Substitutions for λ-free higher-order terms

Vincent Trélat

May 3, 2024

Abstract

This theory provides a formalization of substitutions on λ-free higher-order terms, establishing a structured framework with the expected algebraic properties. It introduces a type construction for the rigorous definition and manipulation of substitutions. The main theorem of this theory proves the existence of fixed-point substitutions under acyclicity, a theorem that is too often regarded as trivial in the literature [1, 3].

Contents

1 Introduction

This theory is based on J. Blanchette's Formalization of Recursive Path Orders for Lambda-Free Higher-Order Terms [2] which defines λ-free higher-order terms.

1.1 Preliminary lemmas

The following lemma and definitions would be worth adding in the theory *Lambda-Free-RPOs.Lambda-Free-Term*.

lemma *sub-trans*: ‹*sub x y* ⟹ *sub y z* ⟹ *sub x z*›
by (*induction z arbitrary*: *x y*, *blast*, *elim sub-AppE*; *simp add*: *sub-arg sub-fun*)

definition *subterms* :: ‹('*s*, '*v*) *tm* ⇒ ('*s*, '*v*) *tm set*› **where**
‹*subterms t* ≡ {*u*. *sub u t*}›

definition *proper-subterms* :: ‹('*s*, '*v*) *tm* ⇒ ('*s*, '*v*) *tm set*› **where**
‹*proper-subterms t* ≡ {*u*. *proper-sub u t*}›

The following lemmas are also helpful in the following and could be easily lifted higher in the hierarchy of theories.

lemmas *mult-Suc-left* =
mult-Suc-right[*unfolded add.commute*[*of m* ‹*m*∗*n*› **for** *m n*]]
— Although this result is immediate, it might be worth adding it to Nat symmetrically.

lemma *inject-nat-in-fset-ninj*:
‹*finite S* ⟹ (*range* (*f*::*nat*⇒-) ⊆ *S*) ⟹ (∃ *x y*. *x* ≠ *y* ∧ *f x* = *f y*)›
apply (*induction S rule*: *finite-induct*, *fast*)
subgoal for *x FS*
using
finite-insert[*of x FS*]
infinite-UNIV-char-0 inj-on-finite[*of f UNIV* ‹{*x*} ∪ *FS*›]
unfolding *inj-def* **by** *blast*.

lemma *wfPD*: ‹*wfP P* ⟹ *wfp-on P A*›
— This destruction rule for *wfP* could be added to the theory *Open-Induction.Restricted-Predicates*
by (*simp add*: *wfP-eq-minimal wfp-on-iff-minimal*)

lemma *set-decr-chain-empty*:
fixes *u* :: ‹*nat* ⇒ '*a set*›
assumes *pord*: ‹⋀*n*. *u n* ≠ ∅ ⟹ *u* (*n*+*1*) ⊂ *u n*›
and *fin*: ‹⋀*n*. *finite* (*u n*)›
shows ‹∃ *k*. *u k* = ∅›
— This lemma could easily be generalized to any partial order and any minimal element and integrated to the theory *Well-Quasi-Orders.Minimal-Elements*.
proof−

```
  have ‹wfp-on (in-rel finite-psubset) (range u)›
    using wf-finite-psubset wfPD wf-in-rel
    by blast
  with fin have wfpon: ‹wfp-on (⊂) (range u)›
    unfolding finite-psubset-def in-rel-def wfp-on-def
    by (auto iff: image-iff, metis)
  then interpret minimal-element ‹(⊂)› ‹range u›
    by (unfold-locales; simp add: po-on-def)

  show ?thesis
    using pord wf
    unfolding wfp-on-def[simplified, unfolded Suc-eq-plus1]
    using rangeI[of u]
    by meson
qed

lemma distinct-in-fset:
  ‹finite E ⟹ card E = n ⟹ distinct xs ⟹ set xs ⊆ E ⟹ length xs ≤ n›
  by (induction n, simp, metis card-mono distinct-card)
```

2 Substitutions

This section embeds substitutions in a proper type, lifting basic operations like substitution application (i.e. *substitution* as an operation on terms) and composition.

2.1 Substitutions for terms

Substitutions in *Lambda-Free-RPOs.Lambda-Free-RPOs* [2] are not defined as a type, they are implicitly used as functions from variables to terms. However, not all functions from variables to terms are substitutions, which motivates the introduction of a proper type *subst* fitting the specification of a substitution, namely that only finitely many variables are not mapped to themselves.

```
abbreviation 𝒱 where ‹𝒱 ≡ Hd ∘ Var›

lemma inj-𝒱: ‹inj 𝒱›
  by (intro injI, simp)

lemma fin-var-restr:‹finite (𝒱 ‘ E) ⟹ finite E›
  using inj-𝒱 finite-imageI image-inv-f-f
  by metis

definition is-subst :: ‹('v ⇒ ('s, 'v) tm) ⇒ bool› where
  ‹is-subst σ ≡ finite {t. is-Hd t ∧ is-Var (head t) ∧ subst σ t ≠ t}›
```

If type-checking on terms was enforced in *is-subst*, the above definition could be expressed as follows in a more concise way:

is-subst $\sigma \equiv$ *finite* $\{subst\ \sigma\ t \neq t\}$

Without type-checking, the definition must range over variables and not over terms since *App x x* is a valid term, even though it does not type check. If $x\sigma = x$, then $(x(x))\sigma = x(x)$. This inductively allows infinitely many fixpoints of the substitution σ.

With type-checking, the second definition would only add finitely many terms, namely type-correct applied terms of the form *App y x* where x and y are substitutable variables.

```
lemma subst-𝒱: ‹is-subst 𝒱›
unfolding is-subst-def
proof −
  {
    fix t::‹('s, 'v) tm›
    assume asm: ‹is-Hd t› ‹is-Var (head t)›
    define v where ‹v = var (head t)›
    hence t-eq:‹t = Hd (Var v)›
      by (simp add: asm(1) asm(2))

    hence ‹subst 𝒱 t = t›
      by simp
  }
  thus ‹finite {t. (is-Hd t ∧ is-Var (head t) ∧ subst (Hd ∘ Var) t ≠ t)}›
    using not-finite-existsD by blast
qed

typedef ('s, 'v) subst = ‹{σ::('v ⇒ ('s, 'v) tm). is-subst σ}›
  using subst-𝒱 by blast

setup-lifting type-definition-subst

lift-definition 𝒱' :: ‹('s, 'v) subst› is ‹𝒱›
  using subst-𝒱 by blast
```

Informally, $\mathcal{V}$ is almost the identity function since it casts variables to themselves (as terms), but it has the type *'v* ⇒ (*'s*, *'v*) *tm*. $\mathcal{V}$ is thus lifted to $\mathcal{V}'$ that applies on substitutions. The fact that $\mathcal{V}'$ leaves ground terms unchanged follows from the definition of *subst* and is obtained by lifting. $\mathcal{V}'$ *is* the identity substitution.

```
lift-definition
  subst-app :: ‹('s, 'v) tm ⇒ ('s, 'v) subst ⇒ ('s, 'v) tm› (‹-·-› [56,55] 55) is
  ‹λ x σ. subst σ x›.

lemma sub-subst': ‹sub x t ⟹ sub (x·σ) (t·σ)›
— This lemma is a lifted version of sub ?s ?t ⟹ sub (subst ?ϱ ?s) (subst ?ϱ ?t).
  by (simp add: subst-app.rep-eq sub-subst)
```

