
   qed
  qed
```

lemma *critical-Peak-steps*:
fixes $R :: ('f, 'v)$ *trs* **and** S
assumes *cp*: $(l, m, r) \in$ *critical-Peaks* $R\ S$
shows $(m, l) \in$ *rstep* S $(m,r) \in$ *rstep* R $(m,r) \in$ *rrstep* R
proof −
from *cp* [*unfolded critical-Peaks-def*]
obtain $\sigma 1\ \sigma 2\ l1\ l2\ r1\ r2\ C$ **where** *id*: $r = r1 \cdot \sigma 1\ l = (C \cdot_c \sigma 1)\langle r2 \cdot \sigma 2\rangle\ m = (C \cdot_c \sigma 1)\langle l1 \cdot \sigma 1\rangle$
and *r1*: $(C\langle l1\rangle, r1) \in R$ **and** *r2*: $(l2, r2) \in S$ **and** *mgu*: *mgu-vd ren l1 l2* = *Some* $(\sigma 1, \sigma 2)$ **by** *auto*
have $(C\langle l1\rangle \cdot \sigma 1, r) \in$ *rrstep* R **unfolding** *id*
by (*rule rrstepI* [*of* $C\langle l1\rangle$ *r1 - - σ1*] *r1*, *insert r1*, *auto*)
thus $(m,r) \in$ *rrstep* R **unfolding** *id* **by** *auto*
thus $(m,r) \in$ *rstep* R **by** (*rule rrstep-imp-rstep*)
from *mgu-vd-sound* [*OF mgu*] **have** *change*: $C\langle l1\rangle \cdot \sigma 1 = (C \cdot_c \sigma 1)\langle l2 \cdot \sigma 2\rangle$ **by** *simp*
have $(C\langle l1\rangle \cdot \sigma 1, l) \in$ *rstep* S **unfolding** *change id*
by (*rule rstepI* [*OF r2*, *of - - σ2*], *auto*)
thus $(m, l) \in$ *rstep* S **unfolding** *id* **by** *auto*
qed

lemma *critical-Peak-to-pair*: **assumes** $(l, m, r) \in$ *critical-Peaks* $R\ R$
shows $\exists\ b.\ (b, l, r) \in$ *critical-pairs* $R\ R$
using *assms* **unfolding** *critical-Peaks-def critical-pairs-def* **by** *blast*

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-pairs* $R\ R \vee (b, r, l) \in$ *critical-pairs* $R\ R))$
using *critical-Peaks-main-rstep*[*OF st2 st1*]
proof
assume $\exists\, C\ l\ m\ r\ \sigma.$
$t1 = C\langle l \cdot \sigma\rangle \wedge s = C\langle m \cdot \sigma\rangle \wedge t2 = C\langle r \cdot \sigma\rangle \wedge ((l, m, r) \in$ *critical-Peaks* $R\ R \vee (r, m, l) \in$ *critical-Peaks* $R\ R)$
then obtain $C\ l\ m\ r\ \sigma$ **where** *id*: $t1 = C\langle l \cdot \sigma\rangle\ t2 = C\langle r \cdot \sigma\rangle$ **and** *disj*: $((l, m, r) \in$ *critical-Peaks* $R\ R \vee (r, m, l) \in$ *critical-Peaks* $R\ R)$
by *blast*
from *critical-Peak-to-pair disj* **obtain** b **where** $(b,l,r) \in$ *critical-pairs* $R\ R \vee (b,r,l) \in$ *critical-pairs* $R\ R$ **by** *blast*
with *id* **show** *?thesis* **by** *blast*
qed *auto*

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

```
  shows WCR (rstep R)
proof
  let ?R = rstep R
  let ?CP = critical-pairs R R
  fix s t1 t2
  assume steps: (s, t1) ∈ ?R (s, t2) ∈ ?R
  let ?p = λ s'. (t1, s') ∈ ?R^* ∧ (t2, s') ∈ ?R^*
  from critical-pairs-main [OF steps]
  have ∃ s'. ?p s'
  proof
    assume ∃ C b l r σ. t1 = C⟨l · σ⟩ ∧ t2 = C⟨r · σ⟩ ∧ ((b, l, r) ∈ ?CP ∨ (b,
r, l) ∈ ?CP)
    then obtain C b l r σ where id: t1 = C⟨l · σ⟩ t2 = C⟨r · σ⟩
      and mem: (b, l, r) ∈ ?CP ∨ (b, r, l) ∈ ?CP by blast
    show ?thesis
    proof (cases l = r)
      case True
      then show ?thesis unfolding id by auto
    next
      case False
    note sub-ctxt = rsteps-closed-ctxt [OF rsteps-closed-subst [OF rsteps-closed-subst]]
      from mem show ?thesis
      proof
        assume mem: (b, l, r) ∈ ?CP
        from cp [OF mem False] obtain l' r' s' τ where id2: l = l' · τ r = r' · τ
and steps: (l', s') ∈ ?R^* (r', s') ∈ ?R^*
          unfolding instance-rule-def by auto
        show ∃ s'. ?p s' unfolding id id2
          by (rule exI [of - C⟨s' · τ · σ⟩], rule conjI, rule sub-ctxt [OF steps(1)],
rule sub-ctxt [OF steps(2)])
      next
        assume mem: (b, r, l) ∈ ?CP
        from cp [OF mem] False obtain l' r' s' τ where id2: r = l' · τ l = r' · τ
and steps: (l', s') ∈ ?R^* (r', s') ∈ ?R^*
          unfolding instance-rule-def by auto
        show ∃ s'. ?p s' unfolding id id2
          by (rule exI [of - C⟨s' · τ · σ⟩], rule conjI, rule sub-ctxt [OF steps(2)],
rule sub-ctxt [OF steps(1)])
      qed
    qed
  qed auto
  then show (t1, t2) ∈ join ?R by auto
qed

lemma critical-pairs-fork:
  fixes R :: ('f, 'v) trs and S
  assumes cp: (b, l, r) ∈ critical-pairs R S
  shows (r, l) ∈ (rstep R)^-1 O rstep S
proof −
```

from *cp* **obtain** *m* **where** (*l*,*m*,*r*) ∈ *critical-Peaks R S*
unfolding *critical-pairs-def critical-Peaks-def* **by** *blast*
from *critical-Peak-steps*(*1*−*2*)[*OF this*] **show** *?thesis* **by** *auto*
qed

lemma *critical-pairs-fork'*: **assumes** (*b*,*l*,*r*) ∈ *critical-pairs R S*
shows (*l*,*r*) ∈ (*rstep S*)⁀−*1 O rstep R*
using *critical-pairs-fork*[*OF assms*] **by** *auto*

lemma *critical-pairs-complete*:
fixes *R* :: ('*f*, '*v*) *trs*
assumes *cp*: (*b*, *l*, *r*) ∈ *critical-pairs R R*
and *no-join*: (*l*, *r*) ∉ (*rstep R*)↓
shows ¬ *WCR* (*rstep R*)
proof
from *critical-pairs-fork* [*OF cp*] **obtain** *u* **where** *ul*: (*u*, *l*) ∈ *rstep R* **and** *ur*: (*u*, *r*) ∈ *rstep R* **by** *force*
assume *wcr*: *WCR* (*rstep R*)
with *ul ur no-join* **show** *False* **unfolding** *WCR-on-def* **by** *auto*
qed

lemma *critical-pair-lemma*:
fixes *R* :: ('*f*, '*v*) *trs*
shows *WCR* (*rstep R*) ⟷
(∀ (*b*, *s*, *t*) ∈ *critical-pairs R R*. (*s*, *t*) ∈ (*rstep R*)↓)
(**is** *?l* = *?r*)
proof
assume *?l*
with *critical-pairs-complete* [**where** *R* = *R*] **show** *?r* **by** *auto*
next
assume *?r*
show *?l*
proof (*rule critical-pairs*)
fix *b s t*
assume (*b*, *s*, *t*) ∈ *critical-pairs R R*
with ‹*?r*› **have** (*s*, *t*) ∈ *join* (*rstep R*) **by** *auto*
then obtain *u* **where** *s*: (*s*, *u*) ∈ (*rstep R*)⁀*
and *t*: (*t*, *u*) ∈ (*rstep R*)⁀* **by** *auto*
show ∃ *s' t' u*. *instance-rule* (*s*, *t*) (*s'*, *t'*) ∧ (*s'*, *u*) ∈ (*rstep R*)⁀* ∧ (*t'*, *u*) ∈ (*rstep R*)⁀*
proof (*intro exI conjI*)
show *instance-rule* (*s*, *t*) (*s*, *t*) **by** *simp*
qed (*insert s t*, *auto*)
qed
qed

lemma *critical-pairs-exI*:
fixes *σ* :: ('*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-pairs* $P\ R$
proof −
from *mgu-vd-complete* [*OF unif*]
obtain $\mu 1\ \mu 2$ **where** *mgu*: *mgu-vd ren* $l''\ l' =$ *Some* $(\mu 1, \mu 2)$ **by** *blast*
show *?thesis*
by (*intro exI*, *rule critical-pairsI* [*OF P R l l'' mgu refl refl b*])
qed

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*
by (*induct ys*) *simp-all*

lemma *concat-partition-by* [*simp*]:
sum-list ys = *length xs* ⟹ *concat* (*partition-by xs ys*) = *xs*
by (*induct ys arbitrary*: *xs*) *simp-all*

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*)
using *concat-nth* [*OF assms*(*1*, *2*) *refl*]
unfolding *concat-partition-by* [*OF assms*(*3*)] **by** *simp*

lemma *map-map-partition-by*:
map (*map f*) (*partition-by xs ys*) = *partition-by* (*map f xs*) *ys*
by (*induct ys arbitrary*: *xs*) (*auto simp*: *take-map drop-map*)

lemma *length-partition-by* [*simp*]:
length (*partition-by xs ys*) = *length ys*
by (*induct ys arbitrary*: *xs*) *simp-all*

lemma *partition-by-Nil* [*simp*]:
partition-by [] *ys* = *replicate* (*length ys*) []
by (*induct ys*) *simp-all*

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*
using *assms*
proof (*induct ys arbitrary*: *xss*)
case (*Cons y ys xss*)
then show *?case* **by** (*cases xss*; *fastforce*)
qed *simp*

lemma *partition-by-nth*:
i < *length ys* ⟹ *partition-by xs ys* ! *i* = *take* (*ys* ! *i*) (*drop* (*sum-list* (*take i ys*)) *xs*)
proof (*induct ys arbitrary*: *xs i*)
case (*Cons x xs i*)
thus *?case* **by** (*cases i*, *auto simp*: *ac-simps*)
qed *simp*

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 −
have *partition-by* (*xs* @ *y* # *ys*) *zs* ! *k* =
take (*zs* ! *k*) (*drop* (*sum-list* (*take k zs*)) (*xs* @ *y* # *ys*))
using *assms* **by** (*auto simp*: *partition-by-nth*)
moreover have *zs* ! *k* + *sum-list* (*take k zs*) ≤ *length xs*
using *assms* **by** (*simp add*: *sum-list-take-eq*)
ultimately show *?thesis* **by** *simp*

qed

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 −
have *partition-by* (*xs* @ y # *ys*) *zs* ! k =
take (*zs* ! k) (*drop* (*sum-list* (*take k zs*)) (*xs* @ y # *ys*))
using *assms* **by** (*auto simp*: *partition-by-nth*)
moreover have *sum-list* (*take k zs*) > *length xs*
using *assms* **by** (*auto simp*: *sum-list-take-eq*)
ultimately show *?thesis* **by** (*auto*) (*metis Suc-diff-Suc drop-Suc-Cons*)
qed

lemma *length-partition-by-nth*:
sum-list ys = *length xs* ⟹ $i <$ *length ys* ⟹ *length* (*partition-by xs ys* ! i) = *ys*
! i
proof (*induct ys arbitrary*: *xs i*)
case (*Cons y ys xs i*)
thus *?case* **by** (*cases i*, *auto*)
qed *simp*

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 −
from *assms* **have** $j <$ *length* (*partition-by xs ys* ! i) **by** (*simp only*: *length-partition-by-nth*)
then have *partition-by xs ys* ! i ! $j \in$ *set* (*partition-by xs ys* ! i) **by** *auto*
with *assms*(*2*) **have** *partition-by xs ys* ! i ! $j \in$ *set* (*concat* (*partition-by xs ys*))
by *auto*
then show *?thesis* **using** *assms* **by** *simp*
qed

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*
unfolding *partition-by-idx-def*
proof −
let *?n* = *partition-by* [*0*..<*length xs*] *ys* ! i ! j
show *?n* < *length xs*
using *partition-by-nth-nth-elem*[*OF* - *assms*(*2*,*3*), *of* [*0*..<*length xs*]] *assms*(*1*)
by *simp*
have *li*: $i <$ *length* (*partition-by* [*0*..<*length xs*] *ys*) **using** *assms*(*2*) **by** *simp*
have *lj*: $j <$ *length* (*partition-by* [*0*..<*length xs*] *ys* ! i)
using *assms* **by** (*simp add*: *length-partition-by-nth*)
have *partition-by* (*map* ((!) *xs*) [*0*..<*length xs*]) *ys* ! i ! j = *xs* ! *?n*

by (*simp only*: *map-map-partition-by*[*symmetric*] *nth-map*[*OF li*] *nth-map*[*OF lj*])
then show *partition-by xs ys ! i ! j = xs ! ?n* **by** (*simp add*: *map-nth*)
qed

lemma *map-length-partition-by* [*simp*]:
sum-list ys = length xs ⟹ map length (*partition-by xs ys*) = *ys*
by (*intro nth-equalityI*, *auto simp*: *length-partition-by-nth*)

lemma *map-partition-by-nth* [*simp*]:
i < length ys ⟹ map f (*partition-by xs ys ! i*) = *partition-by* (*map f xs*) *ys ! i*
proof (*induct ys arbitrary*: *i xs*)
case (*Cons y ys i xs*)
thus *?case* **by** (*cases i*, *simp-all add*: *take-map drop-map*)
qed *simp*

lemma *sum-list-partition-by* [*simp*]:
sum-list ys = length xs ⟹
sum-list (*map* (*λx*. *sum-list* (*map f x*)) (*partition-by xs ys*)) = *sum-list* (*map f xs*)
by (*induct ys arbitrary*: *xs*) (*simp-all*, *metis append-take-drop-id sum-list-append map-append*)

lemma *partition-by-map-conv*:
partition-by xs ys = *map* (*λi*. *take* (*ys ! i*) (*drop* (*sum-list* (*take i ys*)) *xs*)) [*0* ..< *length ys*]
by (*rule nth-equalityI*) (*simp-all add*: *partition-by-nth*)

lemma *UN-set-partition-by-map*:
sum-list ys = length xs ⟹ (⋃*x*∈*set* (*partition-by* (*map f xs*) *ys*). ⋃ (*set x*)) = ⋃(*set* (*map f xs*))
by (*induct ys arbitrary*: *xs*)
(*simp-all add*: *drop-map take-map*, *metis UN-Un append-take-drop-id set-append*)

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*)
by (*induct ys arbitrary*: *xs*) (*simp-all*, *metis UN-Un append-take-drop-id set-append*)

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*)
by *auto* (*metis atLeast0LessThan in-set-conv-nth length-partition-by lessThan-iff*)

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 (*induct ys arbitrary*: *xs*)
case (*Cons y ys*)

then show *?case*
apply (*subst* (*2*) *append-take-drop-id* [*of y xs*, *symmetric*])
apply (*simp only*: *set-append*)
apply *auto*
done
qed *simp*

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*
by (*induct ys arbitrary*: *xs*) (*auto simp*: *drop-take ac-simps split*: *split-min*)

lemma *partition-by-concat2*:
partition-by xs (*concat ys*) =
concat (*map* (λ*i* . *partition-by* (*partition-by xs* (*map sum-list ys*) ! *i*) (*ys* ! *i*)) [*0*..<*length ys*])
proof −
have ∗: *map* (λ*i* . *partition-by* (*partition-by xs* (*map sum-list ys*) ! *i*) (*ys* ! *i*)) [*0*..<*length ys*] =
map (λ(*x*,*y*). *partition-by x y*) (*zip* (*partition-by xs* (*map sum-list ys*)) *ys*)
using *zip-nth-conv*[*of partition-by xs* (*map sum-list ys*) *ys*] **by** *auto*
show *?thesis* **unfolding** ∗ **by** (*induct ys arbitrary*: *xs*) (*auto simp*: *partition-by-append2*)
qed

lemma *partition-by-partition-by*:
length xs = *sum-list* (*map sum-list ys*) ⟹
partition-by (*partition-by xs* (*concat ys*)) (*map length ys*) =
map (λ*i*. *partition-by* (*partition-by xs* (*map sum-list ys*) ! *i*) (*ys* ! *i*)) [*0*..<*length ys*]
by (*auto simp*: *partition-by-concat2 intro*: *partition-by-concat-id*)

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*
by (*induct t*) (*simp-all add*: *map-idI*)

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*)
apply (*intro ext*)
subgoal for *x* **by** (*induct x*, *auto*)
done

lemma *mctxt-of-term-term-of-mctxt-id* [*simp*]:
num-holes C = *0* ⟹ *mctxt-of-term* (*term-of-mctxt C*) = *C*
by (*induct C*) (*simp-all add*: *map-idI*)

lemma *vars-mctxt-of-term*[*simp*]: *vars-mctxt* (*mctxt-of-term t*) = *vars-term t*
by (*induct t*, *auto*)

lemma *num-holes-mctxt-of-term* [*simp*]:
num-holes (*mctxt-of-term t*) = *0*
by (*induct t*) *simp-all*

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*
by (*induct C*) *simp-all*

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)
by (*induct t*) (*simp-all cong*: *map-cong*)

definition *if-Fun-in-set F* $= (\lambda t.$ *is-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)* $\in F$
by (*simp-all add*: *if-Fun-in-set-def*)

lemma *if-Fun-in-set-mono*:
$F \subseteq G \Longrightarrow$ *if-Fun-in-set F t* $\Longrightarrow$ *if-Fun-in-set G t*
by (*auto simp*: *if-Fun-in-set-def*)

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* $= [t]$
by (*cases t*) *auto*

lemma *cap-till-funasD* [*dest*]:
fn $\in$ *funas-mctxt (cap-till-funas F t)* $\Longrightarrow$ *fn* $\in F \Longrightarrow$ *False*
proof (*induct t*)
case (*Fun f ts*)
then show *?case* **by** (*cases* $(f,$ *length ts*$) \in F$) *auto*
qed *simp*

lemma *cap-till-funas*:
$\forall$ *fn* $\in$ *funas-mctxt (cap-till-funas F t).* *fn* $\notin F$
by *auto*

lemma *uncap-till*:

$\forall s \in set\ (uncap\text{-}till\ P\ t).\ P\ s$
by (*induct t*) *simp-all*

lemma *uncap-till-singleton*:
assumes $s \in set\ (uncap\text{-}till\ P\ t)$
shows $uncap\text{-}till\ P\ s = [s]$
using *assms*
proof (*induct t*)
case (*Fun f ts*)
then show *?case* **by** (*cases P* (*Fun f ts*)) *auto*
qed *simp*

lemma *uncap-till-idemp* [*simp*]:
$map\ (uncap\text{-}till\ P)\ (uncap\text{-}till\ P\ t) = map\ (\lambda s.\ [s])\ (uncap\text{-}till\ P\ t)$
by (*intro map-cong* [*OF refl*] *uncap-till-singleton*) *simp-all*

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

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 (*insert len1 len2*, *induct C arbitrary*: *xs ys*)
case *MHole* **then show** *?case* **using** *Hole* **by** (*cases xs*; *cases ys*) *auto*
next

case (*MVar v*) **then show** *?case* **using** *Var* **by** *auto*
next
case (*MFun f Cs*) **then show** *?case* **using** *Fun*[*of Cs xs ys f*] **by** (*auto simp*: *length-partition-by-nth*)
qed

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*
using *fill-holes-induct2*[*of C xs xs* λ *C xs* -. *P C xs*] *assms* **by** *simp*

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 (*induct C ts rule*: *fill-holes-induct*)
case (*MFun f Cs ts*)
have (∃ *i* < *length Cs*. *g* ∈ *funas-term* (*fill-holes* (*Cs* ! *i*) (*partition-holes* (*concat* (*partition-holes ts Cs*)) *Cs* ! *i*)))
⟷ (∃ *C* ∈ *set Cs*. *g* ∈ *funas-mctxt C*) ∨ (∃ *us* ∈ *set* (*partition-holes ts Cs*). ∃ *t* ∈ *set us*. *g* ∈ *funas-term t*)
using *MFun* **by** (*auto simp*: *ex-set-conv-ex-nth*)
then show *?case* **by** *auto*
qed *auto*

lemma *fill-holes-MHole*:
length ts = *1* ⟹ *ts* ! *0* = *u* ⟹ *fill-holes MHole ts* = *u*
by (*cases ts*) *simp-all*

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*)
using *assms* **by** (*simp add*: *length-partition-by-nth*)

lemma *concat-partition-holes-upt*:
assumes $i \leq$ *length cs*
shows *concat* [*partition-holes ts cs* ! *j*. *j* ← [*0* ..< *i*]] =
take (*sum-list* [*num-holes* (*cs* ! *j*). *j* ← [*0* ..< *i*]]) *ts*
using *assms*
proof (*induct i arbitrary*: *ts*)
case (*Suc i*)
then have *i'*: $i <$ *length cs* **by** (*metis less-eq-Suc-le*)
then have ∗: $i <$ *length* (*map num-holes cs*) **by** *simp*
then have *i''*: $i \leq$ *length cs* **by** *auto*
show *?case*
unfolding *upt-Suc-append*[*OF le0*] *map-append concat-append Suc*(*1*)[*OF i''*]
concat.simps append-Nil2
unfolding *sum-list-append take-add*
unfolding *list.map*(*2*)
unfolding *partition-by-nth* [*OF* ∗]
unfolding *take-map nth-map* [*OF i'*]
unfolding *take-upt-idx*[*OF i'*]
unfolding *map-map o-def* **by** *auto*
qed (*auto*)

lemma *partition-holes-step*:
partition-holes ts (*C* # *Cs*) = *take* (*num-holes C*) *ts* # *partition-holes* (*drop* (*num-holes C*) *ts*) *Cs*
by *simp*

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*
using *assms* **by** (*metis nth-map-conv*)

lemma *partition-holes-concat-id*:
assumes *length sss* = *length cs*
and ⋀ *i*. $i <$ *length cs* ⟹ *num-holes* (*cs* ! *i*) = *length* (*sss* ! *i*)
shows *partition-holes* (*concat sss*) *cs* = *sss*
using *assms* **by** (*intro partition-by-concat-id*) *auto*

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*]]
by (*simp add*: *partition-by-nth take-map*)

lemma *fill-holes-arbitrary*:
assumes *lCs*: *length Cs* = *length ts*
and *lss*: *length ss* = *length ts*

and *rec*: $\bigwedge i.\ i < length\ ts \Longrightarrow num\text{-}holes\ (Cs\ !\ i) = length\ (ss\ !\ i) \wedge f\ (Cs\ !\ i)\ (ss\ !\ i) = ts\ !\ i$
shows $map\ (\lambda i.\ f\ (Cs\ !\ i)\ (partition\text{-}holes\ (concat\ ss)\ Cs\ !\ i))\ [0\ ..<\ length\ Cs] = ts$
proof −
have *sum-list* (*map num-holes Cs*) = *length* (*concat ss*) **using** *assms*
by (*auto simp*: *length-concat map-nth-eq-conv intro*: *arg-cong*[*of - - sum-list*])
moreover have *partition-holes* (*concat ss*) *Cs* = *ss*
using *assms* **by** (*auto intro*: *partition-by-concat-id*)
ultimately show *?thesis* **using** *assms* **by** (*auto intro*: *nth-equalityI*)
qed

lemma *fill-holes-MFun*:
assumes *lCs*: *length Cs* = *length ts*
and *lss*: *length ss* = *length ts*
and *rec*: $\bigwedge i.\ i < length\ ts \Longrightarrow num\text{-}holes\ (Cs\ !\ i) = length\ (ss\ !\ i) \wedge fill\text{-}holes\ (Cs\ !\ i)\ (ss\ !\ i) = ts\ !\ i$
shows *fill-holes* (*MFun f Cs*) (*concat ss*) = *Fun f ts*
unfolding *fill-holes.simps term.simps*
by (*rule conjI*[*OF refl*], *rule fill-holes-arbitrary*[*OF lCs lss rec*])

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*
using *assms*
proof (*induct C ss ts rule*: *fill-holes-induct2*)
case (*MFun f Cs ss ts*)
then show *?case* **by** (*intro arg-cong*[*of - - concat*] *nth-equalityI*) *auto*
qed *auto*

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

lemma *eqfE*:
assumes *t* =$_f$ (*D*, *ss*) **shows** *t* = *fill-holes D ss num-holes D* = *length ss*
using *assms*[*unfolded eq-fill.simps*] **by** *auto*

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

```
    and⋀ i. i < length Cs ⟹ ts ! i =f (Cs ! i, sss ! i)
  shows Fun f ts =f (MFun f Cs, concat sss)
proof
  have num-holes (MFun f Cs) = sum-list (map num-holes Cs) by simp
  also have map num-holes Cs = map length sss
    by (rule nth-equalityI, insert assms eqfE[OF assms(3)], auto)
  also have sum-list (...) = length (concat sss) unfolding length-concat ..
  finally show num-holes (MFun f Cs) = length (concat sss) .
  show Fun f ts = fill-holes (MFun f Cs) (concat sss)
     by (rule fill-holes-MFun[symmetric], insert assms(1,2) eqfE[OF assms(3)],
auto)
qed

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
    ⋀ i. i < length Cs ⟹ ts ! i =f (Cs ! i, sss ! i)
    ss = concat sss
proof −
  from eqfE[OF assms] have fh: s = fill-holes (MFun f Cs) ss
    and nh: sum-list (map num-holes Cs) = length ss by auto
  from fh obtain ts where s: s = Fun f ts by (cases s, auto)
  from fh[unfolded s]
  have ts: ts = map (λi. fill-holes (Cs ! i) (partition-holes ss Cs ! i)) [0..<length
Cs]
    (is - = map (?f Cs ss) -)
    by auto
  let ?sss = partition-holes ss Cs
  from nh
  have ∗: length ?sss = length Cs ⋀i. i < length Cs ⟹ ts ! i =f (Cs ! i, ?sss !
i) ss = concat ?sss
    by (auto simp: ts)
  have len: length ts = length Cs unfolding ts by auto
  assume ass: ⋀ts sss. s = Fun f ts ⟹
              length ts = length Cs ⟹
             length sss = length Cs ⟹ (⋀i. i < length Cs ⟹ ts ! i =f (Cs ! i, sss
! i)) ⟹ ss = concat sss ⟹ thesis
  show thesis
    by (rule ass[OF s len ∗])
qed

lemma eqf-MHoleE:
  assumes s =f (MHole, ss)
  shows ss = [s]
  using assms
proof (cases ss)
  case (Cons x xs) with assms show ?thesis by (cases xs) (auto dest: eqfE)
qed (auto dest: eqfE)
```

```
fun mctxt-of-ctxt :: ('f, 'v) ctxt ⇒ ('f, 'v) mctxt
  where
    mctxt-of-ctxt Hole = MHole |
    mctxt-of-ctxt (More f ss₁ C ss₂) =
    MFun f (map mctxt-of-term ss₁ @ mctxt-of-ctxt C # map mctxt-of-term ss₂)

lemma num-holes-mctxt-of-ctxt [simp]:
  num-holes (mctxt-of-ctxt C) = 1
  by (induct C) simp-all

lemma mctxt-of-term: t =_f (mctxt-of-term t, [])
proof (induct t)
  case (Var x)
  show ?case by auto
next
  case (Fun f ts)
  let ?ss = map (λ -. []) ts
  have id: concat ?ss = [] by simp
  have ?case = (Fun f ts =_f (MFun f (map mctxt-of-term ts), concat ?ss)) unfolding id by simp
  also have ...
    by (rule eqf-MFunI, insert Fun[unfolded set-conv-nth], auto)
  finally show ?case .
qed

lemma mctxt-of-ctxt [simp]:
  C⟨t⟩ =_f (mctxt-of-ctxt C, [t])
proof (induct C)
  case (More f bef C aft)
  let ?sss = map (λ -. []) bef @ [t] # map (λ -. []) aft
  let ?ts = map mctxt-of-term bef @ mctxt-of-ctxt C # map mctxt-of-term aft
  have id: concat ?sss = [t] by (induct bef, auto)
  have ?case =
    (Fun f (bef @ C⟨t⟩ # aft) =_f (MFun f ?ts, concat ?sss))
    unfolding id by simp
  also have ...
  proof (rule eqf-MFunI)
    fix i
    assume i: i < length ?ts
    show (bef @ C⟨t⟩ # aft) ! i =_f (?ts ! i, ?sss ! i)
      using More i
      by (cases i < length bef, simp add: nth-append mctxt-of-term,
          cases i = length bef, auto simp: nth-append mctxt-of-term)
  qed auto
  finally show ?case .
qed auto

lemma fill-holes-ctxt-main':
  assumes num-holes C = Suc (length bef + length aft)
```

shows $\exists\ D.\ (\forall\ s.\ \mathit{fill\text{-}holes}\ C\ (\mathit{bef}\ @\ s\ \#\ \mathit{aft}) = D\ \langle s\rangle) \wedge (C = \mathit{MFun}\ f\ \mathit{cs} \longrightarrow D \neq \Box)$
using *assms*
proof (*induct C arbitrary*: *bef aft*)
case *MHole*
show *?case*
by (*rule exI*[*of* - $\Box$], *insert MHole*, *auto*)
next
case (*MFun f cs*)
note *IH* = *MFun*(*1*)
note *holes* = *MFun*(*2*)
let *?p* = λ *bef aft b a D cs s*. *map* (λi. *fill-holes* (*cs* ! *i*)
(*partition-holes* (*bef* @ *s* # *aft*) *cs* ! *i*)) [*0*..<*length cs*] =
b @ *D*$\langle s\rangle$ # *a*
from *holes IH*
have $\exists$ *b D a*. $\forall$ *s*. *?p bef aft b a D cs s*
proof (*induct cs arbitrary*: *bef*)
case (*Cons c ccs*)
have *len*: *length* (*c* # *ccs*) = *Suc* (*length ccs*) **by** *simp*
show *?case*
proof (*cases num-holes c* $\leq$ *length bef*)
case *True*
then have *bef* = *take* (*num-holes c*) *bef* @ *drop* (*num-holes c*) *bef*
$\wedge$ *length* (*take* (*num-holes c*) *bef*) = *num-holes c* **by** *auto*
then obtain *bc ba* **where** *bef*: *bef* = *bc* @ *ba* **and** *lbc*: *length bc* = *num-holes c* **by** *blast*
from *Cons*(*2*) **have** *nh*: *num-holes* (*MFun f ccs*) = *Suc* (*length ba* + *length aft*) **unfolding** *bef*
by (*simp add*: *lbc*)
from *Cons*(*1*)[*OF nh Cons*(*3*)] **obtain** *b D a* **where** *IH*: $\bigwedge$ *s*. *?p ba aft b a D ccs s* **by** *auto*
show *?thesis* **unfolding** *len map-upt-Suc bef*
by (*intro exI*[*of* - *fill-holes c bc* # *b*] *exI*[*of* - *D*] *exI*[*of* - *a*], *insert IH lbc*, *auto*)
next
case *False*
then have $\exists$ *la*. *num-holes c* = *Suc* (*length bef* + *la*) **by** *arith*
then obtain *la* **where** *nhc*: *num-holes c* = *Suc* (*length bef* + *la*) **..**
from *Cons*(*2*) *nhc* **have** *length* (*take la aft*) = *la* **by** *auto*
from *Cons*(*3*)[*of c bef take la aft*, *unfolded this*, *OF* - *nhc*]
obtain *D* **where** *D*: $\forall$ *s*. *fill-holes c* (*bef* @ *s* # *take la aft*) = *D*$\langle s\rangle$ **by** *auto*
show *?thesis* **unfolding** *len map-upt-Suc*
by (*rule exI*[*of* - *Nil*], *rule exI*[*of* - *D*], *simp add*: *nhc D*)
qed
qed *auto*
then obtain *b D a* **where** *main*: $\bigwedge$ *s*. *?p bef aft b a D cs s* **by** *blast*
show *?case* **by** (*rule exI*[*of* - *More f b D a*], *insert main*, *auto*)
qed *simp*

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* ⟩
using *assms fill-holes-ctxt-main′* **by** *fast*

lemma *fill-holes-ctxt*:
assumes *nh*: *num-holes C* = *length ss*
and *i*: *i* < *length ss*
obtains *D* **where** ⋀ *s*. *fill-holes C* (*ss*[*i* := *s*]) = *D* ⟨ *s* ⟩
proof −
from *id-take-nth-drop*[*OF i*] **obtain** *bef aft* **where** *ss*: *ss* = *bef* @ *ss* ! *i* # *aft*
and *bef*: *bef* = *take i ss* **by** *blast*
from *bef i* **have** *bef*: *length bef* = *i* **by** *auto*
note *len* = *arg-cong*[*OF ss*, *of length*]
from *len nh*
have *num-holes C* = *Suc* (*length bef* + *length aft*) **by** *simp*
from *fill-holes-ctxt-main*[*OF this*] **obtain** *D* **where** *id*: ⋀ *s*. *fill-holes C* (*bef* @ *s* # *aft*) = *D* ⟨ *s* ⟩ **by** *blast*
{
fix *s*
have *ss*[*i* := *s*] = *bef* @ *s* # *aft* **unfolding** *arg-cong*[*OF ss*, *of* λ *ss*. *ss*[*i* := *s*]]
using *i bef* **by** *auto*
with *id*[*of s*] **have** *fill-holes C* (*ss*[*i* := *s*]) = *D* ⟨*s* ⟩ **by** *simp*
}
then have *main*: ∃ *D*. ∀ *s*. *fill-holes C* (*ss*[*i* := *s*]) = *D* ⟨*s* ⟩ **by** *blast*
assume ⋀ *D*. ⟦⋀*s*. *fill-holes C* (*ss*[*i* := *s*]) = *D*⟨*s*⟩⟧ ⟹ *thesis*
with *main* **show** *thesis* **by** *blast*
qed

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*
by (*induct C*, *auto intro*: *nth-equalityI*)

lemma *num-holes-map-vars-mctxt* [*simp*]:
num-holes (*map-vars-mctxt vw C*) = *num-holes C*
proof (*induct C*)
case (*MFun f Cs*)
then show *?case* **by** (*induct Cs*, *auto*)
qed *auto*

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 (*induct C arbitrary*: *t ss*)
case (*MFun f Cs s ss*)
from *eqf-MFunE*[*OF MFun*(*2*)] **obtain** *ts sss* **where** *s*: *s* = *Fun f ts* **and** *len*: *length ts* = *length Cs length sss* = *length Cs*
and *IH*: ⋀ *i*. *i* < *length Cs* ⟹ *ts* ! *i* =$_f$ (*Cs* ! *i*, *sss* ! *i*) **and** *ss*: *ss* = *concat sss* **by** *metis*
{
fix *i*
assume *i*: *i* < *length Cs*
then have *Cs* ! *i* ∈ *set Cs* **by** *auto*
from *MFun*(*1*)[*OF this IH*[*OF i*]] **have** *map-vars-term vw* (*ts* ! *i*) =$_f$ (*map-vars-mctxt vw* (*Cs* ! *i*), *map* (*map-vars-term vw*) (*sss* ! *i*)) **.**
} note *IH* = *this*
show *?case* **unfolding** *map-vars-mctxt.simps ss map-concat s term.map*
by (*rule eqf-MFunI*, *insert IH len*, *auto*)
next
case (*MHole t ss*)
from *eqfE*[*OF this*]
show *?case* **by** (*cases ss*, *auto*)
next
case (*MVar v t ss*)
from *eqfE*[*OF this*]
show *?case* **by** (*cases ss*, *auto*)
qed

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 −
from *eqfE*[*OF map-vars-term-eq-fill*[*OF eqfI*[*OF refl nh*]]]
show *?thesis* **by** *simp*
qed

lemma *split-term-eqf*:
t =$_f$ (*cap-till P t*, *uncap-till P t*)
proof (*induct t*)
case (*Fun f ts*)
show *?case*
proof (*cases P* (*Fun f ts*))
case *False*
then have *?thesis* = (*Fun f ts* =$_f$ (*MFun f* (*map* (*cap-till P*) *ts*), *concat* (*map* (*uncap-till P*) *ts*)))
by *simp*
also have . . .
proof (*rule eqf-MFunI*)
fix *i*
presume *i* < *length ts*
moreover then have *ts* ! *i* ∈ *set ts* **by** *auto*

```
    ultimately show ts ! i =f (map (cap-till P) ts ! i, map (uncap-till P) ts ! i)
      using Fun by auto
  qed simp-all
  finally show ?thesis .
 qed auto
qed auto

lemma fill-holes-cap-till-uncap-till-id [simp]:
 fill-holes (cap-till P t) (uncap-till P t) = t
proof −
 have t =f (cap-till P t, uncap-till P t) by (metis split-term-eqf)
 from eqfE [OF this] show ?thesis by simp
qed

lemma num-holes-cap-till [simp]:
 num-holes (cap-till P t) = length (uncap-till P t)
 using eqfE [OF split-term-eqf] by auto

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 (induct t)
 case (Fun f ts)
 then show ?case by (induct ts, auto)
qed simp

lemma split-vars-vars-term-list: snd (split-vars t) = vars-term-list t
proof (induct t)
 case (Fun f ts)
 then show ?case by (auto simp: vars-term-list.simps o-def, induct ts, auto)
qed (auto simp: vars-term-list.simps)

lemma split-vars-vars-term: set (snd (split-vars t)) = vars-term t
 using arg-cong[OF split-vars-vars-term-list[of t], of set] by auto

