Termination Restricted to Right-Forward Closures

René Thiemann[1], Dieter Hofbauer[2], Ulysse Le Huitouze[3], and Johannes Waldmann[4]

[1]University of Innsbruck, Austria
[2]ASW Saarland, Germany
[3]University of Rennes, France
[4]HTWK Leipzig, Germany

August 26, 2026

Abstract

In this AFP entry we formalize Dershowitz' theorem that termination of a term rewrite system (TRS) starting from arbitrary terms is equivalent to termination starting from terms in the right-forward closures of right-hand sides, provided that the TRS is right-linear or orthogonal [1]. Our proof deviates from the original one in that no reorderings of steps in infinite derivations are required, making it more precise in its argumentation. We also integrate a later result that one can weaken orthogonality to locally confluent overlay TRSs [2].

In order to arrive at these statements, we had to formalize two further results: Following Gramlich, we prove that for locally confluent overlay TRSs the notions of termination and innermost termination coincide [3]; and we verify that narrowing with a right-linear TRS preserves linearity of terms.

For further details, we refer to our FSCD 2026 paper [4].

Contents

1 Rewriting with the Innermost Strategy

theory *Innermost-Rewriting*
imports *First-Order-Rewriting.Trs*
begin

inductive-set *inn-rstep* :: ('*f*, '*v*) *trs* ⇒ ('*f*, '*v*) *term rel* **for** *R*
where *inn-rstep*[*intro*]: (*l*,*r*) ∈ *R* ⟹ *set* (*args* (*l* · σ)) ⊆ *NF-trs R* ⟹ (*C*⟨*l* · σ⟩, *C*⟨*r* · σ⟩) ∈ *inn-rstep R*

lemma *inn-rstep-rstep*: *inn-rstep R* ⊆ *rstep R*
by (*standard*, *auto dest*: *inn-rstep.cases*)

lemma *inn-rstep-ctxt*: **assumes** (*s*,*t*) ∈ *inn-rstep R*
shows (*C*⟨*s*⟩, *C*⟨*t*⟩) ∈ *inn-rstep R*
using *assms*
proof (*induct*)
case ∗: (*inn-rstep l r* σ *D*)
from *inn-rstep*[*OF* ∗, *of C* ∘$_c$ *D*]
show *?case* **by** *auto*
qed

lemma *ctxt-closed-inn-rstep*[*simp*, *intro*]: *ctxt.closed* (*inn-rstep R*)
using *inn-rstep-ctxt*[*of* - - *R*] **by** *fast*

inductive-set *inn-nrrstep* :: ('*f*, '*v*) *trs* ⇒ ('*f*, '*v*) *term rel* **for** *R*
where *inn-nrrstep*[*intro*]: (*l*,*r*) ∈ *R* ⟹ *set* (*args* (*l* · σ)) ⊆ *NF-trs R* ⟹ *C* ≠ *Hole* ⟹ (*C*⟨*l* · σ⟩, *C*⟨*r* · σ⟩) ∈ *inn-nrrstep R*

lemma *inn-nrrstep-nrrstep*: *inn-nrrstep R* ⊆ *nrrstep R*
by (*standard*, *elim inn-nrrstep.cases*, *intro nrrstepI*, *auto*)

lemma *inn-nrrstep-inn-rstep*: *inn-nrrstep R* ⊆ *inn-rstep R*
by (*standard*, *elim inn-nrrstep.cases*, *blast*)

lemma *inn-nrrstep-ctxt*: **assumes** (*s*,*t*) ∈ *inn-nrrstep R*
shows (*C*⟨*s*⟩, *C*⟨*t*⟩) ∈ *inn-nrrstep R*
using *assms*
proof (*induct*)
case ∗: (*inn-nrrstep l r* σ *D*)
from *inn-nrrstep*[*OF* ∗(*1*−*2*), *of C* ∘$_c$ *D*] ∗(*3*)
show *?case* **by** (*cases C*, *auto*)
qed

lemma *inn-rstep-ctxt-inn-nrrstep*: **assumes** (*s*,*t*) ∈ *inn-rstep R*

```
  and C ≠ Hole
  shows (C⟨s⟩, C⟨t⟩) ∈ inn-nrrstep R
  using assms
proof (induct)
  case ∗: (inn-rstep l r σ D)
  from inn-nrrstep[OF ∗(1−2), of C ∘c D] ∗(3)
  show ?case by (cases C, auto)
qed

lemma NF-inn-rstep-rstep: NF (inn-rstep R) = NF (rstep R)
proof
  show NF-trs R ⊆ NF (inn-rstep R) using inn-rstep-rstep[of R] by auto
  show NF (inn-rstep R) ⊆ NF-trs R
  proof
    fix t
    assume t: t ∈ NF (inn-rstep R)
    obtain n where size t = n by auto
    with t show t ∈ NF-trs R
    proof (induct n arbitrary: t rule: less-induct)
      case (less n t)
      show ?case
      proof (rule ccontr)
        assume ¬ ?thesis
        then obtain s where (t,s) ∈ rstep R by auto
        then obtain C l r σ where (l,r) ∈ R and t: t = C ⟨ l · σ ⟩ and s = C ⟨
r · σ ⟩ by auto
        from inn-rstep[OF this(1), of σ C, folded this(2,3)] less(2)
        obtain u where u: u ∈ set (args (l · σ)) and uNF: u ∉ NF-trs R by auto
        from supt-imp-args[of l · σ] u
        have l · σ ▷ u by fast
        with t have tu: t ▷ u
          using subterm.order.strict-trans2 by auto
        with less(3) have size u < n using supt-size by auto
        from less(1)[OF this - refl] uNF have uNF: u ∉ NF (inn-rstep R) by auto
        from tu obtain C where t = C ⟨ u ⟩ by blast
        from less(2)[unfolded this] uNF show False
          by (meson NF-iff-no-step inn-rstep-ctxt)
      qed
    qed
  qed
qed

lemma NF-inn-nrrstep-nrrstep: NF (inn-nrrstep R) = NF (nrrstep R)
proof
  show NF (nrrstep R) ⊆ NF (inn-nrrstep R) using inn-nrrstep-nrrstep[of R] by
auto
  show NF (inn-nrrstep R) ⊆ NF (nrrstep R)
  proof
    fix s
```

```
    assume NF: s ∈ NF (inn-nrrstep R)
    show s ∈ NF (nrrstep R)
    proof (rule ccontr)
      assume ¬ ?thesis
      then obtain t where (s,t) ∈ nrrstep R by auto
      then obtain C l r σ where s: s = C⟨l · σ⟩ and t = C⟨r · σ⟩ and (l,r) ∈
R and C: C ≠ Hole
        by (meson nrrstepE)
      hence l · σ ∉ NF (rstep R) by auto
      from this[folded NF-inn-rstep-rstep]
      have l · σ ∉ NF (inn-rstep R) by auto
      then obtain t where (l · σ, t) ∈ inn-rstep R by auto
      from inn-rstep-ctxt-inn-nrrstep[OF this C] s
      have s ∉ NF (inn-nrrstep R) by auto
      with NF show False by auto
    qed
  qed
qed

inductive-set inn-rrstep :: ('f, 'v) trs ⇒ ('f, 'v) term rel for R
  where inn-rrstep[intro]: (l,r) ∈ R ⟹ set (args (l · σ)) ⊆ NF-trs R ⟹ (l · σ,
r · σ) ∈ inn-rrstep R

lemma inn-rrstep-rrstep: inn-rrstep R ⊆ rrstep R
  by (standard, elim inn-rrstep.cases, intro rrstepI, auto)

lemma inn-rrstep-inn-rstep: inn-rrstep R ⊆ inn-rstep R
  by (standard, elim inn-rrstep.cases)
    (metis ctxt.cop-nil inn-rstep.intros)


lemma inn-rstep-iff-inn-rrstep-or-inn-nrrstep: inn-rstep R = (inn-rrstep R ∪ inn-nrrstep
R)
proof
  show inn-rstep R ⊆ inn-rrstep R ∪ inn-nrrstep R
  proof (rule subrelI)
    fix s t assume (s,t) ∈ inn-rstep R
    from inn-rstep.cases[OF this] obtain l r C σ where step: (l,r) ∈ R set (args
(l · σ)) ⊆ NF-trs R
      s = C⟨l · σ⟩ t = C⟨r · σ⟩ by metis
    from inn-rrstep[OF step(1−2)] inn-nrrstep[OF step(1−2), of C] step(3−4)
    show (s,t) ∈ inn-rrstep R ∪ inn-nrrstep R by (cases C, auto)
  qed
next
  show inn-rrstep R ∪ inn-nrrstep R ⊆ inn-rstep R
    using inn-rrstep-inn-rstep inn-nrrstep-inn-rstep by auto
qed

lemma inn-rstep-cases[consumes 1, case-names root nonroot]:
```

```
  ⟦(s,t) ∈ inn-rstep R; (s,t) ∈ inn-rrstep R ⟹ P; (s,t) ∈ inn-nrrstep R ⟹ P⟧
⟹ P
  by (auto simp: inn-rstep-iff-inn-rrstep-or-inn-nrrstep)

lemma inn-nrrstep-args:
  assumes (s, t) ∈ inn-nrrstep R
  shows ∃f ss ts. s = Fun f ss ∧ t = Fun f ts ∧ length ss = length ts
    ∧ (∃j<length ss. (ss!j, ts!j) ∈ inn-rstep R ∧ (∀ i<length ss. i ≠ j ⟶ ss!i =
ts!i))
  using assms
proof (cases, goal-cases)
  case ∗: (1 l r σ C)
  from ∗(5) obtain f bef D aft where C: C = More f bef D aft by (cases C, auto)
  define ss where ss = bef @ D⟨l · σ⟩ # aft
  define ts where ts = bef @ D⟨r · σ⟩ # aft
  define j where j = length bef
  show ?thesis
  proof (intro exI conjI)
    show s = Fun f ss unfolding ∗ ss-def C by simp
    show t = Fun f ts unfolding ∗ ts-def C by simp
    show length ss = length ts unfolding ss-def ts-def by simp
    show j < length ss unfolding j-def ss-def by simp
    show (ss ! j, ts ! j) ∈ inn-rstep R
      unfolding ss-def ts-def j-def using inn-rstep[OF ∗(3,4), of D] by auto
    show ∀ i<length ss. i ≠ j ⟶ ss ! i = ts ! i
      unfolding ss-def ts-def j-def by (auto simp: nth-append)
  qed
qed



lemma Tinf-imp-SN-inn-nrrstep: assumes t ∈ Tinf (inn-rstep R)
  shows SN-on (inn-nrrstep R) {t}
proof
  fix g
  assume g 0 ∈ {t} and ∀ i. (g i, g (Suc i)) ∈ inn-nrrstep R
  hence steps: ⋀ i. (g i, g (Suc i)) ∈ inn-nrrstep R
    and t: t = g 0 by auto
  from steps[of 0] obtain f ts where g0: g 0 = Fun f ts
    using inn-nrrstep-nrrstep[of R] nrrstep-imp-Fun by blast
  define n where n = length ts
  have ∃ ts. g i = Fun f ts ∧ length ts = n for i
  proof (induct i)
    case 0
    show ?case unfolding g0 n-def by auto
  next
    case (Suc i)
    then obtain ts where g i = Fun f ts and length ts = n by auto
     with steps[of i] inn-nrrstep-nrrstep[of R] show ?case using nrrstep-args by
```

fastforce
qed
hence $\forall\ i.\ \exists\ ts.\ g\ i = Fun\ f\ ts \wedge length\ ts = n$ **by** *auto*
from *choice*[*OF this*] **obtain** *ts* **where** *g*: $g\ i = Fun\ f\ (ts\ i)$ **and** *len*: $length\ (ts\ i) = n$ **for** *i* **by** *blast*
have $\exists\ j < n.\ (ts\ i\ !\ j,\ ts\ (Suc\ i)\ !\ j) \in inn\text{-}rstep\ R \wedge (\forall\ k < n.\ k \neq j \longrightarrow ts\ (Suc\ i)\ !\ k = ts\ i\ !\ k)$ **for** *i*
using *steps*[*of i*, *unfolded g*] **using** *len*[*of i*] *len*[*of Suc i*]
using *inn-nrrstep-args* **by** *force*
hence $\forall\ i.\ \exists\ j < n.\ (ts\ i\ !\ j,\ ts\ (Suc\ i)\ !\ j) \in inn\text{-}rstep\ R \wedge (\forall\ k < n.\ k \neq j \longrightarrow ts\ (Suc\ i)\ !\ k = ts\ i\ !\ k)$
by *blast*
from *choice*[*OF this*] **obtain** *j*
where *j*: $j\ i < n\ (ts\ i\ !\ j\ i,\ ts\ (Suc\ i)\ !\ j\ i) \in inn\text{-}rstep\ R$
$k < n \Longrightarrow k \neq j\ i \Longrightarrow ts\ (Suc\ i)\ !\ k = ts\ i\ !\ k$ **for** *i k*
by *blast*
hence $range\ j \subseteq \{..<n\}$ **by** *auto*
hence *finite* (*range j*)
using *finite-subset* **by** *blast*
from *pigeonhole-infinite*[*OF - this*] **obtain** *J*
where *inf*: *infinite* $\{i.\ j\ i = J\}$ **by** *auto*
define *I* **where** $I = \{i.\ j\ i = J\}$
obtain *i* **where** *i*: $i \in I$ **unfolding** *I-def* **using** *inf* **by** *force*
from *this*[*unfolded I-def*] *j* **have** *J*: $J < n$ **by** *auto*
define *h* **where** $h\ i = ts\ i\ !\ J$ **for** *i*
{
fix *i*
assume $i \in I$
from *this*[*unfolded I-def*] *J* **have** $j\ i = J$ **by** *auto*
from *j*(*2*)[*of i*, *unfolded this*, *folded h-def*]
have $(h\ i,\ h\ (Suc\ i)) \in inn\text{-}rstep\ R$ **.**
} **note** *inI* = *this*
{
fix *i*
assume $i \notin I$
hence $J \neq j\ i$ **unfolding** *I-def* **by** *auto*
from *j*(*3*)[*OF J this*]
have $(h\ i,\ h\ (Suc\ i)) \in \{}\widehat{}=$ **by** (*auto simp*: *h-def*)
} **note** *ninI* = *this*
have *chain*: *chain* $(inn\text{-}rstep\ R \cup \{}\widehat{}=)\ h$ **using** *inI ninI* **by** *auto*
have *inf*: *INFM j*. $(h\ j,\ h\ (Suc\ j)) \in inn\text{-}rstep\ R$ **using** *inf*[*folded I-def*] *inI*
by (*metis* (*no-types*, *lifting*) *INFM-iff-infinite INFM-mono I-def mem-Collect-eq*)
from *chain inf*
have $\neg$ *SN-rel-on-alt* $(inn\text{-}rstep\ R)\ (\{}\widehat{}=)\ \{h\ 0\}$
unfolding *SN-rel-on-alt-def* **by** *blast*
hence *nSN*: $\neg$ *SN-on* $(inn\text{-}rstep\ R)\ \{h\ 0\}$
unfolding *SN-rel-on-conv*[*symmetric*]
unfolding *SN-rel-on-Id* **by** *auto*
have $t \rhd h\ 0$ **unfolding** *t g h-def* **using** *J*[*folded len*[*of 0*]] **by** *simp*

with *assms*[*unfolded Tinf-def*] **have** *SN-on* (*inn-rstep R*) {*h 0*} **by** *auto*
with *nSN* **show** *False* **..**
qed

lemma *Tinf-inn-rstep-imp-first-root-step*: **assumes** $s \in$ *Tinf* (*inn-rstep R*)
shows $\exists$ *t u*. (*s*,*t*) $\in$ (*inn-nrrstep R*)$\hat{}*$ $\wedge$ (*t*,*u*) $\in$ *inn-rrstep R* $\wedge$ $\neg$ *SN-on* (*inn-rstep R*) {*u*}
proof −
from *assms* **have** $\neg$ *SN-on* (*inn-rstep R*) {*s*} **unfolding** *Tinf-def* **by** *auto*
then obtain *f* **where** *f0*: *f 0* = *s* **and** *steps*: $\bigwedge$ *i*. (*f i*, *f* (*Suc i*)) $\in$ *inn-rstep R* **by** *auto*
show *?thesis*
proof (*cases* $\exists$ *i*. (*f i*, *f* (*Suc i*)) $\in$ *inn-rrstep R*)
case *True*
define *i* **where** *i* = (*LEAST i*. (*f i*, *f* (*Suc i*)) $\in$ *inn-rrstep R*)
from *LeastI-ex*[*OF True*, *folded i-def*]
have *inn-rrstep*: (*f i*, *f* (*Suc i*)) $\in$ *inn-rrstep R* **.**
have *nSN*: $\neg$ *SN-on* (*inn-rstep R*) {*f j*} **for** *j* **using** *steps*
by (*simp add*: *chain-imp-not-SN-on steps*)
have *nrr*: (*f j*, *f* (*Suc j*)) $\in$ *inn-nrrstep R* **if** *j* < *i* **for** *j*
using *not-less-Least*[*OF that*[*unfolded i-def*]] *steps*[*of j*]
using *inn-rstep-cases* **by** *blast*
hence *steps*: (*s*, *f i*) $\in$ (*inn-nrrstep R*)$\hat{}*$ **unfolding** *f0*[*symmetric*]
by (*metis rtrancl-fun-conv*)
with *nSN*[*of Suc i*] *inn-rrstep* **show** *?thesis* **by** *metis*
next
case *False*
with *steps* **have** *nrr*: $\bigwedge$ *i*. (*f i*, *f* (*Suc i*)) $\in$ *inn-nrrstep R*
by (*meson inn-rstep-cases*)
hence $\neg$ *SN-on* (*inn-nrrstep R*) {*s*} **using** *f0* **by** *auto*
with *Tinf-imp-SN-inn-nrrstep*[*OF assms*]
have *False* **by** *auto*
thus *?thesis* **..**
qed
qed

end

2 Gramlich's Criterion to Prove Termination by Innermost Termination

Gramlich showed that for locally confluent overlay TRSs, innermost termination and termination coincide. We formalize this result in this theory.