Application for substitutions (i.e. *substitution*) is lifted from the function *subst* and denoted as usual in the literature with a post-fix notation: *subst* σ x is denoted by $x{\cdot}\sigma$.

```
lemma subst-alt-def:‹finite {t. (𝒱 t)·σ ≠ 𝒱 t}›
proof (transfer)
  fix σ :: ‹'v ⇒ ('s, 'v) tm›
  assume asm: ‹is-subst σ›
  have
  ‹{t. is-Hd t ∧ is-Var (head t) ∧ subst σ t ≠ t} = 𝒱 ` {t. subst σ (𝒱 t) ≠ 𝒱 t}›
    apply (standard; clarsimp)
    subgoal for x
      apply (cases x)
      subgoal for hd by (cases hd; simp)
      subgoal by simp
    done.
  then show ‹finite {t. subst σ (𝒱 t) ≠ 𝒱 t}›
    using fin-var-restr asm unfolding is-subst-def
    by simp
qed
```

The lemma above provides an alternative definition for substitution. Yet, there is a subtlety since Isabelle does not provide support for dependent types: one shall understand this lemma as the meta-implication "if σ is of type *subst* then the aforementioned set is finite". A true alternative definition should state an equivalence, however the converse implication makes no sense in Isabelle.

```
lemma subst-eq-sub:‹sub s t ⟹ t·σ = t·ϑ ⟹ s·σ = s·ϑ›
  by (transfer fixing: s t, induction t arbitrary: s; fastforce)
```

Composition for substitutions is also lifted as follows.

```
lift-definition
  rcomp :: ‹('s, 'v) subst ⇒ ('s, 'v) subst ⇒ ('s, 'v) subst›
  (infixl ‹○› 55) is ‹λ σ ϑ. subst ϑ ∘ (subst σ ∘ 𝒱)›
proof (transfer)
  fix σ::‹'v ⇒ ('s, 'v) tm› and ϑ::‹'v ⇒ ('s, 'v) tm›

  assume asm: ‹is-subst σ› ‹is-subst ϑ›

  define S where ‹S ≡ {t. is-Hd t ∧ is-Var (head t) ∧ subst σ t ≠ t}›
  define T where ‹T ≡ {t. is-Hd t ∧ is-Var (head t) ∧ subst ϑ t ≠ t}›
  define TS where ‹TS ≡
    {t. is-Hd t ∧ is-Var (head t) ∧ subst (λx. subst ϑ (subst σ (𝒱 x))) t ≠ t}›

  note fin =
    asm(1)[unfolded is-subst-def, folded S-def]
    asm(2)[unfolded is-subst-def, folded T-def]

  have TS-simp:
```

```
  ‹TS = {t. is-Hd t ∧ is-Var (head t) ∧ subst (subst ϑ ∘ (subst σ ∘ 𝒱)) t ≠ t}›
  proof −
    have eq:‹(λx. subst ϑ (subst σ (Hd (Var x)))) = subst ϑ ∘ (subst σ ∘ 𝒱)›
      by (standard, simp)
    then show ?thesis
      by (simp add: TS-def[simplified eq])
  qed

  have ‹TS ⊆ S ∪ T›
    unfolding S-def T-def TS-def
    apply (intro subsetI, simp)
    subgoal for t
      apply (cases t; simp)
      subgoal for x by (cases x; auto)
    done
  done

  from finite-subset[OF this finite-UnI[OF fin], simplified TS-simp]
  show ‹is-subst (subst ϑ ∘ (subst σ ∘ 𝒱))›
  unfolding is-subst-def .
qed

lemma rcomp-subst-simp:
  ‹(x::('s, 'v) tm)·(σ ∘ ϑ) = (x·σ)·ϑ›
  by (transfer fixing: x, induction x; simp add: hd.case-eq-if)

lift-definition
  set-image-subst :: ‹('s, 'v) tm set ⇒ ('s, 'v) subst ⇒ ('s, 'v) tm set›
  (infixl ‹·› 90) is ‹λ S σ. subst σ ‘ S› .

lemma set-image-subst-collect:
  ‹S·σ = {x·σ | x. x ∈ S}›
  by (transfer, blast)
```

2.2 Substitutions as a monoid

First, we state two introduction lemmas for allowing extensional reasoning on substitutions. The first one is on terms and the second one is for terms that are variables.

```
lemma subst-ext-tmI:
  fixes σ::‹('s, 'v) subst› and ϑ::‹('s, 'v) subst›
  shows ‹∀ (x::('s, 'v) tm). (x·σ) = (x·ϑ) ⟹ σ = ϑ›
proof (transfer, rule ccontr)
  fix σ::‹'v ⇒ ('s, 'v) tm› and ϑ::‹'v ⇒ ('s, 'v) tm›
  assume asm:‹is-subst σ› ‹is-subst ϑ› ‹∀ x. subst σ x = subst ϑ x› ‹σ ≠ ϑ›

  from asm(4) obtain v where v-def:‹σ v ≠ ϑ v›
    by fast
  hence ‹subst σ (Hd (Var v)) ≠ subst ϑ (Hd (Var v))›
```

```
    by simp

  with asm(3) show False
    by blast
qed

lemma subst-ext-tmI':
  fixes σ::‹('s, 'v) subst› and ϑ::‹('s, 'v) subst›
  shows ‹∀ x. (𝒱 x)·σ = (𝒱 x)·ϑ ⟹ σ = ϑ›
  by (transfer, simp, blast)

lemmas subst-extI = subst-ext-tmI subst-ext-tmI'
```

The three following lemmas state that $\mathcal{V}'$ is the neutral element for composition. Although uniqueness follows from the definition of a neutral element, the proof of this claim is given below.

```
lemma 𝒱'-id-tm [simp]:
  fixes x::‹(-,-) tm›
  shows ‹(x·𝒱') = x›
  by (transfer fixing: x, induction x; simp add: hd.case-eq-if)

lemma 𝒱'-neutral-rcomp[simp]:
  ‹σ ∘ 𝒱' = σ›
  ‹𝒱' ∘ σ = σ›
  by (intro subst-ext-tmI allI, simp add: rcomp-subst-simp)+

lemma unique-𝒱':
  ‹(⋀σ. σ ∘ η = σ) ⟹ η = 𝒱'›
  ‹(⋀σ. η ∘ σ = σ) ⟹ η = 𝒱'›
proof −
  assume ‹σ ∘ η = σ› for σ
  from 𝒱'-neutral-rcomp(2)[of η, symmetric, simplified this]
  show ‹η = 𝒱'›.
next
  assume ‹η ∘ σ = σ› for σ
  from 𝒱'-neutral-rcomp(1)[of η, symmetric, simplified this]
  show ‹η = 𝒱'›.
qed

lemma 𝒱'-iff:‹σ = 𝒱' ⟷ (∀ x. (𝒱 x)·σ = (𝒱 x))›
  by (intro iffI; simp add: 𝒱'.rep-eq subst-app.rep-eq subst-ext-tmI')

lemma rcomp-assoc[simp]:
  fixes σ::‹('s, 'v) subst›
    and ϑ::‹('s, 'v) subst›
    and γ::‹('s, 'v) subst›
  shows ‹(σ ∘ ϑ) ∘ γ = σ ∘ (ϑ ∘ γ)›
  by (intro subst-extI, simp add: rcomp-subst-simp)
```

Knowing that the composition of substitutions ($\circ$) is associative and has a neutral element $\mathcal{V}'$, we may embed substitutions in an algebraic structure with a monoid structure and enjoy Isabelle's lemmas on monoids.

global-interpretation *subst-monoid*: *monoid rcomp* $\mathcal{V}'$
by (*unfold-locales*; *simp*)

3 Acyclic substitutions

3.1 Definitions and auxiliary lemmas

The iteration on substitutions is defined below and is followed by several algebraic properties.