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 (induct t)
 case (Fun f ts)
 have ?case = (Fun f (map (λt. t · σ) ts)
   =f (MFun f (map (map-vars-mctxt vw ∘ (fst ∘ split-vars)) ts), concat (map
(map σ ∘ (snd ∘ split-vars)) ts)))
   by (simp add: map-concat)
 also have ...
```

proof (*rule eqf-MFunI*, *unfold length-map*)
fix *i*
assume *i*: $i < length\ ts$
then have *mem*: $ts\ !\ i \in set\ ts$ **by** *auto*
show $map\ (\lambda t.\ t \cdot \sigma)\ ts\ !\ i =_f (map\ (map\text{-}vars\text{-}mctxt\ vw \circ (fst \circ split\text{-}vars))\ ts\ !\ i,\ map\ (map\ \sigma \circ (snd \circ split\text{-}vars))\ ts\ !\ i)$
using *Fun*[*OF mem*] *i* **by** *auto*
qed *auto*
finally show *?case* **by** *simp*
qed *auto*

lemma *split-vars-eqf-subst*: $t \cdot \sigma =_f (fst\ (split\text{-}vars\ t),\ (map\ \sigma\ (snd\ (split\text{-}vars\ t))))$
using *split-vars-eqf-subst-map-vars-term*[*of t* σ $\lambda\ x.\ x$] **by** *simp*

lemma *split-vars-into-subst-map-vars-term*:
assumes *split*: $split\text{-}vars\ l = (C,xs)$
and *len*: $length\ ts = length\ xs$
and *id*: $\bigwedge i.\ i < length\ xs \Longrightarrow \sigma\ (xs\ !\ i) = ts\ !\ i$
shows $l \cdot \sigma =_f (map\text{-}vars\text{-}mctxt\ vw\ C,ts)$
proof −
from *split-vars-eqf-subst-map-vars-term*[*of l* σ *vw*, *unfolded split*]
have $l \cdot \sigma =_f (map\text{-}vars\text{-}mctxt\ vw\ C,\ map\ \sigma\ xs)$ **by** *simp*
also have $map\ \sigma\ xs = ts$
by (*rule nth-equalityI*, *insert len id*, *auto*)
finally show *?thesis* **.**
qed

lemma *split-vars-into-subst*:
assumes *split*: $split\text{-}vars\ l = (C,xs)$
and *len*: $length\ ts = length\ xs$
and *id*: $\bigwedge i.\ i < length\ xs \Longrightarrow \sigma\ (xs\ !\ i) = ts\ !\ i$
shows $l \cdot \sigma =_f (C,ts)$
using *split-vars-into-subst-map-vars-term*[*OF split len id*, *of* $\lambda\ x.\ x$] **by** *simp*

lemma *eqf-funas-term*:
$t =_f (C,ss) \Longrightarrow funas\text{-}term\ t = funas\text{-}mctxt\ C \cup \bigcup (funas\text{-}term\ `\ set\ ss)$
proof (*induct C arbitrary*: *t ss*)
case (*MFun f Cs t ss*)
from *eqf-MFunE*[*OF MFun*(*2*)] **obtain** *ts sss* **where**
t: $t = Fun\ f\ ts$ **and** *len*: $length\ ts = length\ Cs\ length\ sss = length\ Cs$
and *args*: $\bigwedge i.\ i < length\ Cs \Longrightarrow ts\ !\ i =_f (Cs\ !\ i,\ sss\ !\ i)$
and *ss*: $ss = concat\ sss$ **by** *auto*
let *?lhs* = $\bigcup \{funas\text{-}term\ (ts\ !\ i) \mid i.\ i < length\ Cs\}$
let *?f1* = $\lambda\ i.\ funas\text{-}mctxt\ (Cs\ !\ i)$
let *?f2* = $\lambda\ i.\ \bigcup (funas\text{-}term\ `\ set\ (sss\ !\ i))$
let *?f* = $\lambda\ i.\ ?f1\ i \cup ?f2\ i$
{
fix *i*

assume *i*: *i* < *length Cs*
then have *mem*: *Cs* ! *i* ∈ *set Cs* **by** *auto*
note *MFun*(*1*)[*OF mem args*[*OF i*]]
} **note** *IH* = *this*
have *funas-term t* = *insert* (*f*,*length Cs*) *?lhs*
unfolding *t* **using** *len* **by** (*auto simp*: *set-conv-nth*)
also have *?lhs* = ⋃ {*?f i* | *i*. *i* < *length Cs*} **using** *IH* **by** *blast*
also have ... = ⋃ {*?f1 i* | *i*. *i* < *length Cs*} ∪ ⋃ {*?f2 i* | *i*. *i* < *length Cs*} **by** *auto*
also have *insert* (*f*,*length Cs*) ... = (*insert* (*f*,*length Cs*) (⋃ {*?f1 i* | *i*. *i* < *length Cs*})) ∪ ⋃ {*?f2 i* | *i*. *i* < *length Cs*} **by** *auto*
also have *insert* (*f*,*length Cs*) (⋃ {*?f1 i* | *i*. *i* < *length Cs*}) = *funas-mctxt* (*MFun f Cs*)
by (*auto simp*: *set-conv-nth*)
also have ⋃ {*?f2 i* | *i*. *i* < *length Cs*} = ⋃(*funas-term* ‘ *set ss*) **unfolding** *ss len*(*2*)[*symmetric*]
using *set-conv-nth*[*of sss*] **by** *auto*
finally show *?case* **.**
next
case *MVar*
from *eqfE*[*OF this*]
show *?case* **by** *auto*
next
case *MHole*
from *eqfE*[*OF this*] **show** *?case* **by** (*cases ss*, *auto*)
qed

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*)
using *ass*
proof (*induct t* (*D*,*ss*) *rule*: *eq-fill.induct*)
case (*eqfI t*)
from *eqfI*(*2*) *eqfI*(*4*)[*unfolded eqfI*(*2*)[*symmetric*]] *eqfI*(*3*,*5*,*6*)
show *?case* **unfolding** *eqfI*(*1*)
proof (*induct D ss ts rule*: *fill-holes-induct2*)
case (*MVar v*) **then show** *?case* **using** *all-ctxt-closed-sig-reflE*[*OF ctxt*] **by** *auto*
next
case (*MHole s*′ *t*′) **then show** *?case* **by** *auto*
next
case (*MFun f Cs ss ts*)
let *?ss* = (*map* (λ*i*. *fill-holes* (*Cs* ! *i*) (*partition-holes ss Cs* ! *i*)) [*0*..<*length Cs*])
let *?ts* = (*map* (λ*i*. *fill-holes* (*Cs* ! *i*) (*partition-holes ts Cs* ! *i*)) [*0*..<*length Cs*])

```
    note ∗ = all-ctxt-closedD[OF ctxt, of f ?ts ?ss, unfolded length-map length-upt
minus-nat.diff-0]
   show ?case unfolding fill-holes.simps MFun(4) concat-partition-by[OF MFun(1)]
concat-partition-by[OF MFun(2)]
    proof (intro conjI eqfI ∗)
      fix i assume i: i < length Cs
      then have ∗: i < length ?ss i < length ?ts by auto
     from ∗(1) MFun(1,5) have g1: funas-term (fill-holes (Cs ! i) (partition-holes
ss Cs ! i)) ⊆ F
        by (auto simp: subset-eq)
      with ∗(1) show funas-term (?ss ! i) ⊆ F by auto
     from ∗(2) MFun(2,6) have g2: (⋃ a∈set (partition-holes ts Cs ! i). funas-term
a) ⊆ F
        unfolding set-concat
        by (auto simp: subset-eq all-set-conv-all-nth[of partition-holes ts Cs])
      with ∗(2) MFun(1,2,5) show funas-term (?ts ! i) ⊆ F
        by (auto simp: funas-term-fill-holes-iff subset-eq)
      {
        fix j assume j: j < length (partition-holes ts Cs ! i)
        from partition-by-nth-nth[of map num-holes Cs ss i j]
          partition-by-nth-nth[of map num-holes Cs ts i j]
          i j MFun(1,2,4)
        have (partition-holes ss Cs ! i ! j, partition-holes ts Cs ! i ! j) ∈ R by simp
      }
     with i show (?ss ! i, ?ts ! i) ∈ R by (auto intro!: conjunct1[OF MFun(3)[OF
i - g1 g2]])
    next
      show (f, length Cs) ∈ F using MFun(5) by auto
    next
      show Fun f ?ts =f (MFun f Cs, ts) using MFun(2) by (intro eq-fill.intros)
auto
    qed simp
  qed
qed

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)
```

by (*induct t*) *simp-all*

lemma *ground-eq-fill*: $t =_f (C,ss) \Longrightarrow$ *ground* $t =$ (*ground-mctxt* $C \wedge (\forall\ s \in set\ ss.\ ground\ s)$)
proof (*induct C arbitrary*: *t ss*)
case (*MVar x*)
from *eqfE*[*OF this*] **show** *?case* **by** *simp*
next
case (*MHole t ss*)
from *eqfE*[*OF this*] **show** *?case* **by** (*cases ss*, *auto*)
next
case (*MFun f Cs s ss*)
from *eqf-MFunE*[*OF MFun*(*2*)] **obtain** *ts sss* **where** *s*: $s = Fun\ f\ ts$ **and** *len*: *length ts* = *length Cs length sss* = *length Cs*
and *IH*: $\bigwedge i.\ i < length\ Cs \Longrightarrow ts\ !\ i =_f (Cs\ !\ i, sss\ !\ i)$ **and** *ss*: *ss* = *concat sss* **by** *metis*
{
fix *i*
assume *i*: $i < length\ Cs$
then have $Cs\ !\ i \in set\ Cs$ **by** *simp*
from *MFun*(*1*)[*OF this IH*[*OF i*]]
have *ground* $(ts\ !\ i) =$ (*ground-mctxt* $(Cs\ !\ i) \wedge (\forall\, a{\in}set\ (sss\ !\ i).\ ground\ a)$) **.**
} note *IH* = *this*
note *conv* = *set-conv-nth*
have *?case* = $((\forall\, x{\in}set\ ts.\ ground\ x) = ((\forall\, x{\in}set\ Cs.\ ground\text{-}mctxt\ x) \wedge (\forall\, a{\in}set\ sss.\ \forall\, x{\in}set\ a.\ ground\ x)))$
unfolding *s ss* **by** *simp*
also have ... **unfolding** *conv*[*of ts*] *conv*[*of Cs*] *conv*[*of sss*] *len* **using** *IH* **by** *auto*
finally show *?case* **by** *simp*
qed

lemma *ground-fill-holes*:
assumes *nh*: *num-holes C* = *length ss*
shows *ground* (*fill-holes C ss*) = (*ground-mctxt* $C \wedge (\forall\ s \in set\ ss.\ ground\ s)$)
by (*rule ground-eq-fill*[*OF eqfI*[*OF refl nh*]])

lemma *split-vars-ground*: *split-vars* $t = (C,xs) \Longrightarrow$ *ground-mctxt C*
proof (*induct t arbitrary*: *C xs*)
case (*Fun f ts C xs*)
from *Fun*(*2*)[*simplified*] **obtain** *Cs* **where** *C*: $C = MFun\ f\ Cs$ **and** *Cs*: *Cs* = *map* (*fst* ∘ *split-vars*) *ts* **by** *auto*
show *?case* **unfolding** *C ground-mctxt.simps*
proof
fix *C*
assume $C \in set\ Cs$
from *this*[*unfolded Cs*] **obtain** *t* **where** *t*: $t \in set\ ts$ **and** *C*: $C = fst\ (split\text{-}vars\ t)$ **unfolding** *o-def* **by** *auto*
from *C* **obtain** *xs* **where** *split*: *split-vars* $t = (C,xs)$ **by** (*cases split-vars t*,

auto)
  show *ground-mctxt C*
   by (*rule Fun*(*1*)[*OF t split*])
 qed
qed *auto*

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*)
 using *assms*
proof (*induct C arbitrary*: *xs*)
 case (*MHole xs*)
 then show *?case* **by** (*cases xs*, *auto*)
next
 case (*MFun f Cs xs*)
 have *fill-holes* (*MFun f Cs*) (*map Var xs*) =$_f$ (*MFun f Cs*, *map Var xs*)
  by (*rule eqfI*, *insert MFun*(*3*), *auto*)
 from *eqf-MFunE*[*OF this*]
 obtain *ts xss* **where** *fh*: *fill-holes* (*MFun f Cs*) (*map Var xs*) = *Fun f ts*
  and *lent*: *length ts* = *length Cs*
  and *lenx*: *length xss* = *length Cs*
  and *args*: ⋀*i*. *i* < *length Cs* ⟹ *ts* ! *i* =$_f$ (*Cs* ! *i*, *xss* ! *i*)
  and *id*: *map Var xs* = *concat xss* **by** *auto*
 from *arg-cong*[*OF id*, *of map the-Var*] **have** *id2*: *xs* = *concat* (*map* (*map the-Var*) *xss*)
  by (*metis map-concat length-map map-nth-eq-conv term.sel*(*1*))
 {
  fix *i*
  assume *i*: *i* < *length Cs*
  then have *mem*: *Cs* ! *i* ∈ *set Cs* **by** *auto*
  with *MFun*(*2*) **have** *ground*: *ground-mctxt* (*Cs* ! *i*) **by** *auto*
  have *map Var* (*map the-Var* (*xss* ! *i*)) = *map id* (*xss* ! *i*) **unfolding** *map-map o-def map-eq-conv*
  proof
   fix *x*
   assume *x* ∈ *set* (*xss* ! *i*)
   with *lenx i* **have** *x* ∈ *set* (*concat xss*) **by** *auto*
   from *this*[*unfolded id*[*symmetric*]] **show** *Var* (*the-Var x*) = *id x* **by** *auto*
  qed
  then have *idxss*: *map Var* (*map the-Var* (*xss* ! *i*)) = *xss* ! *i* **by** *auto*
  note *rec* = *eqfE*[*OF args*[*OF i*]]
  note *IH* = *MFun*(*1*)[*OF mem ground*, *of map the-Var* (*xss* ! *i*), *unfolded rec*(*2*) *idxss rec*(*1*)[*symmetric*]]
  from *IH* **have** *split-vars* (*ts* ! *i*) = (*Cs* ! *i*, *map the-Var* (*xss* ! *i*)) **by** *auto*
  note *this idxss*
 }
 note *IH* = *this*
 have *?case* = (*map fst* (*map split-vars ts*) = *Cs* ∧ *concat* (*map snd* (*map split-vars ts*)) = *concat* (*map* (*map the-Var*) *xss*))

unfolding *fh* **unfolding** *id2* **by** *auto*
also have ...
proof (*rule conjI*[*OF nth-equalityI arg-cong*[*of - - concat*, *OF nth-equalityI*, *rule-format*]], *unfold length-map lent lenx*)
fix *i*
assume *i*: *i* < *length Cs*
with *arg-cong*[*OF IH*(*2*)[*OF this*], *of map the-Var*]
IH[*OF this*] **show** *map snd* (*map split-vars ts*) ! *i* = *map* (*map the-Var*) *xss* ! *i* **using** *lent lenx* **by** *auto*
qed (*insert IH lent*, *auto*)
finally show *?case* **.**
qed *auto*

lemma *ground-map-mctxt*[*simp*]: *ground-mctxt* (*map-mctxt fg C*) = *ground-mctxt C*
by (*induct C*, *auto*)

lemma *num-holes-map-mctxt*[*simp*]: *num-holes* (*map-mctxt fg C*) = *num-holes C*
proof (*induct C*)
case (*MFun f Cs*)
then show *?case* **by** (*induct Cs*, *auto*)
qed *auto*

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 −
from *split-vars-ground*[*OF split*] **have** *ground*: *ground-mctxt C* **by** *simp*
from *split-vars-num-holes*[*of t*, *unfolded split*] **have** *nh*: *num-holes C* = *length xs* **by** *auto*
show *?thesis*
by (*rule split-vars-ground-vars*[*OF ground nh*])
qed

lemma *subst-eq-map-decomp*:
assumes *t* · *σ* = *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*))
using *assms*
proof (*induct t arbitrary*: *s*)
case (*Var x s*)
show *?case*
by (*intro exI*[*of - MHole*] *exI*[*of - [x]] *exI*[*of - [s]], *insert Var*, *auto*)
next
case (*Fun g ts s*)
from *Fun*(*2*) **obtain** *f ss* **where** *s*: *s* = *Fun f ss* **and** *g*: *g* = *fg f* **by** (*cases s*, *auto*)
from *Fun*(*2*)[*unfolded s*] **have** *id*: *map* (λ *t*. *t* · *σ*) *ts* = *map* (*map-funs-term fg*)

ss **by** *auto*
from *arg-cong*[*OF this*, *of length*] **have** *len*: *length ts* = *length ss* **by** *auto*
from *map-nth-conv*[*OF id*] **have** *args*: $\bigwedge$ *i*. *i* < *length ts* ⟹ *ts* ! *i* · *σ* = *map-funs-term fg* (*ss* ! *i*) **by** *auto*
let *?P* = λ *C xs δs i*. *ss* ! *i* =$_f$ (*C*, *δs*) ∧
split-vars (*ts* ! *i*) = (*map-mctxt fg C*, *xs*) ∧
(∀ *i*<*length xs*. *σ* (*xs* ! *i*) = *map-funs-term fg* (*δs* ! *i*))
{
fix *i*
assume *i*: *i* < *length ts*
then have *mem*: *ts* ! *i* ∈ *set ts* **by** *auto*
note *IH* = *Fun*(*1*)[*OF this args*[*OF i*]]
}
then have ∀ *i*. ∃ *C xs δs*. *i* < *length ts* ⟶ *?P C xs δs i* **by** *blast*
from *choice*[*OF this*] **obtain** *Cs* **where** ∀ *i*. ∃ *xs δs*. *i* < *length ts* ⟶ *?P* (*Cs i*) *xs δs i* **by** *blast*
from *choice*[*OF this*] **obtain** *xss* **where** ∀ *i*. ∃ *δs*. *i* < *length ts* ⟶ *?P* (*Cs i*) (*xss i*) *δs i* **by** *blast*
from *choice*[*OF this*] **obtain** *δss* **where** *IH*: $\bigwedge$ *i*. *i* < *length ts* ⟹ *?P* (*Cs i*) (*xss i*) (*δss i*) *i* **by** *blast*
let *?n* = [*0* ..< *length ts*]
let *?Cs* = *map Cs ?n*
let *?C* = *MFun f ?Cs*
let *?xs* = *concat* (*map xss ?n*)
let *?δs* = *concat* (*map δss ?n*)
let *?g* = *fg f*
show *?case* **unfolding** *s g*
proof (*rule exI*[*of* - *?C*], *rule exI*[*of* - *?xs*], *rule exI*[*of* - *?δs*], *intro conjI*)
show *Fun f ss* =$_f$ (*?C*, *?δs*)
by (*rule eqf-MFunI*, *insert IH len*, *auto*)
next
have
(*split-vars* (*Fun ?g ts*) = (*map-mctxt fg ?C*, *?xs*))
= (*map* (*fst* ∘ *split-vars*) *ts* = *map* (*map-mctxt fg* ∘ *Cs*) [*0*..<*length ss*]
∧*concat* (*map* (*snd* ∘ *split-vars*) *ts*) = *?xs*)
(**is** *?goal* = -)
using *len* **by** *auto*
also have ...
by (*rule conjI*[*OF nth-map-conv arg-cong*[*of* - - *concat*, *OF nth-equalityI*]], *insert IH len*, *auto*)
finally show *?goal* **.**
next
show ∀ *i* < *length ?xs*. *σ* (*?xs* ! *i*) = *map-funs-term fg* (*?δs* ! *i*)
proof (*rule concat-all-nth*, *unfold length-map length-upt*)
fix *i*
assume *i* < *length ts* − *0*
then have *i*: *i* < *length ts* **by** *auto*
from *IH*[*OF i*] **have** *split-vars* (*ts* ! *i*) = (*map-mctxt fg* (*Cs i*), *xss i*) **by** *blast*
from *split-vars-map-mctxt*[*OF this*] *split-vars-num-holes*[*of fill-holes* (*Cs i*)

```
(map Var (xss i))]
      have len: length (xss i) = num-holes (Cs i) by simp
      also have ... = length (δss i) by (rule eqfE(2), insert IH[OF i], auto)
      finally
      show length (map xss [0..<length ts] ! i) = length (map δss [0..<length ts] !
i) using i by auto
    qed (insert IH, auto)
  qed
qed

lemma map-funs-term-fill-holes:
  num-holes C = length ss ⟹
    map-funs-term fg (fill-holes C ss) =f (map-mctxt fg C, map (map-funs-term fg)
ss)
proof (induct C arbitrary: ss)
  case (MHole ss)
  then show ?case by (cases ss, auto)
next
  case MVar then show ?case by auto
next
  case (MFun f Cs ss)
  from MFun(2) have fill-holes (MFun f Cs) ss =f (MFun f Cs, ss) by auto
  from eqf-MFunE[OF this] obtain ts sss where fh: fill-holes (MFun f Cs) ss =
Fun f ts
    and lts: length ts = length Cs
    and lsss: length sss = length Cs
    and args: ⋀i. i < length Cs ⟹ ts ! i =f (Cs ! i, sss ! i)
    and sss: ss = concat sss by auto
  {
    fix i
    assume i: i < length Cs
    then have mem: Cs ! i ∈ set Cs by auto
    from MFun(1)[OF mem]  eqfE[OF args[OF i]] have
      map-funs-term fg (ts ! i) =f (map-mctxt fg (Cs ! i), map (map-funs-term fg)
(sss ! i)) by auto
  } note IH = this
  show ?case unfolding fh
    unfolding map-mctxt.simps sss map-concat term.simps
  proof (rule eqf-MFunI, unfold length-map)
    fix i
    assume i: i < length Cs
    have (map (map-funs-term fg) ts ! i =f (map (map-mctxt fg) Cs ! i, map (map
(map-funs-term fg)) sss ! i)) =
      (map-funs-term fg (ts ! i) =f (map-mctxt fg (Cs ! i), map (map-funs-term fg)
(sss ! i))) (is ?goal = -)
      using i lts lsss by auto
    also have ... by (rule IH[OF i])
    finally show ?goal .
  qed (auto simp: lsss lts)
```

qed

lemma *eqf-MVarE*:
assumes $s =_f (MVar\ x, ss)$
shows $s = Var\ x\ ss = []$
by (*insert eqfE*[*OF assms*], *cases s*; *cases ss*, *auto*)+

lemma *eqf-imp-subt*:
assumes *s*: $s =_f (C, ts)$
and *t*: $t \in set\ ts$
shows $s \trianglerighteq t$
proof −
from *t* **obtain** *bef aft* **where** *ts*: $ts = bef\ @\ t\ \#\ aft$
by (*metis split-list*)
note $s = eqfE[OF\ s[unfolded\ ts], simplified]$
from *fill-holes-ctxt-main*[*OF s*(*2*)] **obtain** *D* **where** *fill-holes C* $(bef\ @\ t\ \#\ aft) = D\langle t\rangle$ **by** *auto*
from *this*[*folded s*(*1*)] **show** *?thesis* **by** *auto*
qed

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 −
from *t* **obtain** *bef aft* **where** *ts*: $ts = bef\ @\ t\ \#\ aft$
by (*metis split-list*)
from *eqfE*[*OF s*[*unfolded ts*]] **have** *s*: $s = fill\text{-}holes\ (MFun\ f\ cs)\ (bef\ @\ t\ \#\ aft)$ $num\text{-}holes\ (MFun\ f\ cs) = Suc\ (length\ bef + length\ aft)$ **by** *auto*
from *fill-holes-ctxt-main'*[*OF s*(*2*)] **obtain** *D*
where *D*:$fill\text{-}holes\ (MFun\ f\ cs)\ (bef\ @\ t\ \#\ aft) = D\langle t\rangle$ **and** $D \neq \Box$ **by** *blast*
from *this*[*folded s*(*1*)] **show** *?thesis* **by** *auto*
qed

fun *poss-mctxt* :: $('f, 'v)\ mctxt \Rightarrow pos\ set$
where
$poss\text{-}mctxt\ (MVar\ x) = \{[]\}$ |
$poss\text{-}mctxt\ MHole = \{\}$ |
$poss\text{-}mctxt\ (MFun\ f\ cs) = \{[]\} \cup \bigcup (set\ (map\ (\lambda\ i.\ (\lambda\ p.\ i\ \#\ p)\ `\ poss\text{-}mctxt\ (cs\ !\ i))\ [0\ ..<\ length\ cs]))$

lemma *poss-mctxt-simp* [*simp*]:
$poss\text{-}mctxt\ (MFun\ f\ cs) = \{[]\} \cup \{i\ \#\ p \mid i\ p.\ i < length\ cs \wedge p \in poss\text{-}mctxt\ (cs\ !\ i)\}$
by *auto*
declare *poss-mctxt.simps*(*3*)[*simp del*]

lemma *poss-mctxt-map-vars-mctxt* [*simp*]:

poss-mctxt (*map-vars-mctxt f C*) = *poss-mctxt C*
by (*induct C*) *auto*

fun *hole-poss* :: ('*f*, '*v*) *mctxt* ⇒ *pos set*
where
hole-poss (*MVar x*) = {} |
hole-poss MHole = {[]} |
hole-poss (*MFun f cs*) = ⋃(*set* (*map* (λ *i*. (λ *p*. *i* # *p*) ‘ *hole-poss* (*cs* ! *i*)) [*0* ..< *length cs*]))

lemma *hole-poss-simp* [*simp*]:
hole-poss (*MFun f cs*) = {*i* # *p* | *i p*. *i* < *length cs* ∧ *p* ∈ *hole-poss* (*cs* ! *i*)}
by *auto*
declare *hole-poss.simps*(*3*)[*simp del*]

lemma *hole-poss-empty-iff-num-holes-0*: *hole-poss C* = {} ⟷ *num-holes C* = *0*
by (*induct C*; *fastforce simp*: *set-conv-nth*)

lemma *mctxt-of-term-fill-holes* [*simp*]:
fill-holes (*mctxt-of-term t*) [] = *t*
proof (*induct t*)
case (*Fun f ts*)
then have *fill-holes* (*mctxt-of-term* (*Fun f ts*)) [] = *Fun f* (*map* (λ*i*. (*ts*!*i*)) [*0*..<*length ts*])
unfolding *mctxt-of-term.simps partition-holes-fill-holes-conv partition-by-Nil map-map* **by** *auto*
also have ... = *Fun f ts* **using** *map-nth* **by** *auto*
ultimately show *?case* **by** *auto*
qed (*auto*)

lemma *hole-pos-not-in-poss-mctxt*:
assumes *p* ∈ *hole-poss C*
shows *p* ∉ *poss-mctxt C*
using *assms*
by (*induct C arbitrary*: *p*) *auto*

lemma *hole-pos-in-filled-fun-poss*:
assumes *is-Fun t*
shows *hole-pos E* ∈ *fun-poss* ((*E* ·$_c$ *σ*)⟨*t* · *σ*⟩)
using *assms*
by (*induct E*) (*auto simp*: *append-Cons-nth-middle*)

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*: $C \cdot mc\ \sigma \cdot mc\ \delta = C \cdot mc\ \sigma \circ_s \delta$
proof (*induct C*)
case (*MVar x*)
define *t* **where** $t = \sigma\ x$
show *?case* **by** (*simp add*: *t-def*[*symmetric*] *subst-compose-def*, *induct t*, *auto*)
qed *auto*

lemma *subst-apply-mctxt-cong*: $(\bigwedge x.\ x \in vars\text{-}mctxt\ C \Longrightarrow \sigma\ x = \tau\ x) \Longrightarrow C \cdot mc\ \sigma = C \cdot mc\ \tau$
by (*induct C*, *auto*)

lemma *vars-mctxt-subst*: *vars-mctxt* $(C \cdot mc\ \sigma) = \bigcup$ (*vars-term* ‘ σ ‘ *vars-mctxt C*)
by (*induct C*, *auto*)

lemma *subst-apply-mctxt-numholes*:
shows *num-holes* $(c \cdot mc\ \sigma) =$ *num-holes c*
proof (*induct c arbitrary*: σ)
case (*MFun f cs*)
have *num-holes* (*MFun f cs* $\cdot mc\ \sigma$) = *sum-list* [*num-holes* $(c \cdot mc\ \sigma)$. $c \leftarrow cs$]
unfolding *subst-apply-mctxt.simps num-holes.simps map-map comp-def* **by** *auto*
also have ... = *sum-list* [*num-holes c* . $c \leftarrow cs$] **using** *MFun*(*1*)
by (*metis* (*lifting*, *no-types*) *map-cong*)
ultimately show *?case* **by** *auto*
qed (*auto*)

lemma *subst-apply-mctxt-fill-holes*:
assumes *nh*: *num-holes c* = *length ts*
shows (*fill-holes c ts*) $\cdot\ \sigma$ = *fill-holes* $(c \cdot mc\ \sigma)$ [*ti* $\cdot\ \sigma$. *ti* $\leftarrow$ *ts*]
using *nh*
proof (*induct c arbitrary*: *ts*)
case *MHole*
then obtain *t* **where** *ts*: *ts* = [*t*]
unfolding *num-holes.simps* **unfolding** *One-nat-def* **using** *Suc-length-conv length-0-conv* **by** *metis*
show *?case* **unfolding** *ts* **by** *simp*
next
case (*MVar x*)
then have *ts*: *ts* = [] **using** *length-0-conv* **by** *auto*
show *?case* **unfolding** *ts* **by** *auto*
next
case (*MFun f cs*)
note *IH* = *MFun*(*1*)
note *nh* = *MFun*(*2*)[*unfolded num-holes.simps*]
let *?c* = *MFun f cs*
let *?csσ* = *map* $(\lambda c.\ c \cdot mc\ \sigma)$ *cs*

```
  {
    fix i
    assume i: i < length cs
    have nh-map: ⋀ j. j < length cs ⟹ num-holes (cs!j) = num-holes (?csσ ! j)
      using nth-map subst-apply-mctxt-numholes by metis

    have fill-holes (cs ! i) (partition-holes ts cs ! i) · σ =
      fill-holes ((cs ! i) ·mc σ) (partition-holes [ ti · σ . ti ← ts] cs ! i)
      using IH [OF nth-mem [OF i]] and nh and i by auto
    also have ... = fill-holes (?csσ ! i) (partition-holes [ti · σ . ti ← ts] ?csσ ! i)
        unfolding nth-map[OF i] using partition-holes-map-ctxt[OF - nh-map]
length-map by metis
     ultimately have fill-holes (cs ! i) (partition-holes ts cs ! i) · σ = fill-holes
(?csσ ! i) (partition-holes [ti · σ . ti ← ts] ?csσ ! i)
      by auto
  } note ith = this

  have fill-holes ?c ts · σ = Fun f [fill-holes (?csσ ! i) (partition-holes [ ti · σ . ti
← ts] ?csσ ! i) . i ← [0..<length cs]]
     unfolding partition-holes-fill-holes-conv map-map using ith using comp-def
by auto
  also have ... = fill-holes (?c ·mc σ) [ ti · σ . ti ← ts]
    unfolding subst-apply-mctxt.simps partition-holes-fill-holes-conv length-map ..
  ultimately show ?case by auto
qed

lemma subst-apply-mctxt-sound:
  assumes t =f (c,ts)
  shows t · σ =f (c ·mc σ, [ ti · σ . ti ← ts])
proof (rule eqfI, insert subst-apply-mctxt-numholes subst-apply-mctxt-fill-holes[OF
eqfE(2)[OF assms]] eqfE[OF assms] eqfE(2)[OF assms, symmetric], auto) qed

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
  by (induct C) (auto intro: nth-equalityI)

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
using *assms* **by** (*induct ts arbitrary*: *n*) *auto*

lemma *fill-holes-mctxt-MHole* [*simp*]:
length ts = *Suc 0* ⟹ *fill-holes-mctxt MHole ts* = *hd ts*
by (*cases ts*) *simp-all*

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*]]
by (*simp add*: *partition-by-nth take-map*)

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*)))
unfolding *zip-nth-conv* [*of Cs partition-holes ts Cs*, *simplified*]
and *partition-holes-fill-holes-mctxt-conv* **by** *simp*

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*⟩)
by (*induct C arbitrary*: *t*)
(*simp-all del*: *fill-holes-mctxt.simps add*: *partition-holes-fill-holes-mctxt-conv′*)

lemma *fill-holes-mctxt-mctxt-of-ctxt-MHole* [*simp*]:
fill-holes-mctxt (*mctxt-of-ctxt C*) [*MHole*] = *mctxt-of-ctxt C*
by (*induct C*) (*simp-all del*: *fill-holes-mctxt.simps add*: *partition-holes-fill-holes-mctxt-conv′*)

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*)))
unfolding *zip-nth-conv* [*of Cs partition-holes ts Cs*, *simplified*]
and *partition-holes-fill-holes-conv* **by** *simp*

lemma *fill-holes-mctxt-MFun-replicate-length* [*simp*]:
fill-holes-mctxt (*MFun c* (*replicate* (*length Cs*) *MHole*)) *Cs* = *MFun c Cs*
unfolding *partition-holes-fill-holes-mctxt-conv′*
by (*induct Cs*) *simp-all*

lemma *fill-holes-MFun-replicate-length* [*simp*]:
fill-holes (*MFun c* (*replicate* (*length ts*) *MHole*)) *ts* = *Fun c ts*
unfolding *partition-holes-fill-holes-conv′*
by (*induct ts*) *simp-all*

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*)

using *assms*
proof (*induct C arbitrary*: *Ds*)
case *MHole*
then show *?case* **by** (*cases Ds*) *simp-all*
next
case (*MFun f Cs*)
then have *num-holes*: *sum-list* (*map num-holes Cs*) = *length Ds* **by** *simp*
let *?ys* = *partition-holes Ds Cs*
have $\bigwedge i.$ *i* < *length Cs* $\Longrightarrow$ *?f* (*Cs* ! *i*) (*?ys* ! *i*) = *?g* (*Cs* ! *i*) (*?ys* ! *i*)
using *MFun* **by** (*metis nth-mem num-holes.simps*(*3*) *length-partition-holes-nth*)
then have ($\bigcup i \in \{0$ *..<* *length Cs*$\}$. *?f* (*Cs* ! *i*) (*?ys* ! *i*)) =
($\bigcup i \in \{0$ *..<* *length Cs*$\}$. *?g* (*Cs* ! *i*) (*?ys* ! *i*)) **by** *simp*
then show *?case*
using *num-holes*
unfolding *partition-holes-fill-holes-mctxt-conv*
by (*simp add*: *UN-Un-distrib UN-upt-len-conv* [*of* - - $\lambda x. \bigcup$ (*set x*)] *UN-set-partition-by-map*)
qed *simp*

lemma *fill-holes-mctxt-MFun*:
assumes *lCs*: *length Cs* = *length ts*
and *lss*: *length ss* = *length ts*
and *rec*: $\bigwedge$ *i*. *i* < *length ts* $\Longrightarrow$ *num-holes* (*Cs* ! *i*) = *length* (*ss* ! *i*) $\wedge$ *fill-holes-mctxt* (*Cs* ! *i*) (*ss* ! *i*) = *ts* ! *i*
shows *fill-holes-mctxt* (*MFun f Cs*) (*concat ss*) = *MFun f ts*
unfolding *fill-holes-mctxt.simps mctxt.simps*
by (*rule conjI*[*OF refl*], *rule fill-holes-arbitrary*[*OF lCs lss rec*])

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*)
using *assms*
proof (*induct C arbitrary*: *Ds*)
case *MHole*
then show *?case* **by** (*cases Ds*) *simp-all*
next
case (*MFun f Cs*)
then have $*$: *map* (*num-holes* $\circ$ ($\lambda i.$ *fill-holes-mctxt* (*Cs* ! *i*) (*partition-holes Ds Cs* ! *i*))) [*0..<length Cs*] =
map ($\lambda i.$ *sum-list* (*map num-holes* (*partition-holes Ds Cs* ! *i*))) [*0* *..<* *length Cs*]
and *sum-list* (*map num-holes Cs*) = *length Ds*
by *simp-all*
then show *?case*
using *map-upt-len-conv* [*of* $\lambda x.$ *sum-list* (*map num-holes x*) *partition-holes Ds Cs*]
unfolding *partition-holes-fill-holes-mctxt-conv* **by** (*simp add*: $*$)
qed *simp*

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*]]
using *assms*(*1*)[*symmetric*] *assms*(*2*)
proof (*induct c ds arbitrary*: *ss rule*: *fill-holes-induct*)
case (*MFun f Cs ds ss*)
define *qs* **where** *qs* = *map* (λ*i*. *fill-holes-mctxt* (*Cs* ! *i*) (*partition-holes ds Cs* ! *i*)) [*0*..<*length Cs*]
then have *qs*: ⋀*i*. *i* < *length Cs* ⟹ *fill-holes-mctxt* (*Cs* ! *i*) (*partition-holes ds Cs* ! *i*) = *qs* ! *i*
length qs = *length Cs* **by** *auto*
define *zs* **where** *zs* = *map* (λ*i*. *fill-holes* (*ds* ! *i*) (*partition-holes ss ds* ! *i*)) [*0*..<*length ds*]
{
fix *i* **assume** *i*: *i* < *length Cs*
from *MFun*(*1*) **have** ∗: *map length* (*partition-holes ds Cs*) = *map num-holes Cs* **by** *auto*
have ∗∗: *length ss* = *sum-list* (*map sum-list* (*partition-holes* (*map num-holes ds*) *Cs*))
using *MFun*(*1*) *MFun*(*3*)[*symmetric*] *num-holes-fill-holes-mctxt*[*of MFun f Cs ds*]
by (*auto simp*: *comp-def map-map-partition-by*[*symmetric*])
have *partition-by* (*partition-by ss*
(*map* (λ*i*. *num-holes* (*fill-holes-mctxt* (*Cs* ! *i*) (*partition-holes ds Cs* ! *i*))) [*0*..<*length Cs*]) ! *i*)
(*partition-holes* (*map num-holes ds*) *Cs* ! *i*) = *partition-holes* (*partition-holes ss ds*) *Cs* ! *i*
using *i MFun*(*1*) *MFun*(*3*) *partition-by-partition-by*[*OF* ∗∗]
by (*auto simp*: *comp-def num-holes-fill-holes-mctxt*
intro!: *arg-cong*[*of* - - λ*x*. *partition-by* (*partition-by ss x* ! -) -] *nth-equalityI*)
then have *map* (λ*j*. *fill-holes* (*partition-holes ds Cs* ! *i* ! *j*)
(*partition-holes* (*partition-holes ss qs* ! *i*)
(*partition-holes ds Cs* ! *i*) ! *j*)) [*0*..<*num-holes* (*Cs* ! *i*)] =
partition-holes zs Cs ! *i*
using *MFun*(*1*,*3*)
by (*auto simp*: *zs-def qs-def i comp-def partition-by-nth-nth intro*: *nth-equalityI*)
}
then show *?case* **using** *MFun* **by** (*simp add*: *qs-def* [*symmetric*] *qs zs-def* [*symmetric*])
qed *auto*

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 (*rule eqfI*)

```
  note l-nh-i = eqfE(2)[OF insts]

  from partition-holes-concat-id[OF - l-nh-i] len-ds len-sss
  have concat-sss: partition-holes (concat sss) ds = sss by auto

  then show nh: num-holes (fill-holes-mctxt c ds) = length (concat sss)
    unfolding num-holes-fill-holes-mctxt [OF len-ds [symmetric]] length-concat
    by (metis l-nh-i len-ds len-sss nth-map-conv)

   have ts: ts = [fill-holes (ds ! i) (partition-holes (concat sss) ds ! i) . i ←
[0..<num-holes c]] (is - = ?fhs)
  proof (rule nth-equalityI)
    show l-fhs: length ts = length ?fhs unfolding length-map
      by (metis diff-zero len-ts length-upt)
    fix i
    assume i: i < length ts
    then have i': i < length [0..<num-holes c]
      by (metis diff-zero len-ts length-upt)
    show ts!i = ?fhs ! i
      unfolding nth-map[OF i']
      using eqfE(1)[OF insts[unfolded len-ds, OF i[unfolded len-ts]]]
      by (metis concat-sss i' len-ds len-sss map-nth nth-map)
  qed
  note ts = this

  show fill-holes c ts = fill-holes (fill-holes-mctxt c ds) (concat sss)
    unfolding fill-holes-mctxt-fill-holes[OF len-ds nh] ts ..
qed

lemma poss-mctxt-fill-holes-mctxt:
  assumes p ∈ poss-mctxt C
  shows p ∈ poss-mctxt (fill-holes-mctxt C Cs)
  using assms
proof (induct p arbitrary: C Cs)
  case (Cons a p C Cs)
  thus ?case by (cases C, auto)
next
  case (Nil C Cs)
  thus ?case by (cases C, auto)
qed

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 −
  let ?Ds = [(if i = j then D else MHole). j ← [0 ..< num-holes C]]
  have num-holes C = length ?Ds by simp
  then show ?thesis using assms by (auto split: if-splits)
qed

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 −
  let ?Cs = [ if i = j then Ci else MHole . j ← [0..<num-holes C]]
  let ?ts = bef @ si # aft
  let ?sss = [ [b]. b ← bef ] @ (ts # [ [a]. a ← aft])

  have
    l-Cs : length ?Cs = num-holes C and
    l-ts : length ?ts = num-holes C and
    l-sss : length ?sss = num-holes C
    unfolding length-append length-map list.size(4) using eqfE(2)[OF s] by auto

  have i-le-nh: i < num-holes C unfolding i eqfE(2)[OF s] length-append by
(auto iff: trans-less-add1)
  have concat-sss: concat ?sss = bef @ ts @ aft by auto

  {
    fix j
    assume j: j < i
    then have j': j < length [0..<num-holes C] using i-le-nh length-upt by auto
    have ?sss!j = [bef!j] by (metis append-Cons-nth-left i j length-map nth-map)
    moreover have ?ts!j = bef!j by (metis append-Cons-nth-left i j)
    moreover from nth-map[OF j'] j' j have ?Cs!j = MHole by force
    ultimately have ?ts!j =_f (?Cs!j,?sss!j) using eqfI by auto
  } note j-le-i = this

  from i-le-nh have ?Cs!i = Ci by auto
 moreover from i-le-nh have ?sss!i = ts by (metis i length-map nth-append-length)
  moreover have ?ts!i = si using nth-append-length i by auto
  ultimately have j-eq-i: ?ts!i =_f (?Cs!i,?sss!i) using si by auto

  {
    fix j
    assume j: j > i and j': j < num-holes C
    then have j'': j < length [0..<num-holes C] by auto

    have j''': (j − i) − 1 < length aft using
        j'[unfolded eqfE(2)[OF s] length-append[of bef] list.size(4)] j
      unfolding i by auto
```

from *nth-append*[*of* [[*b*]. *b* ← *bef*] - *j*, *unfolded length-map*[*of* - *bef*] *i*[*symmetric*]]

have *?sss*!*j* = (*ts* # [[*a*]. *a* ← *aft*]) ! (*j* − *i*) **using** *j* **by** *auto*
moreover have ... = [[*a*]. *a* ← *aft*] ! ((*j* − *i*) − *1*) **using** *nth-Cons-pos j* **by** *simp*
moreover have ... = [*aft* ! ((*j* − *i*) − *1*)] **using** *j'''' length-map nth-map* **by** *auto*
ultimately have *sssj*: *?sss*!*j* = [*aft* ! ((*j* − *i*) − *1*)] **by** *auto*

have *Csj*: *?Cs*!*j* = *MHole* **using** *nth-map*[*OF j''*] *j'' j* **by** *force*

have *?ts*!*j* = (*si* # *aft*) ! (*j* − *i*) **unfolding** *nth-append*[*of bef*] *i*[*symmetric*] **using** *j* **by** *simp*
moreover have ... = *aft* ! ((*j* − *i*) − *1*) **by** (*metis j neq0-conv nth-Cons' zero-less-diff*)
ultimately have *?ts*!*j* = *aft* ! ((*j* − *i*) − *1*) **by** *auto*
then have *?ts*!*j* =$_f$ (*?Cs*!*j*,*?sss*!*j*) **using** *sssj Csj* **by** *auto*
} **note** *j-gr-i* = *this*

from *j-le-i j-eq-i j-gr-i* **have** ⋀ *j*. *j* < *length ?Cs* ⟹ *?ts*!*j* =$_f$ (*?Cs*!*j*,*?sss*!*j*)
using *l-Cs linorder-neqE-nat* **by** *metis*

from *fill-holes-mctxt-sound*[*OF l-Cs l-sss l-ts this*, *unfolded concat-sss*, *folded compose-mctxt.simps*]
show *?thesis* **unfolding** *eqfE*(*1*)[*OF s*] **by** *simp*
qed