theory *Gramlich-Innermost-Switch*
imports

First-Order-Rewriting.Critical-Pairs
Innermost-Rewriting
begin

lemma *WCR-on-rstep-imp-WCR-on-nrrstep*: **assumes** *WCR-on* (*rstep R*) {*t*. *SN-on* (*rstep R*) {*t*}}
shows *WCR-on* (*nrrstep R*) {*t*. *SN-on* (*nrrstep R*) {*t*}}
proof
fix *s t u*
assume *SN*: $s \in$ {*t*. *SN-on* (*nrrstep R*) {*t*}}
and *st*: (*s*, *t*) ∈ *nrrstep R*
and *su*: (*s*, *u*) ∈ *nrrstep R*
from *st*[*unfolded nrrstep-iff-arg-rstep*]
obtain *f ss i t′* **where**
s: *s* = *Fun f ss*
and *i*: *i* < *length ss*
and *t*: *t* = *Fun f* (*ss*[*i* := *t′*])
and *st*: (*ss* ! *i*, *t′*) ∈ *rstep R*
by *auto*
from *nrrstep-imp-pos-term*[*OF su*[*unfolded s*]] **obtain** *j u′* **where**
j: *j* < *length ss*
and *u*: *u* = *Fun f* (*ss*[*j* := *u′*])
and *su*: (*ss* ! *j*, *u′*) ∈ *rstep R*
by *auto*
show (*t*, *u*) ∈ (*nrrstep R*)↓
proof (*cases i* = *j*)
case *False*
define *v* **where** *v*: *v* = *Fun f* (*ss*[*i* := *t′*, *j* := *u′*])
have *tv*: (*t*,*v*) ∈ *nrrstep R* **unfolding** *t v* **using** *False i j su*
using *nrrstep-iff-arg-rstep* **by** *force*
have *v*: *v* = *Fun f* (*ss*[*j* := *u′*,*i* := *t′*]) **unfolding** *v* **using** *i j False*
by (*simp add*: *list-update-swap*)
have *uv*: (*u*,*v*) ∈ *nrrstep R* **unfolding** *u v* **using** *False i j st*
using *nrrstep-iff-arg-rstep* **by** *force*
from *tv uv* **show** *?thesis* **by** *blast*
next
case *True*
from *SN*[*unfolded s*] **have** *set ss* ⊆ {*t*. *SN-on* (*rstep R*) {*t*}}
by *simp* (*metis SN-nrrstep-imp-args-SN-rstep SN-on-subset-SN-terms term.sel*(*4*))
with *i* **have** *ss* ! *i* ∈ {*t*. *SN-on* (*rstep R*) {*t*}} **by** (*auto simp*: *set-conv-nth*)
from *WCR-onD*[*OF assms this st su*[*folded True*]] **obtain** *v′*
where *tv*: (*t′*, *v′*) ∈ (*rstep R*)* **and** *uv*: (*u′*, *v′*) ∈ (*rstep R*)* **by** *auto*
define *v* **where** *v*: *v* = *Fun f* (*ss*[*i* := *v′*])
have (*t*,*v*) ∈ (*nrrstep R*)* **unfolding** *t v* **using** *tv i*
by (*rule arg-rsteps-into-nrrsteps*)
moreover have (*u*,*v*) ∈ (*nrrstep R*)* **unfolding** *u v True*[*symmetric*] **using** *uv i*
by (*rule arg-rsteps-into-nrrsteps*)
ultimately show *?thesis* **by** *blast*

qed
qed

lemma *SN-innermost-switch-locally-confluent-overlay-local*:
assumes *WCR*: *WCR-on* (*rstep R*) {*t*. *SN-on* (*rstep R*) {*t*}}
and *overlay*: ⋀ *l r*. (*False*,*l*,*r*) ∉ *critical-pairs ren R R*
and *wf*: *wf-trs R*
shows *SN-on* (*inn-rstep R*) = *SN-on* (*rstep R*)
proof (*rule ccontr*)
assume ¬ *?thesis*
then obtain *T* **where** *SN-on* (*inn-rstep R*) *T* ≠ *SN-on* (*rstep R*) *T* **by** *blast*
with *SN-on-subset1*[*OF - inn-rstep-rstep*, *of R*]
have *nSN*: ¬ (*SN-on* (*rstep R*) *T*)
and *SN*: *SN-on* (*inn-rstep R*) *T* **by** *blast+*
then obtain *init* **where** *init* ∈ *T* **and** *nSN*: ¬ *SN-on* (*rstep R*) {*init*} **by** *fast*
with *SN* **have** *SN*: *SN-on* (*inn-rstep R*) {*init*} **by** *fast*
from *not-SN-imp-subt-Tinf*[*OF nSN*] **obtain** *s*
where *init-s*: *init* ⊵ *s* **and** *Tinf*: *s* ∈ *Tinf* (*rstep R*) **by** *auto*
from *ctxt-closed-SN-on-subt*[*OF ctxt-closed-inn-rstep SN init-s*]
have *SN*: *SN-on* (*inn-rstep R*) {*s*} **.**
from *Tinf-rstep-imp-first-root-step*[*OF Tinf*]
obtain *t u* **where** *tu*: (*s*, *t*) ∈ (*nrrstep R*)*
(*t*, *u*) ∈ *rrstep R*
t ∈ *Tinf* (*rstep R*)
¬ *SN-on* (*rstep R*) {*u*}
by *blast*
from *tu*(*4*) **obtain** *v* **where** *uv*: *u* ⊵ *v v* ∈ *Tinf* (*rstep R*)
using *not-SN-imp-subt-Tinf* **by** *blast*
let *?P* = λ (*n* :: *nat*) *s t*. *s* ∈ *Tinf* (*rstep R*) ∧ (*s*,*t*) ∈ *rrstep R O* {⊵} ∧ *t* ∈ *Tinf* (*rstep R*)
let *?Q* = λ (*n* :: *nat*) *s1 t1 s2 t2*. (*t1*,*s2*) ∈ (*nrrstep R*)*
have *P0*: *?P 0 t v* **using** *tu uv* **by** *auto*
have ∃ *S T*. *S 0* = *t* ∧
T 0 = *v* ∧
(∀ *n*. *?P n* (*S n*) (*T n*) ∧ (*T n*, *S* (*Suc n*)) ∈ (*nrrstep R*)*)
proof (*rule dependent-nat-choice2-start*[**where** *?Q* = *?Q* **and** *?P* = *?P*, *OF P0*])
fix *s t n*
assume *s* ∈ *Tinf* (*rstep R*) ∧ (*s*, *t*) ∈ *rrstep R O* {⊵} ∧ *t* ∈ *Tinf* (*rstep R*)
hence *t* ∈ *Tinf* (*rstep R*) **by** *auto*
from *Tinf-rstep-imp-first-root-step*[*OF this*]
obtain *s′ u* **where** *s′u*: (*t*, *s′*) ∈ (*nrrstep R*)* (*s′*, *u*) ∈ *rrstep R s′* ∈ *Tinf* (*rstep R*)
¬ *SN-on* (*rstep R*) {*u*}
by *auto*
from *s′u*(*4*) **obtain** *v* **where** *uv*: *u* ⊵ *v v* ∈ *Tinf* (*rstep R*)
using *not-SN-imp-subt-Tinf* **by** *blast*
show ∃ *s′ t′*. *?P* (*Suc n*) *s′ t′* ∧ (*t*, *s′*) ∈ (*nrrstep R*)* **using** *s′u uv* **by** *blast*

```
  qed
  then obtain S U where
    start: S 0 = t U 0 = v and
    STinf: ⋀ n. S n ∈ Tinf (rstep R) and
    steps: ⋀ n. (S n, U n) ∈ rrstep R O {⊵} and
    UTinf: ⋀ n. U n ∈ Tinf (rstep R) and
    nsteps: ⋀ n. (U n, S (Suc n)) ∈ (nrrstep R)*
    by blast
  from steps have ∀ n. ∃ u. (S n, u) ∈ rrstep R ∧ u ⊵ U n by blast
  from choice[OF this] obtain T where
    rsteps: ⋀ n. (S n, T n) ∈ rrstep R and
    subt: ⋀ n. T n ⊵ U n by blast
  from rrstepE[OF rsteps] have ∀ n. ∃ l r σ. (l,r) ∈ R ∧ S n = l · σ ∧ T n = r
· σ
    by metis
  from choice[OF this] obtain l where ∀ n. ∃ r σ. (l n,r) ∈ R ∧ S n = l n · σ
∧ T n = r · σ
    by metis
  from choice[OF this] obtain r where ∀ n. ∃ σ. (l n,r n) ∈ R ∧ S n = l n · σ
∧ T n = r n · σ
    by metis
  from choice[OF this] obtain σ where rsteps: ⋀ n. (l n,r n) ∈ R
    and S: ⋀ n. S n = l n · σ n
    and T: ⋀ n. T n = r n · σ n
    by blast
  {
    fix x n
    assume x ∈ vars-term (l n)
    hence l n ▷ Var x using rsteps(1)[of n] wf
      unfolding wf-trs-def by force
    hence l n · σ n ▷ Var x · σ n by blast
    from this[folded S] STinf[of n]
    have SN-on (rstep R) {σ n x} by (auto simp: Tinf-def)
  } note SN-vars = this
  have vars: vars-term (r n) ⊆ vars-term (l n) for n
    using wf rsteps[of n] unfolding wf-trs-def by auto
  {
    fix n
    have ∀x∈vars-term (r n). (Var x · σ n, U n) ∉ {⊵}
    proof
      fix x
      assume x ∈ vars-term (r n)
      with vars have x ∈ vars-term (l n) by auto
      from SN-vars[OF this]
      have SN-on (rstep R) {σ n x} .
      with UTinf[of n, unfolded Tinf-def]
      show (Var x · σ n, U n) ∉ {⊵} using SN-on-imp-SN-on-subt[of R]
        by auto
    qed
```

```
  from subt-instance-and-not-subst-imp-subt[OF subt[of n, unfolded T] this]
  have ∃ u⊴r n. U n = u · σ n .
}
hence ∀ n. ∃ u⊴r n. U n = u · σ n by metis
from choice[OF this] obtain u where
  subt: ⋀ n. r n ⊵ u n and U: ⋀ n. U n = u n · σ n
  by auto
define δ where δ n x = (SOME u. (σ n x, u) ∈ (inn-rstep R)⁻¹) for n x
{
  fix x n
  assume x ∈ vars-term (l n)
  from SN-vars[OF this]
  have SN-on (rstep R) {σ n x} .
  hence SN-on (inn-rstep R) {σ n x} using inn-rstep-rstep[of R] by fast
  hence ∃ u. (σ n x, u) ∈ (inn-rstep R)⁻¹ by (rule SN-on-imp-normalizability)
  from someI-ex[OF this, folded δ-def]
  have NF: (σ n x, δ n x) ∈ (inn-rstep R)! .
  hence steps: (σ n x, δ n x) ∈ (inn-rstep R)*
    and NF: δ n x ∈ NF-trs R by (auto simp: NF-inn-rstep-rstep)
  hence (σ n x, δ n x) ∈ (rstep R)* using inn-rstep-rstep[of R]
    using rtrancl-mono by blast
  note NF steps this
} note σδ = this

  have fun-l: is-Fun (l n) for n using rsteps[of n] wf unfolding wf-trs-def by
force
  have lσδ: (l n · σ n, l n · δ n) ∈ (nrrstep R)⁻* for n
    by (rule term-subst-rsteps-nrrsteps[OF - fun-l], rule σδ(3), auto)

  have args: set (args (l n · δ n)) ⊆ NF-trs R for n
  proof
    fix v
    assume v ∈ set (args (l n · δ n))
    hence sub: l n · δ n ▷ v
      by (metis fun-l is-Fun-Fun-conv subst-apply-eq-Var supt.arg term.distinct(1)
          term.exhaust-sel)
    {
      fix y
      assume (v,y) ∈ rstep R
      then obtain l' r' μ C where v = C ⟨ l' · μ ⟩ and lr': (l',r') ∈ R by auto
      with sub have l n · δ n ▷ l' · μ
        by (metis ctxt-supteq subterm.less-trans suptI)
      hence l n · δ n ⊵ l' · μ and neq: l n · δ n ≠ l' · μ by auto
      from supteq-subst-cases'[OF this(1)]
      have False
      proof
        from lr' have NF: l' · μ ∉ NF-trs R by auto
        assume ∃ x. x ∈ vars-term (l n) ∧ δ n x ⊵ l' · μ
        with σδ(1)[of - n] NF show False by fastforce
```

```
    next
      assume ∃ w ◁ l n. is-Fun w ∧ l' · μ = w · δ n
      then obtain w where ∗: w ◁ l n is-Fun w l' · μ = w · δ n by auto
      from mgu-vd-complete[OF ∗(3)[symmetric], of ren]
      obtain μ1 μ2 where mgu: mgu-vd ren w l' = Some (μ1, μ2) by auto
      from ∗ neq have w ◁ l n by auto
      then obtain C where l n = C ⟨ w ⟩ and C ≠ Hole by auto
      from critical-pairsI[OF rsteps(1) lr' this(1) ∗(2) mgu refl refl refl] this(2)
        overlay show False by auto
    qed
  }
  thus v ∈ NF-trs R by auto
qed
hence lNF: l n · δ n ∈ NF (nrrstep R) for n
  by (rule args-NF-rstep-imp-NF-nrrstep)

define W where W n = l n · δ n for n
have W0-NF: W 0 ∈ NF (nrrstep R) using lNF[of 0] by (auto simp: W-def)

from WCR-on-rstep-imp-WCR-on-nrrstep[OF WCR]
have WCR: WCR-on (nrrstep R) {t. SN-on (nrrstep R) {t}} .


{
  fix n
  from inn-rstep[OF rsteps(1), of n δ n Hole, OF args]
  have inn-rstep: (l n · δ n, r n · δ n) ∈ inn-rstep R by auto

  from vars[of n] subt[of n]
  have varsu: vars-term (u n) ⊆ vars-term (l n)
    using supteq-imp-vars-term-subset by blast
  from UTinf[unfolded U, of n] have ¬ SN-on (rstep R) {u n · σ n}
    unfolding Tinf-def by auto
  hence fun-u: is-Fun (u n) using
    SN-vars[of - n] varsu
    by (cases u n, auto)

  have uσδ: (u n · σ n, u n · δ n) ∈ (nrrstep R)^∗
  proof (rule term-subst-rsteps-nrrsteps[OF - fun-u], rule σδ(3))
    fix x
    show x ∈ vars-term (u n) ⟹ x ∈ vars-term (l n) using vars[of n]
      subt[of n] by (meson in-mono supteq-imp-vars-term-subset)
  qed

  from UTinf[of n, unfolded U]
  have SNu: SN-on (nrrstep R) {u n · σ n} by (rule Tinf-imp-SN-nrrstep)

  with uσδ have SN-on (nrrstep R) {u n · δ n} by (rule steps-preserve-SN-on)
```

hence *SN-on* (*inn-nrrstep R*) {*u n* · *δ n*} **using** *inn-nrrstep-nrrstep* **by** *fast*
then obtain *v* **where** *iuv*: (*u n* · *δ n*, *v*) ∈ (*inn-nrrstep R*)^* **and** *vNF*: *v* ∈ *NF* (*nrrstep R*)
unfolding *NF-inn-nrrstep-nrrstep*[*symmetric*] **by** (*meson SN-reaches-NF*)

from *uσδ iuv vNF* **have** *uv*: (*u n* · *σ n*, *v*) ∈ (*nrrstep R*)^!
using *rtrancl-mono*[*OF inn-nrrstep-nrrstep*, *of R*] **by** *force*

from *nsteps*[*of n*, *unfolded U S*] *lσδ*[*of Suc n*] *lNF*[*of Suc n*]
have *ul*: (*u n* · *σ n*, *l* (*Suc n*) · *δ* (*Suc n*)) ∈ (*nrrstep R*)^!
by *blast*

from *Newman-local*[*OF SNu WCR*]
have *CR-on* (*nrrstep R*) {*u n* · *σ n*} **.**
from *CR-on-same-NF*[*OF this - ul uv*]
have *v*: *v* = *l* (*Suc n*) · *δ* (*Suc n*) **by** *simp*

with *iuv*
have (*u n* · *δ n*, *l* (*Suc n*) · *δ* (*Suc n*)) ∈ (*inn-nrrstep R*)^* **by** *auto*
hence *iul*: (*u n* · *δ n*, *l* (*Suc n*) · *δ* (*Suc n*)) ∈ (*inn-rstep R*)^*
using *rtrancl-mono*[*OF inn-nrrstep-inn-rstep*[*of R*]] **by** *auto*

have (*r n* · *δ n*, *u n* · *δ n*) ∈ {⊵} **using** *subt*[*of n*] **by** *auto*

from *inn-rstep this iul*
have (*W n*, *W* (*Suc n*)) ∈ *inn-rstep R O* {⊵} *O* (*inn-rstep R*)^*
unfolding *W-def* **by** *blast*

} **note** *inf-W-seq* = *this*

have (*t*, *W 0*) ∈ (*nrrstep R*)^* **using** ‹*S 0* = *t*› **unfolding** *S W-def* **using** *lσδ* **by** *auto*
with *tu*(*1*) **have** (*s*, *W 0*) ∈ (*nrrstep R*)^* **by** *auto*
with *W0-NF* **have** *sW0*: (*s*, *W 0*) ∈ (*nrrstep R*)^! **by** *auto*
from *Tinf-imp-SN-nrrstep*[*OF Tinf*]
have *SNs*: *SN-on* (*nrrstep R*) {*s*} **.**
hence *SN-on* (*inn-nrrstep R*) {*s*} **using** *inn-nrrstep-nrrstep* **by** *blast*
then obtain *v* **where** (*s*,*v*) ∈ (*inn-nrrstep R*)^!
by (*metis SN-on-imp-normalizability*)
hence *isv*: (*s*,*v*) ∈ (*inn-nrrstep R*)^* **and** *v*: *v* ∈ *NF* (*nrrstep R*)
using *NF-inn-nrrstep-nrrstep*[*of R*] **by** *auto*
from *isv* **have** (*s*,*v*) ∈ (*nrrstep R*)^* **using** *rtrancl-mono*[*OF inn-nrrstep-nrrstep*] **by** *auto*
with *v* **have** *sv*: (*s*,*v*) ∈ (*nrrstep R*)^! **by** *auto*
from *CR-on-same-NF*[*OF Newman-local*[*OF SNs WCR*] - *sv sW0*]
have *v* = *W 0* **by** *auto*
with *isv* **have** (*s*,*W 0*) ∈ (*inn-nrrstep R*)^* **by** *auto*
hence (*s*,*W 0*) ∈ (*inn-rstep R*)^* **using** *rtrancl-mono*[*OF inn-nrrstep-inn-rstep*]

by *auto*
with *SN*
have *SN-W0*: *SN-on* (*inn-rstep R*) {*W 0*}
by (*metis steps-preserve-SN-on*)

from *inf-W-seq* **have** ¬ *SN-on* (*inn-rstep R O* {⊵} *O* (*inn-rstep R*)*) {*W 0*} **by** *blast*

moreover from *SN-on-r-imp-SN-on-supt-union-r*[*OF ctxt-closed-inn-rstep SN-W0*]
have *SN-on* ({▷} ∪ *inn-rstep R*) {*W 0*} **.**
from *SN-on-trancl*[*OF this*]
have ∗: *SN-on* (({▷} ∪ *inn-rstep R*)$^+$) {*W 0*} **.**
have *id*: {⊵} = {▷}$^=$ **by** *auto*
have *SN-on* (*inn-rstep R O* {⊵} *O* (*inn-rstep R*)*) {*W 0*}
by (*rule SN-on-subset1*[*OF* ∗], *unfold id*, *regexp*)

ultimately show *False* **..**
qed

lemma *SN-innermost-switch-locally-confluent-overlay*:
assumes *WCR*: *WCR-on* (*rstep R*) {*t*. *SN-on* (*rstep R*) {*t*}}
and *overlay*: ⋀ *l r*. (*False*,*l*,*r*) ∉ *critical-pairs ren R R*
and *wf*: *wf-trs R*
shows *SN* (*inn-rstep R*) = *SN* (*rstep R*)
unfolding *SN-innermost-switch-locally-confluent-overlay-local*[*OF assms*] **..**

end

3 Linearity Preservation by Unification

theory *Linear-Unification*
imports
First-Order-Terms.Unification-More
begin

A sufficient criterion to ensure that $t \cdot \sigma$ is linear.