In order to show these properties, we give three different definitions for iterated substitutions. In short, the first one is simply the iteration of composition using Isabelle's ($\frown\frown$) operator. This can be understood as follows:

$$\sigma^n \triangleq (\underbrace{\sigma \circ (\sigma \circ (\ldots (\sigma}_{n \text{ times}} \circ \mathcal{V}') \ldots))$$

Using properties from the monoid structure, this can be written as

$$\sigma^n \triangleq \underbrace{\sigma \circ \cdots \circ \sigma}_{n \text{ times}}$$

The two other definitions are inductively defined using those two schemes:

$$\sigma^{n+1} = \sigma \circ \sigma^n$$
$$\sigma^{n+1} = \sigma^n \circ \sigma$$

We prove that these three definitions are equivalent and use them in the proofs of the properties that follow.

definition *iter-rcomp* :: ‹('*s*, '*v*) *subst* ⇒ *nat* ⇒ ('*s*, '*v*) *subst*›
(‹-⁻› [*200*, *0*] *1000*) **where** ‹$\sigma^n \equiv ((\circ)\ \sigma\ \frown\frown\ n)\ \mathcal{V}'$›

lemma *iter-rcomp-Suc-right*:‹$\sigma^{Suc\ n} = \sigma^n \circ \sigma$›
unfolding *iter-rcomp-def funpow-Suc-right comp-def* $\mathcal{V}'$*-neutral-rcomp*
by (*induction n*; *simp*)

lemma *iter-rcomp-Suc-left*:‹$\sigma^{Suc\ n} = \sigma \circ \sigma^n$›
unfolding *iter-rcomp-def funpow-Suc-right comp-def* $\mathcal{V}'$*-neutral-rcomp*
by (*induction n*; *simp*)

fun *iter-rcomp'* :: ‹('*s*, '*v*) *subst* ⇒ *nat* ⇒ ('*s*, '*v*) *subst*›
where

```
  ‹iter-rcomp' σ 0 = 𝒱'›
| ‹iter-rcomp' σ (Suc n) = σ ∘ (iter-rcomp' σ n)›
lemma iter-rcomp-eq-iter-rcomp':‹σ^n = iter-rcomp' σ n›
  by (induction n; simp add: iter-rcomp-def)

fun iter-rcomp'' :: ‹('s, 'v) subst ⇒ nat ⇒ ('s, 'v) subst›
  where
  ‹iter-rcomp'' σ 0 = 𝒱'›
| ‹iter-rcomp'' σ (Suc n) = (iter-rcomp'' σ n) ∘ σ›

lemma iter-rcomp-eq-iter-rcomp'':‹σ^n = iter-rcomp'' σ n›
  by (induction n, simp add: iter-rcomp-def, simp add: iter-rcomp-Suc-right)

lemmas iter-rcomp'-eq-iter-rcomp'' =
  iter-rcomp-eq-iter-rcomp'[symmetric, simplified iter-rcomp-eq-iter-rcomp'']
```

The following lemmas show some algebraic properties on iterations of substitutions, namely that for any σ, the function $n \mapsto \sigma^n$ i.e. *iter-rcomp* σ is a magma homomorphism between $(\mathbb{N}, +)$ and $(subst, \circ)$. Since $\sigma^0 \equiv \mathcal{V}'$, it is even a (commutative) monoid homomorphism.

```
lemma iter-comp-add-morphism: ‹(σ^n) ∘ (σ^k) = σ^(n+k)›
  unfolding iter-rcomp-eq-iter-rcomp' iter-rcomp'-eq-iter-rcomp''
  by(induction k; simp add:
    rcomp-assoc[of ‹iter-rcomp'' σ n› ‹iter-rcomp'' σ k› σ for k, symmetric])

lemmas iter-comp-com-add-morphism =
  iter-comp-add-morphism[
    of σ n k for σ n k,
    simplified add.commute,
    unfolded iter-comp-add-morphism[of σ k n, symmetric]]
```

There is a similar property with multiplication, stated as follows:

$$\forall \sigma, n, k. (\sigma^n)^k = \sigma^{n \times k}$$

This is shown by the following lemma. The next one shows commutativity.

```
lemma iter-comp-mult-morphism: ‹(σ^n)^k = σ^(n*k)›
  unfolding iter-rcomp-eq-iter-rcomp' iter-rcomp'-eq-iter-rcomp''
  by (induction k; simp only:
    mult-Suc-left iter-comp-add-morphism[
      symmetric,
      unfolded iter-rcomp-eq-iter-rcomp' iter-rcomp'-eq-iter-rcomp'']
    iter-rcomp''.simps mult-0-right)

lemmas iter-comp-com-mult-morphism =
  iter-comp-mult-morphism[
    of σ n k for σ k n,
    simplified mult.commute,
    unfolded iter-comp-mult-morphism[of σ k n, symmetric]]
```

Some simplification rules are added to the rules to help automatize subsequent proofs.

lemma *iter-rcomp-$\mathcal{V}'$[simp]*: ‹$\mathcal{V}'^{m} = \mathcal{V}'$›
unfolding *iter-rcomp-eq-iter-rcomp'*
by (*induction n*; *simp*)

lemma *iter-rcomp-0*[*simp*]: ‹$\sigma^{0} = \mathcal{V}'$›
unfolding *iter-rcomp-eq-iter-rcomp'* **by** *simp*

lemma *iter-rcomp-1*[*simp*]: ‹$\sigma^{Suc\ 0} = \sigma$›
unfolding *iter-rcomp-eq-iter-rcomp'* **by** *simp*

definition *dom* :: ‹('s, 'v) *subst* ⇒ ('s, 'v) *tm set*› **where**
‹*dom* $\sigma \equiv \{\mathcal{V}\ x \mid x.\ (\mathcal{V}\ x)\cdot\sigma \neq \mathcal{V}\ x\}$›

definition *ran* :: ‹('s, 'v) *subst* ⇒ ('s, 'v) *tm set*› **where**
‹*ran* $\sigma \equiv (\lambda x.\ x\cdot\sigma)$ ‘ *dom* σ›

lemma *no-sub-in-dom-subst-eq*: ‹$(\forall\ x \in dom\ \sigma.\ \neg\ sub\ x\ t) \Longrightarrow t = t\cdot\sigma$›
unfolding *dom-def*
by (*induction t*)
((*simp add*: *subst-app.rep-eq split*: *hd.split*, *metis sub-refl*),
(*simp add*: *subst-app.rep-eq*, *metis sub-arg sub-fun*))