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*
using *assms* **by** (*induct rule*: *subseq-induct2*, *auto*)

lemma *mctxt-fill-partially-fills-length* [*simp*]:
assumes *subseq ss ts*
shows *length* (*mctxt-fill-partially-fills ss ts*) = *length ts*
using *assms* **by** (*induct rule*: *subseq-induct2*, *auto*)

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 (*induct ss ts rule*: *subseq-induct2*, *goal-cases*)
case (*3 s ss ts*)
have *ls-one*: ⋀ *as*. *sum-list* (*1* # *as*) = *sum-list as* + *Suc 0*
by (*metis One-nat-def Suc-eq-plus1 Suc-eq-plus1-left sum-list-simps*(*2*))
from *3* **show** *?case*
unfolding *mctxt-fill-partially-mctxts.simps list.size*
by (*metis* (*full-types*) *One-nat-def Suc-eq-plus1 ls-one list.map*(*2*) *num-holes.simps*(*2*))
next
case (*4 s ss t ts*)
have *ls-zero*: ⋀ *as*. *sum-list* (*0* # *as*) = *sum-list as* **by** (*metis sum-list-simps*(*2*) *monoid-add-class.add.left-neutral*)
have *else*:
mctxt-fill-partially-mctxts (*s* # *ss*) (*t* # *ts*) = *mctxt-of-term t* # *mctxt-fill-partially-mctxts* (*s* # *ss*) *ts*
using *4*(*1*) **by** *auto*
show *?case*
unfolding *else list.map*(*2*) *num-holes-mctxt-of-term ls-zero 4*(*5*) **..**
qed (*auto iff*: *assms*)

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*, *mctxt-fill-partially-fills ss ts* ! *i*)
proof (*rule eqfI*, *goal-cases*)
let *?zipped*= *zip* (*mctxt-fill-partially-mctxts ss ts*) (*mctxt-fill-partially-fills ss ts*)

have *l*: *length ?zipped* = *length ts*
unfolding *length-zip mctxt-fill-partially-mctxts-length*[*OF sl*] *mctxt-fill-partially-fills-length*[*OF sl*] **by** *auto*

have *fh*: *ts* = *map* (λ (*ci*,*tsi*) . *fill-holes ci tsi*) *?zipped*
proof (*induct ss ts rule*: *subseq-induct2*, *goal-cases*)
case (*2 ts*) **then show** *?case*
by (*induct ts*, *insert mctxt-of-term-fill-holes*, *auto*)
qed (*insert sl*, *auto*)

have *nh*: *list-all* (λ (*ci*,*tsi*) . *num-holes ci* = *length tsi*) *?zipped*
proof (*induct ss ts rule*: *subseq-induct2*, *goal-cases*)
case (*2 ts*) **then show** *?case*
by (*induct ts*, *insert num-holes-mctxt-of-term*, *auto*)

qed (*insert sl*, *auto*)

{
fix *i*
assume *i* < *length ts*
then have
i1: *i* < *length* (*mctxt-fill-partially-mctxts ss ts*) **and**
i2: *i* < *length* (*mctxt-fill-partially-fills ss ts*)
unfolding *mctxt-fill-partially-mctxts-length*[*OF sl*] *mctxt-fill-partially-fills-length*[*OF sl*] **by** *auto*
} note *i* = *this*

case *1*
then show *?case* **using** *fh nth-zip*[*OF i*(*1*) *i*(*2*)]
by (*metis* (*lifting*, *no-types*) *1 list-update-id list-update-same-conv map-update split-conv*)
case *2* **then show** *?case* **using** *nh*[*unfolded list-all-length*] *nth-zip*[*OF i*(*1*) *i*(*2*)]
by (*auto simp*: *i*(*1*) *i*(*2*))
qed

lemma *mctxt-fill-partially*:
assumes *ss*: *subseq ss ts*
and *t*: *t* =$_f$ (*c*,*ts*)
shows ∃ *d*. *t* =$_f$ (*d*,*ss*)
proof −
let *?ds* = *mctxt-fill-partially-mctxts ss ts*
let *?sss* = *mctxt-fill-partially-fills ss ts*

have *fill-holes c ts* =$_f$ (*fill-holes-mctxt c ?ds*, *concat ?sss*)
using
fill-holes-mctxt-sound eqfE(*2*)[*OF t*,*symmetric*] *mctxt-fill-partially-sound*[*OF ss*]
mctxt-fill-partially-mctxts-length[*OF ss*] *mctxt-fill-partially-fills-length*[*OF ss*]
by *metis*
also have *concat ?sss* = *ss* **by** (*induct ss ts rule*: *subseq-induct2*, *insert ss*, *auto*)
ultimately show *?thesis* **by** (*metis eqfE*(*1*) *t*)
qed

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*)
using *assms*
by (*induct C ts rule*: *fill-holes-induct*) (*auto*)

lemma *fill-holes-mctxt-of-ctxt* [*simp*]:
fill-holes (*mctxt-of-ctxt C*) [*t*] = *C*⟨*t*⟩
proof −
have *C*⟨*t*⟩ =$_f$ (*mctxt-of-ctxt C*, [*t*]) **by** (*metis mctxt-of-ctxt*)
from *eqfE* [*OF this*] **show** *?thesis* **by** *simp*

qed

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 −
have *fill-holes* (*cap-till P s*) *?ss* =$_f$
(*fill-holes-mctxt* (*cap-till P s*) (*map mctxt-of-term* (*take i* (*uncap-till P s*)) @
C # *map mctxt-of-term* (*drop* (*Suc i*) (*uncap-till P s*))),
concat (*map* (λ-. []) (*take i* (*uncap-till P s*)) @ *ts* #
map (λ-. []) (*drop* (*Suc i*) (*uncap-till P s*))))
(**is** - =$_f$ (*fill-holes-mctxt* - *?ts*, *concat ?us*))
proof (*rule fill-holes-mctxt-sound*)
show *length ?ss* = *num-holes* (*cap-till P s*)
using *assms* **by** *simp*
next
show *length ?ts* = *num-holes* (*cap-till P s*)
using *assms* **by** *simp*
next
show *length ?us* = *num-holes* (*cap-till P s*)
using *assms* **by** *simp*
next
fix *j*
assume *j* < *length ?ts*
with *assms* **have** *j*: *j* < *length* (*uncap-till P s*) **by** *simp*
show *?ss* ! *j* =$_f$ (*?ts* ! *j*, *?us* ! *j*)
using *assms* **and** *j* **by** (*cases j* = *i*) (*auto simp*: *nth-append*)
qed
note ∗ = *eqfE*(*1*) [*OF this*]
show *?thesis* **by** (*simp add*: *compose-cap-till-def* ∗)
qed

lemma *in-uncap-till-funas*:
assumes *root*: *root u* = *Some fn fn* ∈ *F*
and *t* = *C*⟨*u*⟩
shows ∃ *i* < *length* (*uncap-till-funas F t*). ∃ *D*. *uncap-till-funas F t* ! *i* = *D*⟨*u*⟩ ∧
mctxt-of-ctxt C = *compose-cap-till-funas F t i* (*mctxt-of-ctxt D*)
using ‹*t* = *C*⟨*u*⟩›
proof (*induct t arbitrary*: *C*)

```
  case (Var x)
  then show ?case using root by (cases C) (auto simp: wf-trs-def)
next
  case (Fun f ts)
  define t where [simp]: t = Fun f ts
  show ?case
  proof (cases (f, length ts) ∈ F)
    case True
    then show ?thesis using Fun.prems by (auto simp: compose-cap-till-def)
  next
    case False
    show ?thesis
    proof (cases C)
      case Hole
      then show ?thesis using Fun.prems and False and root by auto
    next
      case (More - ss₁ D -)
      moreover define j where j = length ss₁
      ultimately have j: j < length ts ts ! j = D⟨u⟩
        and C: C = More f (take j ts) D (drop (Suc j) ts)
        using Fun.prems by (auto)
      then have D⟨u⟩ ∈ set ts by (auto simp: in-set-conv-nth)
      then obtain i and E
        where i: i < length (uncap-till-funas F (D⟨u⟩)) uncap-till-funas F (D⟨u⟩) !
i = E⟨u⟩
          and D: mctxt-of-ctxt D = compose-cap-till-funas F (D⟨u⟩) i (mctxt-of-ctxt
E)
        using Fun by blast
      obtain k where k: take k (uncap-till-funas F t) =
          concat (map (uncap-till-funas F) (take j ts)) @ take i (uncap-till-funas F
(D⟨u⟩))
        k < length (uncap-till-funas F t) uncap-till-funas F t ! k = E⟨u⟩
         drop (Suc k) ((uncap-till-funas F) t) = drop (Suc i) ((uncap-till-funas F)
(D⟨u⟩)) @ concat (map (uncap-till-funas F) (drop (Suc j) ts))
       using False and i and j and take-nth-drop-concat [of j map (uncap-till-funas
F) ts (uncap-till-funas F) (D⟨u⟩) i E⟨u⟩]
        by (auto simp: take-map drop-map)
      moreover have mctxt-of-ctxt C = compose-cap-till-funas F t k (mctxt-of-ctxt
E)
      proof −
        have ∗: compose-cap-till-funas F t k (mctxt-of-ctxt E) =
          fill-holes-mctxt (MFun f (map (cap-till-funas F) ts)) (concat (
            map (map mctxt-of-term ∘ uncap-till-funas F) (take j ts) @
            (map mctxt-of-term (take i (uncap-till-funas F D⟨u⟩)) @
            mctxt-of-ctxt E #
            map mctxt-of-term (drop (Suc i) (uncap-till-funas F D⟨u⟩))) #
            map (map mctxt-of-term ∘ uncap-till-funas F) (drop (Suc j) ts)))
          (is - = fill-holes-mctxt - (concat ?ss))
          using False and k
```

```
      by (simp del: fill-holes-mctxt.simps add: compose-cap-till-def map-concat)
    also have ... = MFun f (map mctxt-of-term (take j ts) @ mctxt-of-ctxt D
#
      map mctxt-of-term (drop (Suc j) ts)) (is - = MFun f ?ts)
    proof (rule fill-holes-mctxt-MFun)
      show length (map (cap-till-funas F) ts) = length ?ts using j by simp
    next
      show length ?ss = length ?ts by simp
    next
      fix n
      assume n < length ?ts
      then have n: n < length ts using j by simp
      show num-holes (map (cap-till-funas F) ts ! n) = length (?ss ! n) ∧
        fill-holes-mctxt (map (cap-till-funas F) ts ! n) (?ss ! n) = ?ts ! n
      proof (cases n = j)
        case False
        then have ∗: ?ss ! n = map mctxt-of-term (uncap-till-funas F (ts ! n))
          ?ts ! n = mctxt-of-term (ts ! n)
          using n and j by (auto simp: nth-append min-def)
        have num-holes (map (cap-till-funas F) ts ! n) = length (?ss ! n)
          using n by (simp add: ∗ )
        moreover have fill-holes-mctxt (map (cap-till-funas F) ts ! n) (?ss ! n)
= ?ts ! n
          using n by (auto simp: ∗ )
        ultimately show ?thesis by blast
      next
        case True
        then have ∗: ?ss ! n =
          map mctxt-of-term (take i (uncap-till-funas F D⟨u⟩)) @ mctxt-of-ctxt E
#
            map mctxt-of-term (drop (Suc i) (uncap-till-funas F D⟨u⟩))
          ?ts ! n = mctxt-of-ctxt D
          using n and j by (auto simp: nth-append)
        have fill-holes-mctxt (map (cap-till-funas F) ts ! n) (?ss ! n) = ?ts ! n
          unfolding ∗ by (simp add: D compose-cap-till-def True j)
        moreover have num-holes (map (cap-till-funas F) ts ! n) = length (?ss
! n)
          unfolding ∗ using i by (simp add: j True)
        ultimately show ?thesis by blast
      qed
    qed
    finally show ?thesis by (simp add: C)
  qed
  ultimately show ?thesis unfolding t-def by blast
 qed
qed

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*)
using *assms*
proof (*induct C arbitrary*: *ts*)
case *MHole*
then show *?case* **by** (*cases ts*) *simp-all*
next
case (*MFun f Cs*)
let *?ts* = *partition-holes ts Cs*
let *?us* = *partition-holes* (*map* (*uncap-till-funas F*) *ts*) *Cs*
have ∗: *fill-holes* (*MFun f Cs*) *ts* =
Fun f (*map* (λ*i*. *fill-holes* (*Cs* ! *i*) (*?ts* ! *i*)) [*0* ..< *length Cs*])
unfolding *partition-holes-fill-holes-conv* **..**
have ∀ *i* < *length Cs*. *uncap-till-funas F* (*fill-holes* (*Cs* ! *i*) (*?ts* ! *i*)) = *concat* (*?us* ! *i*)
proof (*intro allI impI*)
fix *i*
assume *i* < *length Cs*
then have *Cs* ! *i* ∈ *set Cs* **by** *simp*
from *MFun.hyps* [*OF this*, *of ?ts* ! *i*] **and** *MFun.prems* **and** ‹*i* < *length Cs*›
show *uncap-till-funas F* (*fill-holes* (*Cs* ! *i*) (*?ts* ! *i*)) = *concat* (*?us* ! *i*)
by (*auto iff*: *UN-subset-iff*)
qed
then have ∗∗: *map* (*uncap-till-funas F* ∘ (λ*i*. *fill-holes* (*Cs* ! *i*) (*?ts* ! *i*))) [*0*..<*length Cs*] =
map (*concat* ∘ (λ*i*. (*?us* ! *i*))) [*0* ..<*length Cs*] **by** *simp*
have ∗∗∗: *sum-list* (*map num-holes Cs*) = *length* (*map* (*uncap-till-funas F*) *ts*)
using *MFun.prems* **by** *simp*
show *?case*
using *MFun.prems*
apply (*simp add*: ∗ ∗∗ *del*: *fill-holes.simps*)
by (*auto simp*: *o-def map-upt-len-same-len-conv* [*OF length-partition-holes*])
qed *simp*

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-till-funas F*) *ts*)
by (*rule uncap-till-funas-fill-holes-cancel* [*OF assms ground-cap-till-funas*, *of F*]) *auto*

lemma *Ball-atLeast0LessThan-partition-holes-conv* [*simp*]:
(∀ *i* ∈ {*0* ..< *length Cs*}. ∀ *x* ∈ *set* (*partition-holes xs Cs* ! *i*). *P x*) =
(∀ *x* ∈ ⋃(*set* (*map set* (*partition-holes xs Cs*))). *P x*)
using *Ball-atLeast0LessThan-partition-by-conv* [*of map num-holes Cs xs*] **by** *simp*

lemma *ground-fill-holes-mctxt* [*simp*]:
num-holes C = *length Ds* ⟹

ground-mctxt (*fill-holes-mctxt C Ds*) ⟷ *ground-mctxt C* ∧ (∀ *D* ∈ *set Ds*. *ground-mctxt D*)
proof (*induct C arbitrary*: *Ds*)
case *MHole*
then show *?case* **by** (*cases Ds*) *simp-all*
next
case (*MFun f Cs*)
then have ∗: (∀ *i*∈{*0*..<*length Cs*}.
ground-mctxt (*fill-holes-mctxt* (*Cs* ! *i*) (*partition-holes Ds Cs* ! *i*))) =
(∀ *i*∈{*0*..<*length Cs*}.
ground-mctxt (*Cs* ! *i*) ∧ (∀ *a*∈*set* (*partition-holes Ds Cs* ! *i*). *ground-mctxt a*))
and ∗∗: *sum-list* (*map num-holes Cs*) = *length Ds*
by *simp-all*
show *?case*
unfolding *partition-holes-fill-holes-mctxt-conv*
by (*simp add*: ∗ *ball-conj-distrib Ball-set-partition-by* [*OF* ∗∗])
qed *simp*

lemma *concat-map-uncap-till-funas-map-subst-apply-uncap-till-funas* [*simp*]:
concat (*map* (*uncap-till-funas F*) (*map* (λ*s*. *s* · *σ*) (*uncap-till-funas F t*))) = *uncap-till-funas F* (*t* · *σ*)
proof (*induct t*)
case (*Fun f ts*)
then have ∗: *map* (*uncap-till-funas F* ∘ (λ*t*. *t* · *σ*)) *ts* =
map concat (*map* (*map* (*uncap-till-funas F*) ∘ *map* (λ*s*. *s* · *σ*) ∘ *uncap-till-funas F*) *ts*) **by** *simp*
show *?case*
by (*simp add*: ∗ *map-concat concat-map-concat* [*symmetric*])
qed *simp*

lemma *concat-uncap-till-subst-conv*:
concat (*map* (λ*i*. *uncap-till-funas F* ((*uncap-till-funas F t* ! *i*) · *σ*)) [*0* ..< *length* (*uncap-till-funas F t*)]) =
uncap-till-funas F (*t* · *σ*)
proof −
have *concat* (*map* (*uncap-till-funas F*) (*map* (λ*i*.
(*uncap-till-funas F t* ! *i*) · *σ*) [*0* ..< *length* (*uncap-till-funas F t*)])) = *uncap-till-funas F* (*t* · *σ*)
unfolding *map-upt-len-conv* [*of* λ*s*. *s* · *σ uncap-till-funas F t*]
unfolding *concat-map-uncap-till-funas-map-subst-apply-uncap-till-funas* **..**
then show *?thesis* **by** (*simp add*: *o-def*)
qed

lemma *the-root-uncap-till-funas*:
is-Fun t ⟹ *the* (*root t*) ∈ *F* ⟹ *uncap-till-funas F t* = [*t*]
by (*cases t*) *simp-all*

lemma *funas-cap-till-subset*:
funas-mctxt (*cap-till P t*) ⊆ *funas-term t*

by (*induct t*) *auto*

lemma *funas-uncap-till-subset*:
s ∈ *set* (*uncap-till P t*) ⟹ *funas-term s* ⊆ *funas-term t*
proof (*induct t arbitrary*: *s*)
case (*Fun f ts*)
then show *?case* **by** (*cases P* (*Fun f ts*)) *auto*
qed *simp*

lemma *ground-mctxt-subst-apply-context* [*simp*]:
ground-mctxt C ⟹ *C* ·*mc* σ = *C*
by (*induct C*) (*simp-all add*: *map-idI*)

lemma *vars-term-fill-holes* [*simp*]:
num-holes C = *length ts* ⟹ *ground-mctxt C* ⟹
vars-term (*fill-holes C ts*) = ⋃(*vars-term* ‘ *set ts*)
proof (*induct C arbitrary*: *ts*)
case *MHole*
then show *?case* **by** (*cases ts*) *simp-all*
next
case (*MFun f Cs*)
then have ∗: *length* (*partition-holes ts Cs*) = *length Cs* **by** *simp*
let *?f* = λ*x*. ⋃ *y* ∈ *set x*. *vars-term y*
show *?case*
using *MFun*
unfolding *partition-holes-fill-holes-conv*
by (*simp add*: *UN-upt-len-conv* [*OF* ∗, *of ?f*] *UN-set-partition-by*)
qed *simp*

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 ..

end

lemma *inf-mctxt-idem* [*simp*]:
fixes $C :: ('f, 'v)\ mctxt$
shows $C \sqcap C = C$
by (*induct C*) (*auto simp*: *zip-same-conv-map intro*: *map-idI*)

lemma *inf-mctxt-MHole2* [*simp*]:
$C \sqcap MHole = MHole$
by (*induct C*) *simp-all*

lemma *inf-mctxt-comm* [*ac-simps*]:
$(C :: ('f, 'v)\ mctxt) \sqcap D = D \sqcap C$
by (*induct C D rule*: *inf-mctxt.induct*) (*fastforce simp*: *in-set-conv-nth intro*!: *nth-equalityI*)+

lemma *inf-mctxt-assoc* [*ac-simps*]:
fixes $C :: ('f, 'v)\ mctxt$
shows $C \sqcap D \sqcap E = C \sqcap (D \sqcap E)$
proof (*induction C D arbitrary*: *E rule*: *inf-mctxt.induct*)
case (*1 D E*)
then show *?case* **by** (*cases E*, *auto*)
next
case (*2-1 v E*)
then show *?case* **by** (*cases E*, *auto*)
next
case (*2-2 v va E*)
then show *?case* **by** (*cases E*, *auto*)
next
case (*3 x y E*)
then show *?case* **by** (*cases E*, *auto*)
next
case (*4 f Cs g Ds E*)
then show *?case*
by (*cases E*; *fastforce simp*: *in-set-conv-nth intro*!: *nth-equalityI*)
next
case (*5-1 v va vb E*)
then show *?case* **by** (*cases E*, *auto*)
next
case (*5-2 v va vb E*)
then show *?case* **by** (*cases E*, *auto*)
qed

instantiation *mctxt* :: (*type*, *type*) *order*
begin

definition $(C :: ('a, 'b)\ mctxt) \leq D \longleftrightarrow C \sqcap D = C$
definition $(C :: ('a, 'b)\ mctxt) < D \longleftrightarrow C \leq D \wedge \neg\ D \leq C$

instance
by (*standard*, *simp-all add*: *less-eq-mctxt-def less-mctxt-def ac-simps*, *metis inf-mctxt-assoc*)

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* ⟹ (⋀*i*. *i* < *length cs* ⟹ *less-eq-mctxt' (cs* ! *i) (ds* ! *i))*
⟹ *less-eq-mctxt' (MFun f cs) (MFun f ds)*

lemma *less-eq-mctxt-prime*: *C* ≤ *D* ⟷ *less-eq-mctxt' C D*
proof
assume *less-eq-mctxt' C D* **then show** *C* ≤ *D*
by (*induct C D rule*: *less-eq-mctxt'.induct*) (*auto simp*: *less-eq-mctxt-def intro*: *nth-equalityI*)
next
assume *C* ≤ *D* **then show** *less-eq-mctxt' C D* **unfolding** *less-eq-mctxt-def*
by (*induct C D rule*: *inf-mctxt.induct*)
(*auto split*: *if-splits simp*: *set-zip intro*!: *less-eq-mctxt'.intros nth-equalityI elim*!: *nth-equalityE*, *metis*)
qed

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* ⟹ *C* ≤ *MHole*
C = *MHole* ∨ *C* = *MVar v* ⟹ *C* ≤ *MVar v*
C = *MHole* ∨ *C* = *MFun f cs* ∧ *length cs* = *length ds* ∧ (∀ *i*. *i* < *length cs* ⟶ *cs* ! *i* ≤ *ds* ! *i*) ⟹ *C* ≤ *MFun f ds*
unfolding *less-eq-mctxt-prime* **by** (*cases C*) (*auto intro*: *less-eq-mctxt'.intros*)

lemma *less-eq-mctxt-MHoleE2*:
assumes *C* ≤ *MHole*
obtains (*MHole*) *C* = *MHole*
using *assms* **unfolding** *less-eq-mctxt-prime* **by** (*cases C*, *auto*)

lemma *less-eq-mctxt-MVarE2*:
assumes *C* ≤ *MVar v*
obtains (*MHole*) *C* = *MHole* | (*MVar*) *C* = *MVar v*
using *assms* **unfolding** *less-eq-mctxt-prime* **by** (*cases C*) *auto*

lemma *less-eq-mctxt-MFunE2*:
assumes *C* ≤ *MFun f ds*
obtains (*MHole*) *C* = *MHole*
| (*MFun*) *cs* **where** *C* = *MFun f cs length cs* = *length ds* ⋀*i*. *i* < *length cs* ⟹ *cs* ! *i* ≤ *ds* ! *i*
using *assms* **unfolding** *less-eq-mctxt-prime* **by** (*cases C*) *auto*

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*
by (*cases D*) (*auto intro*: *less-eq-mctxtI2*)

lemma *less-eq-mctxt-MVarE1*:
assumes *MVar v* $\leq$ *D*
obtains (*MVar*) *D = MVar v*
using *assms* **by** (*cases D*) (*auto elim*: *less-eq-mctxtE2*)

lemma *less-eq-mctxt-MFunE1*:
assumes *MFun f cs* $\leq$ *D*
obtains (*MFun*) *ds* **where** *D = MFun f ds length cs = length ds* $\bigwedge i.$ *i < length cs* $\Longrightarrow$ *cs ! i* $\leq$ *ds ! i*
using *assms* **by** (*cases D*) (*auto elim*: *less-eq-mctxtE2*)

lemmas *less-eq-mctxtE1 = less-eq-mctxt-MVarE1 less-eq-mctxt-MFunE1*

instance *mctxt* :: (*type*, *type*) *semilattice-inf*
apply (*intro-classes*)
by (*auto simp*: *less-eq-mctxt-def inf-mctxt-assoc* [*symmetric*])
(*metis inf-mctxt-comm inf-mctxt-assoc inf-mctxt-idem*)+

fun *inf-mctxt-args* :: ('*f*, '*v*) *mctxt* $\Rightarrow$ ('*f*, '*v*) *mctxt* $\Rightarrow$ ('*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* $\wedge$ *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*]
by (*cases C*) *simp-all*

lemma *fill-holes-mctxt-replicate-MHole* [*simp*]:
fill-holes-mctxt C (*replicate* (*num-holes C*) *MHole*) = *C*
proof (*induct C*)
case (*MFun f Cs*)
{ fix *i* **assume** *i < length Cs*
then have *partition-holes* (*replicate* (*sum-list* (*map num-holes Cs*)) *MHole*) *Cs*

```
! i =
      replicate (num-holes (Cs ! i)) MHole
       using partition-by-nth-nth[of map num-holes Cs replicate (sum-list (map
num-holes Cs)) MHole]
     by (auto intro!: nth-equalityI)
  } note ∗ = this
  show ?case using MFun[OF nth-mem] by (auto simp: ∗ intro!: nth-equalityI)
qed auto

lemma num-holes-inf-mctxt:
  num-holes (C ⊓ D) = length (inf-mctxt-args C D)
  by (induct C D rule: inf-mctxt.induct)
    (auto simp: in-set-zip length-concat intro!: arg-cong [of - - sum-list])

lemma length-inf-mctxt-args:
  length (inf-mctxt-args D C) = length (inf-mctxt-args C D)
  by (metis inf.commute num-holes-inf-mctxt)

lemma inf-mctxt-args-same [simp]:
  inf-mctxt-args C C = replicate (num-holes C) MHole
proof (induct C)
  case (MFun f Cs)
  have ∗: ⋀C. num-holes C = length (inf-mctxt-args C C)
    using num-holes-inf-mctxt [of C C for C] by auto
  let ?xs = map (case-prod inf-mctxt-args) (zip Cs Cs)
  have ∀ i < length Cs.
    inf-mctxt-args (Cs ! i) (Cs ! i) = replicate (num-holes (Cs ! i)) MHole using
MFun by auto
  then have ∀ i < length ?xs. ∀ j < length (?xs ! i). ?xs ! i ! j = MHole by auto
 then have ∀ i < length (concat ?xs). concat ?xs ! i = MHole by (metis nth-concat-two-lists)

  then show ?case by (auto simp: ∗ intro!: nth-equalityI)
qed simp-all

lemma inf-mctxt-inf-mctxt-args:
  fill-holes-mctxt (C ⊓ D) (inf-mctxt-args C D) = C
proof (induct C D rule: inf-mctxt.induct)
  case (4 f Cs g Ds)
  then show ?case
  proof (cases f = g ∧ length Cs = length Ds)
    case True
    with 4 have ∀ i < length Cs.
      fill-holes-mctxt (Cs ! i ⊓ Ds ! i) (inf-mctxt-args (Cs ! i) (Ds ! i)) = Cs ! i
      by (force simp: set-zip)
   moreover have partition-holes (concat (map (case-prod inf-mctxt-args) (zip Cs
Ds)))
      (map (case-prod (⊓)) (zip Cs Ds)) = map (case-prod inf-mctxt-args) (zip Cs
Ds)
      by (rule partition-by-concat-id) (simp-all add: num-holes-inf-mctxt)
```

ultimately show *?thesis*
using *fill-holes-mctxt.simps* [*simp del*]
by (*auto simp*: *partition-holes-fill-holes-mctxt-conv intro*!: *nth-equalityI*)
qed *auto*
qed *auto*

lemma *inf-mctxt-inf-mctxt-args2*:
fill-holes-mctxt (*C* ⊓ *D*) (*inf-mctxt-args D C*) = *D*
unfolding *inf-mctxt-comm* [*of C D*] **by** (*rule inf-mctxt-inf-mctxt-args*)

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 ..

end

lemma *sup-mctxt-idem* [*simp*]:
fixes *C* :: ('*f*, '*v*) *mctxt*
shows *C* ⊔ *C* = *C*
by (*induct C*) (*auto simp*: *zip-same-conv-map intro*: *map-idI*)

lemma *sup-mctxt-MHole* [*simp*]: *C* ⊔ *MHole* = *C*
by (*induct C*) *simp-all*

lemma *sup-mctxt-comm* [*ac-simps*]:
fixes *C* :: ('*f*, '*v*) *mctxt*
shows *C* ⊔ *D* = *D* ⊔ *C*
by (*induct C D rule*: *sup-mctxt.induct*) (*fastforce simp*: *in-set-conv-nth intro*!: *nth-equalityI*)+

(⊔) 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: $f = g \Longrightarrow$ *length Cs* = *length Ds* $\Longrightarrow \forall\, i <$ *length Ds*. (*Cs* ! *i*, *Ds* ! *i*) $\in$ *comp-mctxt* $\Longrightarrow$
(*MFun f Cs*, *MFun g Ds*) $\in$ *comp-mctxt*

lemma *comp-mctxt-refl*:
(*C*, *C*) $\in$ *comp-mctxt*
by (*induct C*) (*auto intro*: *comp-mctxt.intros*)

lemma *comp-mctxt-sym*:
assumes (*C*, *D*) $\in$ *comp-mctxt*
shows (*D*, *C*) $\in$ *comp-mctxt*
using *assms* **by** (*induct*) (*auto intro*: *comp-mctxt.intros*)

lemma *sup-mctxt-assoc* [*ac-simps*]:
assumes (*C*, *D*) $\in$ *comp-mctxt* **and** (*D*, *E*) $\in$ *comp-mctxt*
shows $C \sqcup D \sqcup E = C \sqcup (D \sqcup E)$
using *assms* **by** (*induct C D arbitrary*: *E*) (*auto elim*!: *comp-mctxt.cases intro*!: *nth-equalityI*)

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* ($\leq$) ($<$)
by (*standard*) (*simp add*: *less-eq-mctxt-def*)

lemma *sup-mctxt-ge1* [*simp*]:
assumes (*C*, *D*) $\in$ *comp-mctxt*
shows $C \leq C \sqcup D$
using *assms* **by** (*induct C D*) (*auto simp*: *less-eq-mctxt-def intro*: *nth-equalityI*)

lemma *sup-mctxt-ge2* [*simp*]:
assumes (*C*, *D*) $\in$ *comp-mctxt*
shows $D \leq C \sqcup D$
using *assms* **by** (*induct*) (*auto simp*: *less-eq-mctxt-def intro*: *nth-equalityI*)

lemma *sup-mctxt-least*:
assumes (*D*, *E*) $\in$ *comp-mctxt*
and $D \leq C$ **and** $E \leq C$
shows $D \sqcup E \leq C$
using *assms*
proof (*induct arbitrary*: *C*)
case (*MFun f g Cs Ds*)
then show *?case*
apply (*auto simp*: *less-eq-mctxt-def elim*!: *inf-mctxt.elims intro*!: *nth-equalityI*)[*1*]
apply (*metis* (*erased*, *lifting*) *length-map nth-map nth-zip split-conv*)
by (*metis mctxt.distinct*(*5*))+
qed *auto*

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*
using *assms*
proof (*induct C D arbitrary*: i)
case (*MHole2 C*)
then show *?case* **by** (*cases C*) *simp-all*
next
case (*MFun f g Cs Ds*)
then have [*simp*]: $f = g$ *length Ds* = *length Cs* **by** *auto*
let *?xs* = *map* (*case-prod inf-mctxt-args*) (*zip Cs Ds*)
let *?ys* = *map* (*case-prod inf-mctxt-args*) (*zip Ds Cs*)
obtain m **and** n **where** $*$: $i =$ *sum-list* (*map length* (*take m ?xs*)) + n
$m <$ *length Cs* $n <$ *length* (*inf-mctxt-args* (*Cs* ! m) (*Ds* ! m))
and *inf-mctxt-args* (*MFun f Cs*) (*MFun g Ds*) ! $i =$ *inf-mctxt-args* (*Cs* ! m) (*Ds* ! m) ! n
using *MFun.prems* **by** (*auto dest*: *less-length-concat*)
moreover have *concat ?ys* ! $i =$ (*map* (*case-prod inf-mctxt-args*) (*zip Ds Cs*)) ! m ! n
by (*rule concat-nth*)
(*insert* $*$, *auto intro*: *arg-cong* [*of - - sum-list*]
simp: *map-nth-eq-conv length-inf-mctxt-args*)
ultimately show *?case* **using** *MFun*(*3*) **by** *simp*
qed *auto*

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 −
have [*simp*]: *length ss* = *length ts* **using** *assms* **by** (*auto dest*!: *eqfE*)
have [*simp*]: $t =$ *fill-holes C ts* **using** *assms* **by** (*auto dest*: *eqfE*)
have (s, *fill-holes C ts*) $\in$ (*rstep R*)*
using *assms* **by** (*intro eqf-all-ctxt-closed-step* [*of UNIV - s C ss*, *THEN conjunct1*]) *auto*
then show *?thesis* **by** *simp*
qed

fun *sup-mctxt-args* :: ('*f*, '*v*) *mctxt* ⇒ ('*f*, '*v*) *mctxt* ⇒ ('*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* ∧ *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
by (*cases C*) *simp-all*

lemma *num-holes-sup-mctxt-args*:
assumes (*C*, *D*) ∈ *comp-mctxt*
shows *num-holes C = length* (*sup-mctxt-args C D*)
using *assms* **by** (*induct*) (*auto simp*: *length-concat intro*!: *arg-cong* [*of* - - *sum-list*] *nth-equalityI*)

lemma *sup-mctxt-sup-mctxt-args*:
assumes (*C*, *D*) ∈ *comp-mctxt*
shows *fill-holes-mctxt C* (*sup-mctxt-args C D*) = *C* ⊔ *D*
using *assms*
proof (*induct*)
note *fill-holes-mctxt.simps* [*simp del*]
case (*MFun f g Cs Ds*)
then show *?case*
proof (*cases f = g* ∧ *length Cs = length Ds*)
case *True*
with *MFun* **have** ∀ *i* < *length Cs*.
fill-holes-mctxt (*Cs* ! *i*) (*sup-mctxt-args* (*Cs* ! *i*) (*Ds* ! *i*)) = *Cs* ! *i* ⊔ *Ds* ! *i*
and ∗: ∀ *i* < *length Cs*. (*Cs* ! *i*, *Ds* ! *i*) ∈ *comp-mctxt* **by** (*force simp*: *set-zip*)+
moreover have *partition-holes* (*concat* (*map* (*case-prod sup-mctxt-args*) (*zip Cs Ds*)))
Cs = map (*case-prod sup-mctxt-args*) (*zip Cs Ds*)
using *True* **and** ∗ **by** (*intro partition-by-concat-id*) (*auto simp*: *num-holes-sup-mctxt-args*)
ultimately show *?thesis*
using ∗ **and** *True* **by** (*auto simp*: *partition-holes-fill-holes-mctxt-conv intro*!: *nth-equalityI*)
qed *auto*
qed *auto*

lemma *sup-mctxt-args*:
assumes (*C*, *D*) ∈ *comp-mctxt*
shows *sup-mctxt-args C D = inf-mctxt-args* (*C* ⊔ *D*) *C*
using *assms* **by** (*induct*) (*auto intro*!: *arg-cong* [*of* - - *concat*] *nth-equalityI*)

lemma *term-for-mctxt*:
fixes *C* :: ('*f*, '*v*) *mctxt*
obtains *t* **and** *ts* **where** *t* =*f* (*C*, *ts*)
proof −
obtain *ts* :: ('*f*, '*v*) *term list* **where** *num-holes C = length ts* **by** (*metis Ex-list-of-length*)
then have *fill-holes C ts* =*f* (*C*, *ts*) **by** *blast*
show *?thesis* **by** (*standard*) *fact*
qed

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 (goal-cases)
  case 1
  obtain u and us where u =_f (C ⊔ D, us) by (metis term-for-mctxt)
  then have u: u = fill-holes (C ⊔ D) us
    and ∗: length us = num-holes (C ⊔ D) by (auto dest: eqfE)
  define Cs Ds where Cs = sup-mctxt-args C D
    and Ds = sup-mctxt-args D C
 then have sup1: C ⊔ D = fill-holes-mctxt C Cs and sup2: C ⊔ D = fill-holes-mctxt
D Ds
   using assms by (auto simp: sup-mctxt-sup-mctxt-args comp-mctxt-sym ac-simps)
  then have u1: u = fill-holes (fill-holes-mctxt C Cs) us
    and u2: u = fill-holes (fill-holes-mctxt D Ds) us by (simp-all add: u)
  define ss ts where ss = map (λi. fill-holes (Cs ! i) (partition-holes us Cs ! i))
[0 ..< num-holes C]
   and ts = map (λi. fill-holes (Ds ! i) (partition-holes us Ds ! i)) [0 ..< num-holes
D]
  have u = fill-holes C ss
    using assms
   by (simp add: ∗ u1 sup1 ss-def fill-holes-mctxt-fill-holes Cs-def num-holes-sup-mctxt-args)
  moreover have u = fill-holes D ts
    using assms [THEN comp-mctxt-sym]
   by (simp add: ∗ u2 sup2 ts-def fill-holes-mctxt-fill-holes Ds-def num-holes-sup-mctxt-args)
  ultimately have u =_f (C, ss) and u =_f (D, ts) by (auto simp: ss-def ts-def)
  from 1[OF this] show thesis .
qed

lemma eqf-comp-mctxt:
  assumes s =_f (C, ss) and s =_f (D, ts)
  shows (C, D) ∈ comp-mctxt
  using assms
proof (induct s arbitrary: C D ss ts)
  case (Var x C D)
  then show ?case
    by (cases C D rule: mctxt.exhaust [case-product mctxt.exhaust])
      (auto simp: eq-fill.simps intro: comp-mctxt.intros)
next
  case (Fun f ss C D us vs)
  { fix Cs and Ds
    assume ∗: C = MFun f Cs D = MFun f Ds and ∗∗: length Cs = length Ds
    have ?case
    proof (unfold ∗, intro comp-mctxt.MFun [OF refl ∗∗] allI impI)
      fix i
      assume i < length Ds
      then show (Cs ! i, Ds ! i) ∈ comp-mctxt
        using Fun by (auto simp: ∗ ∗∗ elim!: eqf-MFunE) (metis nth-mem)
    qed }
  with Fun.prems show ?case
    by (cases C D rule: mctxt.exhaust [case-product mctxt.exhaust])
      (auto simp: eq-fill.simps dest: map-eq-imp-length-eq intro: comp-mctxt.intros)
```

