

# Substitutions for $\lambda$ -free higher-order terms

Vincent Trélat

June 20, 2024

## Abstract

This theory provides a formalization of substitutions on  $\lambda$ -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

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	Preliminary lemmas . . . . .	2
<b>2</b>	<b>Substitutions</b>	<b>3</b>
2.1	Substitutions for terms . . . . .	3
2.2	Substitutions as a monoid . . . . .	6
<b>3</b>	<b>Acyclic substitutions</b>	<b>8</b>
3.1	Definitions and auxiliary lemmas . . . . .	8
3.2	Acyclicity . . . . .	12
3.3	Fixed-point substitution . . . . .	13

# 1 Introduction

This theory is based on J. Blanchette's Formalization of Recursive Path Orders for Lambda-Free Higher-Order Terms [2] which defines  $\lambda$ -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*:  $\langle \text{sub } x \ y \implies \text{sub } y \ z \implies \text{sub } x \ z \rangle$

**by** (*induction*  $z$  arbitrary:  $x \ y$ , *blast*, *elim sub-AppE*; *simp add*: *sub-arg sub-fun*)

**definition** *subterms* ::  $\langle ('s, 'v) \text{ tm} \Rightarrow ('s, 'v) \text{ tm set} \rangle$  **where**

$\langle \text{subterms } t \equiv \{u. \text{sub } u \ t\} \rangle$

**definition** *proper-subterms* ::  $\langle ('s, 'v) \text{ tm} \Rightarrow ('s, 'v) \text{ tm set} \rangle$  **where**

$\langle \text{proper-subterms } t \equiv \{u. \text{proper-sub } u \ t\} \rangle$

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  $m \langle m * n \rangle$  **for**  $m \ n$ ]]

— Although this result is immediate, it might be worth adding it to *Nat* symmetrically.

**lemma** *inject-nat-in-fset-ninj*:

$\langle \text{finite } S \implies (\text{range } (f :: \text{nat} \Rightarrow -) \subseteq S) \implies (\exists x \ y. x \neq y \wedge f \ x = f \ y) \rangle$

**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  $f \ UNIV \ \langle \{x\} \cup FS \rangle$ ]

**unfolding** *inj-def* **by** *blast*.

**lemma** *wfPD*:  $\langle \text{wfP } P \implies \text{wfp-on } P \ A \rangle$

— 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**  $u :: \langle \text{nat} \Rightarrow 'a \ \text{set} \rangle$

**assumes** *por*:  $\langle \bigwedge n. u \ n \neq \emptyset \implies u \ (n+1) \subset u \ n \rangle$

**and** *fin*:  $\langle \bigwedge n. \text{finite } (u \ n) \rangle$

**shows**  $\langle \exists k. u \ k = \emptyset \rangle$

— 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  $\mathcal{V}$  where ⟨ $\mathcal{V} \equiv Hd \circ Var$ ⟩

```

```

lemma inj- $\mathcal{V}$ : ⟨inj  $\mathcal{V}$ ⟩
  by (intro injI, simp)

```

```

lemma fin-var-restr: ⟨finite ( $\mathcal{V} \text{ ' } E$ ) ⟹ finite  $E$ ⟩
  using inj- $\mathcal{V}$  finite-imageI image-inv-f-f
  by metis

```

```

definition is-subst :: ⟨('v ⇒ ('s, 'v) tm) ⇒ bool⟩ where
  ⟨is-subst  $\sigma \equiv$  finite {t. is-Hd t ∧ is-Var (head t) ∧ subst  $\sigma$  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*  $\sigma \equiv \text{finite } \{subst \ \sigma \ t \neq t\}$

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  $\sigma$ .