lemma *subst-eq-on-domI*:
‹$(\forall x.\ x \in dom\ \sigma \vee x \in dom\ \vartheta \longrightarrow x\cdot\sigma = x\cdot\vartheta) \Longrightarrow \sigma = \vartheta$›
by (*intro subst-ext-tmI' allI*)
(*metis comp-apply no-sub-in-dom-subst-eq sub-HdE*)

lemma *subst-finite-dom*:‹*finite* (*dom* σ)›
unfolding *dom-def* **using** *subst-alt-def*[*of* σ] **by** *simp*

lemma *$\mathcal{V}'$-emp-dom*: ‹*dom* $\mathcal{V}' = \emptyset$›
unfolding *dom-def* **by** *simp*

lemma *var-not-in-dom* [*simp*]: ‹$\mathcal{V}\ x \notin dom\ \sigma \Longrightarrow ((\mathcal{V}\ x)\cdot\sigma^{n}) = \mathcal{V}\ x$›
unfolding *dom-def*
by (*induction n*; *simp add*: *iter-rcomp-Suc-left rcomp-subst-simp*)

lemma *ran-alt-def*:‹*ran* $\sigma = \{(\mathcal{V}\ x)\cdot\sigma \mid x.\ (\mathcal{V}\ x)\cdot\sigma \neq \mathcal{V}\ x\}$›
unfolding *ran-def dom-def*
by *blast*

definition *is-ground-subst* :: ‹('s, 'v) *subst* ⇒ *bool*› **where**
‹*is-ground-subst* $\sigma \equiv$ (*ground* ‘ *ran* σ) = {*True*}›

lemma *is-ground-subst-alt-def*:
‹*is-ground-subst* $\sigma \longleftrightarrow (ran\ \sigma \neq \emptyset) \wedge (\forall x.\ (\mathcal{V}\ x)\cdot\sigma \neq \mathcal{V}\ x \longrightarrow ground\ ((\mathcal{V}\ x)\cdot\sigma))$›
unfolding *is-ground-subst-def ran-def dom-def*

```
  by (standard, auto)

lemma ground-subst-grounds:‹is-ground-subst σ ⟹ x ∈ dom σ ⟹ ground (x·σ)›
  unfolding dom-def
  by (induction x) (auto simp add: is-ground-subst-alt-def)

lemma iter-on-ground:‹ground (x·σ) ⟹ n > 0 ⟹ x·σ^n = x·σ›
  using ground-imp-subst-iden
  unfolding iter-rcomp-eq-iter-rcomp'
  by (induction n, blast)
    (simp add: rcomp-subst-simp, simp add: subst-app.rep-eq)

lemma true-subst-nempty-vars:
  ‹σ ≠ 𝒱 ⟹ {t. is-Hd t ∧ is-Var (head t) ∧ subst σ t ≠ t} ≠ {}›
proof −
  assume ‹σ ≠ 𝒱›
  then obtain t where ‹σ t ≠ 𝒱 t›
    by blast
  moreover have ‹is-Hd (𝒱 t)› ‹is-Var (head (𝒱 t))›
    by simp+
  ultimately show ?thesis
    by fastforce
qed

lemma true-subst-nemp-im:‹ran σ = {} ⟹ σ = 𝒱'›
  unfolding ran-def dom-def
  by (transfer, auto simp add: is-subst-def dest: true-subst-nempty-vars)

lemma ground-subst-imp-no-var-mapped-on-var:
  ‹is-ground-subst σ ⟹ (∀ x y. x ≠ y ⟶ (𝒱 x)·σ ≠ (𝒱 y))›
proof −
  have ‹¬ (∀ x y. x ≠ y ⟶ (𝒱 x)·σ ≠ (𝒱 y)) ⟹ ¬ is-ground-subst σ›
  proof −
    assume ‹¬ (∀ x y. x ≠ y ⟶ (𝒱 x)·σ ≠ (𝒱 y))›
    then obtain x y where xy-def:‹x ≠ y› ‹(𝒱 x)·σ = (𝒱 y)›
      by blast
    hence
      ‹(𝒱 x)·σ≠(𝒱 x)›
      ‹𝒱 x ∈ {x. is-Var (head x) ∧ x · σ ≠ x}›
      ‹¬ ground ((𝒱 x)·σ)›
      by fastforce+

    then show ‹¬ is-ground-subst σ›
      unfolding is-ground-subst-def ran-def dom-def
      by blast
  qed

  from contrapos-pp[
    rotated,
```

of ‹$\forall\ x\ y.\ x \neq y \longrightarrow (\mathcal{V}\ x)\cdot\sigma \neq (\mathcal{V}\ y)$› ‹*is-ground-subst* σ›,
OF this]
show ‹*is-ground-subst* $\sigma \Longrightarrow (\forall\ x\ y.\ x \neq y \longrightarrow (\mathcal{V}\ x)\cdot\sigma \neq (\mathcal{V}\ y))$›.
qed

lemma *ran-$\mathcal{V}'$-empty*:‹*ran* $\mathcal{V}' = \emptyset$›
unfolding *ran-def dom-def* **using** *$\mathcal{V}'$-iff*
by *fast*

lemma *non-ground-$\mathcal{V}'$*: ‹¬ *is-ground-subst* $\mathcal{V}'$›
using *is-ground-subst-alt-def ran-$\mathcal{V}'$-empty* **by** *blast*

3.2 Acyclicity

A substitution is said to be *acyclic* if no variable x in the domain of σ occurs as a subterm of $x{\cdot}\sigma^n$ for any $(0::'a) < n$.

definition *is-acyclic* :: ‹('*s*, '*v*) *subst* ⇒ *bool*› **where**
‹*is-acyclic* $\sigma \equiv (\forall\ x \in dom\ \sigma.\ \forall\ n > 0.\ x \notin subterms\ (x{\cdot}\sigma^n))$›

lemma *is-acyclicE*:‹ *is-acyclic* $\sigma \Longrightarrow x \in dom\ \sigma \Longrightarrow n > 0 \Longrightarrow x \notin subterms\ (x{\cdot}\sigma^n)$›
unfolding *is-acyclic-def* **by** *blast*

lemma *non-acyclic-$\mathcal{V}'$*: ‹*is-acyclic* $\mathcal{V}'$›
using *$\mathcal{V}'$-emp-dom is-acyclic-def* **by** *fast*

lemma *acyclic-iter-dom-eq*:‹*is-acyclic* $\sigma \Longrightarrow dom\ \sigma = dom\ \sigma^n$› **if** ‹$n > 0$› **for** n
using *that* **unfolding** *dom-def is-acyclic-def subterms-def*
by (*induction n*, *fast*, *simp*)
(*smt* (*verit*, *ccfv-SIG*)
Collect-cong
bot-nat-0.not-eq-extremum
$\mathcal{V}'$-id-tm
iter-rcomp-eq-iter-rcomp'
iter-rcomp'.simps
mem-Collect-eq
rcomp-subst-simp
sub-refl
zero-less-Suc)

lemma *acyclic-iter*: ‹*is-acyclic* $\sigma \Longrightarrow n > 0 \Longrightarrow$ *is-acyclic* σ^n›
unfolding *is-acyclic-def subterms-def*
by (*auto simp add*: *iter-comp-mult-morphism dom-def dest*: *CollectD*,
use var-not-in-dom[*unfolded dom-def*] **in** *fastforce*)