qed

lemma *comp-mctxt-iff*:
  $(C, D) \in$ *comp-mctxt* $\longleftrightarrow$ $(\exists\, s\ ss\ ts.\ s =_f (C, ss) \wedge s =_f (D, ts))$
  by (*blast elim*!: *comp-mctxt-eqfE intro*: *eqf-comp-mctxt*)

lemma *hole-poss-parallel-pos* [*simp*]:
  assumes $p \in$ *hole-poss* C **and** $q \in$ *hole-poss* C **and** $p \neq q$
  shows *parallel-pos* p q
  using *assms* **by** (*induct C arbitrary*: p q) (*fastforce dest*!: *nth-mem*)+

lemma *eq-fill-induct* [*consumes 1*, *case-names MHole MVar MFun*]:
  assumes $t =_f (C, ts)$
    and $\bigwedge t.$ P t *MHole* $[t]$
    and $\bigwedge x.$ P (*Var* x) (*MVar* x) []
    and $\bigwedge f\ ss\ Cs\ ts.$ $[\![$*length* Cs = *length* ss; *sum-list* (*map num-holes* Cs) = *length* ts;
      $\forall\, i <$ *length* $ss.$ $ss\ !\ i =_f (Cs\ !\ i,$ *partition-holes* $ts\ Cs\ !\ i) \wedge$
        P $(ss\ !\ i)$ $(Cs\ !\ i)$ $($*partition-holes* $ts\ Cs\ !\ i)]\!]$
      $\Longrightarrow$ P (*Fun* f ss) (*MFun* f Cs) ts
  shows P t C ts
  using *assms*(*1*)
proof (*induct t arbitrary*: C ts)
  case (*Var* x)
  then show *?case*
    using *assms*(*2*, *3*) **by** (*cases* C; *cases* ts) (*auto elim*: *eq-fill.cases*)
next
  case (*Fun* f ss C ts)
  { assume C = *MHole* **and** ts = [*Fun* f ss]
    with *Fun.hyps* **have** *?case* **using** *assms*(*2*) **by** *auto* **}**
  moreover
  { fix *Cs*
    assume *C*: C = *MFun* f *Cs* **and** *sum-list* (*map num-holes Cs*) = *length ts*
      and *length Cs* = *length ss*
      and *Fun f ss* = *fill-holes* (*MFun f Cs*) *ts*
    moreover then have $\forall\, i <$ *length ss.* $ss\ !\ i =_f (Cs\ !\ i,$ *partition-holes ts Cs* $!$ $i)$
      by (*auto simp del*: *fill-holes.simps*
          simp: *partition-holes-fill-holes-conv intro*!: *eq-fill.intros*)
       (*metis* (*no-types*, *lifting*) *add.left-neutral length-map length-upt nth-map-upt*)
    moreover with *Fun.hyps*(*1*) **have** $\forall\, i <$ *length ss.*
      P $(ss\ !\ i)$ $(Cs\ !\ i)$ $($*partition-holes ts Cs* $!$ $i)$ **by** *auto*
    ultimately have *?case* **using** *assms*(*4*) [*of Cs ss ts f*] **by** *auto* **}**
  ultimately show *?case*
    using *Fun.prems* **by** (*elim eq-fill.cases*) (*auto*, *cases* C; *cases ts*, *auto*)
qed

lemma *hole-poss-subset-poss*:
  assumes $s =_f (C, ss)$

shows *hole-poss C* ⊆ *poss s*
using *assms* **by** (*induct rule*: *eq-fill-induct*) *auto*

fun *hole-num*
where
hole-num [] *MHole* = *0* |
hole-num (*i* # *q*) (*MFun f Cs*) = *sum-list* (*map num-holes* (*take i Cs*)) + *hole-num q* (*Cs* ! *i*)

lemma *hole-poss-nth-subt-at*:
assumes *t* =$_f$ (*C*, *ts*) **and** *p* ∈ *hole-poss C*
shows *hole-num p C* < *length ts* ∧ *t* |- *p* = *ts* ! *hole-num p C*
using *assms*
proof (*induct arbitrary*: *p rule*: *eq-fill-induct*)
case (*MFun f ss Cs ts*)
let *?ts* = *partition-holes ts Cs*
from *MFun* **obtain** *i* **and** *q* **where** [*simp*]: *p* = *i* # *q*
and *i* < *length ss* **and** *q* ∈ *hole-poss* (*Cs* ! *i*) **by** *auto*
with *MFun.hyps* **have** *ss* ! *i* =$_f$ (*Cs* ! *i*, *?ts* ! *i*)
and *j*: *hole-num q* (*Cs* ! *i*) < *length* (*?ts* ! *i*) (**is** *?j* < *length* -)
and ∗: *?ts* ! *i* ! *hole-num q* (*Cs* ! *i*) = *ss* ! *i* |- *q*
by *auto*
let *?k* = *sum-list* (*map length* (*take i ?ts*)) + *?j*
have *i* < *length ?ts* **using** ‹*i* < *length ss*› **and** *MFun* **by** *auto*
with *partition-by-nth-nth-old* [*OF this j*] **and** *MFun* **and** *concat-nth-length* [*OF this j*]
have *?ts* ! *i* ! *?j* = *ts* ! *?k* **and** *?k* < *length ts* **by** (*auto*)
moreover with ∗ **have** *ts* ! *?k* = *Fun f ss* |- *p* **using** ‹*i* < *length ss*› **by** *simp*
ultimately show *?case* **using** *MFun.hyps*(*2*) **by** (*auto simp*: *take-map* [*symmetric*])
qed *auto*

lemma *eqf-Fun-MFun*:
assumes *Fun f ss* =$_f$ (*MFun g Cs*, *ts*)
shows *g* = *f* ∧ *length Cs* = *length ss* ∧ *sum-list* (*map num-holes Cs*) = *length ts* ∧
(∀ *i* < *length ss*. *ss* ! *i* =$_f$ (*Cs* ! *i*, *partition-holes ts Cs* ! *i*))
using *assms* **by** (*induct Fun f ss MFun g Cs ts rule*: *eq-fill-induct*) *auto*

lemma *fill-holes-eq-Var-cases*:
assumes *num-holes C* = *length ts*
and *fill-holes C ts* = *Var x*
obtains *C* = *MHole* ∧ *ts* = [*Var x*] | *C* = *MVar x* ∧ *ts* = []
using *assms* **by** (*induct C*; *cases ts*) *auto*

lemma *num-holes-inf-mctxt-le*:
assumes *s* =$_f$ (*C*, *ts*) **and** *s* =$_f$ (*D*, *us*)
shows *num-holes* (*C* ⊓ *D*) ≤ *num-holes C* + *num-holes D*
using *assms*
proof (*induct C D arbitrary*: *s ts us rule*: *inf-mctxt.induct*)

```
  case (4 f Cs g Ds)
  show ?case
  proof (cases f = g ∧ length Cs = length Ds)
    case False
    with 4 show ?thesis by (auto elim!: eq-fill.cases dest!: map-eq-imp-length-eq)
  next
    case True
    then have [simp]: g = f length Ds = length Cs by simp-all
    have IH: ∀(C, D) ∈ set (zip Cs Ds). num-holes (C ⊓ D) ≤ num-holes C +
num-holes D
    proof
      fix C D assume ∗: (C, D) ∈ set (zip Cs Ds)
      then obtain i where i < length Cs and zip Cs Ds ! i = (C, D) by (auto
simp: in-set-zip)
      with 4.prems
      have fill-holes (Cs ! i) (partition-holes ts Cs ! i) =f (C, partition-holes ts Cs
! i)
        and fill-holes (Cs ! i) (partition-holes ts Cs ! i) =f (D, partition-holes us
Ds ! i)
        by (auto elim!: eq-fill.cases)
      from 4.hyps [OF True ∗ HOL.refl this]
      show num-holes (C ⊓ D) ≤ num-holes C + num-holes D .
    qed
    have num-holes (MFun f Cs ⊓ MFun g Ds) = sum-list (map (num-holes ∘
case-prod (⊓)) (zip Cs Ds))
      using 4.prems by (auto elim!: eq-fill.cases dest!: map-eq-imp-length-eq)
    moreover have num-holes (MFun f Cs) + num-holes (MFun g Ds) =
      sum-list (map (λ(C, D). num-holes C + num-holes D) (zip Cs Ds))
      using ‹length Ds = length Cs› by (induct rule: list-induct2) simp-all
    ultimately show ?thesis using IH by (auto intro!: sum-list-mono)
  qed
qed (auto elim!: eq-fill.cases)

lemma map-inf-mctxt-zip-mctxt-of-term [simp]:
  map (λ(x, y). x ⊓ y) (zip (map mctxt-of-term ts) (map mctxt-of-term ts)) = map
mctxt-of-term ts
  by (induct ts) simp-all

lemma inf-mctxt-ctxt-apply-term [simp]:
  mctxt-of-term (C⟨t⟩) ⊓ mctxt-of-ctxt C = mctxt-of-ctxt C
  mctxt-of-ctxt C ⊓ mctxt-of-term (C⟨t⟩) = mctxt-of-ctxt C
  by (induct C) simp-all

lemma inf-fill-holes-mctxt-MHoles:
  num-holes C = length Cs ⟹ length Ds = length Cs ⟹
  ∀ i<length Cs. Cs ! i = MHole ∨ Ds ! i = MHole ⟹
  fill-holes-mctxt C Cs ⊓ fill-holes-mctxt C Ds = C
proof (induct C arbitrary: Cs Ds)
  case (MHole Cs Ds)
```

```
  then show ?case by (cases Cs; cases Ds; force)
next
  case (MFun f Bs Cs Ds)
  then show ?case
    unfolding partition-holes-fill-holes-mctxt-conv'
    apply simp
    apply (rule nth-equalityI)
    by (auto simp: partition-by-nth-nth)
qed auto

lemma inf-fill-holes-mctxt-two-MHoles [simp]: num-holes C = 2 ⟹
  fill-holes-mctxt C [MHole, D] ⊓ fill-holes-mctxt C [E, MHole] = C
  by (simp add: inf-fill-holes-mctxt-MHoles nth-Cons')

lemma two-subterms-cases:
  assumes s = C⟨t⟩ and s = D⟨u⟩
  obtains (eq) C = D and t = u
  | (nested1) C' where C' ≠ □ and C = D ∘c C'
  | (nested2) D' where D' ≠ □ and D = C ∘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 (atomize-elim, insert assms, induct s arbitrary: C t D u)
  case (Var x)
  then show ?case by (cases C; cases D; cases t; cases u) auto
next
  case (Fun f ss)
  { fix ts1 C' ts2 and us1 D' us2
    assume [simp]: C = More f ts1 C' ts2 D = More f us1 D' us2
    then have len: length (ts1 @ ts2) + 1 = length ss length (us1 @ us2) + 1 =
length ss
      using Fun.prems by (auto) (metis add-Suc-right length-Cons length-append
nat.inject)
    { assume length ts1 = length us1
      with Fun have [simp]: take (length ts1) ss = ts1 drop (Suc (length ts1)) ss =
ts2
        and [simp]: us1 = take (length ts1) ss us2 = drop (length ts1 + 1) ss
        and nth: C'⟨t⟩ = ss ! length ts1 and mem: C'⟨t⟩ ∈ set ss
        and eq: C'⟨t⟩ = D'⟨u⟩ by auto
      { assume C' = D' and t = u
        then have C = D and t = u by simp-all
        then have ?case by blast }
      moreover
      { fix C'' assume C'' ≠ □ and C' = D' ∘c C''
        then have C'' ≠ □ and C = D ∘c C'' by auto
        then have ?case by blast }
```

moreover
{ fix D'' **assume** $D'' \neq \Box$ **and** $D' = C' \circ_c D''$
then have $D'' \neq \Box$ **and** $D = C \circ_c D''$ **by** *auto*
then have *?case* **by** *blast* **}**
moreover
{ fix E' **assume** [*simp*]: *mctxt-of-ctxt* $C' =$ *fill-holes-mctxt* E' [*MHole*, *mctxt-of-term* u]
mctxt-of-ctxt $D' =$ *fill-holes-mctxt* E' [*mctxt-of-term* t, *MHole*]
num-holes $E' = 2$
define E **where** $E =$ *MFun* f (*map mctxt-of-term* ts_1 @ E' # *map mctxt-of-term* ts_2)
then have *num-holes* $E = 2$ **by** *simp*
moreover have *mctxt-of-ctxt* $C =$ *fill-holes-mctxt* E [*MHole*, *mctxt-of-term* u]
unfolding *E-def* **and** *partition-holes-fill-holes-mctxt-conv'* **by** *simp*
moreover have *mctxt-of-ctxt* $D =$ *fill-holes-mctxt* E [*mctxt-of-term* t, *MHole*]
unfolding *E-def* **and** *partition-holes-fill-holes-mctxt-conv'* **by** *simp*
ultimately have *?case* **by** *blast* **}**
moreover
{ fix E' **assume** [*simp*]: *mctxt-of-ctxt* $C' =$ *fill-holes-mctxt* E' [*mctxt-of-term* u, *MHole*]
mctxt-of-ctxt $D' =$ *fill-holes-mctxt* E' [*MHole*, *mctxt-of-term* t]
num-holes $E' = 2$
define E **where** $E =$ *MFun* f (*map mctxt-of-term* ts_1 @ E' # *map mctxt-of-term* ts_2)
then have *num-holes* $E = 2$ **by** *simp*
moreover have *mctxt-of-ctxt* $C =$ *fill-holes-mctxt* E [*mctxt-of-term* u, *MHole*]
unfolding *E-def* **and** *partition-holes-fill-holes-mctxt-conv'* **by** *simp*
moreover have *mctxt-of-ctxt* $D =$ *fill-holes-mctxt* E [*MHole*, *mctxt-of-term* t]
unfolding *E-def* **and** *partition-holes-fill-holes-mctxt-conv'* **by** *simp*
ultimately have *?case* **by** *blast* **}**
ultimately have *?case* **using** *Fun.hyps* [*OF mem HOL.refl eq*] **by** *blast* **}**
moreover
{ assume $*$: *length* $ts_1 <$ *length* us_1
moreover then have us_1: $us_1 = ts_1$ @ $C'\langle t\rangle$ # *drop* (*length* $ts_1 + 1$) us_1
using *Fun.prems* [*simplified*]
apply (*subst append-take-drop-id* [*symmetric*, *of* - *length* ts_1])
apply (*rule arg-cong2* [**where** $f = (@)$])
apply (*force simp*: *append-eq-append-conv-if*)
apply (*simp add*: *append-eq-append-conv-if*)
apply (*cases* us_1)
by *auto*
(*metis Cons-eq-appendI Cons-nth-drop-Suc calculation drop-Suc-Cons nth-append-length*)
ultimately have *ss*: $ss = ts_1$ @ $C'\langle t\rangle$ # *drop* (*length* $ts_1 + 1$) us_1 @ $D'\langle u\rangle$ # us_2
using *Fun.prems*(*2*, *1*) **by** *auto*

have ts_2: $ts_2 = drop\ (length\ ts_1 + 1)\ us_1\ @\ D'\langle u\rangle\ \#\ us_2$
using *Fun.prems* (*2*, *1*) [*simplified*] **and** $*$
apply (*subst append-take-drop-id* [*symmetric*, *of* - *length* (*drop* (*length* ts_1 + *1*) us_1)])
apply (*rule arg-cong2* [**where** $f = (@)$])
by *auto* (*metis Suc-eq-plus1 append-eq-conv-conj length-drop list.inject ss*)+
define E **where** $E = MFun\ f\ (map\ mctxt\text{-}of\text{-}term\ ts_1\ @\ mctxt\text{-}of\text{-}ctxt\ C'\ \#$
map mctxt-of-term (*drop* (*length* ts_1 + *1*) us_1) @ *mctxt-of-ctxt* D' # *map mctxt-of-term* us_2)
then have *num-holes* $E = 2$ **by** *simp*
moreover have *mctxt-of-ctxt* C = *fill-holes-mctxt* E [*MHole*, *mctxt-of-term* u]
unfolding *E-def* **and** *partition-holes-fill-holes-mctxt-conv'* **by** (*simp add*: $*$ ts_2)
moreover have *mctxt-of-ctxt* D = *fill-holes-mctxt* E [*mctxt-of-term* t, *MHole*]
unfolding *E-def* **and** *partition-holes-fill-holes-mctxt-conv'* **by** (*simp*, *subst* us_1, *simp*)
ultimately have *?case* **by** *blast* **}**
moreover
{ assume $*$: $length\ us_1 < length\ ts_1$
moreover then have ts_1: $ts_1 = us_1\ @\ D'\langle u\rangle\ \#\ drop\ (length\ us_1 + 1)\ ts_1$
using *Fun.prems* [*simplified*]
apply (*subst append-take-drop-id* [*symmetric*, *of* - *length* us_1])
apply (*rule arg-cong2* [**where** $f = (@)$])
apply (*force simp*: *append-eq-append-conv-if*)
apply (*simp add*: *append-eq-append-conv-if*)
apply (*cases* ts_1)
by *auto* (*metis Cons-eq-appendI Cons-nth-drop-Suc calculation drop-Suc-Cons nth-append-length*)
ultimately have *ss*: $ss = us_1\ @\ D'\langle u\rangle\ \#\ drop\ (length\ us_1 + 1)\ ts_1\ @\ C'\langle t\rangle\ \#\ ts_2$
using *Fun.prems* **by** *auto*
have us_2: $us_2 = drop\ (length\ us_1 + 1)\ ts_1\ @\ C'\langle t\rangle\ \#\ ts_2$
using *Fun.prems* (*2*, *1*) [*simplified*] **and** $*$
apply (*subst append-take-drop-id* [*symmetric*, *of* - *length* (*drop* (*length* us_1 + *1*) ts_1)])
apply (*rule arg-cong2* [**where** $f = (@)$])
by *auto* (*metis Suc-eq-plus1 append-eq-conv-conj length-drop list.inject ss*)+
define E **where** $E = MFun\ f\ (map\ mctxt\text{-}of\text{-}term\ us_1\ @\ mctxt\text{-}of\text{-}ctxt\ D'\ \#$
map mctxt-of-term (*drop* (*length* us_1 + *1*) ts_1) @ *mctxt-of-ctxt* C' # *map mctxt-of-term* ts_2)
then have *num-holes* $E = 2$ **by** *simp*
moreover have *mctxt-of-ctxt* C = *fill-holes-mctxt* E [*mctxt-of-term* u, *MHole*]
unfolding *E-def* **and** *partition-holes-fill-holes-mctxt-conv'* **by** (*simp*, *subst* ts_1, *simp*)
moreover have *mctxt-of-ctxt* D = *fill-holes-mctxt* E [*MHole*, *mctxt-of-term* t]
unfolding *E-def* **and** *partition-holes-fill-holes-mctxt-conv'* **by** (*simp add*: $*$ us_2)

ultimately have *?case* **by** *blast* **}**
moreover
have *length* ts_1 = *length* us_1 ∨ *length* ts_1 < *length* us_1 ∨ *length* us_1 < *length* ts_1 **by** *arith*
ultimately have *?case* **by** *blast* **}**
moreover
{ assume $C = \Box$ **and** $D \neq \Box$ **then have** *?case* **by** *auto* **}**
moreover
{ assume $C \neq \Box$ **and** $D = \Box$ **then have** *?case* **by** *auto* **}**
moreover
{ assume $C = \Box$ **and** $D = \Box$ **then have** *?case* **using** *Fun* **by** *simp* **}**
ultimately show *?case* **using** *Fun* **by** (*cases C*; *cases D*) *simp-all*
qed

lemma *two-hole-ctxt-inf-conv*:
num-holes E = *2* ⟹
mctxt-of-ctxt C = *fill-holes-mctxt E* [*MHole*, *mctxt-of-term u*] ⟹
mctxt-of-ctxt D = *fill-holes-mctxt E* [*mctxt-of-term t*, *MHole*] ⟹
mctxt-of-ctxt C ⊓ *mctxt-of-ctxt D* = *E*
by *simp*

lemma *map-length-take-partition-by*:
i < *length ys* ⟹ *sum-list ys* = *length xs* ⟹
map length (*take i* (*partition-by xs ys*)) = *take i ys*
by (*metis map-length-partition-by take-map*)

Closure under contexts can be lifted to multihole contexts.

lemma *ctxt-imp-mctxt*:
assumes ∀ *s t C*. (*s*, *t*) ∈ *R* ⟶ (*C*⟨*s*⟩, *C*⟨*t*⟩) ∈ *R*
and (*t*, *u*) ∈ *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)) ∈ *R*
using *assms*
proof (*induct C arbitrary*: ss_1 ss_2)
case (*MFun f Cs*)
let *?f* = λ*x*. *partition-holes* (ss_1 @ *x* # ss_2) *Cs*
let *?ts* = *?f t* **and** *?us* = *?f u*
have ∗: ⋀*x*. *concat* (*?f x*) = ss_1 @ *x* # ss_2
using *MFun.prems* **by** (*intro concat-partition-by*) *simp*
with *less-length-concat* [*of length* ss_1 *?ts*]
obtain *i j* **where** *ij*: *sum-list* (*map length* (*take i ?ts*)) + *j* = *length* ss_1
i < *length Cs j* < *length* (*?ts* ! *i*)
and [*simp*]: *?ts* ! *i* ! *j* = *t* **by** *auto*
have *length* ss_1 = *sum-list* (*map length* (*take i ?us*)) + *j*
using *ij* **using** *MFun.prems*(*3*) **by** (*auto simp*: *take-map* [*symmetric*])
from *concat-nth* [*OF - - this*]
have [*simp*]: *?us* ! *i* ! *j* = *u* **using** *ij* **and** *MFun.prems*(*3*) **by** *auto*
have [*simp*]: *length ?us* = *length ?ts* **by** *simp*
have [*simp*]: *take j* (*?us* ! *i*) = *take j* (*?ts* ! *i*)

drop (*Suc j*) (*?us* ! *i*) = *drop* (*Suc j*) (*?ts* ! *i*)
using *ij* **and** *MFun.prems*(*3*)
by (*auto intro*: *nth-equalityI simp*: *nth-append concat-nth* [*symmetric*] *take-map* [*symmetric*])
from *MFun.hyps* [*of Cs* ! *i*, *OF* - *MFun.prems*(*1*, *2*), *of take j* (*?ts* ! *i*) *drop* (*Suc j*) (*?ts* ! *i*)]
have *step*: (*fill-holes* (*Cs* ! *i*) (*?ts* ! *i*), *fill-holes* (*Cs* ! *i*) (*?us* ! *i*)) ∈ *R*
using *ij* **and** *MFun.prems*
apply *simp*
apply (*subst id-take-nth-drop* [*of j ?ts* ! *i*])
apply *simp*
apply (*subst id-take-nth-drop* [*of j ?us* ! *i*])
apply *auto*
done

let *?Cs* = *map* (*case-prod fill-holes*) (*zip Cs ?ts*)
let *?C* = *More f* (*take i ?Cs*) □ (*drop* (*Suc i*) *?Cs*)
have [*simp*]:
take i (*map* (*case-prod fill-holes*) (*zip Cs ?us*)) = *take i* (*map* (*case-prod fill-holes*) (*zip Cs ?ts*))
drop (*Suc i*) (*map* (*case-prod fill-holes*) (*zip Cs ?us*)) = *drop* (*Suc i*) (*map* (*case-prod fill-holes*) (*zip Cs ?ts*))
using *ij* **and** *MFun.prems*(*3*)
apply (*auto intro*!: *nth-equalityI*)[*2*]
subgoal
using *partition-by-nth-less* [*of* - *i map num-holes Cs* ss_1 *j* - ss_2]
by (*simp add*: *map-length-take-partition-by*)
subgoal using *partition-by-nth-greater* [*of i Suc* (*i* + *k*) **for** *k*, *of* - *map num-holes Cs j* ss_1 - ss_2]
by (*simp add*: *map-length-take-partition-by*)
done
show *?case*
using *MFun.prems*(*1*) [*rule-format*, *OF step*, *of ?C*] **and** *ij*
apply (*clarsimp simp del*: *fill-holes.simps simp*: *partition-holes-fill-holes-conv*′)
apply (*subst id-take-nth-drop* [*of i map* (*case-prod fill-holes*) (*zip Cs ?ts*)], *simp*)
apply (*subst id-take-nth-drop* [*of i map* (*case-prod fill-holes*) (*zip Cs ?us*)], *simp*)
by *auto*
qed *auto*

lemma *mctxt-of-term-fill-holes*′:
num-holes C = *length ts* ⟹ *mctxt-of-term* (*fill-holes C ts*) = *fill-holes-mctxt C* (*map mctxt-of-term ts*)
by (*induct C ts rule*: *fill-holes-induct*) *auto*

lemma *vars-term-fill-holes*′:
num-holes C = *length ts* ⟹ *vars-term* (*fill-holes C ts*) = ⋃(*vars-term* ‘ *set ts*) ∪ *vars-mctxt C*
proof (*induct C ts rule*: *fill-holes-induct*)
case (*MFun f Cs ts*) **then show** *?case*

using *UN-upt-len-conv*[*of partition-holes ts Cs length Cs* $\lambda t.$ ($\bigcup x \in set\ t.$ *vars-term* *x*)]
by (*simp add*: *UN-Un-distrib UN-set-partition-by*)
qed *auto*

lemma *vars-mctxt-linear*: **assumes** $t =_f (C, ts)$
linear-term t
shows *vars-mctxt C* $\cap \bigcup$ (*vars-term* ‘ *set ts*) = {}
using *assms*
proof (*induct C arbitrary*: *t ts*)
case (*MVar x*)
from *eqf-MVarE*[*OF MVar*(*1*)]
show *?case* **by** *auto*
next
case *MHole*
from *eqf-MHoleE*[*OF MHole*(*1*)]
show *?case* **by** *auto*
next
case (*MFun f Cs t ss*)
from *eqf-MFunE*[*OF MFun*(*2*)] **obtain** *ts sss* **where**
∗: *t* = *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* **by** *blast*
{
fix *i*
assume *i*: *i* < *length Cs*
hence *mem*: *Cs* ! *i* ∈ *set Cs* **by** *auto*
from ∗ *i MFun*(*3*) **have** *lin*: *linear-term* (*ts* ! *i*) **by** *auto*
from *MFun*(*1*)[*OF mem* ∗(*4*)[*OF i*] *lin*]
have *vars-mctxt* (*Cs* ! *i*) ∩ $\bigcup$ (*vars-term* ‘ *set* (*sss* ! *i*)) = {} **by** *auto*
} note *IH* = *this*
show *?case*
proof (*rule ccontr*)
assume ¬ *?thesis*
then obtain *x* **where** *xC*: *x* ∈ *vars-mctxt* (*MFun f Cs*) **and** *xss*: *x* ∈ $\bigcup$ (*vars-term* ‘ *set ss*)
by *auto*
from *xC* **obtain** *i* **where** *i*: *i* < *length Cs* **and** *x*: *x* ∈ *vars-mctxt* (*Cs* ! *i*)
by (*auto simp*: *set-conv-nth*)
from *IH*[*OF i*] *x* **have** *xni*: *x* ∉ $\bigcup$ (*vars-term* ‘ *set* (*sss* ! *i*)) **by** *auto*
from ∗(*4*)[*OF i*] **have** *ts* ! *i* =$_f$ (*Cs* ! *i*, *sss* ! *i*) **.**
from *eqfE*[*OF this*] *x* **have** *xi*: *x* ∈ *vars-term* (*ts* ! *i*)
by (*simp add*: *vars-term-fill-holes*′)
from *xss*[*unfolded* ∗ *set-concat*] ∗ **obtain** *j* **where**
j: *j* < *length Cs* **and** *xsj*: *x* ∈ $\bigcup$ (*vars-term* ‘ *set* (*sss* ! *j*))
unfolding *set-conv-nth* **by** *auto*
from ∗(*4*)[*OF j*] **have** *ts* ! *j* =$_f$ (*Cs* ! *j*, *sss* ! *j*) **by** *auto*
from *eqfE*[*OF this*] *xsj j* **have** *xj*: *x* ∈ *vars-term* (*ts* ! *j*)
by (*simp add*: *vars-term-fill-holes*′)

```
    from xi xj i j ‹linear-term t›[unfolded ∗(1)]
    have i = j unfolding ‹length ts = length Cs›[symmetric]
      by (auto simp: is-partition-alt is-partition-alt-def)
    with xni xsj show False by auto
  qed
qed

lemma mctxt-of-term-var-subst:
  mctxt-of-term (t · (Var ∘ f)) = map-vars-mctxt f (mctxt-of-term t)
  by (induct t) auto

lemma subst-apply-mctxt-map-vars-mctxt-conv:
  C ·mc (Var ∘ f) = map-vars-mctxt f C
  by (induct C) auto

lemma map-vars-mctxt-mono:
  C ≤ D ⟹ map-vars-mctxt f C ≤ map-vars-mctxt f D
  by (induct C D rule: less-eq-mctxt-induct) (auto intro: less-eq-mctxtI1)

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
  using assms
proof (induct D arbitrary: C thesis)
  case (MVar x) show ?case using MVar(1)[of MHole] MVar(1)[of MVar -]
MVar(2)
    by (auto elim: less-eq-mctxtE2 intro: less-eq-mctxtI1)
next
  case MHole show ?case using MHole(1)[of MHole] MHole(2) by (auto elim:
less-eq-mctxtE2)
next
  case (MFun g Ds) note MFun' = MFun
  show ?case using MFun(3) unfolding map-vars-mctxt.simps
  proof (cases rule: less-eq-mctxtE2(3))
    case MHole then show ?thesis using MFun(2)[of MHole] by auto
  next
    case (MFun Cs)
    define Cs' where Cs' = map (λi. SOME Ci'. map-vars-mctxt f Ci' = Cs ! i
∧ Ci' ≤ Ds ! i) [0..<length Cs]
    { fix i assume i < length Cs
      obtain Ci' where map-vars-mctxt f Ci' = Cs ! i Ci' ≤ Ds ! i
        using ‹i < length Cs› MFun MFun'(1)[OF nth-mem, of i] MFun'(3) by
(auto elim!: less-eq-mctxtE2)
      then have ∃ Ci'. map-vars-mctxt f Ci' = Cs ! i ∧ Ci' ≤ Ds ! i by blast
    }
    from someI-ex[OF this] have
      length Cs = length Cs' and i < length Cs ⟹ map-vars-mctxt f (Cs' ! i) =
Cs ! i
      i < length Cs ⟹ Cs' ! i ≤ Ds ! i for i by (auto simp: Cs'-def)
```

then show *?thesis* **using** *MFun(1,2) MFun'(3)*
by (*auto intro*!: *MFun'(2)*[*of MFun g Cs'*] *nth-equalityI less-eq-mctxtI2 elim*!: *less-eq-mctxtE2*)
qed
qed

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*)}
by *auto*

declare *all-poss-mctxt.simps*(*3*)[*simp del*]

lemma *all-poss-mctxt-conv*:
all-poss-mctxt C = *poss-mctxt C* ∪ *hole-poss C*
by (*induct C*) *auto*

lemma *root-in-all-poss-mctxt*[*simp*]:
[] ∈ *all-poss-mctxt C*
by (*cases C*) *auto*

lemma *hole-poss-mctxt-of-term*[*simp*]:
hole-poss (*mctxt-of-term t*) = {}
by (*induct t*) *auto*

lemma *poss-mctxt-mctxt-of-term*[*simp*]:
poss-mctxt (*mctxt-of-term t*) = *poss t*
by (*induct t*) *auto*

lemma *hole-poss-subst*: *hole-poss* (*C* ·*mc* σ) = *hole-poss C*
by (*induct C*, *auto*)

lemma *all-poss-mctxt-mctxt-of-term*[*simp*]:
all-poss-mctxt (*mctxt-of-term t*) = *poss t*
by (*induct t*) *auto*

lemma *mctxt-of-term-leq-imp-eq*:
mctxt-of-term t ≤ *C* ⟷ *mctxt-of-term t* = *C*
by (*induct t arbitrary*: *C*) (*auto elim*!: *less-eq-mctxtE1 simp*: *map-nth-eq-conv*)

lemma *mctxt-of-term-inj*:
mctxt-of-term s = mctxt-of-term t ⟷ s = t
proof (*induct s arbitrary*: *t*)
case (*Var x t*)
show *?case* **by** (*cases t*, *auto*)
next
case (*Fun f ss t*)
thus *?case* **by** (*cases t*, *auto simp*: *map-eq-conv′ intro*: *nth-equalityI*)
qed

lemma *all-poss-mctxt-map-vars-mctxt* [*simp*]:
all-poss-mctxt (*map-vars-mctxt f C*) = *all-poss-mctxt C*
by (*induct C*) *auto*

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*)
using *assms*[*symmetric*] **by** (*induct C Ds rule*: *fill-holes-induct*) *force*+

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*
using *assms*
proof (*induct C arbitrary*: *t ss*)
case (*MVar x t ss*)
from *eqfE*[*OF MVar*(*1*)] **obtain** *y* **where** *t* = *Var y σ y* = *Var x ss* = [] **by** (*cases t*, *auto*)
thus *?case* **using** *MVar* **by** (*auto intro*!: *exI*[*of* - *MVar y*])
next
case (*MHole t ss*)
from *eqfE*[*OF MHole*(*1*)]
show *?case* **by** (*cases ss*, *auto intro*!: *exI*[*of* - *MHole*] *exI*[*of* - [*t*]])
next
case (*MFun f Cs t ss*)
show *?case*
proof (*cases is-Fun t*)
case *True*
from *eqf-MFunE*[*OF MFun*(*2*)] **obtain** *tss sss* **where**
tsigma: *t* · *σ* = *Fun f tss* **and** *len*: *length tss* = *length Cs length sss* = *length Cs*
and *args*: ⋀ *i*. *i* < *length Cs* ⟹ *tss* ! *i* =$_f$ (*Cs* ! *i*, *sss* ! *i*)
and *ss*: *ss* = *concat sss* **by** *auto*
from *True tsigma* **obtain** *ts*: *t* = *Fun f ts* **by** (*cases t*, *auto*)
from *tsigma*[*unfolded t*] **have** *ts*: *tss* = *map* (λ *t*. *t* · *σ*) *ts* **by** *auto*
from *len ts* **have** *length ts* = *length Cs* **by** *auto*
note *len* = *this len*

```
    {
      fix i
      assume i: i < length Cs
      hence Cs ! i ∈ set Cs by auto
      note IH = MFun(1)[OF this]
      from ts len i have ts ! i · σ = tss ! i by auto
      also have ... =_f (Cs ! i, sss ! i) using args[OF i] .
      finally have ts ! i · σ =_f (Cs ! i, sss ! i) .
      note IH = IH[OF this]
      from MFun(3)[unfolded t] i len
      have hole-poss (Cs ! i) ⊆ poss (ts ! i) by auto
      note IH = IH[OF this]
    }
    hence ∀ i. ∃ D tsi. i < length Cs ⟶ ts ! i =_f (D, tsi) ∧ Cs ! i = D ·mc σ
∧ sss ! i = map (λti. ti · σ) tsi by blast
    from choice[OF this] obtain D where
      ∀ i. ∃ tsi. i < length Cs ⟶ ts ! i =_f (D i, tsi) ∧ Cs ! i = D i ·mc σ ∧ sss
! i = map (λti. ti · σ) tsi ..
    from choice[OF this] obtain tsi where
      IH: i < length Cs ⟹ ts ! i =_f (D i, tsi i) ∧ Cs ! i = D i ·mc σ ∧ sss ! i =
map (λti. ti · σ) (tsi i) for i
      by auto
    let ?n = [0 ..< length Cs]
    show ?thesis
    proof (rule exI[of - MFun f (map D ?n)], rule exI[of - concat (map tsi ?n)],
intro conjI)
      show MFun f Cs = MFun f (map D ?n) ·mc σ using IH by (auto intro:
nth-equalityI)
      show ss = map (λti. ti · σ) (concat (map tsi ?n)) unfolding ss
        using len(3) IH unfolding map-concat map-map o-def
        by (intro arg-cong[of - - concat], intro nth-equalityI, auto)
      show t =_f (MFun f (map D ?n), concat (map tsi ?n)) unfolding t
        by (intro eqf-MFunI, insert len IH, auto)
    qed
  next
    case False
    then obtain x where t: t = Var x by auto
    with MFun(3) have hole-poss (MFun f Cs) = {} by auto
   hence num: num-holes (MFun f Cs) = 0 using hole-poss-empty-iff-num-holes-0
by blast
    with eqfE[OF MFun(2)] t have ss: ss = [] σ x = fill-holes (MFun f Cs) [] by
auto
    show ?thesis unfolding t ss
    proof (intro exI[of - MVar x] exI[of - Nil] conjI)
      have MVar x ·mc σ = mctxt-of-term (fill-holes (MFun f Cs) []) using ss by
simp
      also have ... = MFun f Cs using num
        by (metis mctxt-of-term-fill-holes mctxt-of-term-term-of-mctxt-id)
      finally show MFun f Cs = MVar x ·mc σ ..
```

qed *auto*
qed
qed

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*
by (*induct C arbitrary*: *p*) *auto*

lemma *subm-at-mctxt-of-term*:
p ∈ *poss t* ⟹ *subm-at* (*mctxt-of-term t*) *p* = *mctxt-of-term* (*subt-at t p*)
by (*induct t arbitrary*: *p*) *auto*

lemma *subm-at-mreplace-at*[*simp*]:
p ∈ *all-poss-mctxt C* ⟹ *subm-at* (*mreplace-at C p D*) *p* = *D*
by (*induct C arbitrary*: *p*) (*auto simp*: *nth-append-take*)

lemma *replace-at-subm-at*[*simp*]:
p ∈ *all-poss-mctxt C* ⟹ *mreplace-at C p* (*subm-at C p*) = *C*
by (*induct C arbitrary*: *p*) (*auto simp*: *id-take-nth-drop*[*symmetric*])

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 (*induct C arbitrary*: *p q*)
let *?hd* = λ*p*. (*case p* :: *pos of i* # - ⇒ *i*)
case (*MFun f Cs*) **then show** *?case*
by (*cases ?hd p* = *?hd q*) (*auto simp*: *nth-append-take less-pos-def nth-append-drop-is-nth-conv nth-append-take-drop-is-nth-conv*)
qed *auto*

lemma *funas-mctxt-sup-mctxt*:
(*C*, *D*) ∈ *comp-mctxt* ⟹ *funas-mctxt* (*C* ⊔ *D*) = *funas-mctxt C* ∪ *funas-mctxt*