lemma *linear-term-subst*: *linear-term t*
⟹ (⋀ *x*. *x* ∈ *vars-term t* ⟹ *linear-term* (*σ x*))
⟹ (⋀ *x y*. *x* ∈ *vars-term t* ⟹ *y* ∈ *vars-term t* ⟹ *x* ≠ *y* ⟹ *vars-term* (*σ x*) ∩ *vars-term* (*σ y*) = {})
⟹ *linear-term* (*t* · *σ*)
proof (*induct t*)
case (*Fun f ts*)
show *?case* **unfolding** *eval-term.simps linear-term.simps*
proof (*intro conjI ballI*)
fix *tsig*
assume *tsig* ∈ *set* (*map* (λ*s*. *s* · *σ*) *ts*)

```
    then obtain ti where ti: ti ∈ set ts and tsig: tsig = ti · σ by auto
    show linear-term tsig unfolding tsig
      by (rule Fun(1)[OF ti], insert Fun(2−) ti, auto)
  next
   from Fun(2)[unfolded linear-term.simps] have part: is-partition (map vars-term
ts) by auto
    show is-partition (map vars-term (map (λs. s · σ) ts))
      unfolding map-map o-def linear-term.simps
      unfolding is-partition-alt is-partition-alt-def length-map
    proof (intro allI impI, goal-cases)
      case (1 i j)
      show ?case
      proof (rule ccontr)
        assume ¬ ?thesis
        then obtain y where y ∈ vars-term (ts ! i · σ) y ∈ vars-term (ts ! j · σ)
          using 1 by auto
        from this[unfolded vars-term-subst] obtain xi xj where
          xij: xi ∈ vars-term (ts ! i) xj ∈ vars-term (ts ! j) and
          y: y ∈ vars-term (σ xi) y ∈ vars-term (σ xj) by auto
      from part[unfolded is-partition-alt is-partition-alt-def length-map, rule-format,
OF 1]
          xij 1 have xi ≠ xj by auto
         from Fun(4)[OF - - this] have vars-term (σ xi) ∩ vars-term (σ xj) = {}
using xij 1
          by force
        with y show False by auto
      qed
    qed
  qed
qed auto
```

Unification of two var disjoint terms where one of them is linear results in a partially linear substitution and linear terms

```
definition vars-mset-left :: (('f,'v)term × ('f,'v)term) multiset ⇒ 'v multiset where
  vars-mset-left m = sum-mset (image-mset (vars-term-ms o fst) m)

definition vars-mset-right :: (('f,'v)term × ('f,'v)term) multiset ⇒ 'v multiset
where
  vars-mset-right m = sum-mset (image-mset (vars-term-ms o snd) m)

definition linear-mset :: 'a multiset ⇒ bool where
  linear-mset m = (∀ x. count m x ≤ 1)


lemma count-sum-mset-image-mset:
  count (sum-mset (image-mset f m)) x = sum-mset (image-mset (λ a. count (f a)
x) m)
  by (induct m, auto)
```

```
lemma linear-vars-term-ms: linear-mset (vars-term-ms t) = linear-term t
  unfolding linear-mset-def
proof (induct t)
  case (Fun f ts)
  show ?case
  proof
    assume cnt: ∀ x. count (vars-term-ms (Fun f ts)) x ≤ 1
    show linear-term (Fun f ts) unfolding linear-term.simps
    proof (intro conjI ballI)
      fix t
      assume t: t ∈ set ts
      {
        fix x
        have count (vars-term-ms t) x ≤ count (vars-term-ms (Fun f ts)) x
          using split-list[OF t] by auto
        hence count (vars-term-ms t) x ≤ 1 using cnt[rule-format, of x] by auto
      }
      with Fun[OF t] show linear-term t by simp
    next
      show is-partition (map vars-term ts) unfolding is-partition-def
      proof (clarsimp)
        fix j i
        assume ji: j < length ts i < j
        show vars-term (ts ! i) ∩ vars-term (ts ! j) = {}
        proof (rule ccontr)
          assume ¬ ?thesis
          then obtain x where x ∈# vars-term-ms (ts ! i) x ∈# vars-term-ms (ts
! j) by auto
          hence count: count (vars-term-ms (ts ! i)) x ≥ 1 count (vars-term-ms (ts
! j)) x ≥ 1
            by auto
          from ji(1) obtain aft bef where ts: ts = bef @ ts ! j # aft and bef: bef
= take j ts
            using id-take-nth-drop by blast
          from ji have i: i < length bef unfolding bef by auto
          from i obtain b m where bef = b @ bef ! i # m
            using id-take-nth-drop by blast
          also have bef ! i = ts ! i unfolding bef using ji by auto
          finally have ts = b @ ts ! i # m @ ts ! j # aft using ts by auto
          from arg-cong[OF this, of mset]
          have count (vars-term-ms (Fun f ts)) x ≥ count (vars-term-ms (ts ! i)) x
+ count (vars-term-ms (ts ! j)) x
            by simp
          with count have count (vars-term-ms (Fun f ts)) x ≥ 2 by linarith
          with cnt[rule-format, of x] show False by auto
        qed
      qed
    qed
  next
```

assume *lin*: *linear-term* (*Fun f ts*)
hence *part*: *is-partition* (*map vars-term ts*) **by** *auto*
show $\forall x.$ *count* (*vars-term-ms* (*Fun f ts*)) $x \leq 1$
proof
fix x
{
fix t
assume $t \in$ *set ts*
from *Fun*[*OF this*] *lin this* **have** *count* (*vars-term-ms t*) $x \leq 1$ **by** *auto*
} note $IH =$ *this*
show *count* (*vars-term-ms* (*Fun f ts*)) $x \leq 1$
proof (*rule ccontr*)
assume $\neg$ *?thesis*
hence *count*: *count* (*vars-term-ms* (*Fun f ts*)) $x \geq 2$ **by** *auto*
hence $x \in\#$ *vars-term-ms* (*Fun f ts*)
by (*metis Suc-1 Suc-le-eq count-greater-eq-one-iff less-imp-le-nat*)
then obtain t **where** t: $t \in$ *set ts* **and** x: $x \in$ *vars-term t* **by** *auto*
hence *count* (*vars-term-ms t*) $x \geq 1$ **by** *simp*
with *IH*[*OF t*] **have** *c1*: *count* (*vars-term-ms t*) $x = 1$ **by** *linarith*
from t **obtain** i **where** i: $i <$ *length ts* **and** t: $t =$ *ts* ! i
by (*auto simp*: *set-conv-nth*)
from t x i
have j: $j <$ *length ts* $\Longrightarrow j \neq i \Longrightarrow x \notin$ *vars-term* (*ts* ! j) **for** j
using *part*[*unfolded is-partition-alt is-partition-alt-def*, *rule-format*, *unfolded length-map*, *of i j*]
by *auto*
define A **where** $A = \{0..<$*length ts*$\} - \{i\}$
have *ts* = *map* (λ *i. ts* ! *i*) [*0* ..< *length ts*]
by (*intro nth-equalityI*, *auto*)
from *arg-cong*[*OF this*, *of mset*]
have *mset ts* = *image-mset* ((!) *ts*) (*mset-set* {*0*..<*length ts*}) **by** *auto*
also have ... =
add-mset (*ts* ! *i*) (*image-mset* ((!) *ts*) (*mset-set A*))
using *i* **by** (*simp add*: *mset-set.remove A-def*)
finally have *id*: *mset ts* = *add-mset* (*ts* ! *i*) (*image-mset* ((!) *ts*) (*mset-set A*)) **.**
from *count*
have *2* $\leq$ *count* ($\sum_{\#}$ (*image-mset vars-term-ms* (*mset ts*))) x **by** *simp*
also have ... = *1* + *count* ($\sum_{\#}$ (*image-mset vars-term-ms* (*image-mset* ((!) *ts*) (*mset-set A*)))) x
unfolding *id* **by** (*simp add*: *c1*[*unfolded t*])
also have *count* ($\sum_{\#}$ (*image-mset vars-term-ms* (*image-mset* ((!) *ts*) (*mset-set A*)))) x
= ($\sum a \in \#$ *image-mset* ((!) *ts*) (*mset-set A*). *count* (*vars-term-ms a*) x)
unfolding *count-sum-mset-image-mset* **..**
also have ... = ($\sum a \in \#$ *image-mset* ((!) *ts*) (*mset-set A*). *0*)
proof (*rule arg-cong*[*of - - sum-mset*], *rule image-mset-cong*)
fix *tj*
show *tj* $\in\#$ *image-mset* ((!) *ts*) (*mset-set A*) $\Longrightarrow$ *count* (*vars-term-ms tj*)

```
  x = 0
        using j unfolding A-def by force
      qed
      also have ... = 0 by simp
      finally show False by simp
    qed
  qed
 qed
qed auto

lemma linear-term-count: assumes linear-term t
 shows count (vars-term-ms t) x ≤ 1
 using assms[folded linear-vars-term-ms, unfolded linear-mset-def] ..


lemma linear-term-Var-subst: linear-term (t · (Var o r)) ⟹ linear-term t
proof (induct t)
 case (Fun f ts)
 hence is-partition (map (λx. ⋃x∈vars-term x. {r x}) ts)
  by (auto simp: o-def vars-term-subst)
 also have map (λx. ⋃x∈vars-term x. {r x}) ts = map ((`) r o vars-term) ts by
auto
 finally have is-partition (map ((`) r ∘ vars-term) ts) .
 hence is-partition (map vars-term ts) unfolding is-partition-def length-map by
auto
 with Fun show ?case by auto
qed auto

lemma vars-mset-right-add[simp]: vars-mset-right (add-mset p E) = vars-term-ms
(snd p) + vars-mset-right E
 unfolding vars-mset-right-def by auto

lemma right-linear-var-disjoint-mgu-mset: fixes E :: (('f,'v)term × ('f,'v)term)
multiset
 and u :: ('f,'v)term
 assumes set-mset (vars-mset-left E) ∩ set-mset (vars-mset-right E) = {}
 and linear-mset (vars-mset-right E)
 and is-mgu σ (set-mset E)
 and vars-term u ∩ set-mset (vars-mset-right E) = {}
 and linear-term u
shows linear-term (u · σ)
proof −
 define disjLR where disjLR E = (set-mset (vars-mset-left E) ∩ set-mset (vars-mset-right
E) = {})
   for E :: (('f,'v)term × ('f,'v)term) multiset
 define lin where lin E = linear-mset (vars-mset-right E)
   for E :: (('f,'v)term × ('f,'v)term) multiset
 define disjU where disjU u E = (vars-term u ∩ set-mset (vars-mset-right E) =
{})
```

```
  for u :: ('f,'v)term and E :: (('f,'v)term × ('f,'v)term) multiset
 have lin0: lin E unfolding lin-def using assms by auto
 have disjLR0: disjLR E unfolding disjLR-def using assms by auto
 have disjU0: disjU u E unfolding disjU-def using assms by auto
 have is-mgu0: is-mgu σ (set-mset E) by fact
 from lin0 disjLR0 disjU0 is-mgu0 ‹linear-term u› show ?thesis
 proof (induction E arbitrary: u σ rule: wf-induct[OF wf-unifless])
  case less: (1 E u σ)
  note lin = less.prems(1)
  note disjLR = less.prems(2)
  note disjU = less.prems(3)
  note mgu = less.prems(4)
  note linu = less.prems(5)
  note IH = less.IH[rule-format, OF UNIF1-unifless]
  show ?case
  proof (cases ∃ s t. (s,t) ∈# E ∧ is-Fun s ∧ is-Fun t)
   case True
   then obtain s t F where E: E = add-mset (s,t) F and is-Fun s and is-Fun
t
    by (metis insert-DiffM)
   then obtain f g ss ts where s: s = Fun f ss and t: t = Fun g ts by (cases
s; cases t; auto)
   from mgu[unfolded E s t] have Fun f ss · σ = Fun g ts · σ
    by (auto simp: is-mgu-def)
  hence fg: f = g and len: length ss = length ts by (auto intro: map-eq-imp-length-eq)
   define G where G = F + mset (zip ss ts)
   have UNIF1 Var E G unfolding E s t fg G-def by (rule decomp[OF len])
   note IH = IH[OF this]
   have is-mgu σ (set-mset E) = is-mgu σ (set-mset G) unfolding E s t fg
G-def
    using is-mgu-insert-decomp[OF len, of σ g set-mset F] by auto
   with mgu have mgu: is-mgu σ (set-mset G) by auto
   have vr: vars-mset-right E = vars-mset-right G
    unfolding vars-mset-right-def E s t G-def
    by (simp add: o-def, induct rule: list-induct2[OF len], auto)
   have vl: vars-mset-left E = vars-mset-left G
    unfolding vars-mset-left-def E s t G-def
    by (simp add: o-def, induct rule: list-induct2[OF len], auto)
   from lin have lin: lin G unfolding lin-def vr .
   from disjU have disjU: disjU u G unfolding disjU-def vr .
   from disjLR have disjLR: disjLR G unfolding disjLR-def vl vr .
   note IH = IH[OF lin disjLR disjU mgu linu]
   show ?thesis
    by (rule IH)
  next
   case no-Fun-Fun: False
   show ?thesis
   proof (cases ∃ s t. (s,t) ∈# E ∧ is-Var t)
    case True
```

```
    then obtain s t F where E: E = add-mset (s,t) F and is-Var t
      by (metis insert-DiffM)
    then obtain x where t: t = Var x by (cases s; cases t; auto)
    let ?τ = subst x s
  have x: x ∈# vars-mset-right E unfolding E t by (auto simp: vars-mset-right-def)

    from disjLR x
    have xs: x ∉ vars-term s by (auto simp: E disjLR-def vars-mset-left-def)
    from UNIF1-mono[OF UNIF1-singleton-Var-right[OF this], of F]
    have unif1: UNIF1 ?τ E (subst-mset ?τ F) by (auto simp: E t)
    {
      fix l r
      assume (l,r) ∈# F
      then obtain G where F: F = add-mset (l,r) G
        by (metis insert-DiffM)
      from x disjLR
     have xl: x ∉ vars-term l by (auto simp: E F disjLR-def vars-mset-left-def)
      from lin[unfolded E t lin-def linear-mset-def, rule-format, of x]
      have count (vars-mset-right F) x = 0
        by (auto simp: vars-mset-right-def)
      hence x ∉# vars-mset-right F
        by (simp add: count-eq-zero-iff)
      hence xr: x ∉ vars-term r by (auto simp: F vars-mset-right-def)
      note xl xr
    } note xF = this
    have subst-mset ?τ F = image-mset id F unfolding subst-mset-def
    proof (rule image-mset-cong, clarsimp)
      fix l r
      assume (l,r) ∈# F
      from xF[OF this]
      have xl: x ∉ vars-term l x ∉ vars-term r .
      thus l · ?τ = l ∧ r · ?τ = r by auto
    qed
    hence substF: subst-mset ?τ F = F by auto
    with unif1 have unif1: UNIF1 ?τ E F by auto
    note IH = IH[OF this]
    from is-mgu-UNIF1[OF unif1 mgu] obtain σ' δ
      where mgu': is-mgu σ' (set-mset E) is-mgu δ (set-mset F)
        and id: σ' = ?τ ∘s δ by auto
    from is-mgu-is-mgu-var-renaming[OF mgu'(1) mgu, unfolded id] obtain
      γ where sub: ?τ ∘s δ = σ ∘s (Var ∘ γ) by auto
    have u · δ = u · ?τ ∘s δ
    proof (rule term-subst-eq)
      fix y
      assume y ∈ vars-term u
    with disjU[unfolded disjU-def E t] have y ≠ x unfolding vars-mset-right-def
by auto
      thus δ y = (?τ ∘s δ) y
        by (auto simp: subst-compose-def subst-def)
```

```
    qed
    from this[unfolded sub] have udelt: u · δ = u · σ ∘s (Var ∘ γ) by auto

  have LFE: vars-mset-left F ⊆# vars-mset-left E unfolding vars-mset-left-def
E by auto
  have RFE: vars-mset-right F ⊆# vars-mset-right E unfolding vars-mset-right-def
E by auto
    hence count (vars-mset-right F) y ≤ count (vars-mset-right E) y for y
      by (rule mset-subset-eq-count)
    with lin have lin: lin F unfolding lin-def linear-mset-def
      using dual-order.trans by blast
   from disjLR have disjLR: disjLR F unfolding disjLR-def using LFE RFE
      by (meson disjoint-iff mset-subset-eqD)
    from disjU have disjU: disjU u F unfolding disjU-def using RFE
      by (meson disjoint-iff mset-subset-eqD)
    from IH[OF lin disjLR disjU mgu'(2) linu]
    have IH: linear-term (u · δ) .
    from IH[unfolded udelt]
    have linear-term (u · σ · (Var ∘ γ)) by auto
    thus linear-term (u · σ) by (rule linear-term-Var-subst)
  next
    case no-right-Var: False
    from no-right-Var no-Fun-Fun
    have left-Var: ⋀ s t. (s,t) ∈# E ⟹ is-Var s by auto
    show ?thesis
    proof (cases E = {#})
      case True
      hence is-mgu Var (set-mset E) by auto
      from is-mgu-is-mgu-var-renaming[OF this mgu]
      obtain γ where id: Var = σ ∘s (Var ∘ γ) by auto
      have linear-term (u · Var) using linu by auto
      hence linear-term (u · σ · (Var ∘ γ))
        by (subst (asm) id, auto)
      thus linear-term (u · σ) by (rule linear-term-Var-subst)
    next
      case False
      then obtain s t where (s,t) ∈# E by auto
      with left-Var[OF this] obtain x where
        (Var x,t) ∈# E by auto
      then obtain F where E: E = add-mset (Var x,t) F by (rule mset-add)
      let ?τ = subst x t
      let ?F = subst-mset ?τ F
      let ?F' = image-mset (λp. (fst p · ?τ, snd p)) F
      from disjLR
       have xt: x ∉ vars-term t by (auto simp: E disjLR-def vars-mset-left-def
vars-mset-right-def)
      from UNIF1-mono[OF UNIF1-singleton-Var-left[OF this], of F]
      have unif1: UNIF1 (subst x t) E ?F by (auto simp: E)
      have ?F = ?F'
```

```
      unfolding subst-mset-def
    proof (rule image-mset-cong, clarsimp)
      fix l r
      assume lr: (l,r) ∈# F
      have r · ?τ = r · Var
      proof (rule term-subst-eq)
        fix y
        assume y ∈ vars-term r
        with disjLR[unfolded E disjLR-def] lr have x ≠ y
          by (auto simp: vars-mset-left-def vars-mset-right-def)
        thus ?τ y = Var y by (simp add: subst-def)
      qed
      thus r · ?τ = r by auto
    qed
    note unif1 = unif1[unfolded this]
    note IH = IH[OF this]
    have vr: vars-mset-right ?F' = vars-mset-right F
      unfolding vars-mset-right-def by (induct F, auto)
       have vl: set-mset (vars-mset-left ?F') ⊆ set-mset(vars-mset-left F) ∪
vars-term t
        unfolding vars-mset-left-def by (auto simp: vars-term-subst subst-def
split: if-splits)
    have vars-term-ms t ⊆# vars-mset-right E unfolding E by auto
    hence count (vars-term-ms t) y ≤ count (vars-mset-right E) y for y
      by (rule mset-subset-eq-count)
        with lin have linear-mset (vars-term-ms t) unfolding lin-def lin-
ear-mset-def
      using dual-order.trans by blast
    hence lint: linear-term t
      using linear-vars-term-ms by auto
    have RFE: vars-mset-right ?F' ⊆# vars-mset-right E
      unfolding vr unfolding vars-mset-right-def E by auto
    hence count (vars-mset-right ?F') y ≤ count (vars-mset-right E) y for y
      by (rule mset-subset-eq-count)
    with lin have lin': lin ?F' unfolding lin-def linear-mset-def
      using dual-order.trans by blast
    {
      fix y
      assume y: y ∈# vars-mset-right F y ∈ vars-term t
      hence y ∈# vars-term-ms t by auto
   with y(1) have {#y,y#} ⊆# vars-mset-right E unfolding E vars-mset-right-add
snd-conv
        by (metis add-mset-add-single single-subset-iff subset-mset.add-mono)
      from mset-subset-eq-count[OF this, of y]
        lin[unfolded lin-def linear-mset-def, rule-format, of y]
      have False by simp
    } note disj-F-t = this

    have disjLR': disjLR ?F'
```