3.3 Fixed-point substitution

We define the fixed-point substitution of a substitution σ as the substitution σ^i where $i = \inf\{k \in \mathbb{N} \mid \sigma^k = \sigma^{k+1}\}$.

definition *fp-subst* :: ‹('s, 'v) *subst* ⇒ ('s, 'v) *subst*› (‹-⋆› *1000*) **where**
‹$\sigma^\star \equiv$ *iter-rcomp* σ (*LEAST* n . $\sigma^n = \sigma^{n+1}$)›

lemma *ground-subst-is-fp*:‹*is-ground-subst* $\sigma \Longrightarrow \sigma^\star = \sigma$›
— Ground substitutions have no effect and are therefore fixed-points substitutions. The converse is not true.
proof −
assume *asm*:‹*is-ground-subst* σ›
have *eq*:‹$\sigma^1 = \sigma^2$›
apply (*simp only*: *one-add-one*[*symmetric*],
simp, *intro subst-ext-tmI' allI*, *simp*)
subgoal for x
unfolding *iter-rcomp-eq-iter-rcomp'*
by (*cases* ‹$\mathcal{V}$ $x \in$ *dom* σ›,
simp add: *rcomp-subst-simp*,
metis
ground-subst-grounds[*OF asm*, *THEN ground-imp-subst-iden*]
subst-app.rep-eq,
simp add: *dom-def rcomp-subst-simp*).

note *least-le1*=*Least-le*[*of* ‹λ*n*. $\sigma^n = \sigma^{n+1}$› *1*, *unfolded nat-1-add-1*, *OF this*]
have *neq*:‹$\sigma^0 \neq \sigma^1$›
using *asm non-ground-*$\mathcal{V}'$
unfolding *iter-rcomp-eq-iter-rcomp'*
by *auto*
have ‹(*LEAST* *n*. $\sigma^n = \sigma^{n+1}$) ≠ *0*›
proof (*rule ccontr*)
assume ‹¬ (*LEAST* *n*. $\sigma^n = \sigma^{n+1}$) ≠ *0*›
hence ‹(*LEAST* *n*. $\sigma^n = \sigma^{n+1}$) = *0*› **by** *blast*
from
LeastI[*of* ‹λ*n*. $\sigma^n = \sigma^{n+1}$› *1*, *simplified one-add-one*, *OF eq*,
simplified add-0 this]
neq
show *False* **by** *blast*
qed
moreover have ‹(*LEAST* *x*. $\sigma^x = \sigma^{x+1}$) = *0* ∨ (*LEAST* *x*. $\sigma^x = \sigma^{x+1}$) = *1*›
using *least-le1* **by** *linarith*
ultimately have ‹(*LEAST* *n*. $\sigma^n = \sigma^{n+1}$) = *1*›
by *sat*
then show ‹$\sigma^\star = \sigma$›
unfolding *fp-subst-def iter-rcomp-eq-iter-rcomp'*
by *simp*
qed

In the following, we prove that fixed-point substitutions are well-defined for acyclic substitutions. To help visualise how the proofs are carried out, for any terms x and y and any substitution σ, we denote the fact that x is substituted by y after one application of σ, i.e. that $sub\ y\ (x{\cdot}\sigma)$, by $x \rightarrow_\sigma y$.

Remark. *In fact, automata could be used to model substitutions with variables in the domain as the initial states and variables outside of the domain and constants as final states. The transitions would be given by the successive substitutions. Acyclic substitutions would be represented by a DAG.*

lemma *dom-sub-subst*: ‹$x \in dom\ \sigma \Longrightarrow sub\ x\ (t{\cdot}\sigma) \Longrightarrow \exists\, y \in dom\ \sigma.\ sub\ x\ (y{\cdot}\sigma)$›
— If $x \rightarrow_\sigma t$ for $x \in dom\ \sigma$ and a term t, then there is a variable $y \in dom\ \sigma$ such that $x \rightarrow_\sigma y$.
apply (*induction t*)
subgoal for *t0*
apply (*cases t0*)
unfolding *dom-def* **apply** *fastforce*
by (*metis* (*no-types*, *lifting*) *hd.set*(*4*) *tm.set*(*3*) *ground-imp-subst-iden subst-app.rep-eq sub-HdE*)
subgoal for *t1 t2*
by (*simp add*: *dom-def*) (*metis subst.simps*(*2*) *tm.distinct*(*1*) *subst-app.rep-eq sub-AppE*)
done

lemma *dom-sub-subst-contrapos*:
— For x in the domain, if there is no z in the domain such that $x \rightarrow_\sigma z$, then there is not term t such that $x \rightarrow_\sigma t$.
‹$x \in dom\ \sigma \Longrightarrow \forall\, z \in dom\ \sigma.\ \neg\ sub\ x\ (z{\cdot}\sigma) \Longrightarrow \forall\, t.\ \neg\ sub\ x\ (t{\cdot}\sigma)$›
using *dom-sub-subst* **by** *blast*

lemma *dom-sub-subst-iter*:
‹$x \in dom\ \sigma \Longrightarrow \forall\, z \in dom\ \sigma.\ \neg\ sub\ x\ (z{\cdot}\sigma^n) \Longrightarrow \neg\ sub\ x\ (t{\cdot}\sigma^n)$›
by (*induction n arbitrary*: *x* σ,
simp, *use sub-refl* **in** *blast*,
smt (*verit*, *ccfv-threshold*)
One-nat-def
$\mathcal{V}'$*-id-tm*
$\mathcal{V}'$*-neutral-rcomp*(*1*)
dom-def
dom-sub-subst
iter-comp-add-morphism
iter-rcomp-Suc-right
iter-rcomp-eq-iter-rcomp′
iter-rcomp′.simps(*1*)
mem-Collect-eq
plus-1-eq-Suc
rcomp-subst-simp
sub-refl
subst-eq-sub)

lemma
assumes ‹$x \in dom\ \sigma$› ‹$\forall\, y \in dom\ \sigma.\ sub\ y\ t \longrightarrow \neg\ sub\ x\ (y{\cdot}\sigma)$›
shows *not-sub-subst-if*:‹$\neg\ sub\ x\ (t{\cdot}\sigma)$›
— For x in the domain and any term t, if there is no variable y occurring in t such that $x \to_\sigma y$, then $x \not\to_\sigma t$.
using *assms*
by (*induction t*)
((*smt* (*verit*, *ccfv-threshold*)
hd.case-eq-if
subst.simps(*1*)
$\mathcal{V}'$*-id-tm*
dom-def
$\mathcal{V}'$*.rep-eq*
subst-app.rep-eq
mem-Collect-eq
sub-HdE
sub-refl),
(*smt* (*verit*, *ccfv-threshold*)
subst.simps(*2*)
comp-apply
dom-def
subst-app.rep-eq
mem-Collect-eq
sub-Hd-AppE
sub-arg
sub-fun))