D
by (*induct C D rule*: *comp-mctxt.induct*) (*auto simp*: *zip-nth-conv Un-Union-image*)

lemma *mctxt-of-term-not-hole* [*simp*]:
mctxt-of-term t ≠ *MHole*
by (*cases t*) *auto*

lemma *funas-mctxt-mctxt-of-term* [*simp*]:
funas-mctxt (*mctxt-of-term t*) = *funas-term t*
by (*induct t*) *auto*

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*
using *assms*
proof (*induct C p D rule*: *mreplace-at.induct*)
case (*2 f Cs i p D*) **then have** *Cs*: *Cs* = *take i Cs* @ *Cs* ! *i* # *drop* (*Suc i*) *Cs*
by (*auto simp*: *id-take-nth-drop*)
show *?case* **using** *2* **by** (*subst* (*2*) *Cs*) *auto*
qed *auto*

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
show *?R* ⊆ *?L* **using** *assms*
proof (*induct C p D rule*: *mreplace-at.induct*)
case (*1 C D*) **then show** *?case* **by** (*cases C*) *auto*
next
case (*2 f Cs i p D*) **then have** *Cs*: *Cs* = *take i Cs* @ *Cs* ! *i* # *drop* (*Suc i*) *Cs*
by (*auto simp*: *id-take-nth-drop*)
show *?case* **using** *2* **by** (*subst* (*1*) *Cs*) *auto*
qed *auto*
next
show *?L* ⊆ *?R* **using** *assms* **by** (*auto simp*: *all-poss-mctxt-conv funas-mctxt-mreplace-at*)
qed

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*)
using *assms* **by** (*induct C Cs rule*: *fill-holes-induct*) (*auto simp*: *comp-def*)

lemma *map-vars-mctxt-map-vars-mctxt*[*simp*]:
shows *map-vars-mctxt f* (*map-vars-mctxt g C*) = *map-vars-mctxt* (*f* ∘ *g*) *C*
by (*induct C*) *auto*

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))*
using *funas-term-fill-holes-iff*[*OF assms*] **by** *auto*

lemma *mctxt-neq-mholeE*:
x ≠ MHole ⟹ (⋀v. x = MVar v ⟹ P) ⟹ (⋀f Cs. x = MFun f Cs ⟹ P) ⟹ P
by (*cases x*) *auto*

lemma *prefix-comp-mctxt*:
C ≤ E ⟹ D ≤ E ⟹ (C, D) ∈ comp-mctxt
proof (*induct E arbitrary*: *C D*)
case (*MFun f Es C D*)
then show *?case*
proof (*elim less-eq-mctxtE2*)
fix *Cs Ds*
assume *C*: *C = MFun f Cs* **and** *D*: *D = MFun f Ds*
and *lC*: *length Cs = length Es* **and** *lD*: *length Ds = length Es*
and *Ci*: *⋀i. i < length Cs ⟹ Cs ! i ≤ Es ! i* **and** *Di*: *⋀i. i < length Ds ⟹ Ds ! i ≤ Es ! i*
and *IH*: *⋀E′ C′ D′. E′ ∈ set Es ⟹ C′ ≤ E′ ⟹ D′ ≤ E′ ⟹ (C′, D′) ∈ comp-mctxt*
show (*C, D*) ∈ *comp-mctxt*
by (*auto simp*: *C D lC lD intro*!: *comp-mctxt.intros IH*[*OF - Ci Di*])
qed (*auto intro*: *comp-mctxt.intros*)
qed (*auto elim*: *less-eq-mctxtE2*(*1*,*2*) *intro*: *comp-mctxt.intros*)

lemma *less-eq-mctxt-sup-conv1*:
(*C, D*) ∈ *comp-mctxt ⟹ C ≤ D ⟷ C ⊔ D = D*
by (*induct C D rule*: *comp-mctxt.induct*) (*auto elim*!: *less-eq-mctxtE2 nth-equalityE intro*: *nth-equalityI less-eq-mctxtI2*(*3*))

lemma *less-eq-mctxt-sup-conv2*:
(*C, D*) ∈ *comp-mctxt ⟹ D ≤ C ⟷ C ⊔ D = C*
using *less-eq-mctxt-sup-conv1*[*OF comp-mctxt-sym*] **by** (*auto simp*: *ac-simps*)

lemma *comp-mctxt-mctxt-of-term1*[*dest*!]:
(*C, mctxt-of-term t*) ∈ *comp-mctxt ⟹ C ≤ mctxt-of-term t*
proof (*induct C mctxt-of-term t arbitrary*: *t rule*: *comp-mctxt.induct*)
case (*MHole2 C t*)
then show *?case* **by** (*cases t, auto*)
next
case (*MFun f g Cs Ds*)
then show *?case* **by** (*cases t, auto intro*: *less-eq-mctxtI2*)
qed *auto*

lemmas *comp-mctxt-mctxt-of-term2*[*dest*!] = *comp-mctxt-mctxt-of-term1*[*OF comp-mctxt-sym*]

lemma *mfun-leq-mfunI*:
f = *g* ⟹ *length Cs* = *length Ds* ⟹ (⋀*i*. *i* < *length Ds* ⟹ *Cs* ! *i* ≤ *Ds* ! *i*)
⟹ *MFun f Cs* ≤ *MFun g Ds*
by (*auto simp*: *less-eq-mctxt-def list-eq-iff-nth-eq*)

lemma *prefix-mctxt-sup*:
assumes *C* ≤ (*E* :: ('*f*, '*v*) *mctxt*) *D* ≤ *E* **shows** *C* ⊔ *D* ≤ *E*
using *assms*
by (*induct E arbitrary*: *C D*) (*auto elim*!: *less-eq-mctxtE2 intro*!: *mfun-leq-mfunI*)

lemma *mreplace-at-leqI*:
p ∈ *all-poss-mctxt C* ⟹ *C* ≤ *E* ⟹ *D* ≤ *subm-at E p* ⟹ *mreplace-at C p D*
≤ *E*
by (*induct C p D arbitrary*: *E rule*: *mreplace-at.induct*)
(*auto elim*!: *less-eq-mctxtE1 intro*!: *less-eq-mctxtI1 simp*: *upd-conv-take-nth-drop*[*symmetric*]
nth-list-update)

lemma *prefix-and-fewer-holes-implies-equal-mctxt*:
C ≤ *D* ⟹ *hole-poss C* ⊆ *hole-poss D* ⟹ *C* = *D*
proof (*induct C D rule*: *less-eq-mctxt-induct*)
case (*1 D*) **then show** *?case* **by** (*cases D*) *auto*
next
case (*3 Cs Ds f*)
have *i* < *length Ds* ⟹ *hole-poss* (*Cs* ! *i*) ⊆ *hole-poss* (*Ds* ! *i*) **for** *i* **using** *3*(*1*,*4*)
by *auto*
then show *?case* **using** *3* **by** (*auto intro*!: *nth-equalityI*)
qed *auto*

lemma *compare-mreplace-at*:
p ∈ *poss-mctxt C* ⟹ *mreplace-at C p D* ≤ *mreplace-at C p E* ⟷ *D* ≤ *E*
proof (*induct C arbitrary*: *p*)
case (*MFun f Cs p*)
then show *?case*
by (*cases p*, *auto elim*!: *less-eq-mctxtE2*(*3*) *intro*!: *less-eq-mctxt-intros*(*3*) *simp*:
nth-append nth-Cons'
split: *if-splits*) *auto*
qed *auto*

lemma *merge-mreplace-at*:
p ∈ *poss-mctxt C* ⟹ *mreplace-at C p* (*D* ⊔ *E*) = *mreplace-at C p D* ⊔ *mre-*
place-at C p E
proof (*induct C arbitrary*: *p*)
case (*MFun f Cs p*)
then show *?case* **by** (*cases p*, *auto intro*: *nth-equalityI*)
qed *auto*

lemma *compare-mreplace-atI'*:

$C \leq D \Longrightarrow C' \leq D' \Longrightarrow p \in$ *all-poss-mctxt* $C \Longrightarrow$ *mreplace-at* $C\ p\ C' \leq$ *mreplace-at* $D\ p\ D'$
proof (*induct C D arbitrary*: *p rule*: *less-eq-mctxt-induct*)
case (*3 cs ds f p*)
then show *?case* **by** (*cases p*, *auto intro*!: *less-eq-mctxt-intros*(*3*) *simp*: *nth-append nth-Cons'*)
qed *auto*

lemma *compare-mreplace-atI*:
$C \leq D \Longrightarrow C' \leq D' \Longrightarrow p \in$ *poss-mctxt* $C \Longrightarrow$ *mreplace-at* $C\ p\ C' \leq$ *mreplace-at* $D\ p\ D'$
using *compare-mreplace-atI'* *all-poss-mctxt-conv* **by** *blast*

lemma *all-poss-mctxt-mono*:
$C \leq D \Longrightarrow$ *all-poss-mctxt* $C \subseteq$ *all-poss-mctxt* D
by (*induct C D rule*: *less-eq-mctxt-induct*) *force*+

lemma *all-poss-mctxt-inf-mctxt*:
$(C, D) \in$ *comp-mctxt* $\Longrightarrow$ *all-poss-mctxt* $(C \sqcap D) =$ *all-poss-mctxt* $C \cap$ *all-poss-mctxt* D
by (*induct C D rule*: *comp-mctxt.induct*) *auto*

lemma *less-eq-subm-at*:
$p \in$ *all-poss-mctxt* $C \Longrightarrow C \leq D \Longrightarrow$ *subm-at* $C\ p \leq$ *subm-at* $D\ p$
by (*induct C arbitrary*: *p D*) (*auto elim*: *less-eq-mctxtE1*)

lemma *inf-subm-at*:
$p \in$ *all-poss-mctxt* $(C \sqcap D) \Longrightarrow$ *subm-at* $(C \sqcap D)\ p =$ *subm-at* $C\ p \sqcap$ *subm-at* $D\ p$
proof (*induct C D arbitrary*: *p rule*: *inf-mctxt.induct*)
case (*4 f Cs g Ds p*) **show** *?case* **using** *4*(*1*) *4*(*2*)
by (*auto 4 4 intro*!: *4*(*1*)[*of* (*Cs* ! *i*, *Ds* ! *i*) *Cs* ! *i Ds* ! *i* **for** *i*] *simp*: *set-zip*)
qed *auto*

lemma *less-eq-fill-holesI*:
assumes *length Ds* = *num-holes C length Es* = *num-holes C*
$\bigwedge i.\ i <$ *num-holes* $C \Longrightarrow Ds\ !\ i \leq Es\ !\ i$
shows *fill-holes-mctxt C Ds* $\leq$ *fill-holes-mctxt C Es*
using *assms*(*1*,*2*)[*symmetric*] *assms*(*3*)
by (*induct C Ds Es rule*: *fill-holes-induct2*) (*auto intro*!: *less-eq-mctxtI1 simp*: *partition-by-nth-nth*)

lemma *less-eq-fill-holesD*:
assumes *length Ds* = *num-holes C length Es* = *num-holes C*
fill-holes-mctxt C Ds $\leq$ *fill-holes-mctxt C Es* $i <$ *num-holes C*
shows *Ds* ! $i \leq$ *Es* ! i
using *assms*(*1*,*2*)[*symmetric*] *assms*(*3*,*4*)
proof (*induct C Ds Es arbitrary*: *i rule*: *fill-holes-induct2*)
case (*MFun f Cs Ds Es*)

obtain *j k* **where** *j* < *length Cs k* < *num-holes* (*Cs* ! *j*)
zip Ds Es ! *i* = *partition-holes* (*zip Ds Es*) *Cs* ! *j* ! *k*
using *nth-concat-split*[*of i partition-holes* (*zip Ds Es*) *Cs*] *MFun*(*1*,*2*,*5*) **by** *auto*
moreover then have *f* (*zip Ds Es* ! *i*) = *partition-holes* (*map f* (*zip Ds Es*)) *Cs* ! *j* ! *k* **for** *f*
using *nth-map*[*of k partition-holes* (*zip Ds Es*) *Cs* ! *j f*] *MFun*(*1*,*2*)
length-partition-by-nth[*of map num-holes Cs zip Ds Es*] **by** *simp*
from *this*[*of fst*] *this*[*of snd*] *map-fst-zip*[*of Ds Es*] *map-snd-zip*[*of Ds Es*]
have *Ds* ! *i* = *partition-holes Ds Cs* ! *j* ! *k Es* ! *i* = *partition-holes Es Cs* ! *j* ! *k*
using *MFun*(*1*,*2*,*5*) **by** *simp-all*
ultimately show *?case* **using** *MFun*(*3*)[*of j k*] *MFun*(*1*,*2*,*4*) **by** (*auto elim*: *less-eq-mctxtE1*)
qed *auto*

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*)
using *assms* **by** (*auto intro*: *less-eq-fill-holesI dest*: *less-eq-fill-holesD*)

lemma *fill-holes-mctxt-suffix*[*simp*]:
assumes *length Ds* = *num-holes C* **shows** *C* ≤ *fill-holes-mctxt C Ds*
using *assms*(*1*)[*symmetric*]
by (*induct C Ds rule*: *fill-holes-induct*) (*auto simp*: *less-eq-mctxt-def intro*: *nth-equalityI*)

lemma *fill-holes-mctxt-id*:
assumes *length Ds* = *num-holes C C* = *fill-holes-mctxt C Ds* **shows** *set Ds* ⊆ {*MHole*}
using *assms*(*1*)[*symmetric*] *assms*(*2*)
apply (*induct C Ds rule*: *fill-holes-induct*)
unfolding *set-concat*
by (*auto simp*: *set-conv-nth*[*of partition-holes* - -] *list-eq-iff-nth-eq*[*of* - *map* - -])

lemma *fill-holes-suffix*[*simp*]:
num-holes C = *length ts* ⟹ *C* ≤ *mctxt-of-term* (*fill-holes C ts*)
by (*induct C ts rule*: *fill-holes-induct*) (*auto intro*: *less-eq-mctxtI1*)

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*
using *assms*

proof (*induct C t rule*: *unfill-holes.induct*)
case (*3 f Cs g ts*) **with** *3(1)*[*OF - nth-mem*] *3(2)* **show** *?case*
by (*auto simp*: *less-eq-mctxt-def length-concat*
intro!: *cong*[*of sum-list*, *OF refl*] *nth-equalityI elim*!: *nth-equalityE*)
qed (*auto simp*: *less-eq-mctxt-def*)

lemma *fill-unfill-holes*:
assumes $C \leq$ *mctxt-of-term t*
shows *fill-holes C* (*unfill-holes C t*) = *t*
using *assms*
proof (*induct C t rule*: *unfill-holes.induct*)
case (*3 f Cs g ts*) **with** *3(1)*[*OF - nth-mem*] *3(2)* **show** *?case*
by (*auto simp*: *less-eq-mctxt-def intro*!: *fill-holes-arbitrary elim*!: *nth-equalityE*)
qed (*auto simp*: *less-eq-mctxt-def split*: *if-splits*)

lemma *unfill-fill-holes*:
assumes *length ts* = *num-holes C*
shows *unfill-holes C* (*fill-holes C ts*) = *ts*
using *assms*[*symmetric*]
proof (*induct C ts rule*: *fill-holes-induct*)
case (*MFun f Cs ts*) **then show** *?case*
by (*auto intro*!: *arg-cong*[*of - - concat*] *nth-equalityI*[*of - partition-holes ts Cs*]
simp del: *concat-partition-by*) *auto*
qed *auto*

lemma *unfill-holes-subt*:
assumes $C \leq$ *mctxt-of-term t* **and** $t' \in$ *set* (*unfill-holes C t*)
shows $t' \trianglelefteq t$
using *assms*
proof (*induct C t rule*: *unfill-holes.induct*)
case (*3 f Cs g ts*)
obtain *i* **where** *i* < *length Cs* **and** $t' \in$ *set* (*unfill-holes* (*Cs* ! *i*) (*ts* ! *i*))
using *3* **by** (*auto dest*!: *in-set-idx split*: *if-splits simp*: *less-eq-mctxt-def*)
then show *?case*
using *3(1)*[*OF - nth-mem*[*of i*]] *3(2,3) supteq.subt*[*of ts* ! *i ts t' g*]
by (*auto simp*: *less-eq-mctxt-def elim*!: *nth-equalityE split*: *if-splits*)
qed (*auto simp*: *less-eq-mctxt-def split*: *if-splits*)

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*
using *assms*
by (*induct C D arbitrary*: *p thesis rule*: *less-eq-mctxt-induct*)
(*auto*, *metis less-eq-pos-simps*(*4*))

lemma *concat-map-zip-upt*: **assumes** $\bigwedge i.\ i < n \Longrightarrow$ *length* (*f i*) = *length* (*g i*)
shows *concat* (*map* (λ*i*. *zip* (*f i*) (*g i*)) [*0*..<*n*]) = *zip* (*concat* (*map f* [*0*..<*n*]))
(*concat* (*map g* [*0*..<*n*]))
using *assms* **by** (*induct n arbitrary*: *f g*) (*auto simp*: *map-upt-Suc simp del*:

upt.simps)

lemma *unfill-holes-by-prefix'*:
assumes *num-holes C = length Ds fill-holes-mctxt C Ds ≤ 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*)))
using *assms*
proof (*induct C Ds arbitrary*: *t rule*: *fill-holes-induct*)
case (*MVar v*) **then show** *?case* **by** (*cases t*) (*auto elim*: *less-eq-mctxtE1*)
next
case (*MFun f Cs Ds*)
have [*simp*]: *length ts = length Cs ⟹ map* (λ*i*. *unfill-holes* (*map* (λ*i*. *fill-holes-mctxt* (*Cs* ! *i*) (*partition-holes Ds Cs* ! *i*)) [*0*..<*length Cs*] ! *i*) (*ts* ! *i*)) [*0*..<*length Cs*]
= *map* (λ*i*. *unfill-holes* (*fill-holes-mctxt* (*Cs* ! *i*) (*partition-holes Ds Cs* ! *i*)) (*ts* ! *i*)) [*0*..<*length Cs*] **for** *ts*
by (*auto intro*: *nth-equalityI*)
obtain *ts* **where** *lts*: *length ts = length Cs t = Fun f ts* **and**
pre: *i < length Cs ⟹ fill-holes-mctxt* (*Cs* ! *i*) (*partition-holes Ds Cs* ! *i*) ≤ *mctxt-of-term* (*ts* ! *i*) **for** *i*
using *MFun*(*1*,*3*) **by** (*cases t*) (*auto elim*!: *less-eq-mctxtE2*)
have ∗: *i* ∈ *set* [*0*..<*n*] ⟹ *i < n* **for** *i n* **by** *auto*
have ∗∗∗: *i < length Cs ⟹ Cs* ! *i* ≤ *mctxt-of-term* (*ts* ! *i*) **for** *i*
using *fill-holes-mctxt-suffix*[*of partition-holes Ds Cs* ! *i Cs* ! *i*, *OF length-partition-holes-nth*] *MFun*(*1*) *pre*[*of i*]
by (*auto simp del*: *fill-holes-mctxt-suffix*)
have [*simp*]: *concat* (*map* (λ*i*. *concat* (*map f* (*zip* (*partition-holes Ds Cs* ! *i*) (*unfill-holes* (*Cs* ! *i*) (*ts* ! *i*))))) [*0*..<*length Cs*]) = *concat* (*map f* (*zip Ds* (*concat* (*map* (λ*i*. *unfill-holes* (*Cs* ! *i*) (*ts* ! *i*)) [*0*..<*length Cs*])))) **for** *f*
unfolding *concat-map-concat*[*of map - -*, *unfolded map-map comp-def*]
unfolding *map-map*[*of map f* λ*i*. *zip* (*-* *i*) (*-* *i*), *symmetric*, *unfolded comp-def*] *map-concat*[*symmetric*]
using *MFun*(*1*) *map-nth*[*of partition-holes Ds Cs*] **by** (*auto simp*: *length-unfill-holes*[*OF* ∗∗∗] *concat-map-zip-upt*)
from *lts pre* **show** *?case* **using** *MFun*(*1*) *map-cong*[*OF refl MFun*(*2*)[*OF* ∗], *of* [*0*..<*length Cs*] *id* λ*i*. *ts* ! *i*]
by (*auto simp del*: *map-eq-conv*)
qed *auto*

lemma *unfill-holes-var-subst*:
C ≤ mctxt-of-term t ⟹ unfill-holes (*map-vars-mctxt f C*) (*t* · (*Var* ∘ *f*)) = *map* (λ*t*. *t* · (*Var* ∘ *f*)) (*unfill-holes C t*)
by (*induct C t rule*: *unfill-holes.induct*; (*simp only*: *mctxt-of-term.simps*; *elim less-eq-mctxtE2*)*?*)
(*auto simp*: *map-concat intro*!: *arg-cong*[*of - - concat*])

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* ≤ *D*
  shows *length* (*unfill-holes-mctxt C D*) = *num-holes C*
  using *assms*
proof (*induct C D rule*: *unfill-holes-mctxt.induct*)
  case (*3 f Cs g Ds*) **with** *3*(*1*)[*OF - nth-mem*] *3*(*2*) **show** *?case*
    by (*auto simp*: *less-eq-mctxt-def length-concat intro*!: *cong*[*of sum-list*, *OF refl*] *nth-equalityI elim*!: *nth-equalityE*)
qed (*auto simp*: *less-eq-mctxt-def*)

lemma *fill-unfill-holes-mctxt*:
  assumes *C* ≤ *D*
  shows *fill-holes-mctxt C* (*unfill-holes-mctxt C D*) = *D*
  using *assms*
proof (*induct C D rule*: *unfill-holes-mctxt.induct*)
  case (*3 f Cs g Ds*) **with** *3*(*1*)[*OF - nth-mem*] *3*(*2*) **show** *?case*
    by (*auto simp*: *less-eq-mctxt-def intro*!: *fill-holes-arbitrary elim*!: *nth-equalityE*)
qed (*auto simp*: *less-eq-mctxt-def split*: *if-splits*)

lemma *unfill-fill-holes-mctxt*:
  assumes *length Ds* = *num-holes C*
  shows *unfill-holes-mctxt C* (*fill-holes-mctxt C Ds*) = *Ds*
  using *assms*[*symmetric*]
proof (*induct C Ds rule*: *fill-holes-induct*)
  case (*MFun f Cs ts*) **then show** *?case*
    by (*auto intro*!: *arg-cong*[*of - - concat*] *nth-equalityI*[*of - partition-holes ts Cs*]
        simp del: *concat-partition-by*) *auto*
qed *auto*

lemma *unfill-holes-mctxt-mctxt-of-term*:
  assumes *C* ≤ *mctxt-of-term t*
  shows *unfill-holes-mctxt C* (*mctxt-of-term t*) = *map mctxt-of-term* (*unfill-holes C t*)
  using *assms*
proof (*induct C arbitrary*: *t*)
  case (*MVar x*) **then show** *?case* **by** (*cases t*) (*auto elim*: *less-eq-mctxtE1*)
next
  case *MHole* **then show** *?case* **by** (*cases t*) (*auto elim*: *less-eq-mctxtE1*)
next
  case (*MFun x1a x2*) **then show** *?case*

by (*cases t*) (*auto elim*: *less-eq-mctxtE1 simp*: *map-concat intro*!: *arg-cong*[*of* - - *concat*])
qed

6.1.8 Function symbols of prefixes

lemma *funas-prefix*[*simp*]:
C ≤ *D* ⟹ *fn* ∈ *funas-mctxt C* ⟹ *fn* ∈ *funas-mctxt D*
unfolding *less-eq-mctxt-def*
proof (*induct C D rule*: *inf-mctxt.induct*)
case (*4 f Cs g Ds*)
from *4*(*3*) **obtain** *i* **where** *i* < *length Cs* ∧ *fn* ∈ *funas-mctxt* (*Cs* ! *i*) ∨ *fn* = (*f*, *length Cs*)
by (*auto dest*!: *in-set-idx*)
moreover {
assume *i* < *length Cs* ∧ *fn* ∈ *funas-mctxt* (*Cs* ! *i*)
then have *i* < *length Ds* ∧ *fn* ∈ *funas-mctxt* (*Ds* ! *i*) **using** *4*(*2*)
by (*auto intro*!: *4*(*1*)[*of* - *Cs* ! *i Ds* ! *i*] *split*: *if-splits elim*!: *nth-equalityE simp*: *in-set-conv-nth*)
then have *?case* **by** (*auto*)
}
ultimately show *?case* **using** *4*(*2*) **by** *auto*
qed *auto*

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* ⟹ (*s* · σ,*t* · σ) ∈ *par-rstep R*
| *par-step-fun*[*intro*]: ⟦⋀ *i*. *i* < *length ts* ⟹ (*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*
by (*induct t*, *auto*)

lemma *all-ctxt-closed-par-rstep*[*intro*]: *all-ctxt-closed F* (*par-rstep R*)
unfolding *all-ctxt-closed-def*
by *auto*

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* $\frown\frown$ *n* $\Longrightarrow$
(*Fun f xs*, *Fun f ys*) $\in$ *par-rstep R* $\frown\frown$ *n*
proof(*induct n arbitrary*:*ys*)
case *0*
then have $\forall$ *i*<*length xs*. (*xs* ! *i* = *ys* ! *i*) **by** *simp*
with *0* **show** *?case* **using** *relpow-0-I list-eq-iff-nth-eq* **by** *metis*
next
case (*Suc n*)
let *?c* = λ *z i*. (*xs* ! *i*, *z*) $\in$ *par-rstep R* $\frown\frown$ *n* $\wedge$ (*z*, *ys* ! *i*) $\in$ *par-rstep R*
{ fix *i* **assume** *i* < *length xs*
from *relpow-Suc-E*[*OF Suc*(*3*)[*rule-format*, *OF this*]]
have $\exists$ *z*. (*?c z i*) **by** *metis*
}
with *choice* **have** $\exists$ *zf*. $\forall$ *i* < *length xs*. (*?c* (*zf i*) *i*) **by** *meson*
then obtain *zf* **where** *a*:$\forall$ *i* < *length xs*. (*?c* (*zf i*) *i*) **by** *auto*
let *?zs* = *map zf* [*0*..<*length xs*]
have *len*:*length xs* = *length ?zs* **by** *simp*
from *a map-nth* **have** $\forall$ *i*<*length xs*. (*xs* ! *i*, *?zs* ! *i*) $\in$ *par-rstep R* $\frown\frown$ *n* **by** *auto*
from *Suc*(*1*)[*OF len this*] **have** *n*:(*Fun f xs*, *Fun f ?zs*) $\in$ *par-rstep R* $\frown\frown$ *n* **by** *auto*
from *a map-nth* **have** $\forall$ *i*<*length xs*. (*?zs* ! *i*, *ys* ! *i*) $\in$ *par-rstep R* **by** *auto*
with *par-step-fun len Suc*(*2*) **have** (*Fun f ?zs*, *Fun f ys*) $\in$ *par-rstep R* **by** *auto*
with *n* **show** *?case* **by** *auto*
qed

lemma *ctxt-closed-par-rstep*[*intro*]: *ctxt.closed* (*par-rstep R*)
proof (*rule one-imp-ctxt-closed*)
fix *f bef s t aft*
assume *st*: (*s*,*t*) $\in$ *par-rstep R*
let *?ss* = *bef* @ *s* # *aft*
let *?ts* = *bef* @ *t* # *aft*
show (*Fun f ?ss*, *Fun f ?ts*) $\in$ *par-rstep R*
proof (*rule par-step-fun*)
fix *i*
assume *i* < *length ?ts*
show (*?ss* ! *i*, *?ts* ! *i*) $\in$ *par-rstep R*
using *par-rstep-refl*[*of ?ts* ! *i R*] *st* **by** (*cases i* = *length bef*, *auto simp*: *nth-append*)
qed *simp*
qed

lemma *subst-closed-par-rstep*: (*s*,*t*) $\in$ *par-rstep R* $\Longrightarrow$ (*s* $\cdot$ σ, *t* $\cdot$ σ) $\in$ *par-rstep R*
proof (*induct rule*: *par-rstep.induct*)
case (*root-step s t* τ)
show *?case*
using *par-rstep.root-step*[*OF root-step*, *of* τ $\circ_s$ σ] **by** *auto*
next

```
  case (par-step-var x)
  show ?case by auto
next
  case (par-step-fun ss ts f)
  show ?case unfolding eval-term.simps
    by (rule par-rstep.par-step-fun, insert par-step-fun(2−3), auto)
qed

lemma R-par-rstep: R ⊆ par-rstep R
  using root-step[of - - R Var] by auto

lemma par-rstep-rsteps: par-rstep R ⊆ (rstep R)*
proof
  fix s t
  assume (s,t) ∈ par-rstep R
  then show (s,t) ∈ (rstep R)*
  proof (induct rule: par-rstep.induct)
    case (root-step s t sigma)
    then show ?case by auto
  next
    case (par-step-var x)
    then show ?case by auto
  next
    case (par-step-fun ts ss f)
      from all-ctxt-closedD[of UNIV, OF all-ctxt-closed-rsteps - par-step-fun(3)
par-step-fun(2)]
    show ?case unfolding par-step-fun(3) by simp
  qed
qed

lemma rstep-par-rstep: rstep R ⊆ par-rstep R
  by (rule rstep-subset[OF ctxt-closed-par-rstep subst.closedI R-par-rstep],
      insert subst-closed-par-rstep, auto)

lemma par-rsteps-rsteps: (par-rstep R)* = (rstep R)* (is ?P = ?R)
proof
  from rtrancl-mono[OF par-rstep-rsteps[of R]] show ?P ⊆ ?R by simp
  from rtrancl-mono[OF rstep-par-rstep] show ?R ⊆ ?P .
qed

lemma par-rsteps-union: (par-rstep A ∪ par-rstep B)* =
    (rstep (A ∪ B))*
proof
  show (par-rstep A ∪ par-rstep B)* ⊆ (rstep (A ∪ B))*
    by (metis par-rsteps-rsteps rstep-union rtrancl-Un-rtrancl set-eq-subset)
  show (rstep (A ∪ B))* ⊆ (par-rstep A ∪ par-rstep B)* unfolding rstep-union
    by (meson rstep-par-rstep rtrancl-mono sup-mono)
qed
```

lemma *par-rstep-inverse*: *par-rstep* $(R\hat{}-1) = (\textit{par-rstep}\ R)\hat{}-1$
proof −
  {
    fix *s t* :: $('a,'b)$*term* **and** *R*
    assume $(s,t) \in \textit{par-rstep}\ (R\hat{}-1)$
    hence $(t,s) \in \textit{par-rstep}\ R$
      by (*induct s t*, *auto*)
  }
  from *this*[*of* - - *R*] *this*[*of* - - $R\hat{}-1$]
  show *?thesis* **by** *auto*
qed

lemma *par-rstep-conversion*: $(\textit{rstep}\ R)^{\leftrightarrow *} = (\textit{par-rstep}\ R)^{\leftrightarrow *}$
  unfolding *conversion-def*
  by (*metis par-rsteps-rsteps rtrancl-Un-rtrancl rtrancl-converse*)

lemma *par-rstep-mono*: **assumes** $R \subseteq S$
  shows *par-rstep* $R \subseteq$ *par-rstep* S
proof
  fix *s t*
  show $(s, t) \in \textit{par-rstep}\ R \Longrightarrow (s, t) \in \textit{par-rstep}\ S$
    by (*induct s t rule*: *par-rstep.induct*, *insert assms*, *auto*)
qed

lemma *wf-trs-par-rstep*: **assumes** *wf*: $\bigwedge l\ r.\ (l,r) \in R \Longrightarrow \textit{is-Fun}\ l$
  and *step*: $(\textit{Var}\ x,\ t) \in \textit{par-rstep}\ R$
shows $t = \textit{Var}\ x$
  using *step*
proof (*cases rule*: *par-rstep.cases*)
  case (*root-step l r* σ)
  from *root-step*(*1*) *wf*[*OF root-step*(*3*)] **show** *?thesis* **by** (*cases l*, *auto*)
qed *auto*

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*: $(l \cdot \sigma,\ t) \in \textit{par-rstep}\ R$
shows $(\exists\ \tau.\ t = l \cdot \tau \wedge (\forall\ x \in \textit{vars-term}\ l.\ (\sigma\ x,\ \tau\ x) \in \textit{par-rstep}\ R) \vee$
$(\exists\ C\ l''\ l'\ r'.\ l = C\langle l''\rangle \wedge \textit{is-Fun}\ l'' \wedge (l',r') \in R \wedge (l'' \cdot \sigma = l' \cdot \tau) \wedge$
$((C\ \cdot_c\ \sigma)\ \langle\ r' \cdot \tau\ \rangle,\ t) \in \textit{par-rstep}\ R))$
  using *lin step*
proof (*induction l arbitrary*: *t*)
  case (*Var x t*)
  let *?tau* = $\lambda\ y.\ t$
  show *?case*
    by (*rule exI*[*of* - *?tau*], *rule disjI1*, *insert Var*(*2*), *auto*)
next
  case (*Fun f ss*)
  let *?ss* = *map* $(\lambda\ s.\ s \cdot \sigma)$ *ss*

let *?R = par-rstep R*
from *Fun(3)*
show *?case*
proof (*cases rule*: *par-rstep.cases*)
case (*root-step l r τ*)
show *?thesis*
proof (*rule exI*, *rule disjI2*, *intro exI conjI*)
show (*l*,*r*) ∈ *R* **by** (*rule root-step(3)*)
show *Fun f ss* = □⟨*Fun f ss*⟩ **by** *simp*
show (*Fun f ss*) · *σ* = *l* · *τ* **by** (*rule root-step(1)*)
show ((□ ·$_c$ *σ*)⟨ *r* · *τ* ⟩, *t*) ∈ *?R* **unfolding** *root-step(2)* **using** *par-rstep-refl*
by *simp*
qed *simp*
next
case (*par-step-var x*)
then show *?thesis* **by** *simp*
next
case (*par-step-fun ts ss1 g*)
then have *id*: *ss1* = *?ss g* = *f* **and** *len*: *length ts* = *length ss* **by** *auto*
let *?p1* = λ *τ i*. *ts* ! *i* = *ss* ! *i* · *τ* ∧ (∀ *x* ∈ *vars-term* (*ss* ! *i*). (*σ x*, *τ x*) ∈ *?R*)
let *?p2* = λ *τ i*. (∃ *C l'' l' r'*. *ss* ! *i* = *C*⟨ *l''*⟩ ∧ *is-Fun l''* ∧ (*l'*,*r'*) ∈ *R* ∧ *l''* ·
σ = *l'* · *τ* ∧ ((*C* ·$_c$ *σ*)⟨ *r'* · *τ* ⟩, (*ts* ! *i*)) ∈ *?R*)
let *?p* = λ *τ i*. *?p1 τ i* ∨ *?p2 τ i*
{
fix *i*
assume *i*: *i* < *length ss*
with *par-step-fun(4) id* **have** *i2*: *i* < *length ts* **by** *auto*
from *par-step-fun(3)*[*OF i2*] **have** *step*: (*ss* ! *i* · *σ*, *ts* ! *i*) ∈ *par-rstep R*
unfolding *id nth-map*[*OF i*] **.**
from *i* **have** *mem*: *ss* ! *i* ∈ *set ss* **by** *auto*
from *Fun.prems(1) mem* **have** *linear-term* (*ss* ! *i*) **by** *auto*
from *Fun.IH*[*OF mem this step*] **have** ∃ *τ*. *?p τ i* **.**
}
then have ∀ *i*. ∃ *tau*. *i* < *length ss* ⟶ *?p tau i* **by** *blast*
from *choice*[*OF this*] **obtain** *taus* **where** *taus*: ⋀ *i*. *i* < *length ss* ⟹ *?p* (*taus*
i) *i* **by** *blast*
show *?thesis*
proof (*cases* ∃ *i*. *i* < *length ss* ∧ *?p2* (*taus i*) *i*)
case *True*
then obtain *i* **where** *i*: *i* < *length ss* **and** *p2*: *?p2* (*taus i*) *i* **by** *blast*+
from *par-step-fun(2)*[*unfolded id*] **have** *t*: *t* = *Fun f ts* **.**
from *i* **have** *i'*: *i* < *length ts* **unfolding** *len* **.**
from *p2* **obtain** *C l'' l' r'* **where** *ssi*: *ss* ! *i* = *C* ⟨*l''*⟩ **and** *is-Fun l''* (*l'*,*r'*)
∈ *R l'' · σ* = *l' · taus i*
and *tsi*: ((*C* ·$_c$ *σ*) ⟨ *r'* · *taus i* ⟩, *ts* ! *i*) ∈ *?R* **by** *blast*
from *id-take-nth-drop*[*OF i*, *unfolded ssi*] **obtain** *bef aft* **where** *ss*: *ss* = *bef*
@ *C* ⟨ *l''* ⟩ # *aft*
and *bef*: *bef* = *take i ss*
and *aft*: *aft* = *drop* (*Suc i*) *ss* **by** *blast*

```
      let ?C = More f bef C aft
      let ?r = (C ·c σ) ⟨ r' · taus i ⟩
      let ?sig = map (λ s. s · σ)
      let ?bra = ?sig bef @ ?r # ?sig aft
      have C: (?C ·c σ) ⟨ r' · taus i ⟩ = Fun f ?bra by simp
      show ?thesis unfolding ss
      proof (rule exI[of - taus i], rule disjI2, rule exI[of - ?C], intro exI conjI)
        show is-Fun l'' by fact
        show (l',r') ∈ R by fact
        show l'' · σ = l' · taus i by fact
        show ((?C ·c σ) ⟨ r' · taus i ⟩, t) ∈ ?R unfolding C t
        proof (rule par-rstep.par-step-fun)
          show length ?bra = length ts
            unfolding len unfolding ss by simp
        next
          fix j
          assume j: j < length ts
          show (?bra ! j, ts ! j) ∈ ?R
          proof (cases j = i)
            case True
            then have ?bra ! j = ?r using bef i by (simp add: nth-append)
            then show ?thesis using tsi True by simp
          next
            case False
            from bef i have min (length ss) i = i by simp
             then have ?bra ! j = (?sig bef @ (C ⟨ l'' ⟩ · σ) # ?sig aft) ! j using
False bef i by (simp add: nth-append)
            also have ... = ?sig ss ! j unfolding ss by simp
            also have ... = ss1 ! j unfolding id ..
            finally show ?thesis
              using par-step-fun(3)[OF j] by auto
          qed
        qed
      qed simp
    next
      case False
      with taus have taus: ⋀ i. i < length ss ⟹ ?p1 (taus i) i by blast
      from Fun(2) have is-partition (map vars-term ss) by simp
      from subst-merge[OF this, of taus] obtain τ where tau: ⋀ i x. i < length ss
⟹ x ∈ vars-term (ss ! i) ⟹ τ x = taus i x by auto
      let ?tau = τ
      {
        fix i
        assume i: i < length ss
        then have mem: ss ! i ∈ set ss by auto
        from taus[OF i] have p1: ?p1 (taus i) i .
        have id: ss ! i · (taus i) = ss ! i · τ
          by (rule term-subst-eq, rule tau[OF i, symmetric])
        have ?p1 ?tau i
```

```
      proof (rule conjI[OF - ballI])
        fix x
        assume x: x ∈ vars-term (ss ! i)
        with p1 have step: (σ x, taus i x) ∈ par-rstep R by auto
        with tau[OF i x]
        show (σ x, ?tau x) ∈ par-rstep R by simp
      qed (insert p1[unfolded id], auto)
    } note p1 = this
    have p1: ⋀ i. i < length ss ⟹ ?p1 τ i by (rule p1)
    let ?ss = map (λ s. s · τ) ss
    show ?thesis unfolding par-step-fun(2) id
    proof (rule exI[of - τ], rule disjI1, rule conjI[OF - ballI])
      have ts = map (λ i. ts ! i) [0 ..< (length ts)] by (rule map-nth[symmetric])
      also have ... = map (λ i. ?ss ! i) [0 ..< length ?ss] unfolding len using
p1 by auto
      also have ... = ?ss by (rule map-nth)
      finally have ts: ts = ?ss .
      show Fun f ts = Fun f ss · τ unfolding ts by auto
    next
      fix x
      assume x ∈ vars-term (Fun f ss)
      then obtain s where s: s ∈ set ss and x: x ∈ vars-term s by auto
      from s[unfolded set-conv-nth] obtain i where i: i < length ss and s: s =
ss ! i by auto
      from p1[OF i] x[unfolded s]
      show (σ x, τ x) ∈ par-rstep R by blast
    qed
  qed
 qed
qed

lemma par-rstep-id:
 (s, t) ∈ R ⟹ (s, t) ∈ par-rstep R
 using par-rstep.root-step [of s t R Var] by simp
```