proof (*rule ccontr*)
assume ¬ *?thesis*
from *this*[*unfolded disjLR-def vr*, *simplified*]
obtain *y* **where** *y*: *y* ∈# *vars-mset-left ?F′ y* ∈# *vars-mset-right F*
by *auto*
with *vl* **have** *disj*: *y* ∈# *vars-mset-left F* ∨ *y* ∈ *vars-term t* **by** *auto*
from *y*(*2*) *disjLR*[*unfolded E disjLR-def*]
have *y* ∉ *set-mset* (*vars-mset-left* (*add-mset* (*Var x*, *t*) *F*))
by (*auto simp*: *vars-mset-right-def*)
with *disj* **have** *y* ∈ *vars-term t* **unfolding** *vars-mset-left-def* **by** *auto*
from *disj-F-t*[*OF y*(*2*) *this*]
show *False* **.**
qed
let *?v* = *u* · *?τ*
have *disjU′*: *disjU ?v ?F′*
proof (*rule ccontr*)
assume ¬ *?thesis*
from *this*[*unfolded disjU-def vr*] **obtain** *y*
where *y*: *y* ∈ *vars-term ?v y* ∈# *vars-mset-right F*
by *auto*
from *y*(*1*) **have** *y* ∈ *vars-term u* ∨ *y* ∈ *vars-term t*
by (*auto simp*: *vars-term-subst subst-def split*: *if-splits*)
with *disj-F-t*[*OF y*(*2*)] **have** *y* ∈ *vars-term u* **by** *auto*
with *disjU*[*unfolded disjU-def*] *y*(*2*)
show *False* **unfolding** *E* **by** *auto*
qed
have *linu′*: *linear-term ?v*
proof (*rule linear-term-subst*[*OF linu*])
show *linear-term* (*?τ y*) **for** *y* **using** *lint* **by** (*auto simp*: *subst-def*)
fix *y z*
assume *yz*: *y* ∈ *vars-term u z* ∈ *vars-term u y* ≠ *z*
with *disjU*[*unfolded disjU-def*]
have *y* ∉# *vars-mset-right E z* ∉# *vars-mset-right E* **by** *auto*
hence *yzt*: *y* ∉ *vars-term t z* ∉ *vars-term t* **unfolding** *E* **by** *auto*
show *vars-term* (*?τ y*) ∩ *vars-term* (*?τ z*) = {}
proof (*cases y* = *x* ∨ *z* = *x*)
case *False*
thus *?thesis* **using** *yz* **by** (*auto simp*: *subst-def*)
next
case *True*
thus *?thesis*
proof
assume *yx*: *y* = *x*
with *yz* **have** *z* ≠ *x* **by** *auto*
hence *one*: *vars-term* (*?τ z*) = {*z*} **by** (*auto simp*: *subst-def*)
from *yx* **have** *two*: *vars-term* (*?τ y*) = *vars-term t* **by** *simp*
from *one two yzt* **show** *?thesis* **by** *auto*
next
assume *zx*: *z* = *x*

```
            with yz have y ≠ x by auto
            hence one: vars-term (?τ y) = {y} by (auto simp: subst-def)
            from zx have two: vars-term (?τ z) = vars-term t by simp
            from one two yzt show ?thesis by auto
          qed
        qed
      qed
      note IH = IH[OF lin' disjLR' disjU' - linu']

      from is-mgu-UNIF1[OF unif1 mgu] obtain σ' δ
        where mgu': is-mgu σ' (set-mset E) is-mgu δ (set-mset ?F')
          and id: σ' = ?τ ∘s δ by auto
      from is-mgu-is-mgu-var-renaming[OF mgu'(1) mgu, unfolded id] obtain
        γ where sub: ?τ ∘s δ = σ ∘s (Var ∘ γ) by auto
      let ?γ = Var o γ :: ('f,'v)subst
      from IH[OF mgu'(2)] have IH: linear-term (u · ?τ ∘s δ) by simp
      from this[unfolded sub]
      have linear-term (u · σ · ?γ) by simp
      thus linear-term (u · σ) by (rule linear-term-Var-subst)
    qed
  qed
 qed
qed

lemma right-linear-var-disjoint-mgu: fixes s t :: ('f,'v)term
  assumes disj: vars-term s ∩ vars-term t = {}
  and lin: linear-term t
  and mgu: is-mgu σ {(s,t)}
  and linu: linear-term u
  and disju: vars-term u ∩ vars-term t = {}
shows linear-term (u · σ)
proof (rule right-linear-var-disjoint-mgu-mset[of {#(s,t)#} σ u, OF - - - - linu])
  show is-mgu σ (set-mset {#(s, t)#}) using mgu by auto
  show linear-mset (vars-mset-right {#(s, t)#}) using lin
    unfolding vars-mset-right-def by (auto simp: linear-vars-term-ms lin)
  show vars-term u ∩ set-mset (vars-mset-right {#(s, t)#}) = {}
    using disju unfolding vars-mset-right-def by auto
  show set-mset (vars-mset-left {#(s, t)#}) ∩ set-mset (vars-mset-right {#(s,
t)#}) = {}
    using disj unfolding vars-mset-left-def vars-mset-right-def by auto
qed
```

Corollary: Unification of two linear var disjoint terms results in a linear substitution and linear unified terms.

```
lemma linear-var-disjoint-is-mgu: fixes s t :: ('f,'v)term
  assumes disj: vars-term s ∩ vars-term t = {}
  and lin: linear-term s linear-term t
  and mgu: is-mgu σ {(s,t)}
```

shows *vars-term u ∩ vars-term t = {} ⟹ linear-term u ⟹ linear-term (u · σ)*
and *vars-term u ∩ vars-term s = {} ⟹ linear-term u ⟹ linear-term (u · σ)*
and *linear-term (s · σ) linear-term (t · σ)*
and *linear-term (σ x)*
proof −
from *disj* **have** *disj'*: *vars-term t ∩ vars-term s = {}* **by** *auto*
from *mgu* **have** *mgu'*: *is-mgu σ {(t,s)}* **by** (*simp add*: *is-mgu-insert-swap*)
note *one* = *right-linear-var-disjoint-mgu*[*OF disj lin*(2) *mgu*]
note *two* = *right-linear-var-disjoint-mgu*[*OF disj' lin*(1) *mgu'*]
from *one*[*OF lin*(1) *disj*] **show** *linear-term (s · σ)* **.**
from *two*[*OF lin*(2) *disj'*] **show** *linear-term (t · σ)* **.**
let *?x* = *Var x* :: (*'f*,*'v*)*term*
from *disj* **have** *vars-term ?x ∩ vars-term s = {} ∨ vars-term ?x ∩ vars-term t = {}* **by** *auto*
with *one*[*of ?x*] *two*[*of ?x*] **show** *linear-term (σ x)* **by** *auto*
show *vars-term u ∩ vars-term t = {} ⟹ linear-term u ⟹ linear-term (u · σ)*
using *one* **by** *metis*
show *vars-term u ∩ vars-term s = {} ⟹ linear-term u ⟹ linear-term (u · σ)*
using *two* **by** *metis*
qed
end

4 Narrowing and Linearity Preservation

In this theory we define narrowing and show that narrowing of a linear term with a right-linear TRS results in a linear term again. Moreover, some further results on narrowing are formalized.

theory *Linear-Narrowing*
imports
Automatic-Refinement.Misc
Linear-Unification
First-Order-Rewriting.Trs
begin

4.1 Preparations for Narrowing

definition *subst-term-mset* :: (*'f*,*'v*)*term* ⇒ (*'f*,*'v*)*subst* ⇒ (*'f*,*'v*)*term multiset*
where *subst-term-mset t σ = image-mset σ* (*vars-term-ms t*)

lemma *vars-term-ms-subst-compose-split*:
(*δ* ‘# (∑ *x*∈#*xs*. *vars-term-ms* (*μ x*)), (*μ* ∘*s δ*) ‘# *xs*) ∈ (*mult* {◁})=
∧ (*Ball* (*μ* ‘ *set-mset xs*) *is-Var* ∨ (*δ* ‘# (∑ *x*∈#*xs*. *vars-term-ms* (*μ x*)), (*μ* ∘*s δ*) ‘# *xs*) ∈ *mult* {◁})
(**is** - ∈ *?R*= ∧ -)
proof (*induct xs*)
case *empty*
then show *?case* **by** *auto*
next

case (*add x xs*)
define *a* **where** $a = \{\#\ (\mu \circ_s \delta)\ x\ \#\}$
define *b* **where** $b = (\mu \circ_s \delta)\ `\#\ xs$
define *c* **where** $c = \delta\ `\#\ vars\text{-}term\text{-}ms\ (\mu\ x)$
define *d* **where** $d = \delta\ `\#\ (\sum x{\in}\#xs.\ vars\text{-}term\text{-}ms\ (\mu\ x))$
have *ab*: $(\mu \circ_s \delta)\ `\#\ add\text{-}mset\ x\ xs = a + b$ **by** (*simp add*: *a-def b-def*)
have *cd*: $\delta\ `\#\ (\sum x{\in}\#add\text{-}mset\ x\ xs.\ vars\text{-}term\text{-}ms\ (\mu\ x)) = c + d$ **by** (*simp add*: *c-def d-def*)
have *tr*: *trans* $\{\lhd\}$ **by** (*simp add*: *trans-supt*)
have *irr*: *irrefl-on T* $\{\lhd\}$ **for** $T :: ('a,'b)term\ set$ **by** (*simp add*: *irrefl-onI*)
have *ext*: $(x,y) \in ?R \Longrightarrow (x + z,\ y + z) \in ?R$ **for** *x y z*
using *mult-cancel*[*OF tr irr*, *of x z y*] **by** *auto*
hence *exte*: $(x,y) \in ?R^{=} \Longrightarrow (x + z,\ y + z) \in ?R^{=}$ **for** *x y z*
by *auto*
from *add*
have $(d,b) \in ?R^{=}\ Ball\ (\mu\ `\ set\text{-}mset\ xs)\ is\text{-}Var \vee (d,b) \in ?R$ **by** (*auto simp*: *b-def d-def*)
from *exte*[*OF this*(*1*), *of c*] *ext*[*of d b c*] *this*(*2*)
have *IH*: $(c + d,\ c + b) \in ?R^{=}\ Ball\ (\mu\ `\ set\text{-}mset\ xs)\ is\text{-}Var \vee (c + d,\ c + b) \in ?R$
by (*auto simp*: *ac-simps*)

have *step-main*: $(c,a) \in ?R^{=} \wedge (is\text{-}Var\ (\mu\ x) \vee (c,a) \in ?R)$
proof (*cases* $\mu\ x$)
case (*Var y*)
thus *?thesis* **by** (*auto simp*: *subst-compose-def c-def a-def*)
next
case (*Fun f ts*)
have $(c,a) \in mult1\ \{\lhd\}$
by (*auto simp*: *Fun a-def c-def subst-compose-def intro*!: *mult1I*)
(*metis eval-term.simps*(*2*) *subst-image-subterm term.set-intros*(*4*))
thus *?thesis* **by** (*auto simp add*: *mult-def*)
qed
with *ext*[*of c a b*] *exte*[*of c a b*]
have *step*: $(c + b,\ a + b) \in ?R^{=}\ is\text{-}Var\ (\mu\ x) \vee (c + b,\ a + b) \in ?R$ **by** *auto*

have *one*: $(c + d,\ a + b) \in ?R^{=}$
using *IH step*
by (*meson tr transE trans-mult trans-on-reflcl*)
have *two*: $Ball\ (\mu\ `\ set\text{-}mset\ (add\text{-}mset\ x\ xs))\ is\text{-}Var \vee (c + d,\ a + b) \in ?R$
proof (*cases is-Var* $(\mu\ x)$)
case *False*
from *step*(*2*) *False* **have** $(c + b,\ a + b) \in ?R$ **by** *auto*
with *IH*(*1*) **have** $(c + d,\ a + b) \in ?R$
by (*metis Un-iff pair-in-Id-conv transD trans-mult*)
thus *?thesis* **by** *auto*
next
case *x*: *True*
show *?thesis*

```
    proof (cases Ball (μ ‘ set-mset xs) is-Var)
      case True
      with x show ?thesis by auto
    next
      case False
      with IH(2) have (c + d, c + b) ∈ ?R by auto
      with step(1) have (c + d, a + b) ∈ ?R
        by (metis Un-iff pair-in-Id-conv transD trans-mult)
      thus ?thesis by auto
    qed
  qed

  from one two show ?case
    unfolding cd ab by auto
qed

lemma linear-term-vars-term-ms: assumes vars-term r ⊆ vars-term l
    and linear-term r
  shows vars-term-ms r ⊆# vars-term-ms l
proof (intro mset-subset-eqI, rule ccontr)
  fix x
  assume not: ¬ count (vars-term-ms r) x ≤ count (vars-term-ms l) x
  hence c: count (vars-term-ms r) x ≥ 1 by linarith
  hence x ∈ vars-term r by (auto simp: not-in-iff)
  with assms have x ∈ vars-term l by auto
  hence cl: count (vars-term-ms l) x ≥ 1 by (auto simp: not-in-iff)
  with not have count (vars-term-ms r) x > 1 by linarith
  with linear-term-count[OF assms(2), of x] show False by auto
qed

lemma vars-term-ms-map-vars-term[simp]: vars-term-ms (map-vars-term f t) = f
‘# vars-term-ms t
proof (induct t)
  case (Fun f ts)
  thus ?case by (simp, induct ts, auto)
qed auto

lemma linear-term-map-inj-on: assumes linear-term (map-vars-term f t)
  shows inj-on f (vars-term t)
proof (rule ccontr)
  let ?ft = map-vars-term f t
  from assms have linear-mset (vars-term-ms ?ft)
    using linear-vars-term-ms by blast
  also have vars-term-ms ?ft = f ‘# vars-term-ms t by simp
  finally have lin: linear-mset (f ‘# vars-term-ms t) .
  assume ¬ ?thesis
  then obtain x y where xy: x ∈# vars-term-ms t y ∈# vars-term-ms t x ≠ y f
x = f y
    unfolding inj-on-def by auto
```

define *xs* **where** *xs* = *vars-term-ms t* − {#*x*, *y*#}
from *xy*(*1*−*3*) **have** *vars-term-ms t* = {#*x*,*y*#} + *xs* **unfolding** *xs-def*
by (*metis diff-union-swap insert-DiffM2 insert-subset-eq-iff mset-subset-eq-single subset-eq-diff-conv subset-mset.add-diff-inverse*)
from *arg-cong*[*OF this*, *of* (‘#) *f*] *xy*(*4*)
have *f* ‘# *vars-term-ms t* = {# *f x*, *f x* #} + *f* ‘# *xs* **by** *auto*
hence *count* (*f* ‘# *vars-term-ms t*) (*f x*) ≥ *2* **by** *auto*
with *lin*[*unfolded linear-mset-def*, *rule-format*, *of f x*] **show** *False* **by** *auto*
qed