lemma *dom-sub-subst-iter-Suc*:
‹$x \in dom\ \sigma \Longrightarrow sub\ x\ (t{\cdot}\sigma^{n+1}) \Longrightarrow$
$\exists\, y\ z.\ y \in dom\ \sigma \wedge z \in dom\ \sigma \wedge sub\ x\ (z{\cdot}\sigma) \wedge sub\ z\ (y{\cdot}\sigma^{n})$›
— If $x \to_\sigma^{n+1} t$, then there are variables y and z such that $y \to_\sigma^{n} z \to_\sigma x$.
proof (*induction n arbitrary*: *x t*)
case *0*
then show *?case*
unfolding *add-0 One-nat-def iter-rcomp-Suc-right rcomp-subst-simp*
iter-rcomp-0 $\mathcal{V}'$*-neutral-rcomp* $\mathcal{V}'$*-id-tm*
using *dom-sub-subst sub-refl* **by** *blast*
next
case (*Suc n*)
then obtain t' **where** t'*-def*: ‹$sub\ t'\ (t{\cdot}\sigma^{n+1})$› ‹$sub\ x\ (t'{\cdot}\sigma)$›
by (*metis Suc-eq-plus1 iter-rcomp-Suc-right rcomp-subst-simp sub-refl*)
have ‹$\exists\, w \in dom\ \sigma.\ sub\ w\ t' \wedge sub\ x\ (w{\cdot}\sigma)$›
using *not-sub-subst-if Suc.prems*(*1*) t'*-def*(*2*) **by** *blast*
then obtain *w* **where** *w-def*: ‹$w \in dom\ \sigma$› ‹$sub\ w\ t'$› ‹$sub\ x\ (w{\cdot}\sigma)$›
by *blast*

```
  then obtain y where ‹y ∈ dom σ› ‹sub w (y·σ^(n+1))›
    apply (fold Suc-eq-plus1, unfold iter-rcomp-Suc-right rcomp-subst-simp)
    using sub-subst' sub-trans Suc.IH t'-def by meson

  with w-def show ?case
    by auto
qed

lemma sub-Suc-n-sub-n-sub:
  ‹(∃ x ∈ dom σ. sub z (x·σ^(n+1))) ⟷
  (∃ x y. x ∈ dom σ ∧ y ∈ dom σ ∧ sub z (y·σ) ∧ sub y (x·σ^n))› if ‹z ∈ dom σ›
  using that
proof (intro iffI)
  show ‹z ∈ dom σ ⟹ ∃x∈dom σ. sub z (x·σ^n + 1) ⟹
    ∃x y. x ∈ dom σ ∧ y ∈ dom σ ∧ sub z (y·σ) ∧ sub y (x·σ^n)›
  proof−
    assume ‹∃x∈dom σ. sub z (x·σ^n + 1)› ‹z ∈ dom σ›
    then obtain x where ‹x∈dom σ› ‹sub z (x·σ^(n+1))›
      by blast

    then obtain y where ‹y ∈ dom σ› ‹sub z (y·σ)› ‹sub y (x·σ^n)›
      by (metis Suc-eq-plus1 ‹z ∈ dom σ› iter-rcomp-Suc-right
        not-sub-subst-if rcomp-subst-simp)

    then show ‹∃x y. x ∈ dom σ ∧ y ∈ dom σ ∧ sub z (y·σ) ∧ sub y (x·σ^n)›
      using ‹x ∈ dom σ› by blast
  qed
  show ‹z ∈ dom σ ⟹
    ∃x y. x ∈ dom σ ∧ y ∈ dom σ ∧ sub z (y·σ) ∧ sub y (x·σ^n) ⟹
    ∃x∈dom σ. sub z (x·σ^n + 1)›
  proof −
    assume ‹∃ x y. x ∈ dom σ ∧ y ∈ dom σ ∧ sub z (y·σ) ∧ sub y (x·σ^n)›
    then obtain x y where xy-def:
      ‹x ∈ dom σ› ‹y ∈ dom σ› ‹sub z (y·σ)› ‹sub y (x·σ^n)›
      by blast
    then have ‹sub (y·σ) (x·σ^(n+1))›
      unfolding Suc-eq-plus1[symmetric] iter-rcomp-Suc-right rcomp-subst-simp
      using sub-subst' by blast
    with xy-def(3) have ‹sub z (x·σ^(n+1))›
      using sub-trans by blast
    with xy-def(1) show ‹∃x∈dom σ. sub z (x·σ^n + 1)›
      by fast
  qed
qed
```

The following theorem is the main result of this theory and states that for acyclic substitutions, the fixed-point substitution exists and is well defined. The main idea of the proof is to define a non-negative quantity and show that successively applying the substitution makes it decrease.

For any such iteration n, we define the set of variables that will be substituted by the next iteration of the substitution and denote it by S_n. Formally, S_n is defined as follows:

$$S_n := \{z \in \text{dom } \sigma \mid \exists x \in \text{dom } \sigma.\ x \rightarrow_\sigma^n z\}$$

There is a clear recurrence relation between S_{n+1} and S_n, namely that the variables in S_{n+1} are exactly the variables in S_n that are not sources in S_n, i.e. that have a predecessor – for the subterm relation – in S_n. This is formalized as follows:

$$S_{n+1} = S_n - \{z \in S_n \mid \forall x \in S_n.\ x \not\rightarrow_\sigma z\}$$