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*:  $\langle is-subst \ \mathcal{V} \rangle$

**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**  $\langle('s, 'v) \text{ subst} = \{\sigma::('v \Rightarrow ('s, 'v) \text{ tm}). \text{is-subst } \sigma\}\rangle$

**using** *subst-V* **by** *blast*

**setup-lifting** *type-definition-subst*

**lift-definition**  $\mathcal{V}' :: \langle('s, 'v) \text{ subst}\rangle \text{ is } \langle\mathcal{V}\rangle$

**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  $'v \Rightarrow ('s, 'v) \text{ tm}$ .  $\mathcal{V}$  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* ::  $\langle('s, 'v) \text{ tm} \Rightarrow ('s, 'v) \text{ subst} \Rightarrow ('s, 'v) \text{ tm}\rangle \langle\text{---}\rangle [56,55] \ 55 \ \text{is}$   
 $\langle\lambda \ x \ \sigma. \text{subst } \sigma \ x\rangle.$

**lemma** *sub-subst'*:  $\langle\text{sub } x \ t \Longrightarrow \text{sub } (x \cdot \sigma) \ (t \cdot \sigma)\rangle$

— This lemma is a lifted version of  $\text{sub } ?s \ ?t \Longrightarrow \text{sub } (\text{subst } ?\rho \ ?s) \ (\text{subst } ?\rho \ ?t)$ .

**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*  $\sigma$   $x$  is denoted by  $x \cdot \sigma$ .

**lemma** *subst-alt-def*:  $\langle \text{finite } \{t. (\mathcal{V} t) \cdot \sigma \neq \mathcal{V} t\} \rangle$

**proof** (*transfer*)

**fix**  $\sigma :: \langle 'v \Rightarrow ('s, 'v) tm \rangle$

**assume** *asm*:  $\langle \text{is-subst } \sigma \rangle$

**have**

$\langle \{t. \text{is-Hd } t \wedge \text{is-Var } (\text{head } t) \wedge \text{subst } \sigma t \neq t\} = \mathcal{V} \{t. \text{subst } \sigma (\mathcal{V} t) \neq \mathcal{V} t\} \rangle$

**apply** (*standard*; *clarsimp*)

**subgoal for**  $x$

**apply** (*cases*  $x$ )

**subgoal for** *hd* **by** (*cases* *hd*; *simp*)

**subgoal by** *simp*

**done.**

**then show**  $\langle \text{finite } \{t. \text{subst } \sigma (\mathcal{V} t) \neq \mathcal{V} t\} \rangle$

**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  $\sigma$  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*:  $\langle \text{sub } s t \Longrightarrow t \cdot \sigma = t \cdot \vartheta \Longrightarrow s \cdot \sigma = s \cdot \vartheta \rangle$

**by** (*transfer* *fixing*:  $s$   $t$ , *induction*  $t$  *arbitrary*:  $s$ ; *fastforce*)

Composition for substitutions is also lifted as follows.

**lift-definition**

*recomp* ::  $\langle ('s, 'v) \text{subst} \Rightarrow ('s, 'v) \text{subst} \Rightarrow ('s, 'v) \text{subst} \rangle$

(**infixl**  $\langle \circ \rangle$  55) **is**  $\langle \lambda \sigma \vartheta. \text{subst } \vartheta \circ (\text{subst } \sigma \circ \mathcal{V}) \rangle$

**proof** (*transfer*)

**fix**  $\sigma :: \langle 'v \Rightarrow ('s, 'v) tm \rangle$  **and**  $\vartheta :: \langle 'v \Rightarrow ('s, 'v) tm \rangle$

**assume** *asm*:  $\langle \text{is-subst } \sigma \rangle \langle \text{is-subst } \vartheta \rangle$

**define**  $S$  **where**  $\langle S \equiv \{t. \text{is-Hd } t \wedge \text{is-Var } (\text{head } t) \wedge \text{subst } \sigma t \neq t\} \rangle$

**define**  $T$  **where**  $\langle T \equiv \{t. \text{is-Hd } t \wedge \text{is-Var } (\text{head } t) \wedge \text{subst } \vartheta t \neq t\} \rangle$

**define**  $TS$  **where**  $\langle TS \equiv \{t. \text{is-Hd } t \wedge \text{is-Var } (\text{head } t) \wedge \text{subst } (\lambda x. \text{subst } \vartheta (\text{subst } \sigma (\mathcal{V} x))) t \neq t\} \rangle$

**note** *fin* =

*asm*(1)[*unfolded is-subst-def*, *folded S-def*]

*asm*(2)[*unfolded is-subst-def*, *folded T-def*]