4.2 Narrowing Relations

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

definition
narrows-r-p-s :: (*'f*, *'v*) *trs* ⇒ (*'f*, *'v*) *rule* ⇒ *pos* ⇒ (*'f*,*'v*)*subst* ⇒ (*'f*, *'v*)*trs*
where
narrows-r-p-s R r p μ ≡ {(*s*,*t*). ∃ *μ2*. *p* ∈ *poss s* ∧ *is-Fun* (*s* |- *p*) ∧ *r* ∈ *R* ∧
mgu-vd ren (*s* |- *p*) (*fst r*) = *Some* (*μ*, *μ2*)
∧ *t* = *replace-at* (*s* · *μ*) *p* (*snd r* · *μ2*)}

definition
narrow-step-s :: (*'f*,*'v*) *trs* ⇒ (*'f*,*'v*)*subst* ⇒ (*'f*,*'v*)*trs* **where**
narrow-step-s R μ = {*st* | *st lr p*. *st* ∈ *narrows-r-p-s R lr p μ*}

definition *narrow-step* :: (*'f*,*'v*) *trs* ⇒ (*'f*,*'v*)*trs* **where**
narrow-step R = {*st* | *st lr p μ*. *st* ∈ *narrows-r-p-s R lr p μ*}

lemma *narrow-step-to-s*: *narrow-step R* = ⋃ (*range* (*narrow-step-s R*))
unfolding *narrow-step-def narrow-step-s-def* **by** *auto*

theorem *right-linear-rule-narrowing*: **fixes** *R* :: (*'f*,*'v*)*trs*
assumes *linr*: *linear-term r*
and *lins*: *linear-term s*
and *narr*: (*s*,*t*) ∈ *narrows-r-p-s R* (*l*,*r*) *p μ*
shows *linear-term t*
proof −
define *r1* **where** *r1* = *rename-1 ren*
define *r2* **where** *r2* = *rename-2 ren*
let *?r1* = *map-vars-term r1*
let *?r2* = *map-vars-term r2*
define *s'* **where** *s'* = *?r1 s*
define *l'* **where** *l'* = *?r2 l*
define *r'* **where** *r'* = *?r2 r*
define *sp'* **where** *sp'* = *?r1* (*s* |- *p*)
from *narr*[*unfolded narrows-r-p-s-def*] **obtain** *μ2*
where *p*: *p* ∈ *poss s*

```
    and mgu: mgu-vd ren (s |- p) l = Some (μ, μ2)
    and t: t = replace-at (s · μ) p (r · μ2)
  by auto
from p have sp'-id: ?r1 (s |- p) = sp' unfolding sp'-def by simp
with mgu[unfolded mgu-vd-def mgu-var-disjoint-generic-def,
    folded r1-def r2-def]
obtain γ where mgu: mgu sp' l' = Some γ
  and μ: μ = γ ∘ r1 and μ2: μ2 = γ ∘ r2
  by (auto split: option.splits simp: l'-def)
have μ: u · μ = ?r1 u · γ for u unfolding μ o-def
  by (metis eval-eq-map-vars)
have μ2: u · μ2 = ?r2 u · γ for u unfolding μ2 o-def
  by (metis eval-eq-map-vars)
from lins p have linsp: linear-term (s |- p) by (rule subt-at-linear)
from mgu-sound[OF mgu] have imgu: is-imgu γ {(sp',l')} by auto
hence is-mgu γ {(sp',l')} by (rule is-imgu-imp-is-mgu)
hence mgu: is-mgu γ {(l', sp')}
  using is-mgu-insert-swap by blast
note ren = rename-12[of ren, folded r1-def r2-def]
have vars-term l' ∩ vars-term sp' = {}
  using ren(3) unfolding map-vars-term-eq vars-term-subst l'-def sp'-def
  by auto
note main = right-linear-var-disjoint-mgu[OF this - mgu]
from linsp have linsp': linear-term sp' using ren(1) unfolding sp'-def
  by (metis inj-on-subset linear-term-map-inj-on-linear-term top-greatest)

have main: linear-term u ⟹
  vars-term u ∩ vars-term sp' = {} ⟹
  linear-term (u · γ) for u
  using main[OF linsp', of u] by (simp add: term.set-map)
have p': p ∈ poss s' using p unfolding s'-def by auto

define u where u = replace-at s' p r'
from t[unfolded μ μ2, folded r'-def s'-def]
have t = replace-at (s' · γ) p (r' · γ) by simp
also have ... = u · γ using p'
  by (simp add: ctxt-of-pos-term-subst u-def)
finally have t: t = u · γ .

have sp': sp' = s' |- p unfolding sp'-def s'-def using p by auto
show ?thesis unfolding t
proof (rule main)
  have vsu: vars-term-ms u = vars-ctxt-ms (ctxt-of-pos-term p s') + vars-term-ms
r'
    unfolding u-def vars-term-ms-ctxt-apply by auto
  have vss: vars-term-ms s' = vars-ctxt-ms (ctxt-of-pos-term p s') + vars-term-ms
sp'
    using p' unfolding sp' by (metis ctxt-supt-id vars-term-ms-ctxt-apply)
  have lins': linear-term s' using lins unfolding s'-def using ren(1)
```

```
    by (simp add: inj-on-def linear-term-map-inj-on-linear-term)
  show linear-term u
    unfolding linear-vars-term-ms[symmetric]
    unfolding linear-mset-def
  proof
    fix x
    {
      assume x ∈# vars-ctxt-ms (ctxt-of-pos-term p s')
      hence x ∈ vars-term s' using p'
        by (metis set-mset-vars-term-ms union-iff vss)
      hence x ∈ range r1 unfolding s'-def
        by (auto simp: term.set-map)
    } note one = this
    {
      assume x ∈# vars-term-ms r'
      hence x ∈ range r2 unfolding r'-def
        by (auto simp: term.set-map)
    } note two = this
    show count (vars-term-ms u) x ≤ 1
    proof (cases x ∈# vars-ctxt-ms (ctxt-of-pos-term p s'))
      case True
      from one[OF this] two ren have x ∉# vars-term-ms r' by auto
      hence count (vars-term-ms u) x = count (vars-ctxt-ms (ctxt-of-pos-term p
s')) x
        unfolding vsu using not-in-iff by fastforce
      also have ... ≤ count (vars-term-ms s') x unfolding vss by simp
      also have ... ≤ 1 using lins' unfolding linear-vars-term-ms[symmetric]
linear-mset-def
        by simp
      finally show ?thesis .
    next
      case False
      from linr have linr': linear-term r' unfolding r'-def using ren(2)
        by (simp add: inj-on-def linear-term-map-inj-on-linear-term)
      from False have count (vars-term-ms u) x = count (vars-term-ms r') x
        unfolding vsu using not-in-iff by fastforce
      also have ... ≤ 1 using linr' unfolding linear-vars-term-ms[symmetric]
linear-mset-def
        by simp
      finally show ?thesis .
    qed
  qed

  show vars-term u ∩ vars-term sp' = {}
  proof (rule ccontr)
    assume ¬ ?thesis
    then obtain x where x: x ∈# vars-term-ms u x ∈# vars-term-ms sp' by
auto
    with vsu have disj: x ∈# vars-ctxt-ms (ctxt-of-pos-term p s') ∨ x ∈#
```

vars-term-ms r′ **by** *auto*
from *x(2)* **have** *x* ∈ *range r1* **unfolding** *sp′-def* **by** (*auto simp*: *term.set-map*)
with *ren* **have** *x* ∉ *range r2* **by** *auto*
hence *x* ∉ *vars-term r′* **unfolding** *r′-def* **by** (*auto simp*: *term.set-map*)
with *disj* **have** *x* ∈# *vars-ctxt-ms* (*ctxt-of-pos-term p s′*) **by** *auto*
with *x(2)* **have** {#*x*,*x*#} ⊆# *vars-term-ms s′* **unfolding** *vss*
by (*metis add-mset-add-single single-subset-iff subset-mset.add-mono*)
from *mset-subset-eq-count*[*OF this*, *of x*]
have *count* (*vars-term-ms s′*) *x* ≥ *2*
by *auto*
hence ¬ *linear-mset* (*vars-term-ms s′*) **unfolding** *linear-mset-def*
by (*metis Suc-1 Suc-le-eq linorder-not-less*)
with *lins′* **show** *False* **using** *linear-vars-term-ms* **by** *blast*
qed
qed
qed

theorem *right-linear-rule-narrow-step*:
assumes ⋀ *lr*. *lr* ∈ *R* ⟹ *linear-term* (*snd lr*)
and *lins*: *linear-term s*
and *narr*: (*s*,*t*) ∈ *narrow-step R*
shows *linear-term t*
proof −
from *narr*[*unfolded narrow-step-def*] **obtain** *l r p* μ
where *step*: (*s*,*t*) ∈ *narrows-r-p-s R* (*l*,*r*) *p* μ **by** *auto*
hence (*l*,*r*) ∈ *R* **unfolding** *narrows-r-p-s-def* **by** *auto*
from *assms(1)*[*OF this*] **have** *linear-term r* **by** *auto*
from *right-linear-rule-narrowing*[*OF this lins step*]
show *?thesis* **.**
qed

lemma *narrowing-right-linear-one-step-simulation-r-p-s*: **fixes** *R* :: (′*f*,′*v*)*trs*
assumes (*s* · σ, *t*) ∈ *rstep-r-p-s R* (*l*,*r*) *p* μ
and *lins*: *linear-term s*
and *linr*: *linear-term r*
and *var-cond*: *vars-term r* ⊆ *vars-term l*
shows *p* ∉ *fun-poss s* ∧
(∃ δ. *t* = *s* · δ ∧ (*subst-term-mset s* δ, *subst-term-mset s* σ) ∈ *mult1* ((*rstep R*)^−*1*))
∨
p ∈ *fun-poss s* ∧
(∃ μ*1* μ*2* δ *u*. *t* = *u* · δ ∧ *linear-term u* ∧ (*s*,*u*) ∈ *narrows-r-p-s R* (*l*,*r*) *p* μ*1* ∧ σ = μ*1* ∘*s* δ ∧ μ = μ*2* ∘*s* δ
∧ (*subst-term-mset u* δ, *subst-term-mset s* σ) ∈ (*mult* {◁})^=)
proof −
from *assms(1)*[*unfolded rstep-r-p-s-def′ Let-def*, *simplified*]
have *p*: *p* ∈ *poss* (*s* · σ) **and** *lr*: (*l*, *r*) ∈ *R* **and** *ssig*: *s* · σ |- *p* = *l* · μ
and *t*: *t* = *replace-at* (*s* · σ) *p* (*r* · μ)

```
    by auto
  show ?thesis
  proof (cases p ∈ poss s ∧ is-Fun (s |- p))
    case False
    from pos-into-subst[OF refl p this]
    obtain q1 q2 where pq: p = q1 @ q2 and q1: q1 ∈ poss s and is-Var (s |-
q1) by auto
    then obtain x where sq1: s |- q1 = Var x by auto
    define C where C = ctxt-of-pos-term q1 s
    from q1 have ssig': s · σ = (C ·c σ) ⟨ σ x ⟩
      using sq1 by (metis ctxt-supt-id eval-ctxt eval-term.simps(1) C-def)
    have q2: q2 ∈ poss (σ x) using p pq sq1 by (simp add: q1)
   have match: σ x |- q2 = l · μ unfolding C-def ssig[unfolded ssig' pq, symmetric]

      using p pq sq1 q1 q2 by (metis C-def eval-term.simps(1) ssig' subt-at-subst
subterm-poss-conv)
    define u where u = replace-at (σ x) q2 (r · μ)
    have (σ x, u) ∈ rstep-r-p-s R (l,r) q2 μ
      unfolding u-def rstep-r-p-s-def Let-def using lr match q2
      by (auto intro: replace-at-ident)
    hence step: (σ x, u) ∈ rstep R by (rule rstep-r-p-s-imp-rstep)
      have ctxt: ctxt-of-pos-term p (s · σ) = (ctxt-of-pos-term q1 (s · σ)) ∘c
ctxt-of-pos-term q2 (σ x)
      unfolding pq by (subst ctxt-of-pos-term-append, insert q1 sq1, auto)
    define δ where δ = σ(x := u)
    have t: t = replace-at (s · σ) q1 u
      unfolding t u-def unfolding ctxt by simp
    note repl = linear-term-replace-in-subst[OF lins q1 sq1, of σ δ u, folded t]
    have t: t = s · δ
      by (rule repl, auto simp: δ-def)
    show ?thesis unfolding fun-poss-poss
    proof (rule disjI1, intro conjI[OF False] exI[of - δ] conjI t)
      from q1 sq1 have x ∈ vars-term s
        by (metis subt-at-imp-supteq subteq-Var-imp-in-vars-term)
      hence x ∈# vars-term-ms s by auto
        then obtain xs where vars: vars-term-ms s = add-mset x xs by (rule
mset-add)
      from lins have linear-mset (vars-term-ms s)
        by (simp add: linear-vars-term-ms)
      from this[unfolded linear-mset-def vars, rule-format, of x]
      have xs: x ∉# xs by (simp add: not-in-iff)
      have sσ: subst-term-mset s σ = add-mset (σ x) (image-mset σ xs)
        unfolding vars subst-term-mset-def by auto
      have sδ: subst-term-mset s δ = add-mset u (image-mset σ xs)
        unfolding vars subst-term-mset-def using xs by (auto simp: δ-def, induct
xs, auto)
      show (subst-term-mset s δ, subst-term-mset s σ) ∈ mult1 ((rstep R)^-1)
        unfolding sσ sδ
        by (rule mult1I[of - σ x image-mset σ xs - {#u#} (rstep R)^-1], insert
```

```
step, auto)
    qed
  next
    case True
    hence p: p ∈ poss s is-Fun (s |- p) by auto
    from ssig p have s |- p · σ = l · μ by simp
    from mgu-vd-complete[OF this, of ren] obtain
     μ1 μ2 δ where mgu: mgu-vd ren (s |- p) l = Some (μ1, μ2)
       and sig: σ = μ1 ∘s δ
       and mu: μ = μ2 ∘s δ
       and unif: s |- p · μ1 = l · μ2
      by auto
    define u where u = replace-at (s · μ1) p (r · μ2)
    have su: (s, u) ∈ narrows-r-p-s R (l,r) p μ1
      unfolding narrows-r-p-s-def u-def using mgu p lr by auto
    from right-linear-rule-narrowing[OF linr lins su]
    have linu: linear-term u .
    define C where C = ctxt-of-pos-term p s
    from p have ctxt: ctxt-of-pos-term p (s · σ) = C ·c σ for σ :: ('f,'v)subst
      by (metis ctxt-of-pos-term-subst C-def)
    have tu: t = u · δ unfolding t u-def ctxt
      unfolding sig mu by simp
    show ?thesis unfolding fun-poss-poss
    proof (rule disjI2, intro exI[of - u] exI conjI linu p)
      show t = u · δ by fact
      show (s, u) ∈ narrows-r-p-s R (l,r) p μ1 by fact
      show σ = μ1 ∘s δ by fact
      show μ = μ2 ∘s δ by fact
      define VS where VS = vars-term s
      define mid-set where mid-set = δ `# vars-term-ms (s · μ1)
       have subst-u: subst-term-mset u δ = δ `# vars-ctxt-ms (C ·c μ1) + δ `#
vars-term-ms (r · μ2)
        unfolding u-def subst-term-mset-def ctxt vars-term-ms-ctxt-apply o-def by
simp
    have subst-term-mset s σ = (μ1 ∘s δ) `# vars-term-ms s unfolding subst-term-mset-def
sig by auto
      also have (mid-set, ...) ∈ (mult {◁})= ∧
       (Ball (μ1 ` set-mset (vars-term-ms s)) is-Var ∨ (mid-set, ...) ∈ mult {◁})
        unfolding vars-term-ms-subst mid-set-def
        by (rule vars-term-ms-subst-compose-split)
      finally have one: (mid-set, subst-term-mset s σ) ∈ (mult {◁})=
          Ball (μ1 ` VS) is-Var ∨ (mid-set, subst-term-mset s σ) ∈ mult {◁}
        by (auto simp: VS-def)

      have s-split: s = C ⟨ s |- p ⟩ using p unfolding C-def
        by (simp add: ctxt-supt-id)
      from arg-cong[OF this, of λ t. t · μ1]
      have sμ1: s · μ1 = (C ·c μ1) ⟨ s |- p · μ1 ⟩ by simp
      from linear-term-vars-term-ms[OF var-cond linr]
```

```
    have vars-rl: vars-term-ms r ⊆# vars-term-ms l .
    then obtain xs where vars-l: vars-term-ms l = vars-term-ms r + xs
      by (metis subset-mset.add-diff-inverse)

    define VS' where VS' = (∑ x∈#xs. vars-term-ms (μ2 x))
    have mid-set = δ ‘# vars-ctxt-ms (C ·c μ1) + δ ‘# vars-term-ms (l · μ2)
      unfolding mid-set-def sμ1 vars-term-ms-ctxt-apply unif by auto
    also have ... = δ ‘# vars-ctxt-ms (C ·c μ1) + δ ‘# vars-term-ms (r · μ2)
+ δ ‘# VS'
      by (simp add: vars-l VS'-def)
    also have ... = subst-term-mset u δ + δ ‘# VS'
      unfolding subst-u unfolding vars-term-ms-subst vars-l by auto
    finally have mid-u: mid-set = subst-term-mset u δ + δ ‘# VS' .
    hence subst-term-mset u δ ⊆# mid-set by auto
    with subset-implies-mult[of subst-term-mset u δ mid-set {◁}]
    have two: (subst-term-mset u δ, mid-set) ∈ (mult {◁})=
      by auto
    {
      assume VS' ≠ {#}
      with mid-u have subst-term-mset u δ ⊂# mid-set unfolding mid-u
        by (simp add: subset-mset.less-le)
      hence (subst-term-mset u δ, mid-set) ∈ mult {◁}
        by (rule subset-implies-mult)
    } note two' = this


    from two one show refl-step: (subst-term-mset u δ, subst-term-mset s σ) ∈
(mult {◁})=
      by (meson transD trans-mult trans-on-converse trans-on-reflcl)
  qed
 qed
qed

lemma narrowing-right-linear-one-step-simulation: fixes R :: ('f,'v)trs
 assumes (s · σ, t) ∈ rstep R
   and lins: linear-term s
   and right-lin: ⋀ lr. lr ∈ R ⟹ linear-term (snd lr)
   and var-cond: ⋀ lr. lr ∈ R ⟹ vars-term (snd lr) ⊆ vars-term (fst lr)
 shows (∃ δ. t = s · δ ∧ (subst-term-mset s δ, subst-term-mset s σ) ∈ mult1
((rstep R)^-1))
   ∨ (∃ μ δ u. t = u · δ ∧ linear-term u ∧ (s,u) ∈ narrow-step-s R μ ∧ σ = μ ∘s
δ
        ∧ (subst-term-mset u δ, subst-term-mset s σ) ∈ (mult {◁})=)
proof −
 from assms(1) obtain l r p μ where step: (s · σ, t) ∈ rstep-r-p-s R (l,r) p μ
   by (meson rstep-iff-rstep-r-p-s)
 from this[unfolded rstep-r-p-s-def' Let-def, simplified]
 have lr: (l,r) ∈ R by auto
 from right-lin[OF this] have lin: linear-term r by auto
```

from *var-cond*[*OF lr*] **have** *vars-term r* ⊆ *vars-term l* **by** *auto*
from *narrowing-right-linear-one-step-simulation-r-p-s*[*OF step lins lin this*]
show *?thesis* **unfolding** *narrow-step-s-def* **by** *blast*
qed