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*
using *assms* **unfolding** *par-cond-def*
by (*metis rrstepE rrstepI singletonD*)

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 −
from *assms* **have** *eq*: *s* = *par-left* (*infos* ! *i*) *t* = *par-right* (*infos* ! *i*) **and** *pc*: *par-cond R* (*infos* ! *i*)
by *auto*
show *?thesis* **unfolding** *eq* **using** *par-cond-imp-rrstep*[*OF pc*] **.**
qed

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*
unfolding *par-rstep-mctxt-def* **using** *assms* **by** *auto*

lemma *par-rstep-mctxt-reflI*: (*s*,*s*) ∈ *par-rstep-mctxt R* (*mctxt-of-term s*) []
by (*intro par-rstep-mctxtI*, *auto*)

lemma *par-rstep-mctxt-varI*: (*Var x*, *Var x*) ∈ *par-rstep-mctxt R* (*MVar x*) []
by (*intro par-rstep-mctxtI*, *auto*)

lemma *par-rstep-mctxt-MHoleI*: (*l*,*r*) ∈ *R* ⟹ *s* = *l* · σ ⟹ *t* = *r* · σ ⟹ *infos* = [*Par-Info s t* (*l*,*r*)]
⟹ (*s*,*t*) ∈ *par-rstep-mctxt R MHole infos*
by (*intro par-rstep-mctxtI*, *auto simp*: *par-cond-def*)

lemma *par-rstep-mctxt-funI*:
assumes *rec*: ⋀ *i*. *i* < *length ts* ⟹ (*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*)
unfolding *par-rstep-mctxt-def*
proof (*standard*, *unfold split*, *intro conjI*)
{
fix *i*

assume $i < length\ ts$
from *rec*[*OF this*, *unfolded par-rstep-mctxt-def*]
have $ss\ !\ i =_f (Cs\ !\ i, par\text{-}lefts\ (infos\ !\ i))\ ts\ !\ i =_f (Cs\ !\ i, par\text{-}rights\ (infos\ !\ i))$
par-conds R (*infos* ! *i*) **by** *auto*
} note $* = this$
from $*(3)$[*folded len*(*3*)] **show** *par-conds R* (*concat infos*)
by (*metis in-set-conv-nth nth-concat-split*)
show *Fun f ss* $=_f$ (*MFun f Cs*, *par-lefts* (*concat infos*)) **unfolding** *map-concat*
by (*intro eqf-MFunI*, *insert* $*(1)$ *len*, *auto*)
show *Fun f ts* $=_f$ (*MFun f Cs*, *par-rights* (*concat infos*)) **unfolding** *map-concat*
by (*intro eqf-MFunI*, *insert* $*(2)$ *len*, *auto*)
qed

lemma *par-rstep-mctxt-funI-ex*:
assumes $\bigwedge i.\ i < length\ ts \Longrightarrow \exists\ C\ infos.\ (ss\ !\ i, ts\ !\ i) \in par\text{-}rstep\text{-}mctxt\ R\ C\ infos$
and *length ss* = *length ts*
shows $\exists\ C\ infos.\ (Fun\ f\ ss, Fun\ f\ ts) \in par\text{-}rstep\text{-}mctxt\ R\ C\ infos \wedge C \neq MHole$

proof −
let *?n* = *length ts*
from *assms*(*1*) **have** $\forall\ i.\ \exists\ C\ infos.\ i < ?n \longrightarrow (ss\ !\ i, ts\ !\ i) \in par\text{-}rstep\text{-}mctxt\ R\ C\ infos$ **by** *auto*
from *choice*[*OF this*] **obtain** *C* **where** $\forall\ i.\ \exists\ infos.\ i < ?n \longrightarrow (ss\ !\ i, ts\ !\ i) \in par\text{-}rstep\text{-}mctxt\ R\ (C\ i)\ infos$ **by** *auto*
from *choice*[*OF this*] **obtain** *infos* **where** *steps*: $\bigwedge i.\ i < ?n \Longrightarrow (ss\ !\ i, ts\ !\ i) \in par\text{-}rstep\text{-}mctxt\ R\ (C\ i)\ (infos\ i)$ **by** *auto*
let *?Cs* = *map C* [*0* ..< *?n*]
let *?Is* = *map infos* [*0* ..< *?n*]
show *?thesis*
proof (*intro exI conjI*, *rule par-rstep-mctxt-funI*)
show *length ?Cs* = *?n* **by** *simp*
show *length ?Is* = *?n* **by** *simp*
qed (*insert assms*(*2*) *steps*, *auto*)
qed

Parallel rewriting is closed under multihole-contexts.

lemma *par-rstep-mctxt*:
assumes $s =_f (C, ss)$ **and** $t =_f (C, ts)$
and $\forall i{<}length\ ss.\ (ss\ !\ i, ts\ !\ i) \in par\text{-}rstep\ R$
shows $(s, t) \in par\text{-}rstep\ R$
proof −
have [*simp*]: *length ss* = *length ts* **using** *assms* **by** (*auto dest*!: *eqfE*)
have [*simp*]: *t* = *fill-holes C ts* **using** *assms* **by** (*auto dest*: *eqfE*)
have (*s*, *fill-holes C ts*) ∈ *par-rstep R*
using *assms* **by** (*intro eqf-all-ctxt-closed-step* [*of UNIV - s C ss*, *THEN conjunct1*]) *auto*
then show *?thesis* **by** *simp*

qed

lemma *par-rstep-mctxt-rrstepI* :
assumes $s =_f (C, ss)$ **and** $t =_f (C, ts)$
and $\forall i{<}length\ ss.\ (ss\ !\ i, ts\ !\ i) \in rrstep\ R$
shows $(s, t) \in par\text{-}rstep\ R$
by (*meson assms contra-subsetD par-rstep-mctxt rrstep-imp-rstep rstep-par-rstep*)

lemma *par-rstep-mctxtD*:
assumes $(s, t) \in par\text{-}rstep\ R$
shows $\exists C\ ss\ ts.\ s =_f (C, ss) \wedge t =_f (C, ts) \wedge (\forall i{<}length\ ss.\ (ss\ !\ i, ts\ !\ i) \in rrstep\ R)$
(**is** $\exists C\ ss\ ts.\ ?P\ s\ t\ C\ ss\ ts$)
using *assms*
proof (*induct*)
case (*root-step s t σ*)
then have $(s \cdot \sigma, t \cdot \sigma) \in rrstep\ R$ **by** *auto*
moreover have $s \cdot \sigma =_f (MHole, [s \cdot \sigma])$ **and** $t \cdot \sigma =_f (MHole, [t \cdot \sigma])$ **by** *auto*
ultimately show *?case* **by** *force*
next
case (*par-step-var x*)
have $Var\ x =_f (MVar\ x, [])$ **by** *auto*
then show *?case* **by** *force*
next
case (*par-step-fun ts ss f*)
then have $\forall i{<}length\ ts.\ \exists x.\ ?P\ (ss\ !\ i)\ (ts\ !\ i)\ (fst\ x)\ (fst\ (snd\ x))\ (snd\ (snd\ x))$ **by** *force*
then obtain *g* **where** $\forall i{<}length\ ts.\ ?P\ (ss\ !\ i)\ (ts\ !\ i)\ (fst\ (g\ i))\ (fst\ (snd\ (g\ i)))\ (snd\ (snd\ (g\ i)))$
unfolding *choice-iff'* **by** *blast*
moreover
define *Cs us vs* **where** $Cs = map\ (\lambda i.\ fst\ (g\ i))\ [0\ ..{<}\ length\ ts]$
and $us = map\ (\lambda i.\ fst\ (snd\ (g\ i)))\ [0\ ..{<}\ length\ ts]$
and $vs = map\ (\lambda i.\ snd\ (snd\ (g\ i)))\ [0\ ..{<}\ length\ ts]$
ultimately have [*simp*]: $length\ Cs = length\ ts$
$length\ us = length\ ts\ length\ vs = length\ ts$
and $*$: $\forall i{<}length\ us.\ ss\ !\ i =_f (Cs\ !\ i, us\ !\ i) \wedge ts\ !\ i =_f (Cs\ !\ i, vs\ !\ i) \wedge (\forall j{<}length\ (us\ !\ i).\ (us\ !\ i\ !\ j, vs\ !\ i\ !\ j) \in rrstep\ R)$
by *simp-all*
define *C* **where** $C = MFun\ f\ Cs$
have $Fun\ f\ ss =_f (C, concat\ us)$ **and** $Fun\ f\ ts =_f (C, concat\ vs)$
using $*$ **by** (*auto simp*: *C-def* ‹*length ss = length ts*› *intro*: *eqf-MFunI*)
moreover have $\forall i{<}length\ (concat\ us).\ (concat\ us\ !\ i, concat\ vs\ !\ i) \in rrstep\ R$
using $*$ **by** (*intro concat-all-nth*) (*auto dest*!: *eqfE*)
ultimately show *?case* **by** *blast*
qed

lemma *par-rstep-mctxt-mono*: **assumes** $R \subseteq S$

shows *par-rstep-mctxt R C infos* ⊆ *par-rstep-mctxt S C infos*
using *assms* **unfolding** *par-rstep-mctxt-def par-cond-def* **by** *auto*

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 −
have ∃ *C infos*. *s* =$_f$ (*C*, *par-lefts infos*) ∧ *t* =$_f$ (*C*, *par-rights infos*) ∧ *par-conds R infos* (**is** ∃ *C infos*. *?P s t C infos*)
using *assms*
proof (*induct*)
case (*root-step s t σ*)
thus *?case* **by** (*intro exI*[*of* - *MHole*] *exI*[*of* - [*Par-Info* (*s* · *σ*) (*t* · *σ*) (*s*,*t*)]], *auto simp*: *par-cond-def*)
next
case (*par-step-var x*)
show *?case* **by** (*intro exI*[*of* - *MVar x*] *exI*[*of* - *Nil*], *auto*)
next
case (*par-step-fun ts ss f*)
have ∃ *C infos*. (*Fun f ss*, *Fun f ts*) ∈ *par-rstep-mctxt R C infos* ∧ *C* ≠ *MHole*
by (*intro par-rstep-mctxt-funI-ex*, *insert par-step-fun*, *auto simp*: *par-rstep-mctxt-def*)
then obtain *C infos* **where** (*Fun f ss*, *Fun f ts*) ∈ *par-rstep-mctxt R C infos* **by** *auto*
hence *?P* (*Fun f ss*) (*Fun f ts*) *C infos*
by (*auto simp*: *par-rstep-mctxt-def*)
thus *?case* **by** *blast*
qed
with *that* **show** *?thesis* **by** *blast*
qed

lemma *par-rstep-par-rstep-mctxt-conv*:
(*s*, *t*) ∈ *par-rstep R* ⟷ (∃ *C infos*. (*s*, *t*) ∈ *par-rstep-mctxt R C infos*)
proof
assume (*s*, *t*) ∈ *par-rstep R*
from *par-rstep-mctxtE*[*OF this*] **obtain** *C infos*
where *s* =$_f$ (*C*, *par-lefts infos*) **and** *t* =$_f$ (*C*, *par-rights infos*) **and** *par-conds R infos*
by *metis*
then show ∃ *C infos*. (*s*, *t*) ∈ *par-rstep-mctxt R C infos* **by** (*auto simp*: *par-rstep-mctxt-def*)
next
assume ∃ *C infos*. (*s*, *t*) ∈ *par-rstep-mctxt R C infos*
then show (*s*, *t*) ∈ *par-rstep R*
by (*force simp*: *par-rstep-mctxt-def par-cond-def rrstep-def' set-conv-nth intro*!: *par-rstep-mctxt-rrstepI*)
qed

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*)
unfolding *par-cond-def*
by (*cases info*; *force simp*: *subst.closedD subst-closed-rrstep*)+

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*)
using *assms* **unfolding** *par-rstep-mctxt-def* **by** (*auto simp*: *o-def dest*!: *subst-apply-mctxt-sound*[*of - C - σ*])

lemma *par-rstep-mctxt-MVarE*:
assumes (*s*,*t*) ∈ *par-rstep-mctxt R* (*MVar x*) *infos*
shows *s* = *Var x t* = *Var x infos* = []
using *assms*[*unfolded par-rstep-mctxt-def*]
by (*auto dest*: *eqf-MVarE*)

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 −
from *assms*[*unfolded par-rstep-mctxt-def*, *simplified*]
have *s* =*f* (*MHole*, *par-lefts infos*) *t* =*f* (*MHole*, *par-rights infos*) **and** *par-conds R infos* **by** *auto*
from *eqf-MHoleE*[*OF this*(*1*)] *eqf-MHoleE*[*OF this*(*2*)] *this*(*3*)
obtain *info* **where** ∗: *infos* = [*info*] *s* = *par-left info t* = *par-right info par-cond R info*
by (*cases infos*, *auto*)
from *par-cond-imp-rrstep*[*OF* ∗(*4*)] ∗
have (*s*,*t*) ∈ *rrstep R* **by** *auto*
with ∗ **have** ∃ *info*. *par-left info* = *s* ∧ *par-right info* = *t* ∧ *infos* = [*info*] ∧ (*s*, *t*) ∈ *rrstep R* ∧
par-cond R info **by** *auto*
thus (⋀*info*.
par-left info = *s* ⟹
par-right info = *t* ⟹
infos = [*info*] ⟹
(*s*, *t*) ∈ *rrstep R* ⟹ *par-cond R info* ⟹ *thesis*) ⟹
thesis **by** *blast*

qed

lemma *par-rstep-mctxt-MFunD*:
assumes $(s,t) \in$ *par-rstep-mctxt R* (*MFun f Cs*) *infos*
shows $\exists$ *ss ts Infos*.
$s =$ *Fun f ss* $\wedge$
$t =$ *Fun f ts* $\wedge$
length ss = *length Cs* $\wedge$
length ts = *length Cs* $\wedge$
length Infos = *length Cs* $\wedge$
infos = *concat Infos* $\wedge$
$(\forall\ i <$ *length Cs*. (*ss* ! *i*, *ts* ! *i*) $\in$ *par-rstep-mctxt R* (*Cs* ! *i*) (*Infos* ! *i*))
proof −
from *assms*[*unfolded par-rstep-mctxt-def*]
have *eq*: $s =_f$ (*MFun f Cs*, *par-lefts infos*) $t =_f$ (*MFun f Cs*, *par-rights infos*)
and *pc*: *par-conds R infos*
by *auto*
define *Infos* **where** *Infos* = *partition-holes infos Cs*
let *?sss* = *map par-lefts Infos*
let *?tss* = *map par-rights Infos*
let *?n* = *length Cs*
let *?is* = [*0*..<*?n*]
from *eqfE*[*OF eq*(*1*)]
have *s*: *s* = *Fun f* (*map* (λ*i*. *fill-holes* (*Cs* ! *i*) (*?sss* ! *i*)) *?is*)
and *num*: *num-holes* (*MFun f Cs*) = *length infos*
and *len*: *length Infos* = *?n*
and *infos*: *infos* = *concat Infos*
and *lens*: $\bigwedge$ *i*. *i* < *?n* $\Longrightarrow$ *num-holes* (*Cs* ! *i*) = *length* (*Infos* ! *i*)
by (*auto simp*: *Infos-def*)
note *pc* = *pc*[*unfolded infos set-concat*]
from *eqfE*[*OF eq*(*2*)] *num*
have *t*: *t* = *Fun f* (*map* (λ*i*. *fill-holes* (*Cs* ! *i*) (*?tss* ! *i*)) *?is*)
by (*auto simp*: *Infos-def*)
show *?thesis*
apply (*intro exI*[*of - Infos*] *exI conjI infos len allI impI*)
apply (*rule s*)
apply (*rule t*)
apply *force*
apply *force*
apply (*intro par-rstep-mctxtI*, *insert lens len pc*, *auto*)
done
qed

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 = {}
  by (induct t C rule: vars-below-hole.induct, auto simp: set-zip, blast)

lemma vars-below-hole-mctxt-of-term[simp]: vars-below-hole t (mctxt-of-term u) =
{}
  by (rule vars-below-hole-no-hole, auto)

lemma vars-below-hole-vars-term: vars-below-hole t C ⊆ vars-term t
  by (induct t C rule: vars-below-hole.induct; force simp: set-zip set-conv-nth)

lemma vars-below-hole-subst[simp]: vars-below-hole t (C ·mc σ) = vars-below-hole
t C
  by (induct t C rule: vars-below-hole.induct; fastforce simp: set-zip)

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}
  using assms by (auto simp: set-zip)

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 (induct t D rule: vars-below-hole.induct)
  case (1 t)
  then show ?case by (auto simp: vars-term-subst)
next
  case (3 f ts g Cs)
  then show ?case by (fastforce simp: set-zip)
next
  case (4 x f Cs)
  hence hp: hole-poss (MFun f Cs) = {} by auto
  show ?case unfolding vars-below-hole-no-hole[OF hp] by auto
qed auto


lemma vars-below-hole-eqf: assumes t =_f (C, ts)
  shows vars-below-hole t C = ⋃ (vars-term ‘ set ts)
  using assms
proof (induct C arbitrary: t ts)
  case (MVar x)
  from eqf-MVarE[OF MVar(1)]
  show ?case by auto
next
  case MHole
  from eqf-MHoleE[OF MHole(1)]
```

```
  show ?case by auto
next
  case (MFun f Cs t ss)
  from eqf-MFunE[OF MFun(2)] obtain ts sss where
    *: t = Fun f ts length ts = length Cs length sss = length Cs
    ⋀ i. i < length Cs ⟹ ts ! i =_f (Cs ! i, sss ! i)
    ss = concat sss by blast
  {
    fix i
    assume i: i < length Cs
    hence mem: Cs ! i ∈ set Cs by auto
    from MFun(1)[OF mem *(4)[OF i]]
    have vars-below-hole (ts ! i) (Cs ! i) = ⋃ (vars-term ` set (sss ! i)) .
  } note IH = this
  show ?case unfolding *(1) *(5) set-concat set-conv-nth[of sss] using IH *(2,3)
    by (auto simp: set-zip)
qed
```

```
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
 unfolding par-rstep-var-restr-def using par-rstep-mctxt-mono[OF assms(1)] assms(2)
  by blast

lemma par-rstep-var-restr-refl[simp]: (t, t) ∈ par-rstep-var-restr R V
  unfolding par-rstep-var-restr-def
  by (intro CollectI exI conjI refl, force, rule par-rstep-mctxt-reflI, auto)
```

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: ⋀ x. (δ x, γ x) ∈ par-rstep R
    and st: (s, t) ∈ par-rstep-var-restr R V
    and subst-eq: ⋀ x. x ∉ V ⟹ δ x = γ x
  shows (s · δ, t · γ) ∈ par-rstep R
proof −
  from st[unfolded par-rstep-var-restr-def] subst-eq
  obtain C infos where st: (s, t) ∈ par-rstep-mctxt R C infos
    and subst-eq: ⋀ x. x ∈ vars-below-hole t C ⟹ δ x = γ x
    by auto
  thus ?thesis
  proof (induct C arbitrary: s t infos)
    case (MVar x)
    from par-rstep-mctxt-MVarE[OF this(1)]
    show ?case using subst-R by auto
```

```
  next
    case (MHole s t)
    have (s,t) ∈ par-rstep R
      using MHole.prems(1) par-rstep-par-rstep-mctxt-conv by blast
    hence step: (s · δ, t · δ) ∈ par-rstep R
      by (rule subst-closed-par-rstep)
    have vars-below-hole t MHole = vars-term t by simp
    with MHole(2) have t: t · δ = t · γ by (auto intro: term-subst-eq)
    thus ?case using step by auto
  next
    case (MFun f Cs s t infos)
    let ?n = length Cs
    let ?is = [0..<?n]

    from par-rstep-mctxt-MFunD[OF MFun(2)]
    obtain ss ts Infos
      where s: s = Fun f ss
        and t: t = Fun f ts
        and len: length ss = length Cs
        length ts = length Cs
        length Infos = length Cs
        and infos: infos = concat Infos
        and steps: ⋀ i. i<length Cs ⟹ (ss ! i, ts ! i) ∈ par-rstep-mctxt R (Cs ! i)
(Infos ! i)
      by blast
    {
      fix i
      assume i: i < ?n
      hence mem: Cs ! i ∈ set Cs by auto
      have IH: (ss ! i · δ, ts ! i · γ) ∈ par-rstep R
      proof (rule MFun(1)[OF mem steps[OF i]])
        fix x
        assume x ∈ vars-below-hole (ts ! i) (Cs ! i)
        hence x ∈ vars-below-hole t (MFun f Cs) unfolding t using i len(2)
          by (auto simp: set-zip)
        from MFun(3)[OF this] show δ x = γ x .
      qed
    }
    thus ?case unfolding s t using len(1−2) MFun(1−2) by auto
  qed
qed
```

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) ∈ par-rstep-var-restr R (γ ‘ V)
    and ⋀ x. σ x · (Var o γ) = Var x
  shows (s · σ, t · σ) ∈ par-rstep-var-restr R V
```

proof −
from *assms*(*1*)[*unfolded par-rstep-var-restr-def*, *simplified*]
obtain *C infos* **where** *step*: $(s,\ t) \in$ *par-rstep-mctxt R C infos* **and** *vars*: *vars-below-hole t C* $\cap\ \gamma$ ‘ $V = \{\}$
by *auto*
from *step*[*unfolded par-rstep-mctxt-def*, *simplified*]
have $t =_f$ (*C*, *par-rights infos*) **by** *auto*
hence *hole-poss C* $\subseteq$ *poss t* **by** (*metis hole-poss-subset-poss*)
hence *hp*: *hole-poss* (*C* ·*mc* σ) $\subseteq$ *poss t*
using *hole-poss-subst* **by** *auto*
from *par-rstep-mctxt-subst*[*OF step*, *of* σ]
have *step*: $(s \cdot \sigma,\ t \cdot \sigma) \in$ *par-rstep-mctxt R* (*C* ·*mc* σ) (*map* ($\lambda i.\ i$ ·*pi* σ) *infos*)
.
show $(s \cdot \sigma,\ t \cdot \sigma) \in$ *par-rstep-var-restr R V*
unfolding *par-rstep-var-restr-def*
proof (*standard*, *intro exI conjI*, *rule refl*, *rule step*)
show *vars-below-hole* ($t \cdot \sigma$) (*C* ·*mc* σ) $\cap$ $V = \{\}$
unfolding *vars-below-hole-term-subst*[*OF hp*]
unfolding *vars-below-hole-subst*
proof (*intro equals0I*, *elim IntE*)
fix *x*
assume $x \in \bigcup$ (*vars-term* ‘ σ ‘ *vars-below-hole t C*)
then obtain *y* **where** *y*: $y \in$ *vars-below-hole t C* **and** *x*: $x \in$ *vars-term* (σ *y*) **by** *auto*
from *y vars* **have** *y*: $y \notin \gamma$ ‘ V **by** *auto*
assume $x \in V$
with *assms*(*2*)[*of y*] *y x* **show** *False* **unfolding** *o-def* **by** (*cases* σ *y*, *auto*)
qed
qed
qed

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 ⋀ b l r. (b,l,r) ∈ critical-pairs ren R R ⟹ l = r
    and wf: ⋀ l r. (l,r) ∈ R ⟹ is-Fun l
  shows ∃ u. (t1,u) ∈ par-rstep R ∧ (t2,u) ∈ par-rstep R
proof −
  let ?R = par-rstep R
  let ?CP = critical-pairs ren R R
  {
    fix ls ts σ f r
    assume below: ⋀ i. i < length ls ⟹ ((map (λ l. l · σ) ls) ! i, ts ! i) ∈ ?R
      and rule: (Fun f ls, r) ∈ R
      and len: length ts = length ls
    let ?ls = map (λ l. l · σ) ls
   from weak-ortho(1) rule have lin: linear-term (Fun f ls) unfolding left-linear-trs-def
by auto
    let ?p1 = λ τ i. ts ! i = ls ! i · τ ∧ (∀ x ∈ vars-term (ls ! i). (σ x, τ x) ∈
par-rstep R)
    let ?p2 = λ τ i. (∃ C l'' l' r'. ls ! i = C⟨l''⟩ ∧ is-Fun l'' ∧ (l',r') ∈ R ∧ (l'' ·
σ = l' · τ) ∧ ((C ·c σ) ⟨ r' · τ ⟩, ts ! i) ∈ par-rstep R)
    {
      fix i
      assume i: i < length ls
      then have i2: i < length ts using len by simp
      from below[OF i] have step: (ls ! i · σ, ts ! i) ∈ ?R using i by auto
      from i have mem: ls ! i ∈ set ls by auto
      from lin i have lin: linear-term (ls ! i) by auto
      from par-rstep-linear-subst[OF lin step] have ∃ τ. ?p1 τ i ∨ ?p2 τ i .
    } note p12 = this
    have ∃ u. (r · σ, u) ∈ ?R ∧ (Fun f ts, u) ∈ ?R
    proof (cases ∃ i τ. i < length ls ∧ ?p2 τ i)
      case True
      then obtain i τ where i: i < length ls and p2: ?p2 τ i by blast
      from p2 obtain C l'' l' r' where lsi: ls ! i = C ⟨ l'' ⟩ and l'': is-Fun (l'')
and lr': (l',r') ∈ R
        and unif: l'' · σ = l' · τ and tsi: ((C ·c σ) ⟨ r' · τ ⟩, ts ! i) ∈ ?R  by blast
      from id-take-nth-drop[OF i] obtain bef aft where ls: ls = bef @ C ⟨ l'' ⟩ #
aft and bef: bef = take i ls unfolding lsi by auto
      from i bef have bef: length bef = i by auto
      let ?C = More f bef C aft
      from bef have hp: hole-pos ?C = i # hole-pos C by simp
      have fls: Fun f ls = ?C ⟨ l'' ⟩ unfolding ls by simp
      from mgu-vd-complete[OF unif] obtain μ1 μ2 δ where
        mgu: mgu-vd ren l'' l' = Some (μ1, μ2) and id: l'' · μ1 = l' · μ2
        and sigma: σ = μ1 ∘s δ and tau: τ = μ2 ∘s δ by blast
      let ?sig = map (λ s. s · σ)
      let ?r = (C ·c σ) ⟨ r' · τ⟩
      let ?bra = ?sig bef @ ?r # ?sig aft
     from weak-ortho(2)[OF critical-pairsI[OF rule lr' fls l'' mgu refl refl refl]]

      have id: r · σ = (?C ·c σ) ⟨r' · τ ⟩ unfolding sigma tau by simp
```

also have ... = *Fun f ?bra* **by** *simp*
also have (..., *Fun f ts*) ∈ *?R*
proof (*rule par-rstep.par-step-fun*)
show *length ?bra* = *length ts* **unfolding** *len* **unfolding** *ls* **by** *simp*
next
fix *j*
assume *j*: *j* < *length ts*
show (*?bra* ! *j*, *ts* ! *j*) ∈ *?R*
proof (*cases j* = *i*)
case *True*
then have *?bra* ! *j* = *?r* **using** *bef i* **by** (*simp add*: *nth-append*)
then show *?thesis* **using** *tsi True* **by** *simp*
next
case *False*
then have *?bra* ! *j* = (*?sig bef* @ (*C* ⟨ *l''* ⟩ · *σ*) # *?sig aft*) ! *j* **using** *False bef i* **by** (*simp add*: *nth-append*)
also have ... = *?sig ls* ! *j* **unfolding** *ls* **by** *simp*
finally show *?thesis*
using *below*[*OF j*[*unfolded len*]] **by** *auto*
qed
qed
finally have *step*: (*r* · *σ*, *Fun f ts*) ∈ *?R* **.**
show ∃ *u*. (*r* · *σ*, *u*) ∈ *?R* ∧ (*Fun f ts*, *u*) ∈ *?R*
by (*rule exI*, *rule conjI*[*OF step par-rstep-refl*])
next
case *False*
with *p12*
have ∀ *i*. (∃ *τ*. *i* < *length ls* ⟶ *?p1 τ i*) **by** *blast*
from *choice*[*OF this*] **obtain** *tau* **where** *tau*: ⋀ *i*. *i* < *length ls* ⟹ *?p1* (*tau i*) *i* **by** *blast*
from *lin* **have** *is-partition* (*map vars-term ls*) **by** *auto*
from *subst-merge*[*OF this*, *of tau*] **obtain** *τ* **where** *τ*: ⋀ *i x*. *i* < *length ls* ⟹ *x* ∈ *vars-term* (*ls* ! *i*) ⟹ *τ x* = *tau i x*
by *blast*
obtain *δ* **where** *delta*: *δ* = (*λ x*. *if x* ∈ *vars-term* (*Fun f ls*) *then τ x else σ x*) **by** *auto*
{
fix *i*
assume *i*: *i* < *length ls*
from *tau*[*OF i*] **have** *p*: *?p1* (*tau i*) *i* **.**
have *id1*: *ls* ! *i* · *tau i* = *ls* ! *i* · *τ*
by (*rule term-subst-eq*[*OF τ*[*OF i*, *symmetric*]])
have *id2*: ... = *ls* ! *i* · *δ*
by (*rule term-subst-eq*, *unfold delta*, *insert i*, *auto*)
have *p*: *?p1 δ i* **using** *p* **using** *τ*[*OF i*] **unfolding** *id1 id2* **using** *id2* **unfolding** *delta* **by** *auto*
} **note** *delt* = *this*
have *r-delt*: (*r* · *σ*, *r* · *δ*) ∈ *?R*
proof (*rule all-ctxt-closed-subst-step*)

```
      fix x
      assume x: x ∈ vars-term r
      show (σ x, δ x) ∈ ?R
      proof (cases x ∈ vars-term (Fun f ls))
        case True
        then obtain l where l: l ∈ set ls and x: x ∈ vars-term l by auto
        from l[unfolded set-conv-nth] obtain i where i: i < length ls and l: l =
ls ! i by auto
        from delt[OF i] x l show ?thesis by auto
      next
        case False
        then have δ x = σ x unfolding delta by auto
        then show ?thesis by auto
      qed
    qed auto
    {
      let ?ls = map (λ l. l · δ) ls
      have ts = map (λ i. ts ! i) [0 ..< length ts] by (rule map-nth[symmetric])
      also have ... = map (λ i. ts ! i) [0 ..< length ls] unfolding len by simp
      also have ... = map (λ i. ?ls ! i) [0 ..< length ?ls]
        by (rule nth-map-conv, insert delt[THEN conjunct1], auto)
      also have ... = ?ls
        by (rule map-nth)
      finally have Fun f ts = Fun f ls · δ by simp
    } note id = this
    have l-delt: (Fun f ts, r · δ) ∈ ?R unfolding id
      by (rule par-rstep.root-step[OF rule])
    show ∃ u. (r · σ, u) ∈ ?R ∧ (Fun f ts, u) ∈ ?R
      by (intro exI conjI, rule r-delt, rule l-delt)
  qed
} note root-arg = this
from st1 st2 show ?thesis
proof (induct arbitrary: t2 rule: par-rstep.induct)
  case (par-step-var x t2)
  have t2: t2 = Var x
    by (rule wf-trs-par-rstep[OF wf par-step-var])
  show ∃ u. (Var x,u) ∈ ?R ∧ (t2, u) ∈ ?R unfolding t2
    by (intro conjI exI par-rstep.par-step-var, auto)
next
  case (par-step-fun ts1 ss f t2)
  note IH = this
  show ?case using IH(4)
  proof (cases rule: par-rstep.cases)
    case (par-step-fun ts2)
    from IH(3) par-step-fun(3) have len: length ts2 = length ts1 by simp
    {
      fix i
      assume i: i < length ts1
      then have i2: i < length ts2 using len by simp
```

from *par-step-fun(2)*[*OF i2*] **have** *step2*: (*ss* ! *i*, *ts2* ! *i*) ∈ *?R* **.**
from *IH(2)*[*OF i step2*] **have** ∃ *u*. (*ts1* ! *i*, *u*) ∈ *?R* ∧ (*ts2* ! *i*, *u*) ∈ *?R* **.**
}
then have ∀ *i*. ∃ *u*. (*i* < *length ts1* ⟶ (*ts1* ! *i*, *u*) ∈ *?R* ∧ (*ts2* ! *i*, *u*) ∈ *?R*) **by** *blast*
from *choice*[*OF this*] **obtain** *us* **where** *join*: ⋀ *i*. *i* < *length ts1* ⟹ (*ts1* ! *i*, *us i*) ∈ *?R* ∧ (*ts2* ! *i*, *us i*) ∈ *?R* **by** *blast*
let *?us* = *map us* [*0* ..< *length ts1*]
{
fix *i*
assume *i*: *i* < *length ts1*
from *join*[*OF this*] *i* **have** (*ts1* ! *i*, *?us* ! *i*) ∈ *?R* (*ts2* ! *i*, *?us* ! *i*) ∈ *?R* **by** *auto*
} note *join* = *this*
let *?u* = *Fun f ?us*
have *step1*: (*Fun f ts1*, *?u*) ∈ *?R*
by (*rule par-rstep.par-step-fun*[*OF join(1)*], *auto*)
have *step2*: (*Fun f ts2*, *?u*) ∈ *?R*
by (*rule par-rstep.par-step-fun*[*OF join(2)*], *insert len*, *auto*)
show *?thesis* **unfolding** *par-step-fun(1)* **using** *step1 step2* **by** *blast*
next
case (*root-step l r σ*)
from *wf*[*OF root-step(3)*] *root-step(1)* **obtain** *ls* **where** *l*: *l* = *Fun f ls*
by *auto*
from *root-step(1) l* **have** *ss*: *ss* = *map* (λ *l*. *l* · *σ*) *ls* (**is** - = *?ls*) **by** *simp*
from *root-step(3) l* **have** *rule*: (*Fun f ls*, *r*) ∈ *R* **by** *simp*
from *root-step(2)* **have** *t2*: *t2* = *r* · *σ* **.**
from *par-step-fun(3) ss* **have** *len*: *length ts1* = *length ls* **by** *simp*
from *root-arg*[*OF par-step-fun(1)*[*unfolded ss len*] *rule len*]
show *?thesis* **unfolding** *t2* **by** *blast*
qed
next
case (*root-step l r σ*)
note *IH* = *this*
from *wf*[*OF IH(1)*] *IH(1)* **obtain** *f ls* **where** *l*: *l* = *Fun f ls* **and** *rule*: (*Fun f ls*,*r*) ∈ *R*
by (*cases l*, *auto*)
from *IH(2)*[*unfolded l*] **show** *?case*
proof (*cases rule*: *par-rstep.cases*)
case (*par-step-var x*)
then show *?thesis* **by** *simp*
next
case (*root-step l′ r′ τ*)
then have *t2*: *t2* = *r′* · *τ* **by** *auto*
have *id*: *Fun f ls* = □⟨*Fun f ls*⟩ **by** *simp*
from *mgu-vd-complete*[*OF root-step(1)*, *of ren*] **obtain** *mu1 mu2 delta* **where**
mgu: *mgu-vd ren* (*Fun f ls*) *l′* = *Some* (*mu1*, *mu2*) **and** *sigma*: *σ* = *mu1* ∘$_s$ *delta*

and *tau*: $\tau = mu2 \circ_s delta$ **by** *auto*
from *weak-ortho*(*2*)[*OF critical-pairsI*[*OF rule root-step*(*3*) *id - mgu refl refl*]]
have $r \cdot mu1 = r' \cdot mu2$ **by** *simp*
then have *id*: $r \cdot \sigma = r' \cdot \tau$ **unfolding** *sigma tau* **by** *simp*
show *?thesis* **unfolding** *t2 id* **by** *auto*
next
case (*par-step-fun ts ls' g*)
then have *ls'*: $ls' = map\ (\lambda\ l.\ l \cdot \sigma)\ ls$ **and** *g*: $g = f$ **and** *len*: *length ts = length ls* **by** *auto*
note *par-step-fun = par-step-fun*[*unfolded ls' g len*]
from *root-arg*[*OF par-step-fun*(*3*) *rule len*]
show *?thesis* **unfolding** *par-step-fun*(*2*) **.**
qed
qed
qed

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 −
let *?R* = *par-rstep R*
from *weakly-orthogonal-main*[*OF - - weak-ortho wf*]
have *diamond*: $\bigwedge$ *s t1 t2*. (*s*,*t1*) $\in$ *?R* $\Longrightarrow$ (*s*,*t2*) $\in$ *?R* $\Longrightarrow$ $\exists$ *u*. (*t1*,*u*) $\in$ *?R* $\wedge$ (*t2*,*u*) $\in$ *?R* **.**
show *?thesis*
by (*rule diamond-imp-CR*, *rule diamond-I*, *insert diamond*, *blast*)
qed

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 −
from *weakly-orthogonal-par-rstep-CR*[*OF assms*] **have** *CR* (*par-rstep R*) **.**
then show *?thesis* **unfolding** *CR-on-def join-def rtrancl-converse par-rsteps-rsteps*
.
qed

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*) ∈ *mstep R* |
args: ⋀*f n ss ts*. ⟦*length ss* = *n*; *length ts* = *n*;
∀ *i*<*n*. (*ss* ! *i*, *ts* ! *i*) ∈ *mstep R*⟧ ⟹
(*Fun f ss*, *Fun f ts*) ∈ *mstep R* |
rule: ⋀*l r σ τ*. ⟦(*l*, *r*) ∈ *R*; ∀ *x*∈*vars-term l*. (*σ x*, *τ x*) ∈ *mstep R*⟧ ⟹
(*l* · *σ*, *r* · *τ*) ∈ *mstep R*

lemma *mstep-refl* [*simp*]:
(*t*, *t*) ∈ *mstep R*
by (*induct t*) (*auto intro*: *mstep.intros*)

lemma *mstep-ctxt*:
assumes (*s*, *t*) ∈ *mstep R*
shows (*C*⟨*s*⟩, *C*⟨*t*⟩) ∈ *mstep R*
proof (*induction C*)
case *Hole* **with** *assms* **show** *?case* **by** *simp*
next
case (*More f ss C ts*)
let *?ss* = *ss* @ *C*⟨*s*⟩ # *ts*
let *?ts* = *ss* @ *C*⟨*t*⟩ # *ts*
{ fix *i* **assume** *i* = *length ss*
then have (*?ss* ! *i*, *?ts* ! *i*) ∈ *mstep R*
using *More.IH* **by** *simp* **}**
moreover
{ fix *i* **assume** *i* < *length ss*
then have (*?ss* ! *i*, *?ts* ! *i*) ∈ *mstep R*
by (*simp add*: *nth-append*) **}**
moreover
{ fix *i* **assume** *i* < *length ?ss* **and** *i* > *length ss*
then have (*?ss* ! *i*, *?ts* ! *i*) ∈ *mstep R*
by (*simp add*: *nth-append*) **}**
ultimately
have ∀ *i*<*length ?ss*. (*?ss* ! *i*, *?ts* ! *i*) ∈ *mstep R*
by (*metis linorder-neqE-nat*)
from *mstep.args* [*OF* - - *this*, *simplified*]
show *?case* **by** *simp*
qed

lemma *rstep-imp-mstep*:
assumes (*s*, *t*) ∈ *rstep R*
shows (*s*, *t*) ∈ *mstep R*
using *assms*

proof (*induct*)
  case (*IH C* σ *l r*)
  have $\forall x{\in}$*vars-term l*. (σ *x*, σ *x*) $\in$ *mstep R* **by** *simp*
  from *mstep.rule* [*OF* ‹(*l*, *r*) $\in$ *R*› *this*]
  have (*l* $\cdot$ σ, *r* $\cdot$ σ) $\in$ *mstep R* **by** *simp*
  from *mstep-ctxt* [*OF this*] **show** *?case* **by** *blast*
qed

lemma *rstep-mstep-subset*:
  rstep R $\subseteq$ *mstep R*
  by (*auto simp*: *rstep-imp-mstep*)

lemma *subst-rsteps-imp-rule-rsteps*:
  assumes $\forall x{\in}$*vars-term l*. (σ *x*, τ *x*) $\in$ (*rstep R*)*
    and (*l*, *r*) $\in$ *R*
  shows (*l* $\cdot$ σ, *r* $\cdot$ τ) $\in$ (*rstep R*)*
proof −
  let *?σ*=λ*x*. (*if x* $\in$ *vars-term l then* σ *x else* τ *x*)
  have *l* $\cdot$ σ = *l* $\cdot$ *?σ*
    by (*simp add*: *term-subst-eq-conv*)
  with ‹(*l*, *r*) $\in$ *R*› **have** (*l* $\cdot$ σ, *r* $\cdot$ *?σ*) $\in$ *rstep R*
    by *auto*
  moreover have (*r* $\cdot$ *?σ*, *r* $\cdot$ τ) $\in$ (*rstep R*)*
    by (*rule subst-rsteps-imp-rsteps*) (*insert assms*, *auto*)
  ultimately show *?thesis* **by** *auto*
qed

lemma *mstep-imp-rsteps*:
  assumes (*s*, *t*) $\in$ *mstep R*
  shows (*s*, *t*) $\in$ (*rstep R*)*
  using *assms*
proof (*induct*)
  case (*args f n ss ts*)
  then show *?case* **by** (*metis args-rsteps-imp-rsteps*)
next
  case (*rule l r* σ τ)
  then show *?case* **using** ‹(*l*, *r*) $\in$ *R*› **by** (*metis subst-rsteps-imp-rule-rsteps*)
qed *simp*

lemma *mstep-rsteps-subset*:
  shows *mstep R* $\subseteq$ (*rstep R*)*
  by (*auto simp*: *mstep-imp-rsteps*)

lemma *mstep-mono*: *R* $\subseteq$ *S* $\Longrightarrow$ *mstep R* $\subseteq$ *mstep S*
proof −
  have (*s*,*t*) $\in$ *mstep R* $\Longrightarrow$ *R* $\subseteq$ *S* $\Longrightarrow$ (*s*,*t*) $\in$ *mstep S* **for** *s t*
    by (*induct rule*: *mstep.induct*, *auto intro*: *mstep.intros*)
  thus *R* $\subseteq$ *S* $\Longrightarrow$ *mstep R* $\subseteq$ *mstep S* **by** *auto*
qed