**have** *TS-simp*:

$\langle TS = \{t. \text{is-Hd } t \wedge \text{is-Var } (\text{head } t) \wedge \text{subst } (\text{subst } \vartheta \circ (\text{subst } \sigma \circ \mathcal{V})) t \neq t\} \rangle$   
**proof** –  
**have**  $\text{eq}::\langle (\lambda x. \text{subst } \vartheta (\text{subst } \sigma (\text{Hd } (\text{Var } x)))) = \text{subst } \vartheta \circ (\text{subst } \sigma \circ \mathcal{V}) \rangle$   
**by**  $(\text{standard}, \text{simp})$   
**then show**  $?thesis$   
**by**  $(\text{simp add: TS-def}[\text{simplified eq}])$   
**qed**

**have**  $\langle TS \subseteq S \cup T \rangle$   
**unfolding**  $S\text{-def } T\text{-def } TS\text{-def}$   
**apply**  $(\text{intro subsetI}, \text{simp})$   
**subgoal for**  $t$   
**apply**  $(\text{cases } t; \text{simp})$   
**subgoal for**  $x$  **by**  $(\text{cases } x; \text{auto})$   
**done**  
**done**

**from**  $\text{finite-subset}[\text{OF this finite-UnI}[\text{OF fin}], \text{simplified TS-simp}]$   
**show**  $\langle \text{is-subst } (\text{subst } \vartheta \circ (\text{subst } \sigma \circ \mathcal{V})) \rangle$   
**unfolding**  $\text{is-subst-def}$ .  
**qed**

**lemma**  $\text{rcomp-subst-simp}$ :  
 $\langle (x::('s, 'v) \text{tm}) \cdot (\sigma \circ \vartheta) = (x \cdot \sigma) \cdot \vartheta \rangle$   
**by**  $(\text{transfer fixing: } x, \text{induction } x; \text{simp add: hd.case-eq-if})$

**lift-definition**  
 $\text{set-image-subst} :: \langle ('s, 'v) \text{tm set} \Rightarrow ('s, 'v) \text{subst} \Rightarrow ('s, 'v) \text{tm set} \rangle$   
**(infixl**  $\langle \cdot \rangle$   $90$ ) **is**  $\langle \lambda S \sigma. \text{subst } \sigma \text{ ` } S \rangle$ .

**lemma**  $\text{set-image-subst-collect}$ :  
 $\langle S \cdot \sigma = \{x \cdot \sigma \mid x. x \in S\} \rangle$   
**by**  $(\text{transfer}, \text{blast})$

## 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**  $\text{subst-ext-tmI}$ :  
**fixes**  $\sigma::\langle ('s, 'v) \text{subst} \rangle$  **and**  $\vartheta::\langle ('s, 'v) \text{subst} \rangle$   
**shows**  $\langle \forall (x::('s, 'v) \text{tm}). (x \cdot \sigma) = (x \cdot \vartheta) \implies \sigma = \vartheta \rangle$   
**proof**  $(\text{transfer}, \text{rule ccontr})$   
**fix**  $\sigma::\langle 'v \Rightarrow ('s, 'v) \text{tm} \rangle$  **and**  $\vartheta::\langle 'v \Rightarrow ('s, 'v) \text{tm} \rangle$   
**assume**  $\text{asm}::\langle \text{is-subst } \sigma \rangle \langle \text{is-subst } \vartheta \rangle \langle \forall x. \text{subst } \sigma x = \text{subst } \vartheta x \rangle \langle \sigma \neq \vartheta \rangle$   
**from**  $\text{asm}(4)$  **obtain**  $v$  **where**  $v\text{-def}::\langle \sigma v \neq \vartheta v \rangle$   
**by**  $\text{fast}$   
**hence**  $\langle \text{subst } \sigma (\text{Hd } (\text{Var } v)) \neq \text{subst } \vartheta (\text{Hd } (\text{Var } v)) \rangle$

by *simp*

with *asm*( $\exists$ ) show *False*  
by *blast*

qed

lemma *subst-ext-tmI'*:

fixes  $\sigma::\langle('s, 'v) \text{ subst}\rangle$  and  $\vartheta::\langle('s, 'v) \text{ subst}\rangle$   
shows  $\langle\forall x. (\mathcal{V} x)\cdot\sigma = (\mathcal{V} x)\cdot\vartheta \implies \sigma = \vartheta\rangle$   
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  *$\mathcal{V}'$ -id-tm* [*simp*]:

fixes  $x::\langle(-, -) \text{ tm}\rangle$   
shows  $\langle(x\cdot\mathcal{V}') = x\rangle$   
by (*transfer fixing: x, induction x; simp add: hd.case-eq-if*)

lemma  *$\mathcal{V}'$ -neutral-rcomp*[*simp*]:

$\langle\sigma \circ \mathcal{V}' = \sigma\rangle$   
 $\langle\mathcal{V}' \circ \sigma = \sigma\rangle$   
by (*intro subst-ext-tmI allI, simp add: rcomp-subst-simp*)+

lemma *unique- $\mathcal{V}'$* :

$\langle(\bigwedge\sigma. \sigma \circ \eta = \sigma) \implies \eta = \mathcal{V}'\rangle$   
 $\langle(\bigwedge\sigma. \eta \circ \sigma = \sigma) \implies \eta = \mathcal{V}'\rangle$

proof –

assume  $\langle\sigma \circ \eta = \sigma\rangle$  for  $\sigma$   
from  *$\mathcal{V}'$ -neutral-rcomp*( $\exists$ )[*of  $\eta$ , symmetric, simplified this*]  
show  $\langle\eta = \mathcal{V}'\rangle$ .

next

assume  $\langle\eta \circ \sigma = \sigma\rangle$  for  $\sigma$   
from  *$\mathcal{V}'$ -neutral-rcomp*( $\exists$ )[*of  $\eta$ , symmetric, simplified this*]  
show  $\langle\eta = \mathcal{V}'\rangle$ .

qed

lemma  *$\mathcal{V}'$ -iff*: $\langle\sigma = \mathcal{V}' \longleftrightarrow (\forall x. (\mathcal{V} x)\cdot\sigma = (\mathcal{V} x))\rangle$

by (*intro iffI; simp add:  $\mathcal{V}'$ .rep-eq subst-app.rep-eq subst-ext-tmI'*)

lemma *rcomp-assoc*[*simp*]:

fixes  $\sigma::\langle('s, 'v) \text{ subst}\rangle$   
and  $\vartheta::\langle('s, 'v) \text{ subst}\rangle$   
and  $\gamma::\langle('s, 'v) \text{ subst}\rangle$   
shows  $\langle(\sigma \circ \vartheta) \circ \gamma = \sigma \circ (\vartheta \circ \gamma)\rangle$   
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 ( $\widetilde{\phantom{x}}$ ) operator. This can be understood as follows:

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

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

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

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

$$\begin{aligned} \sigma^{n+1} &= \sigma \circ \sigma^n \\ \sigma^{n+1} &= \sigma^n \circ \sigma \end{aligned}$$

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

**definition** *iter-rcomp* ::  $\langle 's, 'v \rangle \text{ subst} \Rightarrow \text{nat} \Rightarrow \langle 's, 'v \rangle \text{ subst} \rangle$   
 $\langle \cdot \rangle$  [200, 0] 1000) **where**  $\langle \sigma^n \equiv ((\circ) \sigma \widetilde{\phantom{x}} n) \mathcal{V}' \rangle$

**lemma** *iter-rcomp-Suc-right*:  $\langle \sigma^{\text{Suc } n} = \sigma^n \circ \sigma \rangle$

**unfolding** *iter-rcomp-def funpow-Suc-right comp-def  $\mathcal{V}'$ -neutral-rcomp*  
**by** (*induction n; simp*)

**lemma** *iter-rcomp-Suc-left*:  $\langle \sigma^{\text{Suc } n} = \sigma \circ \sigma^n \rangle$

**unfolding** *iter-rcomp-def funpow-Suc-right comp-def  $\mathcal{V}'$ -neutral-rcomp*  
**by** (*induction n; simp*)

**fun** *iter-rcomp'* ::  $\langle 's, 'v \rangle \text{ subst} \Rightarrow \text{nat} \Rightarrow \langle 's, 'v \rangle \text{ subst} \rangle$

**where**



$\langle \text{iter-rcomp}' \sigma 0 = \mathcal{V}' \rangle$   
 $| \langle \text{iter-rcomp}' \sigma (\text{Suc } n) = \sigma \circ (\text{iter-rcomp}' \sigma n) \rangle$   
**lemma** *iter-rcomp-eq-iter-rcomp'*:  $\langle \sigma^n = \text{iter-rcomp}' \sigma n \rangle$   
**by** (*induction n; simp add: iter-rcomp-def*)