lemma *rstep-instance-imp-narrows-r-p-s*: **assumes** *step*: (*s* · σ,*t*) ∈ *rstep-r-p-s R r p* τ
and *notσ*: *p* ∈ *poss s is-Fun* (*s* |- *p*)
shows ∃ μ *μ2* δ *u*. (*s*,*u*) ∈ *narrows-r-p-s R r p* μ ∧ *s* · σ = *s* · μ · δ ∧ *t* = *u* · δ ∧ σ = μ ∘$_s$ δ ∧ τ = *μ2* ∘$_s$ δ
proof −
from *step*[*unfolded rstep-r-p-s-def'*] **have** *r*: *r* ∈ *R* **and** *id*: *s* · σ |- *p* = *fst r* · τ
and *t*: *t* = *replace-at* (*s* · σ) *p* (*snd r* · τ) **by** *auto*
from *notσ*(*1*) *id* **have** *s* |- *p* · σ = *fst r* · τ **by** *auto*
from *mgu-vd-complete*[*OF this*, *of ren*] **obtain** *μ1 μ2* δ **where**
mgu: *mgu-vd ren* (*s* |- *p*) (*fst r*)
= *Some* (*μ1*, *μ2*) **and** σ: σ = *μ1* ∘$_s$ δ **and** τ: τ = *μ2* ∘$_s$ δ **and** *id*: *s* |- *p* · *μ1* = *fst r* · *μ2* **by** *auto*
have *narr*: (*s*,*replace-at* (*s* · *μ1*) *p* (*snd r* · *μ2*)) ∈ *narrows-r-p-s R r p μ1*
unfolding *narrows-r-p-s-def* **using** *notσ r mgu* **by** *auto*
show *?thesis*
proof (*intro exI conjI*, *rule narr*)
show *s* · σ = *s* · *μ1* · δ **unfolding** σ **by** *simp*
show *t* = *replace-at* (*s* · *μ1*) *p* (*snd r* · *μ2*) · δ
unfolding *t* σ τ **using** *notσ*(*1*)
by (*simp add*: *ctxt-of-pos-term-subst*)
qed (*rule* σ, *rule* τ)
qed

lemma *narrows-r-p-s-imp-rstep-r-p-s*: **assumes** *narr*: (*s*,*t*) ∈ *narrows-r-p-s R r p* μ
shows ∃ δ. (*s* · μ, *t*) ∈ *rstep-r-p-s R r p* δ
proof −
from *narr*[*unfolded narrows-r-p-s-def*] **obtain** *μ2* **where** *p*: *p* ∈ *poss s*
and *r*: *r* ∈ *R*
and *mgu*: *mgu-vd ren* (*s* |- *p*) (*fst r*) = *Some* (μ, *μ2*)
and *t*: *t* = *replace-at* (*s* · μ) *p* (*snd r* · *μ2*) **by** *auto*
from *p* **have** *pμ*: *p* ∈ *poss* (*s* · μ) **by** *auto*
from *mgu-vd-sound*[*OF mgu*] **have** *id*: *s* |- *p* · μ = *fst r* · *μ2* **by** *simp*
show *?thesis*
by (*rule exI*[*of* - *μ2*], *unfold rstep-r-p-s-def' t*, *insert r p pμ id*, *auto*)
qed

lemma *narrows-r-p-s-imp-rstep*: **assumes** *narr*: (*s*,*t*) ∈ *narrows-r-p-s R r p* μ
shows (*s* · μ, *t*) ∈ *rstep R*
using *narrows-r-p-s-imp-rstep-r-p-s*[*OF narr*]
using *rstep-r-p-s-imp-rstep* **by** *blast*

lemma *narrow-step-s-to-rstep*: (*s*,*t*) ∈ *narrow-step-s R* μ ⟹ (*s* · μ, *t*) ∈ *rstep R*

using *narrows-r-p-s-imp-rstep*[*of s t R - - μ*]
unfolding *narrow-step-s-def* **by** *auto*

lemma *rstep-imp-narrows-r-p-s*: **assumes** *step*: $(s,t) \in$ *rstep-r-p-s R r p τ*
and *wf*: *wf-trs R*
shows $\exists\ \mu\ \delta\ u.\ (s,u) \in$ *narrows-r-p-s R r p μ* $\wedge\ s = s \cdot \mu \cdot \delta \wedge t = u \cdot \delta \wedge u = t \cdot \mu \wedge \mu \circ_s \delta =$ *Var*
proof −
from *wf*[*unfolded wf-trs-def*] **have** *var-cond*: $\bigwedge\ l\ r.\ (l,r) \in R \Longrightarrow$ *is-Fun l* **by** *force*
from *wf*[*unfolded wf-trs-def*] **have** *var-cond'*: $\bigwedge\ l\ r.\ (l,r) \in R \Longrightarrow$ *vars-term r* $\subseteq$ *vars-term l* **by** *force*

from *step*[*unfolded rstep-r-p-s-def'*] **have** *p*: *p* ∈ *poss s* **and** *r*: *r* ∈ *R* **and** *id*: *s* |- *p* · *Var* = *fst r* · *τ*
and *t*: *t* = *replace-at s p* (*snd r* · *τ*)
by *auto*
obtain *l1 r1* **where** *lr1*: *r* = (*l1*,*r1*) **by** *force*
from *var-cond*[*OF r*[*unfolded lr1*]] **have** *is-Fun*: *is-Fun* (*s* |- *p*) **using** *id lr1* **by** (*cases l1*, *auto*)
from *mgu-vd-complete*[*OF id*, *of ren*] **obtain** *μ1 μ2 δ* **where**
mgu: *mgu-vd ren* (*s* |- *p*) (*fst r*)
= *Some* (*μ1*, *μ2*) **and** *Var*: *Var* = *μ1* ∘*s δ* **and** *τ*: *τ* = *μ2* ∘*s δ* **and** *id*: *s* |- *p* · *μ1* = *fst r* · *μ2* **by** *auto*
define *u* **where** *u* = *replace-at* (*s* · *μ1*) *p* (*snd r* · *μ2*)

have *narr*: (*s*, *u*) ∈ *narrows-r-p-s R r p μ1*
unfolding *narrows-r-p-s-def u-def* **using** *is-Fun p r mgu* **by** *auto*
from *narrows-r-p-s-imp-rstep*[*OF this*] **have** (*s* · *μ1*, *u*) ∈ *rstep R* **.**

with *var-cond'* **have** *vsu*: *vars-term* (*s* · *μ1*) ⊇ *vars-term u*
unfolding *rstep.simps* **by** (*force simp*: *vars-term-ctxt-apply vars-term-subst*)

have ∃ *y*. *μ1 x* = *Var y* **for** *x* **using** *arg-cong*[*OF Var*, *of* λ *σ*. *σ x*, *unfolded subst-compose-def*]
by (*cases μ1 x*, *auto*)
then obtain *μ1'* **where** *μ1*: *μ1* = *Var o μ1'* **unfolding** *o-def* **by** *metis*
have *δmu*: *δ* (*μ1' x*) = *Var x* **for** *x* **using** *arg-cong*[*OF Var*[*unfolded μ1 subst-compose-def o-def*], *of* λ *σ*. *σ x*]
by *simp*
have *u* · (*δ* ∘*s μ1*) = *u* · *Var*
proof (*rule term-subst-eq*)
fix *x*
assume *x* ∈ *vars-term u*
with *vsu* **obtain** *y* **where** *x* ∈ *vars-term* (*μ1 y*)
by (*auto simp*: *vars-term-subst*)
from *this*[*unfolded μ1*]
have *μ1'*: *μ1' y* = *x* **by** *simp*

```
    from δmu[of y, unfolded this]
    have δ x = Var y by auto
    from arg-cong[OF this, of λ t. t · μ1]
    show (δ ∘s μ1) x = Var x unfolding subst-compose-def
      unfolding μ1 o-def using μ1' by simp
  qed
  hence uu: u · δ · μ1 = u by auto

  have s': s = s · μ1 · δ using Var by (simp add: eval-subst)
  have tu: t = u · δ
    unfolding t τ u-def using p Var
    by (simp add: ctxt-of-pos-term-subst)
  have ut: u = t · μ1
    by (subst uu[symmetric], unfold tu, auto)
  show ?thesis
    by (intro exI conjI, rule narr, rule s', auto intro: tu ut simp: Var)
qed

lemma rstep-imp-narrows-s: assumes step: (s,t) ∈ rstep R
  and wf: wf-trs R
shows ∃ μ δ u. (s,u) ∈ narrow-step-s R μ ∧ s = s · μ · δ ∧ t = u · δ ∧ u = t · μ
∧ μ ∘s δ = Var
proof −
  from step obtain r p τ where (s,t) ∈ rstep-r-p-s R r p τ
    using rstep-iff-rstep-r-p-s by blast
  note main = rstep-imp-narrows-r-p-s[OF this]
  from main[OF wf] obtain μ δ u
    where ∗: (s, u) ∈ narrows-r-p-s R r p μ s = s · μ · δ t = u · δ u = t · μ μ ∘s
δ = Var
    by fast
  thus ?thesis unfolding narrow-step-s-def by fast
qed

lemma rstep-imp-narrows: assumes step: (s,t) ∈ rstep R
  and wf: wf-trs R
shows ∃ μ δ u. (s,u) ∈ narrow-step R ∧ t = u · δ ∧ u = t · μ
  using rstep-imp-narrows-s[OF assms] unfolding narrow-step-to-s by fast

lemma narrow-instance-pos-in-subst: assumes
  narr: (t · γ, s) ∈ narrows-r-p-s (R :: ('f,'v)trs) (l,r) p μ
  and pos: p ∉ fun-poss t
  and vars: vars-term r ⊆ vars-term l
  and lin: linear-term (t :: ('f,'v)term) linear-term r
shows ∃γ. t · γ = s
proof −
  from narrows-r-p-s-imp-rstep-r-p-s[OF narr]
  obtain δ where (t · (γ ∘s μ), s) ∈ rstep-r-p-s R (l,r) p δ  by auto
  from narrowing-right-linear-one-step-simulation-r-p-s[OF this lin vars] pos
  show ?thesis by auto
```

qed

lemma *narrow-instance-pos-in-term*: **fixes** $t :: ('f,'v)term$
assumes *narr*: $(t, s) \in narrows\text{-}r\text{-}p\text{-}s\ R\ (l, r)\ p\ \mu 1$
and *narr-inst*: $(t \cdot \gamma, u) \in narrows\text{-}r\text{-}p\text{-}s\ R\ (l, r)\ p\ \mu 2$
shows $\exists \delta.\ s \cdot \delta = u \wedge \gamma \circ_s \mu 2 = \mu 1 \circ_s \delta$
proof −
define C **where** $C = ctxt\text{-}of\text{-}pos\text{-}term\ p\ t$
define tp **where** $tp = t\ |\text{-}\ p$
from *narr*[*unfolded narrows-r-p-s-def fst-conv snd-conv*] **obtain** $\mu 1'$ **where**
p: $p \in poss\ t$ **and** *mgu*: $mgu\text{-}vd\ ren\ tp\ l = Some\ (\mu 1, \mu 1')$
and *s*: $s = (ctxt\text{-}of\text{-}pos\text{-}term\ p\ (t \cdot \mu 1))\langle r \cdot \mu 1' \rangle$ **using** *tp-def* **by** *blast*
have *ctxt*: $ctxt\text{-}of\text{-}pos\text{-}term\ p\ (t \cdot \mu 1) = C \cdot_c \mu 1$ **for** $\mu 1 :: ('f,'v)subst$
using *p* **unfolding** *C-def* **by** (*rule ctxt-of-pos-term-subst*)
from *s this*
have *s*: $s = (C \cdot_c \mu 1)\langle r \cdot \mu 1' \rangle$ **by** *auto*
have *t*: $t = C\ \langle\ tp\ \rangle$ **using** *p* **unfolding** *C-def tp-def*
by (*simp add*: *ctxt-supt-id*)
from *mgu-vd-sound*[*OF mgu*] **have** *mu1*: $tp \cdot \mu 1 = l \cdot \mu 1'$ **by** *auto*
from *p* **have** $t \cdot \gamma\ |\text{-}\ p = tp \cdot \gamma$ **unfolding** *tp-def* **by** *auto*

with *narr-inst*[*unfolded narrows-r-p-s-def fst-conv snd-conv*, *simplified*]
obtain $\mu 2'$ **where** *mgu2*: $mgu\text{-}vd\ ren\ (tp \cdot \gamma)\ l = Some\ (\mu 2, \mu 2')$
and *u*: $u = (ctxt\text{-}of\text{-}pos\text{-}term\ p\ (t \cdot (\gamma \circ_s \mu 2)))\langle r \cdot \mu 2' \rangle$ **by** *auto*
from *u*[*unfolded ctxt*]
have *u*: $u = (C \cdot_c (\gamma \circ_s \mu 2))\langle r \cdot \mu 2' \rangle$ **by** *simp*

from *mgu-vd-sound*[*OF mgu2*] **have** $tp \cdot (\gamma \circ_s \mu 2) = l \cdot \mu 2'$ **by** *auto*
from *mgu-vd-complete*[*OF this*, *of ren*, *unfolded mgu*]
obtain δ **where** *delt*: $\gamma \circ_s \mu 2 = \mu 1 \circ_s \delta\ \mu 2' = \mu 1' \circ_s \delta\ tp \cdot \mu 1 = l \cdot \mu 1'$ **by** *auto*
show *?thesis* **unfolding** *s u* **using** *delt*
by (*intro exI*[*of - δ*], *auto*)
qed

lemma *SN-on-narrows-imp-SN-on-rstep*: **fixes** $R :: ('f,'v)trs$
assumes *SN-on* (*narrow-step R*) {*s*}
and *wf*: *wf-trs R*
shows *SN-on* (*rstep R*) {*s*}
proof
fix *t*
assume *start*: $t\ 0 \in \{s\}$ **and** *steps*: $\forall\ i.\ (t\ i, t\ (Suc\ i)) \in rstep\ R$
hence *t0s*: $t\ 0 \cdot Var = s$ **by** *auto*

define *step-cond* **where** *step-cond* = $(\lambda\ (i :: nat)\ ti\ (\sigma i :: ('f,'v)subst)\ tsi\ (\sigma si :: ('f,'v)subst).\ (ti, tsi) \in narrow\text{-}step\ R)$
note *simu* = *dependent-nat-choice2-start*[*of* $\lambda\ i\ u\ \mu.\ t\ i \cdot \mu = u\ s\ Var$ *step-cond*, *OF t0s*]

```
  {
    fix v and μ :: ('f,'v)subst and   n
    assume v: t n · μ = v
    from steps have (t n, t (Suc n)) ∈ rstep R by auto
    hence (v, t (Suc n) · μ) ∈ rstep R unfolding v[symmetric] ..
    from rstep-imp-narrows[OF this wf]
    obtain μ' u where (v, u) ∈ narrow-step R u = t (Suc n) · μ · μ'
      by blast
    hence ∃ x' y'. t (Suc n) · y' = x' ∧ step-cond n v μ x' y'
      unfolding step-cond-def
      by (intro exI[of - u] exI[of - μ ∘s μ'], auto)
  }
  from simu[OF this, unfolded step-cond-def] obtain u where
    u 0 = s
    (u i, u (Suc i)) ∈ narrow-step R for i
    by blast

  thus False using ‹SN-on (narrow-step R) {s}› by blast
qed

lemma exists-narrow-steps-to-infinite-rsteps:  fixes R :: ('f,'v)trs
  defines Q ≡ λ (n :: nat) s (σ :: ('f,'v)subst) u (δ :: ('f,'v)subst).
      (∃ μ. (s,u) ∈ narrow-step-s R μ ∧ σ = μ ∘s δ)
  assumes wf: wf-trs R
    and single-steps: ⋀ n s σ. P n s σ ⟹ ∃ u δ. P (Suc n) u δ ∧ Q n s σ u δ
    and P0: P 0 r σ
shows ¬ SN-on (narrow-step R) {r}
  ¬ SN-on (rstep R) ((narrow-step R)* `` {r})
proof −
  from dependent-nat-choice2-start[of P r σ Q, OF P0 single-steps]
  obtain s δ where 0: s 0 = r δ 0 = σ
    and ∗: P n (s n) (δ n) ∀ n. Q n (s n) (δ n) (s (Suc n)) (δ (Suc n)) for n by
blast
  from choice[OF ∗(2)[unfolded Q-def]] obtain μ
    where steps: ⋀ i. (s i, s (Suc i)) ∈ narrow-step-s R (μ i)
      and δ: ⋀ i. δ i = μ i ∘s δ (Suc i)
    by auto
  define γ where γ = rec-nat Var (λ i γ. γ ∘s μ i)
  have γ: γ 0 = Var ⋀ i. γ (Suc i) = γ i ∘s μ i unfolding γ-def by auto
  define t where t = r · σ

  have σ: σ = γ i ∘s δ i for i
  proof (induct i)
    case (Suc i)
    thus ?case by (simp add: δ[of i] γ subst-compose-assoc)
  qed (auto simp: 0 γ)
  have rsteps: (s i · μ i, s (Suc i)) ∈ rstep R for i using narrow-step-s-to-rstep[OF
steps[of i]] .
  {
```

```
  fix i
  from rstep-vars-term[OF - rsteps] wf
  have vars-term (s i · μ i) ⊇ vars-term (s (Suc i))
    by (auto simp: wf-trs-def)
} note vars-si = this


have vars: vars-term (s i) ⊆ vars-term (r · γ i) for i
proof (induct i)
  case (Suc i)
  with vars-si[of i] show ?case by (force simp: vars-term-subst γ)
qed (auto simp: 0 γ)

have tsγ: t = r · γ i · δ i for i unfolding t-def σ[of i] 0 by simp


from steps have nsteps: (s i, s (Suc i)) ∈ narrow-step R for i
  unfolding narrow-step-to-s by auto
with 0(1) show ¬ SN-on (narrow-step R) {r} by force




define meas-fun where meas-fun i = size (funs-term-ms (r · γ i)) for i
define dmeas-fun where dmeas-fun i = size (funs-term-ms t) − meas-fun i for
i

{
  fix i
  define rg where rg = r · γ i
  have sub: vars-term (s i) ⊆ vars-term rg using vars[of i] unfolding rg-def by
auto

  let ?diff = ∑# ((funs-term-ms ∘∘ μ) i '# vars-term-ms rg)
  have bnd: meas-fun (Suc i) ≤ size (funs-term-ms t)
    unfolding tsγ[of Suc i] meas-fun-def funs-term-ms-subst-apply by simp

  have funs-term-ms (r · γ (Suc i)) = funs-term-ms (rg · μ i) unfolding γ rg-def
by simp
    also have ... = funs-term-ms rg + sum-mset ((funs-term-ms o μ i) '#
vars-term-ms rg)
    unfolding funs-term-ms-subst-apply o-def by simp
  finally have id: funs-term-ms (r · γ (Suc i)) = funs-term-ms rg + ?diff
    by auto
  from arg-cong[OF this, of size]
  have meas: meas-fun (Suc i) = meas-fun i + size ?diff
    by (auto simp: meas-fun-def rg-def)
  hence meas-fun (Suc i) ≥ meas-fun i by auto
  with bnd have le: dmeas-fun (Suc i) ≤ dmeas-fun i unfolding dmeas-fun-def
```