This implies that the sequence $(S_n)_{n\in\mathbb{N}}$ is strictly monotone for inclusion. Since it is bounded and has its values in finite sets, it is convergent and there is a rank k from which it is constant and equal to the infimum of the range, the empty set.

```
theorem fp-subst: ‹is-acyclic σ ⟹ ∃ n. σ^n = σ^(n+1)›
proof −
  assume acyc:‹is-acyclic σ›
  define S where ‹S ≡ λ n. ⋃(subterms ` ran σ^n) ∩ dom σ›

  have S-alt-def: ‹S n = {z ∈ dom σ. ∃ x ∈ dom σ. sub z (x·σ^n)}› if ‹n > 0› for
n
    unfolding S-def ran-def subterms-def
      acyclic-iter-dom-eq[OF that acyc, symmetric]
    by blast

  have S-σ-unfold:
    ‹S n · σ = {x·σ | x. x ∈ dom σ ∧ (∃ y∈dom σ. sub x (y·σ^n))}› if ‹n > 0› for n
    unfolding set-image-subst-collect[of ‹S n›, unfolded S-def]
      S-def subterms-def
    unfolding ran-def acyclic-iter-dom-eq[OF that acyc, symmetric]
    by blast

  have sources-charac:
    ‹n > 0 ⟹ S (n+1) = S n − {z ∈ S n. ∀ x ∈ S n. ¬ sub z (x·σ)}› for n
  proof (intro subset-antisym subsetI)
    fix z assume ‹n > 0›
    show ‹z ∈ S (n + 1) ⟹ z ∈ S n − {z ∈ S n. ∀ x∈S n. ¬ sub z (x·σ)}›
    proof−
      assume ‹z ∈ S (n + 1)›
      then obtain x where x-def: ‹x ∈ dom σ› ‹sub z (x·σ^(n+1))›
        unfolding S-alt-def[OF zero-less-Suc[of n, unfolded Suc-eq-plus1]]
        by blast
      then have ‹z ∈ S n›
        unfolding S-def ran-def
```

```
      acyclic-iter-dom-eq[OF ‹n > 0› acyc, symmetric]
      subterms-def
    by (simp, metis IntE S-def Suc-eq-plus1 ‹z ∈ S (n + 1)›
      dom-sub-subst-iter iter-rcomp-Suc-left rcomp-subst-simp)
  moreover have ‹∃ x∈S n. sub z (x·σ)›
    using
      S-alt-def[OF ‹n > 0›]
      dom-sub-subst-iter-Suc[of z σ]
      x-def(2) ‹z ∈ S n›
    by fast

  ultimately show ‹z ∈ S n − {z ∈ S n. ∀ x∈S n. ¬ sub z (x·σ)}›
    by blast
 qed
 show ‹z ∈ S n − {z ∈ S n. ∀ x∈S n. ¬ sub z (x·σ)} ⟹ z ∈ S (n + 1)›
 proof−
  assume ‹z ∈ S n − {z ∈ S n. ∀ x∈S n. ¬ sub z (x·σ)}›
  hence ‹z ∈ S n› ‹∃ y∈S n. sub z (y·σ)›
    by blast+
  then obtain x y w where
    ‹x ∈ dom σ› ‹sub z (x·σ^n)›
    ‹y ∈ dom σ› ‹w ∈ dom σ› ‹sub z (y·σ)› ‹sub y (w·σ^n)›
    unfolding S-def subterms-def ran-def
      acyclic-iter-dom-eq[OF ‹n > 0› acyc, symmetric]
    by blast
  show ‹z ∈ S (n+1)›
    unfolding S-alt-def[OF zero-less-Suc[of n], unfolded Suc-eq-plus1]
    apply (simp, intro conjI, use ‹z ∈ S n›[unfolded S-def] in simp)
      using sub-trans[OF ‹sub z (y·σ)›
        sub-subst'[OF ‹sub y (w·σ^n)›, of σ]] ‹w ∈ dom σ›
      unfolding iter-rcomp-Suc-right rcomp-subst-simp
    by (intro bexI[of - w])
 qed
qed

have fin-Sn: ‹finite (S n)› for n
 unfolding S-def using subst-finite-dom
 by blast

have decr: ‹S n ≠ ∅ ⟹ S (n+1) ⊂ S n› for n
proof
 show ‹S n ≠ ∅ ⟹ S (n+1) ⊆ S n›
 proof (cases ‹n > 0›)
  case True
  assume ‹S n ≠ ∅›
  show ?thesis
  proof (intro subsetI, rule ccontr)
    fix x assume ‹x ∈ S(n+1)› ‹x ∉ S n›
    from this(1)[
```

```
      unfolded S-alt-def[OF zero-less-Suc[of n],
        unfolded Suc-eq-plus1],
      simplified,
      unfolded Suc-eq-plus1]
    obtain y where ‹x ∈ dom σ› ‹y ∈ dom σ› ‹sub x (y·σ^(n+1))›
      by blast
    with ‹x ∉ S n›[unfolded S-alt-def[OF True], simplified]
    have ‹∀ z∈dom σ. ¬ sub x (z·σ^n)›
      by blast
    from
      dom-sub-subst-iter[OF ‹x ∈ dom σ› this, of ‹y·σ›]
    show False
      using ‹sub x (y·σ^(n+1))›[
        folded Suc-eq-plus1,
        unfolded iter-rcomp-Suc-left rcomp-subst-simp] ..
  qed
next
  case False
  assume ‹S n ≠ ∅›
  moreover have ‹S 0 = ∅›
    unfolding S-def iter-rcomp-eq-iter-rcomp'
    by (simp add: ran-𝒱'-empty)
  ultimately show ?thesis
    using False by simp
qed

show ‹S n ≠ ∅ ⟹ S (n+1) ≠ S n›
proof (cases n)
  case (Suc m)
  hence ‹n > 0› by simp
  assume ‹S n  ≠ ∅›
  have ‹{z ∈ S n. ∀ x ∈ S n. ¬ sub z (x·σ)} ≠ ∅›
  proof (rule ccontr)
    assume ‹¬ {z ∈ S n. ∀ x∈S n. ¬ sub z (x·σ)} ≠ {}›
    hence ch: ‹∀ x. x ∈ S n ⟶ (∃ y ∈ S n. sub x (y·σ))›
      by simp
    define E where ‹E ≡ λx. {y ∈ S n. sub x (y·σ)}›
    have E-nemp: ‹x ∈ S n ⟹ E x ≠ ∅› for x
      unfolding E-def using ch by fast
    have ‹x ∈ S n ⟹ x ∉ E x› for x
    proof (rule ccontr)
      assume a: ‹x ∈ S n› ‹¬ x ∉ E x›
      from this(1) have ‹x ∈ dom σ›
        unfolding S-def by fast
      with conjunct2[OF a(2)[simplified, unfolded E-def, simplified]]
      acyc[
        unfolded is-acyclic-def subterms-def,
        simplified,
        rule-format,
```

```
    OF this zero-less-Suc[of 0],
    unfolded iter-rcomp-Suc-right,
    simplified]
  show False
    by satx
qed

with ch ‹S n ≠ ∅› obtain x y where ‹x ∈ S n› ‹y ∈ S n› ‹sub y (x·σ)›
  by blast
with acyc have ‹y ≠ x›
  unfolding S-def is-acyclic-def subterms-def
  by (metis IntE One-nat-def 𝒱'-id-tm iter-rcomp-0
    iter-rcomp-Suc-left mem-Collect-eq rcomp-subst-simp zero-less-one)

let ?C = ‹card (S n)›
have incl: ‹E v ⊆ S n› for v
  unfolding E-def by blast

have ‹k > 0 ⟹
  ∃ seq. k = length seq ∧
    seq!0 ∈ S n ∧
    (∀ m < length seq − 1. seq!(m+1) ∈ E (seq!m)) ∧
    distinct seq›
  for k
proof (induction k rule: nat-induct-non-zero)
  case (Suc k)
  then obtain seq where seq-def: ‹k = length seq› ‹seq!0 ∈ S n›
    ‹(∀ m < k−1. seq!(m+1) ∈ E (seq!m))› ‹distinct seq›
    by blast
  hence ‹l < k ⟹ seq!l ∈ S n› for l
    unfolding E-def
    by (metis (no-types, lifting) CollectD Suc-eq-plus1
      less-Suc-eq less-Suc-eq-0-disj less-diff-conv)
  with seq-def obtain v where v-def: ‹v ∈ E (seq!(k−1))›
    using E-nemp Suc.hyps
    by (metis Suc-eq-plus1 Suc-pred' all-not-in-conv less-add-one)

  have seq-sub:
    ‹i < length seq ⟹ j ≤ i ⟹ sub (seq!j) ((seq!i)·σ^(i−j))› for i j
  proof (induction i arbitrary: j)