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-Var l*
    and (*Var x*, *t*) $\in$ *mstep R*
  shows *t* = *Var x*
  using *assms*(2−)
proof *cases*
  case (*rule l r σ τ*)
  then show *?thesis* **using** $*$ **by** (*cases l*, *auto*)
qed *auto*

8.1 Maximal multi-step rewriting.

inductive-set
  mmstep :: ('*f*, '*v*) *trs* ⇒ ('*f*, '*v*) *term rel*
  for *R*
  where
    Var: (*Var x*, *Var x*) ∈ *mmstep R* |
    args: $\bigwedge f\ n\ ss\ ts.$ ⟦*length ss* = *n*; *length ts* = *n*;
    ¬ (∃ (*l*, *r*)∈*R*. ∃ *σ*. *Fun f ss* = *l* · *σ*);
    ∀ *i*<*n*. (*ss* ! *i*, *ts* ! *i*) ∈ *mmstep R*⟧ ⟹
    (*Fun f ss*, *Fun f ts*) ∈ *mmstep R* |
    rule: $\bigwedge l\ r\ \sigma\ \tau.$ ⟦(*l*, *r*) ∈ *R*; ∀ *x*∈*vars-term l*. (*σ x*, *τ x*) ∈ *mmstep R*⟧ ⟹
    (*l* · *σ*, *r* · *τ*) ∈ *mmstep R*

lemma *mmstep-imp-mstep*:
  assumes (*s*, *t*) ∈ *mmstep R*
  shows (*s*, *t*) ∈ *mstep R*
  using *assms* **by** (*induct*) (*auto intro*: *mstep.intros*)

lemma *mmstep-mstep-subset*:
  mmstep R ⊆ *mstep R*
  by (*auto simp*: *mmstep-imp-mstep*)

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*)
  unfolding *rrewrite-def List.maps-def* **using** *match-matches*[*of s*]
  by *force*

lemma *rrewrite-complete*: **assumes** (*s*,*t*) ∈ *rrstep* (*set R*)
  shows ∃ *u*. *u* ∈ *set* (*rrewrite s*)
proof −
  from *assms* **obtain** *l r σ* **where** *lr*: (*l*,*r*) ∈ *set R* **and** *s*: *s* = *l* · *σ* **and** *t*: *t* = *r* · *σ*
    by (*rule rrstepE*)
  from *match-complete'*[*OF s*[*symmetric*]] **obtain** *τ* **where** *match*: *match s l* = *Some τ*
    by *auto*
  with *lr match* **have** *r* · *τ* ∈ *set* (*rrewrite s*) **unfolding** *rrewrite-def List.maps-def* **by** *force*
  thus *?thesis* **..**
qed

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 (*standard*; *clarify*)
  fix *t*
  assume (*s*,*t*) ∈ *rrstep* (*set R*)
  then obtain *l r σ* **where** *lr*: (*l*,*r*) ∈ *set R* **and** *s*: *s* = *l* · *σ* **and** *t*: *t* = *r* · *σ*
    by (*rule rrstepE*)
  from *match-complete'*[*OF s*[*symmetric*]] **obtain** *τ* **where** *match*: *match s l* = *Some τ*
    and *vars*: ⋀ *x*. *x* ∈ *vars-term l* ⟹ *σ x* = *τ x* **by** *auto*
  have *vars'*: ⋀ *x*. *x* ∈ *vars-term r* ⟹ *σ x* = *τ x* **using** *assms*[*OF lr*] *vars* **by** *auto*
  have *t*: *t* = *r* · *τ* **unfolding** *t* **using** *vars'* **by** (*intro term-subst-eq*, *auto*)
  with *lr match* **show** *t* ∈ *set* (*rrewrite s*) **unfolding** *rrewrite-def List.maps-def* **by** *force*
qed (*rule rrewrite-sound*)

```
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 (induct s arbitrary: t rule: rewrite.induct)
  case (1 s t)
  note [simp] = rewrite.simps[of s]
  from 1(2) consider (root) t ∈ set (rrewrite s) |
    (arg) f ss ti i where s = Fun f ss i < length ss ti ∈ set (rewrite (ss ! i)) t =
Fun f (ss[i := ti])
    by (auto simp: set-zip)
  thus ?case
  proof cases
    case root
    with rrewrite-sound[of t s] have (s,t) ∈ rrstep (set R) by auto
    thus ?thesis by (rule rrstep-imp-rstep)
  next
    case (arg f ss ti i)
   from arg(2) have mem: (i, ss ! i) ∈ set (zip [0..<length ss] ss) by (force simp:
set-zip)
    from 1(1)[OF arg(1) mem refl arg(3)]
    have IH: (ss ! i, ti) ∈ rstep (set R) .
    with arg have (s,t) ∈ nrrstep (set R)
      unfolding nrrstep-iff-arg-rstep by blast
    thus ?thesis by (rule nrrstep-imp-rstep)
  qed
qed


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 (standard; clarify)
  fix t
  assume (s,t) ∈ rstep (set R)
  then obtain C u v where s: s = C⟨u⟩ and t: t = C⟨v⟩ and step: (u,v) ∈ rrstep
(set R)
    by blast
  from rrewrite[OF assms, of u] step have step: v ∈ set (rrewrite u) by auto
  show t ∈ set (rewrite s) unfolding s t
  proof (induct C)
    case Hole
    then show ?case using step by (auto simp: rewrite.simps[of u])
  next
    case (More f bef C aft)
    show ?case
```

```
    apply (simp add: rewrite.simps[of Fun f -] set-zip)
    apply (intro disjI2)
    apply (intro exI[of - C⟨u⟩] exI)
    apply (intro conjI exI[of - length bef])
    using More by (auto simp: nth-append)
  qed
qed (rule rewrite-sound)

lemma rewrite-complete: assumes (s,t) ∈ rstep (set R)
  shows ∃ w. w ∈ set (rewrite s)
proof −
  from assms obtain C u v where s: s = C⟨u⟩ and t: t = C⟨v⟩ and step: (u,v)
∈ rrstep (set R)
    by blast
  from rrewrite-complete[OF step] obtain v where step: v ∈ set (rrewrite u) by
auto
  have C⟨v⟩ ∈ set (rewrite s) unfolding s
  proof (induct C)
    case Hole
    then show ?case using step by (auto simp: rewrite.simps[of u])
  next
    case (More f bef C aft)
    show ?case
      apply (simp add: rewrite.simps[of Fun f -] set-zip)
      apply (intro disjI2)
      apply (intro exI[of - C⟨u⟩] exI)
      apply (intro conjI exI[of - length bef])
      using More by (auto simp: nth-append)
  qed
  thus ?thesis by blast
qed
end


lemma rrewrite-mono: set R ⊆ set S ⟹ set (rrewrite R s) ⊆ set (rrewrite S s)
  unfolding rrewrite-def List.maps-def by auto

lemma Union-image-mono: (⋀ x. x ∈ A ⟹ f x ⊆ g x) ⟹ ⋃ (f ‘ A) ⊆ ⋃ (g ‘
A)
  by blast

lemma rewrite-mono: assumes set R ⊆ set S
  shows set (rewrite R s) ⊆ set (rewrite S s)
proof −
  note rrewrite = rrewrite-mono[OF assms]
  show ?thesis
  proof (induct s)
    case (Var x)
    thus ?case using rrewrite unfolding rewrite.simps[of - Var x] by auto
```

```
  next
    case (Fun f ss)
    show ?case unfolding rewrite.simps[of - Fun f ss]
        set-append term.simps set-concat set-map image-comp set-zip o-def
      apply (intro Un-mono, rule rrewrite)
      by (intro Union-image-mono, insert Fun, force simp: set-conv-nth[of ss])
  qed
qed


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
  show is-root-step R s t ⟹ (s, t) ∈ rrstep R
    unfolding is-root-step-def rrstep-def rstep-r-p-s-def'
    by (auto split: option.splits) (force dest: match-list-matches)
  assume (s, t) ∈ rrstep R
  then obtain σ l r where lr: (l,r) ∈ R and id: s = l · σ t = r · σ
    by (metis rrstepE)
  show is-root-step R s t unfolding id
    unfolding is-root-step-def
    by (cases match-list Var [(l, l · σ), (r, r · σ)],
        auto intro!: bexI[OF - lr] dest!: match-list-complete)
qed

lemma is-root-step: is-root-step R s t ⟹ (s, t) ∈ rrstep R
  unfolding rrstep-code .

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 (induct R s t rule: is-rstep.induct)
  case (1 R f ts g ss)
  show ?case
```

proof (*cases* (*Fun f ts*, *Fun g ss*) ∈ *rrstep R*)
case *True*
thus *?thesis* **using** *rrstep-imp-rstep* **by** *auto*
next
case *False*
with *1*(*2*) **obtain** *i* **where**
i: *i* < *length ss* **and**
gf: *g* = *f* **and** *len*: *length ts* = *length ss* **and** *id*: *ss* = *ts*[*i* := *ss* ! *i*]
and *rec*: *is-rstep R* (*ts* ! *i*) (*ss* ! *i*)
by *auto*
from *1*(*1*)[*OF - rec*] *i* **have** (*ts* ! *i*, *ss* ! *i*) ∈ *rstep R* **by** *auto*
thus *?thesis* **unfolding** *gf* **using** *id i len*
by (*metis nrrstep-iff-arg-rstep nrrstep-imp-rstep*)
qed
qed (*insert rrstep-imp-rstep*, *auto*)

lemma *is-rstep-complete*: **assumes** (*s*,*t*) ∈ *rstep R*
shows *is-rstep R s t*
proof −
from *rstepE*[*OF assms*] **obtain** *C s*′ *t*′ **where**
id: *s* = *C* ⟨*s*′⟩ *t* = *C*⟨*t*′⟩ **and** *step*: (*s*′,*t*′) ∈ *rrstep R*
using *rrstepI* **by** *metis*
show *?thesis* **unfolding** *id*
proof (*induct C*)
case *Hole*
then show *?case* **using** *step* **by** (*cases s*′; *cases t*′, *auto*)
next
case (*More f bef C aft*)
show *?case* **unfolding** *intp-actxt.simps is-rstep.simps*
by (*intro disjI1 conjI bexI*[*of - length bef*], *insert More*, *auto*)
qed
qed

lemma *is-rstep*[*simp*]: *is-rstep R s t* ⟷ (*s*,*t*) ∈ *rstep R*
using *is-rstep-sound is-rstep-complete* **by** *auto*

lemma *in-rstep-code*[*code-unfold*]:
st ∈ *rstep R* ⟷ (*case st of* (*s*,*t*) ⇒ *is-rstep R s t*)
by (*cases st*, *auto*)

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*))^* **using** *res*[*unfolded compute-rstep-NF-def*]
proof (*rule compute-NF-sound*)

fix *s t*
assume *first-rewrite R s = Some t*
from *this*[*unfolded first-rewrite-def*] **obtain** *ts* **where** *rewrite R s = t # ts*
by (*cases rewrite R s*, *auto*)
then have *t*: *t ∈ set* (*rewrite R s*) **by** *simp*
from *rewrite-sound*[*OF this*] **show** (*s*,*t*) ∈ *rstep* (*set R*) **.**
qed

lemma *compute-rstep-NF-complete*: **assumes** *res*: *compute-rstep-NF R s = Some t*
shows *t ∈ NF* (*rstep* (*set R*)) **using** *res*[*unfolded compute-rstep-NF-def*]
proof (*rule compute-NF-complete*)
fix *s*
assume *first-rewrite R s = None*
from *this*[*unfolded first-rewrite-def*] **have** *empty*: *rewrite R s* = []
by (*cases rewrite R s*, *auto*)
have *False* **if** (*s*,*t*) ∈ *rstep* (*set R*) **for** *t*
using *rewrite-complete*[*OF that*] *arg-cong*[*OF empty*, *of set*] **by** *auto*
thus *s ∈ NF* (*rstep* (*set R*)) **by** *blast*
qed

lemma *compute-rstep-NF-SN*: **assumes** *SN*: *SN* (*rstep* (*set R*))
shows ∃ *t*. *compute-rstep-NF R s = Some t*
proof −
have ∃ *t*. *compute-NF* (*first-rewrite R*) *s = Some t*
proof (*rule compute-NF-SN*[*OF SN*])
fix *s t*
assume *first-rewrite R s = Some t*
from *this*[*unfolded first-rewrite-def*] **have**
rewrite: *t ∈ set* (*rewrite R s*) **by** (*auto split*: *list.splits*)
from *rewrite-sound*[*OF this*]
show (*s*,*t*) ∈ *rstep* (*set R*) **.**
qed
then show *?thesis* **unfolding** *compute-rstep-NF-def* **.**
qed

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
  using assms
proof (induct i arbitrary: t)
  case 0
  then show ?case by auto
next
  case (Suc i)
  let ?R = λ j. (rstep (set R))^^j
  from Suc(2)
  have t ∈ set (reachable-terms R s i)
    ∨ (∃ u ∈ set (reachable-terms R s i). t ∈ set (rewrite R u)) by (simp add:
Let-def)
  then show ?case
  proof
    assume t ∈ set (reachable-terms R s i)
    from Suc(1)[OF this] obtain j where j ≤ i and (s,t) ∈ ?R j by auto
    then show ?thesis by (intro exI[of - j], auto)
  next
    assume ∃ u ∈ set (reachable-terms R s i). t ∈ set (rewrite R u)
    then obtain u where u: u ∈ set (reachable-terms R s i)
      and 1: t ∈ set (rewrite R u) by auto
    from rewrite-sound[OF 1] have ut: (u,t) ∈ rstep (set R) .
    from Suc(1)[OF u] obtain j where j: j ≤ i and su: (s,u) ∈ ?R j by auto
    from su ut have (s,t) ∈ ?R (Suc j) by auto
    with j show ?thesis by (intro exI[of - Suc j], auto)
  qed
qed

lemma reachable-terms:
  assumes t ∈ set (reachable-terms R s i)
  shows (s,t) ∈ (rstep (set R))^*
  using reachable-terms-nat[OF assms] by (metis relpow-imp-rtrancl)

lemma reachable-terms-one:
  assumes t ∈ set (reachable-terms R s (Suc 0))
  shows (s,t) ∈ (rstep (set R))^=
proof −
  from reachable-terms-nat[OF assms] obtain j where j ≤ 1
    and (s,t) ∈ (rstep (set R))^^j by auto
  then show ?thesis by (cases j, auto)
qed
```

9.2.2 Algorithms to Ensure Joinability

```
definition
  check-join-NF ::
  ('f :: showl, 'v :: showl) rules ⇒
    ('f, 'v) term ⇒ ('f, 'v) term ⇒ 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 −
  note ok = ok[unfolded check-join-NF-def]
  from ok obtain s' where s: compute-rstep-NF R s = Some s' by force
  note ok = ok[unfolded s]
  from ok obtain t' where t: compute-rstep-NF R t = Some t' by force
  from ok[unfolded t] have id: s' = t' by simp
  note seq = compute-rstep-NF-sound
  from seq[OF s] seq[OF t]
  show ?thesis unfolding id by auto
qed


function iterative-join-search-main ::
  ('f,'v) rules ⇒ ('f,'v) term ⇒ ('f,'v) term ⇒ nat ⇒ nat ⇒ 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)
  by pat-completeness auto

termination by (relation measure ( λ (R,s,t,i,n). Suc n − i)) auto

lemma iterative-join-search-main:
  iterative-join-search-main R s t i n ⟹ (s,t) ∈ join (rstep (set R))
proof (induction rule: iterative-join-search-main.induct)
  case (1 R s t i n)
  from 1(2) have i-n: i ≤ n by (simp split: if-splits)
  note IH = 1(1)[OF i-n]
  let ?I = list-inter (reachable-terms R s i) (reachable-terms R t i)
  from 1(2) i-n have ?I ≠ [] ∨ iterative-join-search-main R s t (Suc i) n by auto
  then show ?case
  proof
    assume a: ?I ≠ []
    then obtain u us where u: ?I = u # us by (cases ?I, auto)
    then have d: u ∈ set ?I by auto
   from this[simplified] reachable-terms[of u - - i] have c: (s, u) ∈ (rstep (set R))*
by auto
```

from *d*[*simplified*] *reachable-terms*[*of u - - i*] **have** *e*: (*t*,*u*) ∈ (*rstep* (*set R*))^* **by** *auto*
from *c e* **show** *?thesis* **by** *auto*
next
assume *b*: *iterative-join-search-main R s t* (*Suc i*) *n*
from *IH*[*OF this*] **show** *?thesis* **.**
qed
qed

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*))
by (*rule iterative-join-search-main*, *unfold iterative-join-search-def*)

definition
check-join-BFS-limit ::
nat ⇒ ('*f* :: *showl*, '*v* :: *showl*) *rules* ⇒
('*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*))
by (*rule iterative-join-search*, *insert ok*[*unfolded check-join-BFS-limit-def*], *simp*)

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*)
unfolding *map-funs-rules-def map-funs-trs.simps* **by** *simp*

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*
by (*simp add*: *add-vars-rule-def vars-rule-list-def*)

lemma *add-funs-rule-funs-rule-list-conv* [*simp*]:

add-funs-rule r fs = funs-rule-list r @ fs
by (*simp add*: *add-funs-rule-def funs-rule-list-def*)

lemma *add-funas-rule-funas-rule-list-conv* [*simp*]:
add-funas-rule r fs = funas-rule-list r @ fs
by (*simp add*: *add-funas-rule-def funas-rule-list-def*)

lemma *add-roots-rule-roots-rule-list-conv* [*simp*]:
add-roots-rule r fs = roots-rule-list r @ fs
by (*simp add*: *add-roots-rule-def roots-rule-list-def*)

lemma *add-funas-args-rule-funas-args-rule-list-conv* [*simp*]:
add-funas-args-rule r fs = funas-args-rule-list r @ fs
by (*simp add*: *add-funas-args-rule-def funas-args-rule-list-def*)

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* ⇒ '*f list* ⇒ '*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* ⇒ ('*f* × *nat*) *list* ⇒ ('*f* × *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* ⇒ ('*f* × *nat*) *list* ⇒ ('*f* × *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* ⇒ ('*f* × *nat*) *list* ⇒ ('*f* × *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*
by (*simp add*: *insert-vars-rule-def ac-simps*)

lemma *set-insert-funs-rule* [*simp*]:
set (*insert-funs-rule r xs*) = *funs-term* (*fst r*) ∪ *funs-term* (*snd r*) ∪ *set xs*
by (*simp add*: *insert-funs-rule-def ac-simps*)

lemma *set-insert-funas-rule* [*simp*]:
set (*insert-funas-rule r xs*) = *funas-term* (*fst r*) ∪ *funas-term* (*snd r*) ∪ *set xs*
by (*simp add*: *insert-funas-rule-def ac-simps*)

lemma *set-insert-roots-rule* [*simp*]:
set (*insert-roots-rule r xs*) = *root-set* (*fst r*) ∪ *root-set* (*snd r*) ∪ *set xs*
by (*cases fst r snd r rule*: *term.exhaust* [*case-product term.exhaust*])
(*auto simp*: *insert-roots-rule-def ac-simps*)

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*
by (*simp add*: *insert-funas-args-rule-def ac-simps funas-args-term-def*)

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*
by (*simp add*: *vars-rule-def*)

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* []
by (*simp-all add*: *vars-rule-list-def funs-rule-list-def funas-rule-list-def*
roots-rule-list-def funas-args-rule-list-def)

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* []
by (*induct trs*)
(*simp-all add*: *vars-trs-list-def funs-trs-list-def funas-trs-list-def*
roots-trs-list-def funas-args-trs-list-def)

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*
by (*induct trs arbitrary*: *xs*) (*simp-all add*: *insert-vars-trs-def ac-simps vars-rule-def*)

lemma *set-insert-funs-trs* [*simp*]:
set (*insert-funs-trs trs fs*) = (⋃ *r* ∈ *set trs*. *funs-rule r*) ∪ *set fs*
by (*induct trs arbitrary*: *fs*) (*simp-all add*: *insert-funs-trs-def ac-simps funs-rule-def*)

lemma *set-insert-funas-trs* [*simp*]:
set (*insert-funas-trs trs fs*) = (⋃ *r* ∈ *set trs*. *funas-rule r*) ∪ *set fs*
by (*induct trs arbitrary*: *fs*) (*simp-all add*: *insert-funas-trs-def ac-simps funas-rule-def*)

lemma *set-insert-roots-trs* [*simp*]:
set (*insert-roots-trs trs fs*) = (⋃ *r* ∈ *set trs*. *roots-rule r*) ∪ *set fs*
by (*induct trs arbitrary*: *fs*) (*simp-all add*: *insert-roots-trs-def ac-simps roots-rule-def*)

lemma *set-insert-funas-args-trs* [*simp*]:
set (*insert-funas-args-trs trs fs*) = (⋃ *r* ∈ *set trs*. *funas-args-rule r*) ∪ *set fs*
by (*induct trs arbitrary*: *fs*)
(*simp-all add*: *insert-funas-args-trs-def ac-simps funas-args-rule-def*)

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*}
by (*force simp*: *defined-list-def defined-def elim*!: *root-Some*)

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* $\circ$ *showsl* (*STR* ''$\boxed{\hookleftarrow}$*is not left−linear*$\boxed{\hookleftarrow}$''))

lemma *check-left-linear-trs* [*simp*]:
isOK (*check-left-linear-trs R*) = *left-linear-trs* (*set R*)
unfolding *check-left-linear-trs-def left-linear-trs-def*
by *auto*

definition *check-varcond-subset* :: (-,-) *rules* $\Rightarrow$ *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 ''*) $\circ$ *showsl x*
$\circ$ *showsl* (*STR '' in right−hand side of rule ''*) $\circ$ *showsl-rule rule* $\circ$ *showsl-nl*)
) *R*

lemma *check-varcond-subset* [*simp*]:
isOK (*check-varcond-subset R*) = ($\forall$ (*l*, *r*) $\in$ *set R*. *vars-term r* $\subseteq$ *vars-term l*)
unfolding *check-varcond-subset-def* **by** *force+*

definition *check-varcond-no-Var-lhs* =
check-allm (λ*rule*.
check (*is-Fun* (*fst rule*))
(*showsl* (*STR ''variable left−hand side in rule ''*) $\circ$ *showsl-rule rule* $\circ$ *showsl-nl*))

lemma *check-varcond-no-Var-lhs* [*simp*]:
isOK (*check-varcond-no-Var-lhs R*) $\longleftrightarrow$ ($\forall$ (*l*, *r*) $\in$ *set R*. *is-Fun l*)
by (*auto simp*: *check-varcond-no-Var-lhs-def*)

definition *check-wf-trs* :: (-,-) *rules* $\Rightarrow$ *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*$\boxed{\hookleftarrow}$''') $\circ$ *e*)

lemma *check-wf-trs-conf* [*simp*]:
isOK (*check-wf-trs R*) = *wf-trs* (*set R*)
by (*force simp*: *check-wf-trs-def wf-trs-def*)

definition *check-not-wf-trs* :: (-,-) *rules* $\Rightarrow$ *showsl check* **where**
check-not-wf-trs R = *check* ($\neg$ *isOK* (*check-wf-trs R*)) (*showsl* (*STR ''The TRS is well formed*$\boxed{\hookleftarrow}$''))

lemma *check-not-wf-trs*:
assumes *isOK*(*check-not-wf-trs R*)
shows $\neg$ *SN* (*rstep* (*set R*))
proof −
from *assms* **have** $\neg$ *wf-trs* (*set R*) **unfolding** *check-not-wf-trs-def* **by** *auto*

with *SN-rstep-imp-wf-trs* **show** *?thesis* **by** *auto*
qed

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
assume *?l*
then obtain σ **where** *fst lr* = *fst st* · σ
and *snd lr* = *snd st* · σ **by** (*auto simp*: *instance-rule-def*)
then have ⋀*l t*. (*l*, *t*) ∈ *set ?list* ⟹ *l* · σ = *t* **by** (*auto*)
then have *matchers* (*set ?list*) ≠ {} **by** (*auto simp*: *matchers-def*)
with *match-list-complete*
show *match-list ?d ?list* ≠ *None* **by** *blast*
next
assume *match-list ?d ?list* ≠ *None*
then obtain τ **where** *match-list ?d ?list* = *Some* τ **by** *auto*
from *match-list-sound* [*OF this*]
have *fst lr* = *fst st* · τ **and** *snd lr* = *snd st* · τ **by** *auto*
then show *?l* **by** (*auto simp*: *instance-rule-def*)
qed

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
assume *?l*
show *?r*
proof
fix *x y*
assume (*x*,*y*) ∈ *subst.closure* (*set R*)
then show (*x*,*y*) ∈ *subst.closure* (*set S*)
proof (*induct*)
case (*subst s t* σ)
with ‹*?l*›[*unfolded check-CS-subseteq-def*]
obtain *l r* δ **where** *lr*: (*l*,*r*) ∈ *set S* **and** *s*: *s* = *l* · δ **and** *t*: *t* = *r* · δ
by (*auto simp add*: *instance-rule-def*)
show *?case* **unfolding** *s t*
using *subst.closure.intros*[*OF lr*, *of* δ ∘$_s$ σ]
by *auto*
qed
qed

next
assume *?r*
{
fix *lr*
assume *mem*: *lr* ∈ *set R*
obtain *l r* **where** *lr*: *lr* = (*l*,*r*) **by** (*cases lr*, *auto*)
with *mem* **have** (*l*,*r*) ∈ *subst.closure* (*set R*) **using** *subst.subset-closure* **by** *auto*
with ‹*?r*› **have** (*l*,*r*) ∈ *subst.closure* (*set S*) **by** *auto*
then have *Bex* (*set S*) (*instance-rule lr*) **unfolding** *lr*
proof (*induct*)
case (*subst s t σ*)
then show *?case* **unfolding** *instance-rule-def* **by** *force*
qed
}
thus *?l* **unfolding** *check-CS-subseteq-def* **by** *auto*
qed

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* **unfolding** *reverse-rules-def* **by** *force*

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*)
unfolding *map-funs-rules-wa-def map-funs-trs-wa-def* **by** *auto*

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*)))
unfolding *wf-rule-def* **by** *auto*

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*)
unfolding *wf-rules-impl-def wf-rules-def* **by** *auto*

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*)
by (*cases R*) *auto*

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*)
unfolding *check-linear-trs-def linear-trs-def* **by** *auto*

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*)
unfolding *non-collapsing-impl-def non-collapsing-def list-all-iff* **by** *auto*

type-synonym ('*f*, '*v*) *term-map* = '*f* × *nat* ⇒ ('*f*, '*v*) *term list*

definition *term-map* :: ('*f*::*compare-order*, '*v*) *term list* ⇒ ('*f*, '*v*) *term-map* **where**
term-map ts = *fun-of-map* (*rm.α* (*elem-list-to-rm* (*the* ∘ *root*) *ts*)) []

definition
is-NF-main :: *bool* ⇒ *bool* ⇒ ('*f*::*compare-order*, '*v*) *term-map* ⇒ ('*f*, '*v*) *term*
⇒ *bool*
where
is-NF-main var-cond R-empty m = (*if var-cond then* (λ-. *False*)
else if R-empty then (λ-. *True*)
else (λ*t*. ∀ *u*∈*set* (*supteq-list t*).
if is-Fun u then
∀ *l*∈*set* (*m* (*the* (*root u*))). ¬ *matches u l*
else True))

lemma *neq-root-no-match*:
assumes *is-Fun l* **and** *the* (*root l*) ≠ *the* (*root t*)
shows ¬ *matches t l*
using *assms* **by** (*cases t*) (*force iff*: *matches-iff*)+

lemma *all-not-conv*: $(\forall\, x \in A.\ \neg\ P\ x) = (\neg\ (\exists\, x \in A.\ P\ x))$ **by** *auto*

lemma *efficient-supteq-list-do-not-match*:
assumes $\forall\, l{\in}set\ ls.\ \forall\, u{\in}set\ (supteq\text{-}list\ t).\ the\ (root\ l) \neq the\ (root\ u) \longrightarrow \neg\ matches\ u\ l$
shows
$(\forall\, l{\in}set\ ls.\ \forall\, u{\in}set\ (supteq\text{-}list\ t).\ \neg\ matches\ u\ l) \longleftrightarrow$
$(\forall\, u{\in}set\ (supteq\text{-}list\ t).\ \forall\, l{\in}set\ (term\text{-}map\ ls\ (the\ (root\ u))).$
$\neg\ matches\ u\ l)$
(**is** *?lhs* $\longleftrightarrow$ *?rhs* **is** - $\longleftrightarrow$ $(\forall\, u{\in}set\ ?subs.\ \forall\, l{\in}set\ (?ls\ u).\ \neg\ matches\ u\ l)$)
proof (*intro iffI ballI*)
fix *u l* **assume** $\forall\, l{\in}set\ ls.\ \forall\, u{\in}set\ ?subs.\ \neg\ matches\ u\ l$ **and** $u \in set\ ?subs$
and $l \in set\ (?ls\ u)$
then show $\neg\ matches\ u\ l$
using *elem-list-to-rm.rm-set-lookup*[*of the* $\circ$ *root ls the* (*root u*)]
by (*auto simp*: *o-def term-map-def rm-set-lookup-def*)
next
fix *l u* **assume** *1*: $\forall\, u{\in}set\ ?subs.\ \forall\, l{\in}set\ (?ls\ u).\ \neg\ matches\ u\ l$
and $l \in set\ ls$ **and** $u \in set\ ?subs$
with *assms* **have** $the\ (root\ l) \neq the\ (root\ u) \longrightarrow \neg\ matches\ u\ l$
and $\forall\, l{\in}set\ (?ls\ u).\ \neg\ matches\ u\ l$ **by** *simp+*
with *elem-list-to-rm.rm-set-lookup*[*of the* $\circ$ *root ls the* (*root u*)]
and $\langle l \in set\ ls\rangle$
show $\neg\ matches\ u\ l$ **by** (*auto simp*: *o-def term-map-def rm-set-lookup-def*)
qed

lemma *supteq-list-ex*:
$(\exists\, u{\in}set\ (supteq\text{-}list\ l).\ \exists\, \sigma.\ t \cdot \sigma = u) \longleftrightarrow (\exists\, \sigma.\ l \trianglerighteq t \cdot \sigma)$
unfolding *supteq-list* **by** *auto*

definition *is-NF-trs R* = *is-NF-main* $(\exists\, r{\in}set\ R.\ is\text{-}Var\ (fst\ r))$ $(R = [])$ (*term-map* (*map fst R*))
definition *is-NF-terms Q* = *is-NF-main* $(\exists\, q{\in}set\ Q.\ is\text{-}Var\ q)$ $(Q = [])$ (*term-map Q*)

lemma *is-NF-main-NF-trs-conv*:
is-NF-main $(\exists\, r{\in}set\ R.\ is\text{-}Var\ (fst\ r))$ $(R = [])$ (*term-map* (*map fst R*)) $t \longleftrightarrow$
$t \in NF\text{-}trs\ (set\ R)$
(**is** *is-NF-main ?var ?R ?map t* $\longleftrightarrow$ -)
proof (*intro iffI allI*)
assume *is-NF-main*: *is-NF-main ?var ?R ?map t*
show $t \in NF\text{-}trs\ (set\ R)$
proof (*cases* $\exists\, r{\in}set\ R.\ is\text{-}Var\ (fst\ r)$)
case *True* **with** *is-NF-main*[*unfolded is-NF-main-def*] **show** *?thesis* **by** *simp*
next
case *False*
let *?ts* = *map fst R*
from *False* **have** *allfun*: $\forall\, s{\in}set\ ?ts.\ is\text{-}Fun\ s$ **by** *simp*
with *neq-root-no-match*

```
    have no-match: ∀ s∈set ?ts. ∀ u∈set (supteq-list t). the (root s) ≠ the (root u)
        ⟶ ¬ matches u s by blast
    note is-NF-main = is-NF-main[unfolded is-NF-main-def if-not-P[OF False]]
    show ?thesis
    proof (cases R = [])
      case False
      then have False: (R = []) = False by simp
      have ∀ u∈set (supteq-list t). ∀ l∈set (term-map ?ts (the (root u))). ¬ matches
u l
      proof
        fix u
        assume mem: u ∈ set (supteq-list t)
        show ∀ l∈set (term-map ?ts (the (root u))). ¬ matches u l
        proof (cases u)
          case (Var x)
          show ?thesis
          proof
            fix l
            assume l ∈ set (term-map ?ts (the (root u)))
            with elem-list-to-rm.rm-set-lookup[of the ∘ root ?ts the (root u)]
            have l ∈ set ?ts by (auto simp: o-def term-map-def rm-set-lookup-def)
            then have is-Fun l using allfun by auto
            then have (∀ σ. l · σ ≠ u) using Var by auto
            then show ¬ matches u l using matches-iff by blast
          qed
        next
          case (Fun f us)
          with mem is-NF-main[unfolded False] show ?thesis by auto
        qed
      qed
      then show ?thesis
        unfolding efficient-supteq-list-do-not-match[OF no-match, symmetric]
        unfolding all-not-conv matches-iff
        unfolding supteq-list-ex by auto
    qed auto
  qed
next
  assume NF-trs: t ∈ NF-trs (set R)
  show is-NF-main (∃ r∈set R. is-Var (fst r)) (R = []) (term-map (map fst R)) t
  proof (cases ∃ r∈set R. is-Var (fst r))
    case True
    then obtain l where l ∈ lhss (set R) and is-Var l by auto
    from lhs-var-not-NF[OF this] and NF-trs show ?thesis by simp
  next
    case False note oFalse = this
    let ?ts = map fst R
    from False have ∀ s∈set ?ts. is-Fun s by auto
    with neq-root-no-match
    have
```