by *auto*
  {
   assume ∃ *x* ∈ *vars-term* (*s i*). *is-Fun* (*μ i x*)
   with *sub* **obtain** *x* **where** *x*: *x* ∈# *vars-term-ms rg* **and** *f*: *is-Fun* (*μ i x*) **by** *auto*
    then obtain *xs* **where** *vars*: *vars-term-ms rg* = *add-mset x xs* **by** (*metis mset-add*)
   with *f* **have** *size ?diff* ≠ *0* **by** (*cases μ i x*, *auto*)
   hence *meas-fun* (*Suc i*) > *meas-fun i* **unfolding** *meas* **by** *linarith*
    with *bnd* **have** *dmeas-fun* (*Suc i*) < *dmeas-fun i* **unfolding** *dmeas-fun-def* **by** *auto*
  }
  note *le this*
 } **note** *dmeas* = *this*

 have ∃*j*. ∀ *i*≥*j*. (*dmeas-fun i*, *dmeas-fun* (*Suc i*)) ∈ {(*x*, *y*). *y* ≤ *x*} − {(*x*, *y*). *y* < *x*}
   by (*rule non-strict-ending*[*of dmeas-fun*], *insert dmeas SN-nat-gt*, *auto simp*: *SN-defs*)
 then obtain *j* **where** *dmeas-eq*: ⋀ *i*. *i* ≥ *j* ⟹ *dmeas-fun* (*Suc i*) = *dmeas-fun i* **by** *fastforce*

 have *muVar*: *μ i* ‘ *vars-term* (*s i*) ⊆ *range Var* **if** *i* ≥ *j* **for** *i*
  using *dmeas-eq*[*OF that*] *dmeas*(*2*)[*of i*] **by** *force*

 define *vs* **where** *vs i* = *vars-term* (*s* (*i* + *j*) · *μ* (*i* + *j*)) **for** *i*
 have *fin*[*simp*,*intro*]: *finite* (*vs i*) **for** *i* **unfolding** *vs-def* **by** *auto*
 define *meas-var* **where** *meas-var i* = *card* (*vs i*) **for** *i*
 {
  fix *i*
  assume *i* ≥ *j*
   from *muVar*[*OF this*] **have** ∀ *x*. ∃ *y*. *x* ∈ *vars-term* (*s i*) ⟶ *μ i x* = *Var y* **by** *auto*
  from *choice*[*OF this*] **obtain** *μi* **where** ⋀ *x*. *x* ∈ *vars-term* (*s i*) ⟹ *μ i x* = *Var* (*μi x*)
    by *auto*
  hence *s i* · *μ i* = *s i* · (*Var o μi*)
    by (*intro term-subst-eq*, *auto*)
  hence ∃ *μi*. *s i* · *μ i* = *s i* · (*Var o μi*) **by** *auto*
 }
 hence ∀ *i*. ∃ *μ*′. *i* ≥ *j* ⟶ *s i* · *μ i* = *s i* · (*Var o μ*′) **by** *blast*
 from *choice*[*OF this*] **obtain** *μ*′ **where**
  μ′: ⋀ *i*. *i* ≥ *j* ⟹ *s i* · *μ i* = *s i* · (*Var o* (*μ*′ *i*)) **by** *blast*
 {
  fix *i*
  have *vs* (*Suc i*) = *μ*′ (*Suc* (*i* + *j*)) ‘ *vars-term* (*s* (*Suc* (*i* + *j*)))

```
    using μ'[of Suc (i + j)] vs-def[of Suc i]
    by (auto simp: vars-term-subst)
   also have ... ⊆ μ' (Suc (i + j)) ‘ vs i
    using vars-si unfolding vs-def by auto
   ultimately have sub: vs (Suc i) ⊆ μ' (Suc i + j) ‘ vs i by auto
   from card-mono[OF finite-imageI[OF fin] this]
  have meas-var (Suc i) ≤ meas-var i unfolding meas-var-def using card-image-le[OF
fin]
    by (metis le-trans)
   note sub this
 } note meas-var = this

 have ∃j. ∀i≥j. (meas-var i, meas-var (Suc i)) ∈ {(x, y). y ≤ x} − {(x, y). y <
x}
   by (rule non-strict-ending[of meas-var], insert meas-var SN-nat-gt, auto simp:
SN-defs)
 then obtain k where meas-var-eq: ⋀ i. i ≥ k ⟹ meas-var (Suc i) = meas-var
i by fastforce
 define l where l = Suc k + j


 {
  fix ii
  assume ii: ii ≥ l
  define i where i = ii − Suc 0
  define i' where i' = i − j
   have i ≥ k + j and ii: ii = Suc i and iij: ii ≥ j using ii unfolding l-def
i-def by auto
  hence i: i = i' + j Suc i' + j = Suc i i' ≥ k unfolding i'-def by auto
  from meas-var-eq[OF i(3), unfolded meas-var-def]
  have card: card (vs (Suc i')) = card (vs i') by auto
  from meas-var(1)[of i', unfolded i(2)]
  have vs (Suc i') ⊆ μ' (Suc i) ‘ vs i' by auto
  with card have inj-on (μ' (Suc i)) (vs i')
    by (metis card-image-le card-seteq eq-card-imp-inj-on fin finite-imageI)
  from this[unfolded vs-def, folded i(1)]
  have inj-on (μ' (Suc i)) (vars-term (s i · μ i)) by auto
  with vars-si[of i]
  have inj-on (μ' ii) (vars-term (s ii)) unfolding ii by (metis inj-on-subset)
 } note inj = this

 define sl where sl i = s (i + l) for i
 {
  fix i
  assume i: i ≥ l
  hence ij: i ≥ j unfolding l-def by auto
  let ?V = vars-term (s i)
  from inj[OF i]
  have inj-on (μ' i) ?V .
```

from *the-inv-into-f-f*[*OF this*]
obtain *inv* **where** *inv*: $\bigwedge x.\ x \in ?V \Longrightarrow inv\ (\mu'\ i\ x) = x$ **by** *blast*
have $s\ i \cdot \mu\ i \cdot (Var\ o\ inv) = s\ i \cdot ((Var\ o\ \mu'\ i) \circ_s (Var\ o\ inv))$
using μ'[*OF ij*] **by** *simp*
also have $\ldots = s\ i \cdot Var$
by (*rule term-subst-eq*, *insert inv*, *auto simp*: *o-def subst-compose-def*)
finally have $s\ i \cdot \mu\ i \cdot (Var\ o\ inv) = s\ i$ **by** *simp*
from *rstep-subst*[*OF rsteps*[*of i*], *of* (*Var o inv*), *unfolded this*]
have $\exists\ \gamma.\ (s\ i,\ s\ (Suc\ i) \cdot \gamma) \in rstep\ R$ **by** *auto*
}
hence $\forall\ i.\ \exists\ \gamma.\ (sl\ i,\ sl\ (Suc\ i) \cdot \gamma) \in rstep\ R$ **unfolding** *sl-def* **by** *simp*
from *choice*[*OF this*] **obtain** γ **where** *rsteps*: $(sl\ i,\ sl\ (Suc\ i) \cdot \gamma\ i) \in rstep\ R$ **for** i
by *auto*

define γ' **where** $\gamma' = rec\text{-}nat\ Var\ (\lambda\ i\ gam.\ \gamma\ i \circ_s gam)$
define t **where** $t\ i = sl\ i \cdot \gamma'\ i$ **for** i
have *rsteps*: $(t\ i,\ t\ (Suc\ i)) \in rstep\ R$ **for** i
unfolding *t-def* γ'*-def* **using** *rstep-subst*[*OF rsteps*[*of i*]] **by** *simp*
have *t0*: $t\ 0 = s\ l$ **unfolding** *t-def sl-def* γ'*-def* **by** *simp*
from *rsteps t0* **have** *nSN*: $\neg$ *SN-on* (*rstep R*) $\{s\ l\}$ **by** *auto*

have $s\ l \in (narrow\text{-}step\ R)^*\ ``\ \{r\}$
proof (*induct l*)
case $*$: *0*
with *0* **show** *?case* **by** *auto*
next
case (*Suc i*)
with *nsteps*[*of i*] **show** *?case* **by** (*metis rtrancl-image-advance*)
qed
with *nSN* **show** $\neg$ *SN-on* (*rstep R*) $((narrow\text{-}step\ R)^*\ ``\ \{r\})$
by (*metis SN-onE rsteps t0*)
qed

end
end

5 Dershowitz' Right-Forward Closures for Proving Termination

Right-Forward Closures (RFC) are defined, and it is shown that termination on all terms is equivalent to termination starting on RFC for TRSs that are right-linear, or for overlay TRSs that are locally confluent.

theory *Right-Forward-Closure*
imports
Linear-Narrowing
First-Order-Rewriting.Critical-Pairs

Gramlich-Innermost-Switch
begin

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

definition *right-forw-closure* :: (*'f*,*'v*)*trs* ⇒ (*'f*,*'v*)*term set* **where**
right-forw-closure R = (*narrow-step ren R*)* `` *rhss R*

lemma *right-forw-closure-SN-main*: **fixes** *R* :: (*'f*,*'v*)*trs*
assumes *right-lin*: ⋀ *lr*. *lr* ∈ *R* ⟹ *linear-term* (*snd lr*)
and *wf*: *wf-trs R*
and *nSN*: ¬ *SN* (*rstep R*)
shows ¬ *SN-on* (*rstep R*) (*right-forw-closure R*)
proof −
from *wf* **have** *var-cond*: ⋀ *lr*. *lr* ∈ *R* ⟹ *vars-term* (*snd lr*) ⊆ *vars-term* (*fst lr*)
by (*auto simp*: *wf-trs-def*)
from *wf* **have** *var-cond2*: ⋀ *lr*. *lr* ∈ *R* ⟹ *is-Fun* (*fst lr*)
by (*force simp*: *wf-trs-def*)
note *simu* = *narrowing-right-linear-one-step-simulation*[*of - - - R ren*, *OF - - right-lin var-cond*]
let *?R* = *rstep R*
let *?N* = *narrow-step ren R*
let *?NS* = *narrow-step-s ren R*
let *?RFC* = *right-forw-closure R*
have *R-to-RFC*: *lr* ∈ *R* ⟹ *snd lr* ∈ *?RFC* **for** *lr* **unfolding** *right-forw-closure-def* **by** *auto*
from *nSN* **obtain** *t* **where** ¬ *SN-on* (*rstep R*) {*t*} **unfolding** *SN-defs* **by** *auto*
from *not-SN-imp-subt-Tinf*[*OF this*] **obtain** *t* **where** *t* ∈ *Tinf* (*rstep R*) **by** *auto*
from *Tinf-rstep-imp-first-root-step*[*OF this*]
obtain *tm1 t0* **where** *Tinf*: *tm1* ∈ *Tinf ?R* **and** (*tm1*,*t0*) ∈ *rrstep R* **and** *nt0*: ¬ *SN-on ?R* {*t0*} **by** *auto*
from *this*(*2*) **obtain** *l r0* σ*0* **where** *lr*: (*l*,*r0*) ∈ *R* **and** *tm1*: *tm1* = *l* · σ*0* **and** *t0*: *t0* = *r0* · σ*0*
by (*rule rrstepE*)
have *r0-rfc*: *r0* ∈ *?RFC* **using** *R-to-RFC*[*OF lr*] **by** *simp*
have *linr*: *linear-term r0* **using** *right-lin*[*OF lr*] **by** *simp*
define *SNT* **where** *SNT* = {*t*. *SN-on ?R* {*t*}}
define *stm* :: (*'f*,*'v*)*term* ⇒ - **where** *stm* = *subst-term-mset*
{
fix *x*
assume *x* ∈ *vars-term r0*
with *var-cond*[*OF lr*] **have** *x*: *x* ∈ *vars-term l* **by** *auto*
from *var-cond2*[*OF lr*] **obtain** *f ls* **where** *l*: *l* = *Fun f ls* **by** *auto*
with *x* **obtain** *li* **where** *li*: *li* ∈ *set ls* **and** *x* ∈ *vars-term li* **by** *auto*

```
    hence subt: li · σ0 ⊵ σ0 x
      by (metis eval-term.simps(1) supteq-subst vars-term-supteq)
    hence tm1 ▷ σ0 x unfolding tm1 using li
      by (metis l subst-image-subterm x)
    with Tinf[unfolded Tinf-def] have SN-on ?R {σ0 x} by auto
    hence σ0 x ∈ SNT unfolding SNT-def by auto
  }
  hence r0-SNT: set-mset (stm r0 σ0) ⊆ SNT
    unfolding subst-term-mset-def stm-def by auto
  define Rel where Rel = restrict-SN-supt ?R
  define mRel where mRel = mult (Rel^-1)
  have SN-Rel: SN Rel unfolding Rel-def
    by (rule SN-restrict-SN-supt-rstep)
  hence wf (Rel^-1) by (rule SN-imp-wf)
  hence wf-mRel: wf mRel unfolding mRel-def by (rule wf-mult)
  define RR where RR = ?R
  have R-SNT: (u,v) ∈ ?R ⟹ u ∈ SNT ⟹ v ∈ SNT for u v unfolding SNT-def
    by (simp add: step-preserves-SN-on)

  have supt-SNT: u ▷ v ⟹ u ∈ SNT ⟹ v ∈ SNT for u v unfolding SNT-def
    by fast

  {
    fix s σ δ
    assume SNT: set-mset (stm s σ) ⊆ SNT and decr: (stm s δ, stm s σ) ∈ mult1
(?R^-1)
    {
      fix u
      assume u ∈# stm s δ
      with decr obtain v where v: v ∈# stm s σ and disj: u = v ∨ (v,u) ∈ ?R
        unfolding mult1-def RR-def[symmetric] by force
      from SNT v have v ∈ SNT by auto
      with R-SNT[OF - this] disj have u ∈ SNT by auto
    }
    hence SNT': set-mset (stm s δ) ⊆ SNT by auto
    from decr[unfolded mult1-def, simplified]
    obtain t M K where split: stm s σ = add-mset t M stm s δ = M + K
      and steps: ⋀ u. u ∈# K ⟹ (t, u) ∈ ?R by auto
    {
      fix u
      assume u: u ∈# K
      with split SNT SNT' have SNT: t ∈ SNT u ∈ SNT by auto
      from steps[OF u] have (t,u) ∈ ?R by auto
        with SNT have (u,t) ∈ Rel^-1 unfolding Rel-def restrict-SN-supt-def
restrict-SN-def SNT-def by auto
    }
    hence (stm s δ, stm s σ) ∈ mult1 (Rel^-1) unfolding mult1-def split by auto
    hence (stm s δ, stm s σ) ∈ mRel unfolding mRel-def by (simp add: mult-def)
    note SNT' this
```

} note *rstep-into-mRel = this*

have *tr*: *trans* {◁}
by (*simp add*: *trans-supt*)

{
fix *s t σ δ*
assume *SNT*: *set-mset* (*stm t σ*) ⊆ *SNT* **and** *decr*: (*stm s δ*, *stm t σ*) ∈ *mult* {◁}
from *mult-implies-one-step*[*OF tr decr*] **obtain** *I J K*
where *split*: *stm t σ* = *I* + *J stm s δ* = *I* + *K* **and** *J*: *J* ≠ {#} **and** *step*: ⋀ *k*. *k*∈#*K* ⟹ ∃ *j*∈#*J*. *k* ◁ *j* **by** *auto*
{
fix *u*
assume *u* ∈# *stm s δ*
then obtain *v* **where** *v* ∈# *stm t σ u* = *v* ∨ *v* ▷ *u* **unfolding** *split* **using** *step* **by** *auto*
with *supt-SNT*[*of v u*] *SNT* **have** *u* ∈ *SNT* **by** *auto*
}
hence *SNT'*: *set-mset* (*stm s δ*) ⊆ *SNT* **by** *auto*
{
fix *k*
assume *k*: *k* ∈# *K*
from *step*[*OF this*] **obtain** *j* **where** *j*: *j* ∈# *J j* ▷ *k* **by** *auto*
from *SNT SNT' k j* **have** *k* ∈ *SNT j* ∈ *SNT* **by** (*auto simp*: *split*)
with *j*(*2*) **have** (*j*,*k*) ∈ *Rel* **unfolding** *Rel-def restrict-SN-supt-def restrict-SN-def SNT-def* **by** *auto*
hence ∃ *j* ∈# *J*. (*k*,*j*) ∈ *Rel*^−*1* **using** *j* **by** *auto*
}
hence (*stm s δ*, *stm t σ*) ∈ *mRel*
unfolding *split mRel-def* **using** *one-step-implies-mult*[*OF J*, *of K Rel*^−*1 I*]
by *auto*
note *SNT' this*
} note *supt-into-mRel = this*

define *P* **where** *P s σ* = (*linear-term s* ∧ *set-mset* (*stm s σ*) ⊆ *SNT* ∧ *s* ∈ *?RFC*) **for** *s σ*

define *narr-step* **where** *narr-step s σ u μ δ* = ((*s*, *u*) ∈ *?NS μ* ∧ *σ* = *μ* ∘*s δ*)
for *s u* :: ('*f*,'*v*)*term* **and** *μ σ δ* :: ('*f*,'*v*)*subst*

define *step-cond* **where** *step-cond s σ t u δ* = (*t* = *u* · *δ* ∧
P u δ ∧ ((*stm u δ*, *stm s σ*) ∈ *mRel* ∨ *stm u δ* = *stm s σ* ∧ (∃ *μ*. *narr-step s σ u μ δ*)))
for *s σ t δ u*