    case (Suc l)
    with seq-def have ‹seq!(Suc l) ∈ E (seq!l)›
      by simp
    hence ‹sub (seq!l) ((seq!(Suc l))·σ)›
      unfolding E-def by blast
    with Suc show ?case
    by (smt (verit, ccfv-SIG)
      cancel-comm-monoid-add-class.diff-cancel
      bot-nat-0.not-eq-extremum
```

```
    Suc-diff-Suc
    Suc-lessD
    𝒱'-id-tm
    diff-Suc-Suc
    diff-diff-cancel
    iter-rcomp-0
    iter-rcomp-Suc-left
    le-Suc-eq
    rcomp-subst-simp
    sub-refl
    sub-subst'
    sub-trans
    zero-less-diff)

qed (simp add: sub-refl)

define seq' where ‹seq' ≡ seq @ [v]›
hence ‹sub (seq!(k−1)) (v·σ)›
  using v-def unfolding E-def
  by blast

have ‹v ∈ dom σ›
  using incl v-def
  unfolding E-def S-def
  by fast

have ‹v ∉ set seq›
proof
  assume ‹v ∈ set seq›
  then obtain j where ‹j < k› ‹v = seq!j›
    unfolding in-set-conv-nth seq-def(1) by fast
  with seq-sub[of ‹k−1› j, folded seq-def(1) this(2)] Suc.hyps
  have ‹sub v (seq!(k−1)·σ^(k−1−j))›
    by simp
  from sub-trans[OF
      this
      sub-subst'[OF ‹sub (seq!(k−1)) (v·σ)›, of ‹σ^(k−1−j)›,
    folded rcomp-subst-simp iter-rcomp-Suc-left]]
  show False
    using acyc ‹v ∈ dom σ›
    unfolding is-acyclic-def subterms-def
    by blast
qed

show ?case
  by (intro exI[of - seq'] conjI, unfold seq'-def,
    simp add: seq-def(1),
    simp add:
      seq-def(2)
```

```
        nth-append[
          of seq ‹[v]› 0,
          folded seq-def(1),
          unfolded if-split,
          simplified]
        Suc.hyps,
      intro allI,
      metis
        butlast-snoc
        diff-add-inverse2
        length-append-singleton
        length-butlast
        less-Suc-eq
        less-diff-conv
        nth-append
        nth-append-length
        seq-def(1,3)
        v-def,
      simp add: ‹v ∉ set seq› seq-def(4))
  qed (intro exI[of - ‹[x]›] conjI, (simp add: ‹x ∈ S n›)+)

  then obtain seq where seq-def:
    ‹?C + 1 = length seq› ‹seq!0 ∈ S n› ‹distinct seq›
    ‹(∀ m < length seq − 1. seq!(m+1) ∈ E (seq!m))›
    by auto

  have ‹set seq ⊆ S n›
  proof
    fix x assume ‹x ∈ set seq›
    then obtain i where ‹seq!i = x› ‹i < length seq›
      by (metis in-set-conv-nth)
    then show ‹x ∈ S n›
      by (cases i, use seq-def(2) in simp)
        (metis seq-def(4) One-nat-def Suc-eq-plus1 in-mono incl less-diff-conv)
  qed

  from
    distinct-in-fset[OF fin-Sn, of n ?C, simplified, OF seq-def(3) this]
    seq-def(1)
  show False
    by simp
 qed
 thus ?thesis
  unfolding sources-charac[OF ‹n > 0›]
  by blast
next
 case 0 assume ‹S n ≠ ∅›
 moreover have ‹S 0 = ∅›
  unfolding S-def iter-rcomp-0 ran-𝒱'-empty
```

```
      by blast
    ultimately show ?thesis
      using 0 by simp
  qed
qed

have subset-dom: ‹n > 0 ⟹ S n ⊆ dom σ› for n
  by (induction n) (simp add: S-def)+

have ‹finite (S n)› for n
  by (cases n)
    (simp add:
      S-def subst-finite-dom
      subset-dom[THEN finite-subset[OF - subst-finite-dom[of σ]]])+

moreover have ‹S 0 = ∅›
  unfolding S-def iter-rcomp-eq-iter-rcomp'
  by (simp add: ran-𝒱'-empty)

ultimately obtain k where ‹k > 0› ‹S k = ∅›
  using set-decr-chain-empty[where u=‹λn. S (Suc n)›] decr
  by auto

from this(2)[unfolded S-def subterms-def ran-def
  acyclic-iter-dom-eq[OF this(1) acyc, symmetric], simplified]
have f: ‹{y. y ∈ dom σ ∧ (∃x∈dom σ. sub y (x·σ^k))} = ∅›
  by blast
then have ‹σ^k = σ^(k+1)›
  by (intro subst-eq-on-domI allI impI)
  ((simp add:
      acyclic-iter-dom-eq[OF ‹k > 0› acyc, symmetric]
      acyclic-iter-dom-eq[OF zero-less-Suc[of k] acyc, symmetric]),
    unfold iter-rcomp-Suc-right rcomp-subst-simp,
    simp add: no-sub-in-dom-subst-eq)
then show ?thesis
  by (intro exI[of - k])
qed

lemma fp-subst-comp-stable: ‹is-acyclic σ ⟹ (σ⋆) ∘ (σ⋆) = σ⋆›
proof–
  assume ‹is-acyclic σ›
  with fp-subst obtain m where m-def: ‹σ^m = σ^(m+1)›
    by blast
  define n where ‹n ≡ LEAST n. σ^n = σ^(n+1)›
  have eq: ‹σ⋆ = σ^n›
    unfolding fp-subst-def n-def..
  from m-def have ‹σ^n = σ^(n+1)›
    unfolding n-def
    by (intro LeastI[of ‹λn. σ^n = σ^(n+1)› m])
```

```
  hence ‹σ^n = σ^(n+k)› for k
  proof (induction k)
    case (Suc k)
    show ?case
      unfolding add-Suc-right iter-rcomp-Suc-right
        Suc.IH[OF Suc.prems, symmetric]
      unfolding iter-rcomp-Suc-right[symmetric] Suc-eq-plus1
      using Suc.prems.
  qed simp
  then show ?thesis
    unfolding eq iter-comp-com-add-morphism
    by (induction n) presburger+
qed

lemma fp-subst-stable-iter: ‹is-acyclic σ ⟹ n > 0 ⟹ (σ⋆)^n = σ⋆›
  by (induction n, simp,
    unfold iter-rcomp-Suc-right,
    use fp-subst-comp-stable in fastforce)

lemma fp-subst-stable-fp: ‹is-acyclic σ ⟹ (σ⋆)⋆ = σ⋆›
proof −
  assume ‹is-acyclic σ›
  define m where ‹m ≡ LEAST n. (σ⋆)^n = (σ⋆)^(n+1)›
  show ?thesis
    unfolding fp-subst-def[of ‹σ⋆›, folded m-def]
    using fp-subst-stable-iter[OF ‹is-acyclic σ›]
    by (metis (mono-tags, lifting) LeastI-ex Suc-eq-plus1 m-def zero-less-Suc)
qed

end
```

References

[1] H. Barbosa, P. Fontaine, and A. Reynolds. Congruence closure with free variables. In A. Legay and T. Margaria, editors, *TACAS 2017*, volume 10206 of *LNCS*, pages 214–230, 2017.

[2] J. C. Blanchette, U. Waldmann, and D. Wand. Formalization of recursive path orders for lambda-free higher-order terms. *Archive of Formal Proofs*, September 2016. https://isa-afp.org/entries/Lambda_Free_RPOs.html, Formal proof development.

[3] S. Tourret, P. Fontaine, D. E. Ouraoui, and H. Barbosa. Lifting congruence closure with free variables to λ-free higher-order logic via SAT encoding. In F. Bobot and T. Weber, editors, *SMT 2020*, volume 2854 of *CEUR Workshop Proceedings*, pages 3–14. CEUR-WS.org, 2020.