**fun** *iter-rcomp''* ::  $\langle ('s, 'v) \text{ subst} \Rightarrow \text{nat} \Rightarrow ('s, 'v) \text{ subst} \rangle$   
**where**  
 $\langle \text{iter-rcomp}'' \sigma 0 = \mathcal{V}' \rangle$   
 $| \langle \text{iter-rcomp}'' \sigma (\text{Suc } n) = (\text{iter-rcomp}'' \sigma n) \circ \sigma \rangle$

**lemma** *iter-rcomp-eq-iter-rcomp''*:  $\langle \sigma^n = \text{iter-rcomp}'' \sigma n \rangle$   
**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  $\sigma$ , the function  $n \mapsto \sigma^n$  i.e. *iter-rcomp*  $\sigma$  is a magma homomorphism between  $(\mathbb{N}, +)$  and  $(\text{subst}, \circ)$ . Since  $\sigma^0 \equiv \mathcal{V}'$ , it is even a (commutative) monoid homomorphism.

**lemma** *iter-comp-add-morphism*:  $\langle (\sigma^n) \circ (\sigma^k) = \sigma^{n+k} \rangle$   
**unfolding** *iter-rcomp-eq-iter-rcomp' iter-rcomp'-eq-iter-rcomp''*  
**by**(*induction k; simp add:*  
*rcomp-assoc*[*of*  $\langle \text{iter-rcomp}'' \sigma n \rangle \langle \text{iter-rcomp}'' \sigma k \rangle \sigma$  **for**  $k$ , *symmetric*])

**lemmas** *iter-comp-com-add-morphism* =  
*iter-comp-add-morphism*[  
*of*  $\sigma n k$  **for**  $\sigma n k$ ,  
*simplified add.commute*,  
*unfolded iter-comp-add-morphism*[*of*  $\sigma k n$ , *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*:  $\langle (\sigma^n)^k = \sigma^{n * k} \rangle$   
**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*  $\sigma n k$  **for**  $\sigma k n$ ,  
*simplified mult.commute*,  
*unfolded iter-comp-mult-morphism*[*of*  $\sigma k n$ , *symmetric*]]

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

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

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

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

**definition** *dom* ::  $\langle ('s, 'v)\ subst \Rightarrow ('s, 'v)\ tm\ set \rangle$  **where**  
 $\langle dom\ \sigma \equiv \{ \mathcal{V}\ x \mid x.\ (\mathcal{V}\ x).\ \sigma \neq \mathcal{V}\ x \} \rangle$

**definition** *ran* ::  $\langle ('s, 'v)\ subst \Rightarrow ('s, 'v)\ tm\ set \rangle$  **where**  
 $\langle ran\ \sigma \equiv (\lambda x.\ x.\ \sigma)\ ' dom\ \sigma \rangle$

**lemma** *no-sub-in-dom-subst-eq*:  $\langle (\forall x \in dom\ \sigma.\ \neg\ sub\ x\ t) \Longrightarrow t = t.\ \sigma \rangle$   
**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*:  
 $\langle (\forall x.\ x \in dom\ \sigma \vee x \in dom\ \vartheta \longrightarrow x.\ \sigma = x.\ \vartheta) \Longrightarrow \sigma = \vartheta \rangle$   
**by** (*intro subst-ext-tmI' allI*)  
 (*metis comp-apply no-sub-in-dom-subst-eq sub-HdE*)

**lemma** *subst-finite-dom*:  $\langle finite\ (dom\ \sigma) \rangle$   
**unfolding** *dom-def* **using** *subst-alt-def*[of  $\sigma$ ] **by** *simp*

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

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

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

**definition** *is-ground-subst* ::  $\langle ('s, 'v)\ subst \Rightarrow bool \rangle$  **where**  
 $\langle is-ground-subst\ \sigma \equiv (ground\ ' ran\ \sigma) = \{ True \} \rangle$

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

by (standard, auto)

**lemma** *ground-subst-grounds*:  $\langle is\text{-ground}\text{-subst } \sigma \implies x \in \text{dom } \sigma \implies \text{ground } (x \cdot \sigma) \rangle$   
unfolding *dom-def*  
by (induction *x*) (auto simp add: *is-ground-subst-alt-def*)