{
fix *s* :: ('*f*,'*v*)*term* **and** *σ t*
assume *P s σ* **and** (*s* · *σ*, *t*) ∈ *?R*

```
    hence lin: linear-term s (s · σ, t) ∈ ?R and SNT: set-mset (stm s σ) ⊆ SNT
and RFC: s ∈ ?RFC
      by (auto simp: P-def)
    from simu[OF this(2,1)] obtain μ δ u where
      (t = s · δ ∧ (stm s δ, stm s σ) ∈ mult1 ((rstep R)^-1)) ∨
          (t = u · δ ∧ linear-term u ∧ (s, u) ∈ ?NS μ ∧
          (stm u δ, stm s σ) ∈ (mult {◁})^= ∧
           ((stm u δ, stm s σ) ∈ mult {◁} ∨ narr-step s σ u μ δ)) (is ?A ∨ ?B)
      by (auto simp: stm-def narr-step-def)
    hence ∃ u. t = u · δ ∧ P u δ ∧
              ((stm u δ, stm s σ) ∈ mRel ∨ stm u δ = stm s σ ∧ narr-step s σ u μ
δ)
    proof
      assume ?A
      with rstep-into-mRel[OF SNT, of δ]
      have t = s · δ ∧ (stm s δ, stm s σ) ∈ mRel ∧ set-mset (stm s δ) ⊆ SNT by
blast
      with ‹?A› RFC lin show ?thesis unfolding P-def
        by (intro exI[of - s], auto)
    next
      assume ?B
      hence B: t = u · δ linear-term u (s, u) ∈ ?NS μ (stm u δ, stm s σ) ∈ (mult
{◁})^=
        and disj: (stm u δ, stm s σ) ∈ mult {◁} ∨ narr-step s σ u μ δ by auto
      note supt = supt-into-mRel[OF SNT, of u δ]
     from B(3) have (s,u) ∈ ?N unfolding narrow-step-def narrow-step-s-def by
fastforce
     with RFC have RFC: u ∈ ?RFC unfolding right-forw-closure-def by (metis
rtrancl-image-advance)
      from B(4) supt SNT have SNT: set-mset (stm u δ) ⊆ SNT by auto
      show ?thesis
      proof (cases (stm u δ, stm s σ) ∈ mult {◁})
        case True
        from B True supt lin RFC show ?thesis by (auto simp: P-def)
      next
        case False
        with B(4) have eq: stm u δ = stm s σ by auto
       from disj False eq B lin RFC SNT show ?thesis unfolding P-def by blast
      qed
    qed
    hence ∃ u δ. step-cond s σ t u δ by (auto simp: step-cond-def)
  } note simu = this


  have P0: P r0 σ0 unfolding P-def using linr r0-SNT R-to-RFC[OF lr] by
force
  from nt0 obtain t where t 0 = t0 and steps: ⋀ i. (t i, t (Suc i)) ∈ ?R by
auto
  hence r0 · σ0 = t 0 using t0 by auto
```

with *P0* **have** *r0 · σ 0 = t 0 ∧ P r0 σ0* **by** *auto*
note *dep-choice = dependent-nat-choice2-start[of λ i s sig. s · sig = t i ∧ P s sig r0 σ0*
λ i p1 p2 q1 q2. step-cond p1 p2 (t (Suc i)) q1 q2, OF this]
{
fix *s sig i*
assume *s · sig = t i ∧ P s sig*
with *simu[of s sig t (Suc i)] steps[of i]* **obtain** *u δ* **where**
∗: step-cond s sig (t (Suc i)) u δ **by** *auto*
let *?q = (u,δ)*
from *∗* **have** *u · δ = t (Suc i) ∧ P u δ*
unfolding *step-cond-def* **by** *auto*
with *∗* **have**
∃ u δ. (u · δ = t (Suc i) ∧ P u δ) ∧
step-cond s sig (t (Suc i)) u δ **by** *auto*
}
from *dep-choice[OF this]* **obtain** *r σ* **where**
∗: P (r i) (σ i) r i · σ i = t i step-cond (r i) (σ i) (t (Suc i)) (r (Suc i)) (σ (Suc i)) **for** *i*
by *blast*
define *stmp* **where** *stmp i = stm (r i) (σ i)* **for** *i*
{
fix *i*
have *(stmp i, stmp (Suc i)) ∈ Id ∪ mRel^−1*
unfolding *stmp-def* **using** *∗(3)[of i]* **unfolding** *step-cond-def* **by** *auto*
}
hence *rel-chain*: *∀ i. (stmp i, stmp (Suc i)) ∈ Id ∪ mRel^−1* **by** *auto*
from *wf-mRel* **have** *SN*: *SN (mRel^−1)*
by (*simp add*: *SN-iff-wf*)
have *∃j. ∀i≥j. (stmp i, stmp (Suc i)) ∈ Id − mRel^−1*
by (*rule non-strict-ending*[*OF rel-chain*], *insert SN*, *auto simp*: *SN-def*)
then obtain *j* **where** *eq-steps*: *⋀ i. i ≥ j ⟹ (stmp i, stmp (Suc i)) ∈ Id − mRel^−1* **by** *auto*
from *∗(1)*[*of j, unfolded P-def*]
have *rj*: *r j ∈ ?RFC* **by** *auto*
{
fix *n s σ′*
from *eq-steps*[*of n + j*] *∗(3)*[*of n + j, unfolded step-cond-def*]
obtain *μ* **where** *narr-step (r (n + j)) (σ (n + j)) (r (Suc (n + j))) μ (σ (Suc (n + j)))*
using *stmp-def* **by** *fastforce*
hence *s = r (n + j) ∧ σ′ = σ (n + j) ⟹ ∃u δ. (u = r (Suc n + j) ∧ δ = σ (Suc n + j)) ∧ (∃μ. (s, u) ∈ ?NS μ ∧ σ′ = μ ∘s δ)*
by (*intro exI*[*of - r (Suc (n + j))*] *exI*[*of - σ (Suc (n + j))*], *unfold narr-step-def*, *auto*)
}
from *exists-narrow-steps-to-infinite-rsteps*[*OF wf, of λ i t sig. t = r (i + j) ∧ sig = σ (i + j) ren r j σ j,*
OF this]

have *main*: ¬ *SN-on ?N* {*r j*} ¬ *SN-on ?R* (*?N** `` {*r j*})
by *auto*
from *rj main*(*1*) **have** ¬ *SN-on ?N* (*rhss R*) **unfolding** *right-forw-closure-def*
by (*metis SN-on-Image-rtrancl-iff SN-on-def singletonD*)
from *rj main*(*2*) **show** ¬ *SN-on ?R* (*right-forw-closure R*)
unfolding *right-forw-closure-def*
by (*metis Image-singleton-iff SN-on-def rtrancl-image-advance-rtrancl*)
qed

theorem *right-linear-SN-rstep-RFC*:
assumes *wf*: *wf-trs R*
and *rlin*: ⋀*lr*. *lr* ∈ *R* ⟹ *linear-term* (*snd lr*)
shows *SN* (*rstep R*) ⟷ *SN-on* (*rstep R*) (*right-forw-closure R*) (**is** *?A* = *?B*)
proof
show *?A* ⟹ *?B* **by** *force*
next
assume *?B*
with *right-forw-closure-SN-main*[*OF rlin wf*]
show *?A* **by** *blast*
qed

context
fixes *R* :: (*'f*,*'v*)*trs*
assumes *wf*: *wf-trs R*
and *WCR*: *WCR* (*rstep R*)
and *overlay*: ⋀ *l r*. (*False*,*l*,*r*) ∉ *critical-pairs ren R R*
begin

lemma *WCRO-not-SN-imp-non-terminating-innermost-rhs-instance*:
assumes ¬ *SN* (*rstep R*)
shows ∃ *r* σ. *r* ∈ *rhss R* ∧ ¬ *SN-on* (*inn-rstep R*) {*r* · σ} ∧ σ ‘ *vars-term r* ⊆ *NF* (*rstep R*)
proof −
let *?R* = *rstep R*
let *?IR* = *inn-rstep R*
from *WCR* **have** *WCR-on* (*rstep R*) {*t*. *SN-on* (*rstep R*) {*t*}}
unfolding *WCR-on-def* **by** *auto*
from *SN-innermost-switch-locally-confluent-overlay*[*OF this overlay wf*] *assms*
have ¬ *SN ?IR* **by** *auto*
then obtain *t0* **where** ¬ *SN-on ?IR* {*t0*} **unfolding** *SN-defs* **by** *blast*
from *not-SN-imp-subt-Tinf*[*OF this*] **obtain** *t1* **where** *t1* ∈ *Tinf ?IR* **by** *blast*
from *Tinf-inn-rstep-imp-first-root-step*[*OF this*]
obtain *s t* **where** (*s*, *t*) ∈ *inn-rrstep R* **and** *nSN*: ¬ *SN-on ?IR* {*t*} **by** *auto*
from *inn-rrstep.cases*[*OF this*(*1*)] **obtain** *l r* σ **where** *lr*: (*l*,*r*) ∈ *R*
and *NF*: *set* (*args* (*l* · σ)) ⊆ *NF-trs R* **and** *t*: *t* = *r* · σ **and** *s* = *l* · σ **by** *metis*

```
  hence r: r ∈ rhss R by auto
  from NF have NF: ⋀ u. u ◁ l · σ ⟹ u ∈ NF-trs R
    by (metis (no-types, lifting) NF-subterm subset-iff
        subterm.dual-order.strict-iff-order supteq.cases term.sel(4))
  show ?thesis
  proof (intro exI[of - r] exI[of - σ] conjI r nSN[unfolded t])
    {
      fix x
      assume x ∈ vars-term r
      hence x ∈ vars-term l using lr wf[unfolded wf-trs-def] by auto
      hence l ▷ Var x using lr wf[unfolded wf-trs-def] by (cases l, auto)
      hence l · σ ▷ Var x · σ by blast
      from NF[OF this] have σ x ∈ NF (rstep R) by auto
    }
    thus σ ‘ vars-term r ⊆ NF-trs R by auto
  qed
qed

lemma WCRO-one-step-simulation-by-narrowing: fixes s :: ('f,'v)term
  assumes nSN: ¬ SN-on (inn-rstep R) {s · σ}
    and NF-sigma: σ ‘ vars-term s ⊆ NF (rstep R)
  shows ∃ u μ δ. (s,u) ∈ narrow-step-s ren R μ
    ∧ ¬ SN-on (inn-rstep R) {u · δ}
    ∧ δ ‘ vars-term u ⊆ NF (rstep R)
    ∧ σ = μ ∘s δ
proof −
  let ?Q = lhss R
  let ?IR = inn-rstep R
  let ?R = rstep R
  define VS where VS = vars-term s
  from nSN obtain t where step: (s · σ, t) ∈ ?IR and nSN: ¬ SN-on ?IR {t}
    by (meson step-reflects-SN-on)
  from inn-rstep.cases[OF step] obtain l r C τ
    where Cid: s · σ = C⟨l · τ⟩ t = C⟨r · τ⟩
      and lr: (l, r) ∈ R
      and NF-ltau: set (args (l · τ)) ⊆ NF-trs R
    by metis
  define p where p = hole-pos C
  from Cid have p: p ∈ poss (s · σ) unfolding p-def by auto
  from Cid have unif: s · σ |- p = l · τ unfolding p-def by auto
  from Cid have t: t = replace-at (s · σ) p (r · τ) unfolding p-def by auto
  from NF-ltau have NF-ltau: ⋀ u. u ◁ l · τ ⟹ u ∈ NF ?R
    by (metis (no-types, lifting) NF-subterm subset-iff
        subterm.dual-order.strict-iff-order supteq.cases term.sel(4))
  from poss-subst-choice[OF p] consider (p) p ∈ poss s is-Fun (s |- p)
    | (sigma) x q where x ∈ vars-term s q ∈ poss (σ x) s · σ |- p = σ x |- q
    by auto
  thus ?thesis
  proof cases
```

case *sigma*
from *sigma*(*2*,*3*)[*unfolded unif*] **obtain** C **where** $\sigma\ x = C\ \langle\ l \cdot \tau\ \rangle$
by (*metis ctxt-supt-id*)
from *rstepI*[*OF lr this refl*] *NF-sigma sigma*(*1*) **have** *False* **by** *auto*
thus *?thesis* **..**
next
case p
with *unif* **have** *unif*: s |- $p \cdot \sigma = l \cdot \tau$ **by** *auto*
from *mgu-vd-complete*[*OF this*, *of ren*]
obtain $\mu 1\ \mu 2\ \delta$ **where** *mgu*: *mgu-vd ren* (s |- p) l = *Some* ($\mu 1$, $\mu 2$)
and *sigma*: $\sigma = \mu 1 \circ_s \delta$ **and** *tau*: $\tau = \mu 2 \circ_s \delta$ **and** *unif*: s |- $p \cdot \mu 1 = l \cdot \mu 2$
by *auto*
define C **where** C = *ctxt-of-pos-term p s*
from p **have** *Cmu*: $C \cdot_c \mu 1$ = *ctxt-of-pos-term p* ($s \cdot \mu 1$) **for** $\mu 1$ :: ('*f*,'*v*)*subst*
unfolding *C-def*
by (*auto simp*: *ctxt-of-pos-term-subst*)
define u **where** $u = (C \cdot_c \mu 1)\langle r \cdot \mu 2\rangle$
have (s, u) ∈ *narrows-r-p-s ren R* (*l*,*r*) p $\mu 1$
unfolding *narrows-r-p-s-def u-def Cmu* **using** p *lr mgu* **by** *auto*
hence *narr*: (s, u) ∈ *narrow-step-s ren R* $\mu 1$ **unfolding** *narrow-step-s-def* **by**
blast

have $t = (C \cdot_c \sigma)\ \langle\ r \cdot \tau\ \rangle$ **unfolding** t *Cmu* **by** *simp*
also have ... = $u \cdot \delta$ **unfolding** *sigma tau u-def*
by *simp*
finally have *tu*: $t = u \cdot \delta$ **.**

with *nSN* **have** *nSN*: ¬ *SN-on ?IR* {$u \cdot \delta$} **by** *simp*

from *wf*[*unfolded wf-trs-def*] *lr* **have** *vc*: *vars-term r* ⊆ *vars-term l* **by** *auto*
from *NF-sigma* **have** *NF*: $x \in VS \Longrightarrow \sigma\ x \in NF\ ?R$ **for** x **by** (*auto simp*:
VS-def)

{
fix v
assume $v \in \delta$ ‘ *vars-term u*

then obtain x **where** $x \in$ *vars-term u* **and** v: $v = \delta\ x$ **by** *auto*
from *this*[*unfolded u-def*] **have** $x \in$ *vars-ctxt* ($C \cdot_c \mu 1$) ∨ $x \in$ *vars-term* ($r \cdot$
$\mu 2$)
by (*simp add*: *vars-term-ctxt-apply*)
hence $x \in$ *vars-term* ($s \cdot \mu 1$)
proof
have $s = C\ \langle\ s$ |- $p\ \rangle$ **unfolding** *C-def* **using** p
by (*simp add*: *ctxt-supt-id*)
from *arg-cong*[*OF this*, *of* λ *t*. $t \cdot \mu 1$]
have *smu1*: $s \cdot \mu 1 = (C \cdot_c \mu 1)\langle s$ |- $p \cdot \mu 1\rangle$ **by** *auto*

```
      assume x ∈ vars-ctxt (C ·c μ1)
      thus x ∈ vars-term (s · μ1) unfolding smu1
        by (simp add: vars-term-ctxt-apply)
    next
    from p have vars-sp: vars-term (s |-p) ⊆ vars-term s by (metis vars-term-subt-at)
      assume x ∈ vars-term (r · μ2)
      with vc have  x ∈ vars-term (l · μ2)
        by (auto simp: vars-term-subst)
      from this[folded unif] vars-sp show x ∈ vars-term (s · μ1)
        by (auto simp: vars-term-subst)
    qed
    hence v ∈ δ ‘ vars-term (s · μ1) unfolding v by auto
    from this[unfolded vars-term-subst VS-def[symmetric]]
    have v ∈ δ ‘ ⋃ (vars-term ‘ μ1 ‘ VS) .

    then obtain x y where x ∈ VS and y: y ∈ vars-term (μ1 x) and v: v = δ
y by auto
    from NF[OF this(1)] have NF: σ x ∈ NF ?R by auto
    from y have Var y ◁ μ1 x by auto
    hence Var y · δ ◁ μ1 x · δ by blast
    hence v ◁ σ x unfolding sigma v by (auto simp: subst-compose-def)
    with NF have v ∈ NF ?R by (rule NF-subterm)
  }
  hence NFu: δ ‘ vars-term u ⊆ NF ?R by auto

  show ?thesis
    by (intro exI[of - u], rule exI[of - μ1], rule exI[of - δ], intro conjI narr nSN
sigma NFu)
 qed
qed


lemma WCRO-not-SN-rstep-imp-not-SN-rstep: assumes ¬ SN (rstep R)
 shows ¬ SN-on (rstep R) (right-forw-closure R)
proof −
 define nfs :: bool where nfs = undefined
 define P where P (n :: nat) (r :: ('f,'v)term) σ = (σ ‘ vars-term r ⊆ NF-trs R
∧ ¬ SN-on (inn-rstep R) {r · σ})
   for n r σ
 from WCRO-not-SN-imp-non-terminating-innermost-rhs-instance[OF assms]
 obtain r σ where P0: P 0 r σ and r: r ∈ rhss R
   unfolding P-def by auto
 define Q where Q (n :: nat) s (σ :: ('f,'v)subst) u (δ :: ('f,'v)subst) =
     (∃ μ. (s,u) ∈ narrow-step-s ren R μ ∧ σ = μ ∘s δ) for n s σ u δ
 have P n s σ ⟹ ∃ u δ. P (Suc n) u δ ∧ Q n s σ u δ for s n σ
   using WCRO-one-step-simulation-by-narrowing[of s σ]
   unfolding P-def Q-def by auto
 from exists-narrow-steps-to-infinite-rsteps[OF wf, of P ren r σ, OF this[unfolded
Q-def] P0]
```

have *main*: ¬ *SN-on* (*narrow-step ren R*) {*r*} ¬ *SN-on* (*rstep R*) ((*narrow-step ren R*)* `` {*r*}) **by** *auto*

from *r* **have** (*narrow-step ren R*)* `` {*r*} ⊆ *right-forw-closure R*
unfolding *right-forw-closure-def* **by** *auto*
with *main*(*2*) **show** ¬ *SN-on* (*rstep R*) (*right-forw-closure R*) **unfolding** *SN-defs* **by** *blast*
from *main*(*1*) **have** ¬ *SN-on* (*narrow-step ren R*) (*rhss R*) **using** *r* **by** *fast*
qed
end

theorem *WCRO-SN-rstep-RFC*: **assumes** *wf*: *wf-trs R*
and *WCR*: *WCR* (*rstep R*)
and *overlay*: ⋀ *l r*. (*False*,*l*,*r*) ∉ *critical-pairs ren R R*
shows *SN* (*rstep R*) ⟷ *SN-on* (*rstep R*) (*right-forw-closure R*) (**is** *?A* = *?B*)
proof
show *?A* ⟹ *?B* **by** *force*
next
assume *?B*
with *WCRO-not-SN-rstep-imp-not-SN-rstep*[*OF assms*] **show** *?A* **by** *blast*
qed
end
end

References

[1] N. Dershowitz. Termination of linear rewriting systems (preliminary version). In S. Even and O. Kariv, editors, *Automata, Languages and Programming, 8th Colloquium, Acre (Akko), Israel, July 13-17, 1981. Proceedings*, volume 115 of *Lecture Notes in Computer Science*, pages 448–458. Springer, 1981.

[2] N. Dershowitz and C. Hoot. Natural termination. *Theor. Comput. Sci.*, 142(2):179–207, 1995.

[3] B. Gramlich. Relating innermost, weak, uniform and modular termination of term rewriting systems. In A. Voronkov, editor, *Logic Programming and Automated Reasoning, International Conference, LPAR '92, St. Petersburg, Russia, July 15-20, 1992. Proceedings*, volume 624 of *Lecture Notes in Computer Science*, pages 285–296. Springer, 1992.

[4] R. Thiemann, D. Hofbauer, U. Le Huitouze, and J. Waldmann. New and Formalized Proofs for Right-Forward Closures and Core Matrix Interpretations. In F. Pfenning, editor, *11th International Conference on Formal Structures for Computation and Deduction (FSCD 2026)*, volume 378 of *Leibniz International Proceedings in Informatics (LIPIcs)*,

pages 32:1–32:19, Dagstuhl, Germany, 2026. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.