```
    no-match: ∀ s∈set ?ts. ∀ u∈set (supteq-list t). the (root s) ≠ the (root u)
    ⟶ ¬ matches u s by blast
  show ?thesis
  proof (cases R = [])
    case True then show ?thesis unfolding is-NF-main-def by auto
  next
    case False
    then have False: (R = []) = False by simp
    from NF-trs
    show ?thesis
      unfolding is-NF-main-def False if-False if-not-P[OF oFalse]
      using efficient-supteq-list-do-not-match[OF no-match, symmetric]
      unfolding all-not-conv matches-iff
      unfolding supteq-list-ex set-map by fastforce
  qed
 qed
qed

lemma is-NF-trs [simp]:
  is-NF-trs R = (λ t. t ∈ NF-trs (set R))
  by (rule ext, unfold is-NF-trs-def is-NF-main-NF-trs-conv, simp)

lemma is-NF-terms [simp]:
  is-NF-terms Q = (λ t. t ∈ NF-terms (set Q))
proof (rule ext)
  fix t
  let ?Q = map (λt. (t, t)) Q
  have 1: (∃ t∈set Q. is-Var t) = (∃ t∈set ?Q. is-Var (fst t))
    by (induct Q) (auto simp: o-def)
  have 2: map fst ?Q = Q by (induct Q) simp-all
  have 3: term-map Q = term-map (map fst ?Q) unfolding 2 ..
  have 4: set ?Q = Id-on (set Q) by (induct Q) (auto simp: o-def)
  from is-NF-main-NF-trs-conv[of ?Q t]
  show is-NF-terms Q t = (t ∈ NF-terms (set Q))
    unfolding is-NF-terms-def 1 3 4 unfolding 2 by auto
qed
```

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 com-
puteRuleMapH 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 (λf.
      (case map-of rm f of
        None ⇒ []
      | Some rls ⇒ rls)))

lemma computeRuleMapHSound2: (computeRuleMapH R = None) = (∃ (l, r) ∈
set R. root l = None)
proof (induct R)
  case (Cons rule rules)
  obtain l r where rule: rule = (l,r) by force
  show ?case
  proof (cases l)
    case (Fun f ts)
    show ?thesis
      using rule Cons
    proof (cases computeRuleMapH rules)
      case (Some rm) note oSome = this
      let ?e = List.extract (λ (fa,rls). fa = (f,length ts)) rm
      from Some Fun rule Cons show ?thesis
      proof (cases ?e)
        case (Some res)
       then obtain bef aft g rls where ?e = Some (bef, (g,rls), aft) by (cases res,
force)
        with extract-SomeE[OF this] have rm: rm = bef @ ((f, length ts),rls) #
aft and e: ?e = Some (bef, ((f,length ts),rls), aft)
          by auto
        show ?thesis using Cons
          by (simp add: rule Fun Let-def oSome e)
      qed auto
    qed (insert, auto simp: rule Fun)
  qed (auto simp: rule)
qed (auto simp: rule)

lemma computeRuleMapSound2: (computeRuleMap R = None) = (∃ (l, r) ∈ set
R. root l = None)
  unfolding computeRuleMap-def
  by (simp only: computeRuleMapHSound2[symmetric], cases computeRuleMapH
R, auto)


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
  using assms
proof (induct R arbitrary: rm)
  case (Cons rule rules)
  let ?setl = λ rm. (λ (f,rls). (f,set rls)) ` set rm
  let ?setr = λ R. {((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)} ≠ {}}
  obtain l r where Pair: rule = (l,r) by force
  show ?case
  proof (cases l)
    case (Var v)
    with Cons Pair show ?thesis by simp
  next
    case (Fun f ts)
    with Cons Pair show ?thesis
    proof (cases computeRuleMapH rules)
      case (Some rrm) note oSome = this
      let ?dis = distinct-eq (λ (f,rls) (g,rls'). f = g)
      from Cons(1)[OF Some] have drrm: ?dis rrm and srrm: ?setl rrm = ?setr
rules by auto
      show ?thesis
      proof (cases List.extract (λ (fa,rls). fa = (f,length ts)) rrm)
        case None
        let ?e = ((f,length ts), [(Fun f ts,r)])
        let ?e' = ((f,length ts), {(Fun f ts,r)})
        from None Cons(2) have rm: rm = ?e # rrm by (simp add: Fun Pair
Some None)
        from None[unfolded extract-None-iff] have rrm: ⋀ g n rl. ((g,n),rl) ∈ set
rrm ⟹ (f,length ts) ≠ (g,n) by auto
        then have rrm': ⋀ g n rl. ((g,n),rl) ∈ ?setr rules ⟹ (f,length ts) ≠ (g,n)
by (simp only: srrm[symmetric], auto)
        then have id: {(Fun f ts, r)} = {(l, ra). (l = Fun f ts ∧ ra = r ∨ (l, ra)
∈ set rules) ∧ root l = Some (f, length ts)} by force
        from rrm have dis: ?dis rm
          by (simp add: rm drrm, auto)
        have ?setl rrm = insert ?e' (?setl rrm) by (simp add: rm)
        also have ... = insert ?e' (?setr rules) by (simp add: srrm)
        also have ... = ?setr ( (Fun f ts,r) # rules)
        proof (rule set-eqI, clarify)
          fix g n rls
          show (((g,n),rls) ∈ insert ?e' (?setr rules)) = (((g,n),rls) ∈ ?setr ((Fun f
ts,r) # rules))
          proof (cases (g,n) = (f,length ts))
            case False
            then have (((g,n),rls) ∈ insert ?e' (?setr rules)) = (((g,n),rls) ∈ ?setr
rules) by auto
            also have ... = (((g,n),rls) ∈ ?setr ((Fun f ts,r) # rules)) using False
by auto
```

```
          finally show ?thesis .
        next
          case True note oTrue = this
          show ?thesis
          proof (cases rls = {(Fun f ts, r)})
            case True
            with oTrue show ?thesis by (simp add: id, force)
          next
            case False
             show ?thesis using rrm'[of g n rls] True False by (simp add: False
True id, auto)
          qed
        qed
      qed
      finally show ?thesis
        by (simp only: dis drrm, simp add: Pair Fun)
    next
      case (Some res)
      obtain bef fg rls aft where res = (bef,(fg,rls),aft) by (cases res, force)
       from extract-SomeE[OF Some[simplified this]] Some[simplified this] have
rrm: rrm = bef @ ((f,length ts), rls) # aft
          and e: List.extract (λ (fa, rls). fa = (f,length ts)) rrm = Some (bef,
((f,length ts),rls),aft) by auto
      let ?e = ((f,length ts), (Fun f ts,r) # rls)
      let ?e' = ((f,length ts), insert (Fun f ts,r) (set rls))
      have ((f,length ts),set rls) ∈ ?setl rrm unfolding rrm by auto
      then have rls: set rls = {(l, r) |l r. (l, r) ∈ set rules ∧ root l = Some (f,
length ts)} using Cons(1)[OF oSome] by auto
      obtain ba where ba: ba = bef @ aft by auto
     from Cons(2) e ba have rm: rm = ?e # ba by (simp add: Fun Pair oSome
e)
      from drrm[simplified rrm] have dis: ?dis ba  unfolding distinct-eq-append
ba by auto
    from drrm[simplified rrm] have dis: ?dis rm unfolding rm distinct-eq-append
ba by (auto simp: dis[simplified ba])
      from drrm[simplified rrm distinct-eq-append]
     have diff: (∀ x∈set ba. ¬ (λ(g, rls). (f,length ts) = g) x) by (auto simp: ba)
     have ?setl [((f, length ts),rls)] ∪ ?setl ba = ?setl rrm using rrm ba by auto
      also have ... = ?setr rules by (rule srrm)
      finally have id: ?setl [((f, length ts),rls)] ∪ ?setl ba = ?setr rules .
      have ?setl rm = insert ?e' (?setl ba) by (simp add: rm)
      also have ... = ?setr ((Fun f ts,r) # rules)
      proof (rule set-eqI, clarify)
        fix g n rl
       show (((g,n),rl) ∈ insert ?e' (?setl ba)) = (((g,n),rl) ∈ ?setr ((Fun f ts,r)
# rules))
        proof (cases (g,n) = (f,length ts))
          case False
           then have (((g,n),rl) ∈ insert ?e' (?setl ba)) = (((g,n),rl) ∈ ?setl ba)
```

```
by auto
          also have ... = (((g,n),rl) ∈ ?setr rules) using False by (simp only:
id[symmetric], auto)
          also have ... = (((g,n),rl) ∈ ?setr ((Fun f ts,r) # rules)) using False
by auto
          finally show ?thesis .
        next
          case True note oTrue = this
          show ?thesis
          proof (cases rl = insert (Fun f ts, r) (set rls))
            case True
            then have (((g,n),rl) ∈ insert ?e' (?setl ba)) = True using oTrue by
auto
            also have ... = (((g,n),rl) ∈ ?setr ((Fun f ts,r) # rules)) unfolding
True rls using oTrue by force
            finally show ?thesis .
          next
            case False
             then have (((g,n),rl) ∈ insert ?e' (?setl ba)) = False using diff by
(simp add: True, auto)
            also have ... = (((g,n),rl) ∈ ?setr ((Fun f ts,r) # rules))
            proof (rule ccontr)
              assume ¬ ?thesis
               with True have ((f,length ts),rl) ∈ ?setr ((Fun f ts,r) # rules) by
simp
               then have rl = {(l, ra) |l ra. (l, ra) ∈ set ((Fun f ts, r) # rules) ∧
root l = Some (f, length ts)} by simp
              with False rls show False by auto
            qed
            finally show ?thesis .
          qed
        qed
      qed
      also have ... = ?setr (rule # rules) by (simp add: Pair Fun)
      finally show ?thesis by (simp add: dis)
    qed
  qed simp
 qed
qed force

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 (cases computeRuleMapH R)
 case None
 then show ?thesis using assms unfolding computeRuleMap-def by auto
next
 case (Some rrm)
 note rrm = computeRuleMapHSound[OF this]
```

note *rm* = *assms*[*unfolded computeRuleMap-def*, *simplified Some*, *simplified*, *symmetric*]
show *?thesis*
proof (*cases map-of rrm* (*f*, *n*))
case (*Some rls*)
from *map-of-SomeD*[*OF this*] **have** ((*f*,*n*),*set rls*) ∈ (λ (*f*,*rls*). (*f*, *set rls*)) ‘ *set rrm*
by *auto*
then have *set rls* = {(*l*,*r*) | *l r*. (*l*,*r*) ∈ *set R* ∧ *root l* = *Some* (*f*, *n*)}
by (*simp only*: *rrm*, *simp*)
then show *?thesis* **by** (*simp add*: *rm Some*)
next
case *None*
have *id*: {(*l*, *r*) |*l r*. (*l*, *r*) ∈ *set R* ∧ *root l* = *Some* (*f*, *n*)} = {} (**is** *?set* = {})
proof (*rule ccontr*)
assume ¬ *?thesis*
then obtain *l r* **where** (*l*, *r*) ∈ *set R* ∧ *root l* = *Some* (*f*, *n*) **by** *auto*
with *rrm* **have** (((*f*,*n*), *?set*)) ∈ (λ (*f*,*rls*). (*f*, *set rls*)) ‘ *set rrm* **by** *auto*
with *None*[*unfolded map-of-eq-None-iff*] **show** *False* **by** *force*
qed
then show *?thesis* **by** (*simp only*: *rm None id*, *auto*)
qed
qed

lemma *computeRuleMap-left-vars*:
shows (*computeRuleMap R* ≠ *None*) = (∀ *lr* ∈ *set R*. ∀ *x*. *fst lr* ≠ *Var x*)
proof (*cases computeRuleMap R*)
case *None*
from *None computeRuleMapSound2* **have** ∃ (*l*,*r*) ∈ *set R*. *root l* = *None* **by** *auto*
from *this* **obtain** *l r* **where** (*l*,*r*) ∈ *set R* ∧ *root l* = *None* **by** *auto*
from *this* **have** (*l*,*r*) ∈ *set R* ∧ ¬ (∀ *x*. *fst* (*l*,*r*) ≠ *Var x*) **by** (*cases l*, *auto*)
with *None* **show** *?thesis* **by** *blast*
next
case (*Some rm*)
then have *left*: *computeRuleMap R* ≠ *None* **by** *auto*
from *Some computeRuleMapSound2* **have** ∀ (*l*,*r*) ∈ *set R*. *root l* ≠ *None* **by** *force*
then have ∀ *lr* ∈ *set R*. ∀ *x*. *fst lr* ≠ *Var x* **by** *auto*
with *left* **show** *?thesis* **by** *blast*
qed

lemma *computeRuleMap-defined*: **fixes** *R* :: ('*f*,'*v*)*rules*
assumes *computeRuleMap R* = *Some rm*
shows (*rm* (*f*,*n*) = []) = (¬ *defined* (*set R*) (*f*,*n*))
proof −
from *assms computeRuleMapSound* **have** *rm*: ⋀(*f*::'*f*) *n*. *set* (*rm* (*f*,*n*)) = {(*l*,*r*)

| *l r*. (*l*, *r*) ∈ *set R* ∧ *root l* = *Some* (*f*, *n*)} **by** *force*
show *?thesis*
proof (*cases rm* (*f*,*n*))
case *Nil*
with *rm* **have** ¬ *defined* (*set R*) (*f*,*n*) **unfolding** *defined-def* **by** *force*
with *Nil* **show** *?thesis* **by** *blast*
next
case (*Cons lr RR*)
then have *left*: *rm* (*f*,*n*) ≠ [] **by** *auto*
from *Cons rm*[**where** *f* = *f* **and** *n* = *n*] **have** *defined* (*set R*) (*f*,*n*) **unfolding** *defined-def* **by** (*cases lr*, *force*)
with *left* **show** *?thesis* **by** *blast*
qed
qed

lemma *computeRuleMap-None-not-SN*:
assumes *computeRuleMap R* = *None*
shows ¬ *SN-on* (*rstep* (*set R*)) {*t*}
proof −
from *assms computeRuleMap-left-vars*[*of R*] **obtain** *x r* **where** (*Var x*,*r*) ∈ *set R* **by** *auto*
from *left-var-imp-not-SN*[*OF this*] **show** *?thesis* **.**
qed

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)
  by auto

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
  let ?Prop = λ s t. s = t ∨ vars-term t ∩ V = {} ∧ (s,t) ∈ rrstep (set R)
  {
    fix s t
    assume ?Prop s t
    hence ∃ C infos. (s, t) ∈ par-rstep-mctxt (set R) C infos ∧ vars-below-hole t
C ∩ V = {}
    proof
      assume s = t
        thus ?thesis by (intro exI[of - mctxt-of-term s] exI[of - Nil], auto simp:
par-rstep-mctxt-reflI)
    next
      assume vars-term t ∩ V = {} ∧ (s,t) ∈ rrstep (set R)
      then obtain l r σ where id: s = l · σ t = r · σ and
        lr: (l,r) ∈ set R and
        vars: vars-term t ∩ V = {}
        by (metis rrstepE)
     thus ?thesis by (intro exI[of - MHole] exI[of - [Par-Info s t (l,r)]], auto intro:
par-rstep-mctxt-MHoleI)
    qed
  } note Prop = this
  {
    assume is-par-rstep-var-restr s t
    hence ∃ C infos. (s, t) ∈ par-rstep-mctxt (set R) C infos ∧ vars-below-hole t
C ∩ V = {}
    proof (induct rule: is-par-rstep-var-restr.induct[])
      case 2-1
      thus ?case by (intro Prop, auto)
    next
      case 2-2
      thus ?case by (intro Prop, auto)
    next
      case (1 f ss g ts)
      show ?case
      proof (cases ?Prop (Fun f ss) (Fun g ts))
        case True
        thus ?thesis using Prop by auto
      next
```

```
      case False
      with 1 have args: f = g length ss = length ts list-all2 is-par-rstep-var-restr
ss ts
        by (auto split: if-splits)
       let ?P = λ i C infos. (ss ! i, ts ! i) ∈ par-rstep-mctxt (set R) C infos ∧
vars-below-hole (ts ! i) C ⊆ (UNIV − V)
      { fix i
        assume i:i < length ss
        then have si:ss ! i ∈ set ss by auto
        from i args(2) have ti:ts ! i ∈ set ts by auto
          from args(3) have iprv:is-par-rstep-var-restr (ss ! i) (ts ! i) using i
list-all2-nthD by blast
        with 1(1)[of ss!i ts!i] have pp:∃ C infos. ?P i C infos
          using local.args(1) local.args(2) using si ti by blast
      }
      hence ∀ i. ∃ C infos. i < length ss ⟶ ?P i C infos by blast
      from choice[OF this] obtain C where ∀ i. ∃ infos. i < length ss ⟶ ?P
i (C i) infos by blast
      from choice[OF this] obtain infos where IH: ⋀ i. i < length ss ⟹ ?P i
(C i) (infos i) by blast
      let ?C = MFun f (map C [0..<length ss])
      let ?infos = concat (map infos [0..<length ss])
      show ?thesis
      proof (intro exI[of - ?C] exI[of - ?infos] conjI)
       show vars-below-hole (Fun g ts) ?C ∩ V = {} using IH args(2) unfolding
args(1)
         by (subst vars-below-hole-Fun; force)
        show (Fun f ss, Fun g ts) ∈ par-rstep-mctxt (set R) ?C ?infos unfolding
args(1) using args(2) IH
         by (intro par-rstep-mctxt-funI, auto)
      qed
    qed
  qed
  thus (s, t) ∈ par-rstep-var-restr (set R) V unfolding par-rstep-var-restr-def
by auto
 }
 {
  assume (s, t) ∈ par-rstep-var-restr (set R) V
  from this[unfolded par-rstep-var-restr-def] obtain C infos where
    st: (s, t) ∈ par-rstep-mctxt (set R) C infos and vars: vars-below-hole t C ∩
V = {} by auto
  thus is-par-rstep-var-restr s t
  proof (induct C arbitrary: s t infos)
    case (MVar x)
    from par-rstep-mctxt-MVarE[OF MVar(1)]
    have s = Var x t = Var x by auto
    thus ?case by simp
  next
    case MHole
```

```
      from par-rstep-mctxt-MHoleE[OF MHole(1)]
      have (s,t) ∈ rrstep (set R) by auto
      then show ?case using MHole(2) by (cases s; cases t; auto)
    next
      case (MFun f Cs)
      from par-rstep-mctxt-MFunD[OF MFun(2)]
      obtain ss ts Infos
        where s: s = Fun f ss and
          t: t = Fun f ts and
          len: length ss = length Cs
          length ts = length Cs
          length Infos = length Cs and
          infos: infos = concat Infos and
          steps: ⋀ i. i < length Cs ⟹(ss ! i, ts ! i) ∈ par-rstep-mctxt (set R) (Cs !
i) (Infos ! i)
        by auto
      show ?case unfolding s t is-par-rstep-var-restr.simps
      proof (intro disjI2 conjI refl list-all2-all-nthI, unfold len)
        fix i
        assume i: i < length Cs
        hence mem: Cs ! i ∈ set Cs by auto
        show is-par-rstep-var-restr (ss ! i) (ts ! i)
        proof (rule MFun(1)[OF mem steps[OF i]])
          have vars-below-hole (ts ! i) (Cs ! i) ⊆ vars-below-hole t (MFun f Cs)
            unfolding t using i len
            by (subst vars-below-hole-Fun, auto)
          with MFun(3) show vars-below-hole (ts ! i) (Cs ! i) ∩ V = {} by auto
        qed
      qed auto
    qed
  }
qed
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
  by simp
```

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 −
have *is-par-rstep R s t* = *is-par-rstep-var-restr R* {} *s t*
by (*induct R s t rule*: *is-par-rstep.induct*, *auto simp del*: *is-par-rstep-var-restr simp*: *list-all2-conv-all-nth*)
also have ... ⟷ (*s*, *t*) ∈ *par-rstep-var-restr* (*set R*) {} **by** *simp*
also have ... ⟷ (*s*, *t*) ∈ *par-rstep* (*set R*)
unfolding *par-rstep-var-restr-def par-rstep-par-rstep-mctxt-conv* **by** *auto*
finally show *?thesis* **.**
qed

lemma *par-rstep-code*[*code-unfold*]: (*s*, *t*) ∈ *par-rstep* (*set R*) ⟷ *is-par-rstep R s t* **by** *simp*

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 −
from *assms*
have ∃ *l r*. (*l*,*r*) ∈ *set R* ∧ *t* ∈ *set* (*case match s l of None* ⇒ [] | *Some* σ ⇒ [*r* · σ])
by *auto*
from *this* **obtain** *l r* **where** *one*:
(*l*,*r*) ∈ *set R* ∧ *t* ∈ *set* (*case match s l of None* ⇒ [] | *Some* σ ⇒ [*r* · σ])
by *auto*
from *this* **obtain** σ **where** *two*: *match s l* = *Some* σ ∧ *t* ∈ {*r* · σ} **by** (*cases match s l*, *auto*)
then have *match*: *l* · σ = *s* **using** *match-sound* **by** *auto*
with *one match one two* **have** (*s*,*t*) ∈ *rstep-r-p-s* (*set R*) (*l*,*r*) [] σ **unfolding** *rstep-r-p-s-def* **by** (*simp add*: *Let-def ctxt-supt-id*)
then show (*s*,*t*) ∈ *rrstep* (*set R*) **unfolding** *rstep-iff-rstep-r-p-s rrstep-def* **by** *blast*
qed

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* (λ*ss*. *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*)
using *assms*
proof (*induct s arbitrary*: *t*)
case (*Fun f ss*)
then consider (*root*) *t* ∈ *set* (*root-rewrite R* (*Fun f ss*))
| (*args*) *t* ∈ *set* (*map* (λ*ss*. *Fun f ss*) (*product-lists* (*map* (*parallel-rewrite R*) *ss*)))
by *force*
then show *?case*
proof (*cases*)
case *root*
from *root-rewrite-sound*[*OF this*] **obtain** *l r* σ **where** (*l*, *r*) ∈ *set R*
and *l* · σ = *Fun f ss* **and** *r* · σ = *t*
unfolding *rrstep-def rstep-r-p-s-def* **by** *auto*
then show *?thesis* **using** *par-rstep.intros*(*1*) **by** *metis*
next
case *args*
then obtain *ts* **where** *t*:*t* = *Fun f ts* **and** *ts*:*ts* ∈ *set* (*product-lists* (*map* (*parallel-rewrite R*) *ss*))
by *auto*
then have *len*:*length ss* = *length ts* **using** *in-set-product-lists-length* **by** *force*
{ fix *i*
assume *i*:*i* < *length ts*
have *ts* ! *i* ∈ *set* (*parallel-rewrite R* (*ss* ! *i*))
using *ts*[*unfolded product-lists-set*[*of - ss*]]
by (*auto simp*: *list-all2-map2*[*of* (λ*x ys*. *x* ∈ *set ys*)] *intro*: *list-all2-nthD*[*OF - i*])
with *Fun*(*1*) *len i* **have** (*ss* ! *i*, *ts* ! *i*) ∈ *par-rstep* (*set R*) **by** *auto*
}
from *par-rstep.intros*(*2*)[*OF this len*] **show** *?thesis* **using** *t* **by** *auto*
qed
qed *auto*

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−
  from assms obtain l r where lr:(l,r) ∈ set R (ss, ts) ∈ set (case match s l of
        None ⇒ []
      | Some σ ⇒ (case match t r of
          None ⇒ []
        | Some τ ⇒ [(map σ (filter (λx. x ∈ vars-term r) (vars-distinct l)), map τ
(filter (λx. x ∈ vars-term r) (vars-distinct l)))]))
    unfolding root-steps-substs.simps Let-def by auto
  let ?var-list=filter (λx. x ∈ vars-term r) (vars-distinct l)
  from lr obtain σ where σ:match s l = Some σ
    by fastforce
  from lr obtain τ where τ:match t r = Some τ
    by fastforce
  from lr σ τ have (ss, ts) = (map σ ?var-list, map τ ?var-list)
    by simp
  with lr(1) σ τ show ?thesis
    using match-matches by blast
qed

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−
  from assms(1) obtain f ts where l:l = Fun f ts
    by blast
  from match have ∗:l · τ = s
    by (simp add: match-matches)
  then obtain ss where s:s = Fun f ss
    unfolding l by force
  from assms(2) obtain i where i:i < length ts and x:x ∈ vars-term (ts!i)
    unfolding l by (metis term.sel(4) var-imp-var-of-arg)
  from ∗ have le:length ts = length ss
    unfolding s l by auto
  moreover from ∗ i l s have ts!i · τ = ss!i
    by fastforce
  then have size (τ x) ≤ size (ss!i)
    using vars-term-size x by metis
  with i show ?thesis unfolding s term.size le
   by (metis add.commute add-0 add-Suc in-set-conv-nth less-Suc-eq-le size-list-estimation′)
```

qed

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−
from *assms* **obtain** *x y* **where** *xy*:(*Var x*, *Var y*) ∈ *R x* ≠ *y* **by** *blast*
let *?σ*=(*subst x s*) (*y* := *t*)
from *xy* **have** (*?σ x*, *?σ y*) ∈ *rrstep R*
by (*metis eval-term.simps*(*1*) *rrstepI*)
then show *?thesis*
by (*simp add*: *xy*(*2*))
qed

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−
let *?R*=*remove-trivial-rules R*
from *assms*(*1*) **obtain** *l r vl σ τ* **where** *lr*:(*l*, *r*) ∈ *set ?R* **and** *vl*:*vl* = *filter* (λ*x*. *x* ∈ *vars-term r*) (*vars-distinct l*)
and *s*:*s* = *l* · *σ* **and** *t*:*t* = *r* · *τ* **and** *ss-ts*:(*ss*, *ts*) = (*map σ vl*, *map τ vl*)
using *root-steps-substs-exists* **by** *blast*
from *ss-ts assms*(*2*) **obtain** *x* **where** *s′*:*s′* = *σ x* **and** *x*:*x* ∈ *set vl*
by *auto*
with *s* **have** *s1*:*size s′* ≤ *size s*
unfolding *vl* **by** (*simp add*: *vars-term-size*)
from *ss-ts assms*(*3*) **obtain** *y* **where** *t′*:*t′* = *τ y* **and** *y*:*y* ∈ *set vl*
by *auto*
with *t* **have** *s2*:*size t′* ≤ *size t*
unfolding *vl* **by** (*simp add*: *vars-term-size*)
from *lr* **consider** ¬ *is-Var l* | ¬ *is-Var r*
by *force*
then show *?thesis* **proof**(*cases*)
case *1*
then obtain *f ls* **where** *l*:*l* = *Fun f ls*
by *blast*
from *x* **obtain** *i* **where** *i* < *length ls* **and** *x* ∈ *vars-term* (*ls*!*i*)
unfolding *vl l* **by** (*metis comp-apply filter-is-subset set-remdups set-rev set-vars-term-list subsetD term.sel*(*4*) *var-imp-var-of-arg*)
then have *s′* ◁ *s*
unfolding *s s′ l* **by** (*meson nth-mem subst-image-subterm term.set-intros*(*4*))
then have *size s′* < *size s*
by (*simp add*: *supt-size*)
then show *?thesis* **using** *s2* **by** *simp*

```
  next
    case 2
    then obtain f rs where r:r = Fun f rs
      by blast
    from y obtain i where i < length rs and y ∈ vars-term (rs!i)
      unfolding vl r using var-imp-var-of-arg by force
    then have t' ◁ t
     unfolding t t' r by (meson nth-mem subst-image-subterm term.set-intros(4))
    then have size t' < size t
      by (simp add: supt-size)
    then show ?thesis using s1 by simp
  qed
qed

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))
  by pat-completeness auto

termination proof (relation measure (λ (R, s, t). size s + size t), goal-cases)
  case (2 R f ss g ts x ls rs l r)
  then show ?case using size-root-steps-substs
    unfolding in-measure by (metis case-prod-conv)
next
  case (3 R f ss g ts s t)
  then have size s < size (Fun f ss) and size t < size (Fun g ts)
    by (simp add: elem-size-size-list-size less-Suc-eq)+
  then show ?case by simp
next
  case (4 R v t x xa y z yb)
  then show ?case using size-root-steps-substs
    unfolding in-measure by (metis case-prod-conv)
next
  case (5 R s v x xa y z yb)
  then show ?case using size-root-steps-substs
    unfolding in-measure by (metis case-prod-conv)
qed auto
```

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*
using *assms* **proof**(*induct*)
case (*args f n ss ts*)
then have *list-all2* (*is-mstep R*) *ss ts*
by (*simp add*: *list-all2-all-nthI*)
with *args* **show** *?case*
by *simp*
next
case (*rule l r σ τ*)
show *?case* **proof**(*cases* (*l* · *σ*, *r* · *τ*) ∈ *rrstep* (*set R*))
case *True*
then show *?thesis* **using** *is-mstep.simps* **by** (*metis* (*no-types*, *opaque-lifting*) *funas-term.cases*)
next
case *False*
then show *?thesis* **proof**(*cases is-Var l* ∧ *is-Var r*)
case *True*
with *False* **have** ¬ (∃ *x y*. (*Var x*, *Var y*) ∈ *set R* ∧ *x* ≠ *y*)
using *trivial-rrstep* **by** *metis*
with *True* **obtain** *x* **where** *l*:*l* = *Var x* **and** *r*:*r* = *Var x*
using *rule.hyps*(*1*) **by** *blast*
show *?thesis*
unfolding *l r* **using** *rule*(*2*) *l* **by** *simp*
next
case *False*
let *?R*=*remove-trivial-rules R*
let *?vl*=*filter* (λ*x*. *x* ∈ *vars-term r*) (*vars-distinct l*)
from *rule*(*1*) *False* **obtain** *i* **where** *i*:*i* < *length ?R ?R*!*i* = (*l*, *r*)
by (*metis* (*no-types*, *lifting*) *case-prodI2 fst-conv in-set-conv-nth mem-Collect-eq prod.sel*(*2*) *set-filter*)
obtain *σ′* **where** *sigma*:*match* (*l* · *σ*) *l* = *Some σ′* (∀ *x*∈*vars-term l*. *σ x* = *σ′ x*)
by (*meson match-complete′*)
obtain *τ′* **where** *tau*:*match* (*r* · *τ*) *r* = *Some τ′* (∀ *x*∈*vars-term r*. *τ x* = *τ′ x*)
by (*meson match-complete′*)
let *?matches*=(*map* (λ(*l′*, *r′*). *case*
match (*l* · *σ*) *l′ of None* ⇒ [] | *Some σ* ⇒ (*case match* (*r* · *τ*) *r′ of None* ⇒ []
| *Some τ* ⇒ (*let var-list* = *filter* (λ*x*. *x* ∈ *vars-term r′*) (*vars-distinct l′*)
in [(*map σ var-list*, *map τ var-list*)]))) *?R*)
have *i* < *length ?matches*
using *i*(*1*) **by** *auto*
moreover have (*map σ′ ?vl*, *map τ′ ?vl*) ∈ *set* (*?matches* ! *i*)
using *sigma*(*1*) *tau*(*1*) *i* **unfolding** *Let-def* **by** *simp*
ultimately have (*map σ′ ?vl*, *map τ′ ?vl*) ∈ *set* (*root-steps-substs ?R* (*l*·*σ*) (*r*·*τ*))
unfolding *root-steps-substs.simps* **by** (*metis* (*no-types*, *lifting*) *concat-nth concat-nth-length in-set-conv-nth*)

```
    then obtain j where j:j < length (root-steps-substs ?R (l·σ) (r·τ)) root-steps-substs
?R (l·σ) (r·τ) ! j = (map σ' ?vl, map τ' ?vl)
      by (metis in-set-idx)
      have (λ (ss, ts). list-all2 (is-mstep R) ss ts) ((root-steps-substs ?R (l·σ)
(r·τ))!j)
     proof−
      {fix i assume i:i < length ?vl
       from i have vr:?vl!i ∈ vars-term r
        using nth-mem by force
       from i have vl:?vl!i ∈ vars-term l
        using nth-mem by force
       moreover have σ' (?vl ! i) = σ (?vl ! i)
        using sigma(2) vr vl by simp
       moreover have τ' (?vl ! i) = τ (?vl ! i)
        using vl vr tau(2) by simp
       ultimately have is-mstep R (σ' (?vl !i)) (τ' (?vl !i))
        using rule(2) by force
      }
      then have list-all2 (is-mstep R) (map σ' ?vl) (map τ' ?vl)
       by (simp add: list-all2-conv-all-nth)
      then show ?thesis
       unfolding j(2) by fastforce
     qed
     then have ∗:list-ex (λ (ss, ts). list-all2 (is-mstep R) ss ts) (root-steps-substs
?R (l·σ) (r·τ))
      using j by (meson list-ex-length)
     then show ?thesis
      by (smt (verit) is-mstep.elims(3))
    qed
  qed
qed simp

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−
  let ?R=(remove-trivial-rules R)
  from assms obtain i where i < length (root-steps-substs ?R s t) (λ (ss, ts).
list-all2 (is-mstep R) ss ts) ((root-steps-substs ?R s t)!i)
    using list-ex-length by blast
  then obtain ss' ts' where ss'ts':(ss' , ts') ∈ set (root-steps-substs ?R s t) list-all2
(is-mstep R) ss' ts'
    using nth-mem by fastforce
  with root-steps-substs-exists obtain l r vl σ τ where lr:(l, r) ∈ set R
    and vl:vl = filter (λx. x ∈ vars-term r) (vars-distinct l)
    and l:l · σ = s and r:r · τ = t
```

```
    and στ:(ss', ts') = (map σ vl, map τ vl)
    by (smt (verit, best) mem-Collect-eq set-filter)
  let ?τ=λx. (if x ∈ vars-term r then τ x else σ x)
  from r have r':r · ?τ = t
    by (smt (verit, del-insts) term-subst-eq)
  { fix x assume x:x ∈ vars-term l
    then have (σ x, ?τ x) ∈ mstep (set R) proof(cases x ∈ set vl)
      case True
      then obtain i where i:i < length vl vl ! i = x
        using in-set-idx by force
      then have i1:i < length ss'
        using στ by simp
      from i have i2:i < length ts'
        using στ by simp
      from ss'ts'(2) i1 i2 have is-mstep R (ss' ! i) (ts' ! i)
        using list-all2-nthD by blast
      with assms(2)[OF ss'ts'(1)] i1 i2 have (ss' ! i, ts' ! i) ∈ mstep (set R)
        by auto
      then show ?thesis
        using i στ by auto
    next
      case False
      with vl x show ?thesis by simp
    qed
  }
  then show ?thesis
    using mstep.rule[OF lr] l r' by force
qed

lemma is-mstep-mstep:
  assumes is-mstep R s t
  shows (s, t) ∈ mstep (set R)
  using assms proof (induct rule: is-mstep.induct)
  case (1 R f ss g ts)
  from 1 consider Fun f ss = Fun g ts
    | (rrstep) (Fun f ss, Fun g ts) ∈ rrstep (set R)
   | (root) 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))
    | (args) f = g and length ss = length ts and list-all2 (is-mstep R) ss ts
    by (auto split: if-splits)
  then show ?case proof(cases)
    case root
    show ?thesis using mstep-root-helper[OF root] 1(1) by simp
  next
    case args
    { fix i
      assume i:i < length ss
      then have si:ss ! i ∈ set ss by auto
      from i args(2) have ti:ts ! i ∈ set ts by auto
```

```
    from args(3) have is-mstep R (ss ! i) (ts ! i) using i list-all2-nthD by blast
    with 1(2)[of ss ! i ts ! i] args(1,2) si ti have (ss ! i, ts ! i) ∈ mstep (set R)
      by auto
   }
   then show ?thesis using args(1,2)
    by (simp add: mstep.args)
 qed (simp-all add: rrstep-imp-rstep rstep-imp-mstep)
next
 case (2-1 R v t)
 from 2-1 consider Var v = t
  | (Var v, t) ∈ rrstep (set R)
  | (root) list-ex (λ (ss, ts). list-all2 (is-mstep R) ss ts) (root-steps-substs (remove-trivial-rules
R) (Var v) t)
   by auto
 then show ?case proof(cases)
   case root
   show ?thesis using mstep-root-helper[OF root] 2-1(1) by simp
 qed (simp-all add: rrstep-imp-rstep rstep-imp-mstep)
next
 case (2-2 R s v)
 from 2-2 consider s = Var v
  | (s, Var v) ∈ rrstep (set R)
  | (root) list-ex (λ (ss, ts). list-all2 (is-mstep R) ss ts) (root-steps-substs (remove-trivial-rules
R) s (Var v))
   by auto
 then show ?case proof(cases)
   case root
   show ?thesis using mstep-root-helper[OF root] 2-2(1) by simp
 qed (simp-all add:rrstep-imp-rstep rstep-imp-mstep)
qed

lemma is-mstep[simp]:
 is-mstep R s t ⟷ (s, t) ∈ mstep (set R)
 using is-mstep-mstep mstep-is-mstep by blast

lemma mstep-code[code-unfold]: (s, t) ∈ mstep (set R) ⟷ is-mstep R s t by simp
```

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 $(r, ss) \in set\ (root\text{-}subst\text{-}with\text{-}rhs\ R\ s)$
shows $\exists\ l\ \sigma.\ (l, r) \in set\ R \wedge l \cdot \sigma = s \wedge ss = map\ \sigma\ (vars\text{-}distinct\ r)$
proof−
from *assms* **obtain** l **where** *lr*:$(l,r) \in set\ R\ (r, ss) \in set\ (case\ match\ s\ l\ of$
None ⇒ []
| *Some* σ ⇒ $[(r, map\ \sigma\ (vars\text{-}distinct\ r))]$)
by *auto*
then obtain σ **where** σ:*match s l* = *Some* σ
by *fastforce*
with *lr* **show** *?thesis*
using *match-matches* **by** *force*
qed

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

private lemma ∗: *list-all* $(\lambda(l, r).\ is\text{-}Fun\ l \wedge (vars\text{-}term\ r \subseteq vars\text{-}term\ l))\ R$
using ‹*wf-trs* $(set\ R)$› **unfolding** *wf-trs-def* **by** (*auto simp*: *list-all-iff*)

lemma *varcond*:
$\bigwedge l\ r.\ (l, r) \in set\ R \Longrightarrow is\text{-}Fun\ l \wedge vars\text{-}term\ r \subseteq vars\text{-}term\ l$
using ∗ *Ball-set-list-all case-prodD* **by** (*metis* (*no-types*, *lifting*))

lemma [*termination-simp*]:
assumes $(l, r) \in set\ R$
and *Some* σ = *match* (*Fun g ts*) *l*
and $x \in vars\text{-}term\ r$
shows *size* $(\sigma\ x) <$ *Suc* (*size-list size ts*)
using *assms size-match-subst-Fun varcond*
by (*metis* (*no-types*, *lifting*) *add.right-neutral add-Suc-right subsetD term.size*(*4*))

Compute the list of terms reachable in multi-step from a given term.

fun *mstep-rewrite-main* :: $('f, 'v)\ term \Rightarrow ('f, 'v)\ term\ list$
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 $t \in set\ (mstep\text{-}rewrite\text{-}main\ s)$
shows $(s, t) \in mstep\ (set\ R)$
using *assms*
proof (*induct s arbitrary*: *t rule*:*subterm-induct*)

```
  case (subterm s)
  then show ?case proof(cases s)
    case (Var x)
    with subterm(2) show ?thesis by simp
  next
    case (Fun f ss)
    with subterm consider (root) t ∈ set (concat (map (λ(r,ts).(map (λargs. r ·
(mk-subst Var (zip (vars-distinct r) args)))
          (product-lists (map mstep-rewrite-main ts)))) (root-subst-with-rhs R (Fun
f ss))))
      | (args) t ∈ set (map (λss. Fun f ss) (product-lists (map mstep-rewrite-main
ss)))
      by force
    then show ?thesis
    proof (cases)
      case root
      then obtain r ts where rhs-subst:(r,ts) ∈ set (root-subst-with-rhs R (Fun f
ss))
          t ∈ set (map (λargs. r · (mk-subst Var (zip (vars-distinct r) args)))
(product-lists (map mstep-rewrite-main ts)))
        by force
      from root-steps-subst-rhs-exists[OF rhs-subst(1)] obtain l σ where lr:(l, r)
∈ set R
        and sigma:l · σ = Fun f ss ts = map σ (vars-distinct r) by auto
        from rhs-subst(2) obtain args where args:t = r · (mk-subst Var (zip
(vars-distinct r) args))
        args ∈ set (product-lists (map mstep-rewrite-main ts))
        by auto
      then have len:length args = length ts
        using in-set-product-lists-length by fastforce
      then have len':length args = length (vars-distinct r)
        by (simp add: sigma(2))
      let ?τ=λx. if x ∈ vars-term r then (mk-subst Var (zip (vars-distinct r) args))
x else σ x
      from args(1) have t:t = r · ?τ
        by (simp add: term-subst-eq-conv)
      { fix x
        assume x:x ∈ vars-term l
        have (σ x, ?τ x) ∈ mstep (set R) proof(cases x ∈ vars-term r)
          case True
          then obtain i where i:i < length (vars-distinct r) x = vars-distinct r ! i
            by (metis in-set-idx set-vars-term-list vars-term-list-vars-distinct)
          with True len' have tau-x:?τ x = args!i
            by (simp add: mk-subst-distinct)
          from i sigma(2) have sigma-x:σ x = ts!i
            by simp
          have σ x ◁ Fun f ss
              by (metis is-VarI lr sigma(1) subst-image-subterm term.set-cases(2)
varcond x)
```

```
        with sigma-x have ts!i ◁ Fun f ss by simp
         moreover have args!i ∈ set (mstep-rewrite-main (ts!i)) using args(2)
i(1) len' len
          unfolding product-lists-set list-all2-conv-all-nth by force
        ultimately show ?thesis using subterm(1) sigma-x tau-x unfolding Fun
by presburger
       next
        case False
        then show ?thesis by simp
       qed
      }
      then show ?thesis using mstep.intros(3)[OF lr] sigma(1) unfolding Fun t
       by fastforce
    next
     case args
      then obtain ts where t:t = Fun f ts and ts:ts ∈ set (product-lists (map
mstep-rewrite-main ss))
       by auto
      then have len:length ss = length ts using in-set-product-lists-length by force
      { fix i
       assume i:i < length ts
       have ts ! i ∈ set (mstep-rewrite-main (ss ! i))
        using ts[unfolded product-lists-set[of - ss]]
       by (auto simp: list-all2-map2[of (λx ys. x ∈ set ys)] intro: list-all2-nthD[OF
- i])
       with subterm len i have (ss ! i, ts ! i) ∈ mstep (set R)
        unfolding Fun by auto
      }
      with mstep.intros(2) len t Fun show ?thesis
       by metis
    qed
  qed
qed

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)}
  by (intro exI[of - Nil], auto simp: wf-trs-def)

setup-lifting type-definition-wfTRS

lift-definition get-TRS :: ('f, 'v) wfTRS ⇒ ('f, 'v) rules is λ R. R .

lemma is-wf-get-TRS: wf-trs (set (get-TRS R'))
  by (transfer, auto)

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*
  by (*force simp*: *wf-trs-def list.pred-set split*: *prod.splits*)

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*)
  by *simp*

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 -. [])
  by (*auto simp*: *wf-trs-def*)

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 −
  define *R'* **where** *R'* = *get-wfTRS-sub R*
  have *wf*: *wf-trs* (*set* (*get-TRS R'*))
    by (*transfer*, *auto*)
  have *sub*: *set* (*get-TRS R'*) ⊆ *set R* **unfolding** *R'-def* **by** (*transfer*, *auto*)
 from *mstep-rewrite-main-mstep*[*OF wf*, *folded mstep-rewrite-wf-def*, *OF assms*(*1*)[*unfolded mstep-rewrite-def*, *folded R'-def*]]
  have (*s*, *t*) ∈ *mstep* (*set* (*get-TRS R'*)) **.**
  with *mstep-mono*[*OF sub*] **show** *?thesis* **by** *auto*
qed

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.