**lemma** *iter-on-ground*:  $\langle \text{ground } (x \cdot \sigma) \implies n > 0 \implies x \cdot \sigma^n = x \cdot \sigma \rangle$   
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*:  
 $\langle \sigma \neq \mathcal{V} \implies \{t. is\text{-Hd } t \wedge is\text{-Var } (\text{head } t) \wedge \text{subst } \sigma t \neq t\} \neq \{\} \rangle$   
proof –  
assume  $\langle \sigma \neq \mathcal{V} \rangle$   
then obtain *t* where  $\langle \sigma t \neq \mathcal{V } t \rangle$   
by blast  
moreover have  $\langle is\text{-Hd } (\mathcal{V } t) \rangle \langle is\text{-Var } (\text{head } (\mathcal{V } t)) \rangle$   
by simp+  
ultimately show ?thesis  
by fastforce  
qed

**lemma** *true-subst-nemp-im*:  $\langle \text{ran } \sigma = \{\} \implies \sigma = \mathcal{V}' \rangle$   
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*:  
 $\langle is\text{-ground}\text{-subst } \sigma \implies (\forall x y. x \neq y \longrightarrow (\mathcal{V } x) \cdot \sigma \neq (\mathcal{V } y)) \rangle$   
proof –  
have  $\langle \neg (\forall x y. x \neq y \longrightarrow (\mathcal{V } x) \cdot \sigma \neq (\mathcal{V } y)) \implies \neg is\text{-ground}\text{-subst } \sigma \rangle$   
proof –  
assume  $\langle \neg (\forall x y. x \neq y \longrightarrow (\mathcal{V } x) \cdot \sigma \neq (\mathcal{V } y)) \rangle$   
then obtain *x y* where *xy-def*:  $\langle x \neq y \rangle \langle (\mathcal{V } x) \cdot \sigma = (\mathcal{V } y) \rangle$   
by blast  
hence  
 $\langle (\mathcal{V } x) \cdot \sigma \neq (\mathcal{V } x) \rangle$   
 $\langle \mathcal{V } x \in \{x. is\text{-Var } (\text{head } x) \wedge x \cdot \sigma \neq x\} \rangle$   
 $\langle \neg \text{ground } ((\mathcal{V } x) \cdot \sigma) \rangle$   
by fastforce+  
  
then show  $\langle \neg is\text{-ground}\text{-subst } \sigma \rangle$   
unfolding *is-ground-subst-def ran-def dom-def*  
by blast  
qed

from *contrapos-pp*[  
rotated,

of  $\langle \forall x y. x \neq y \longrightarrow (\mathcal{V} x) \cdot \sigma \neq (\mathcal{V} y) \rangle \langle \text{is-ground-subst } \sigma \rangle$ ,  
*OF this*  
**show**  $\langle \text{is-ground-subst } \sigma \implies (\forall x y. x \neq y \longrightarrow (\mathcal{V} x) \cdot \sigma \neq (\mathcal{V} y)) \rangle$ .  
**qed**

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

**lemma** *non-ground- $\mathcal{V}'$* :  $\langle \neg \text{is-ground-subst } \mathcal{V}' \rangle$   
**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  $\sigma$  occurs as a subterm of  $x \cdot \sigma^n$  for any  $(0 :: 'a) < n$ .

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

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

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

**lemma** *acyclic-iter-dom-eq*:  $\langle \text{is-acyclic } \sigma \implies \text{dom } \sigma = \text{dom } \sigma^n \rangle$  **if**  $\langle n > 0 \rangle$  **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*:  $\langle \text{is-acyclic } \sigma \implies n > 0 \implies \text{is-acyclic } \sigma^n \rangle$   
**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  $\sigma$  as the substitution  $\sigma^i$  where  $i = \inf\{k \in \mathbb{N} \mid \sigma^k = \sigma^{k+1}\}$ .

**definition**  $fp\text{-subst} :: \langle ('s, 'v) \text{ subst} \Rightarrow ('s, 'v) \text{ subst} \rangle \langle \text{-}^* \rangle 1000$  **where**  
 $\langle \sigma^* \equiv \text{iter-rcomp } \sigma \text{ (LEAST } n . \sigma^n = \sigma^{n+1}) \rangle$

**lemma**  $ground\text{-subst-is-fp} : \langle is\text{-ground-subst } \sigma \Longrightarrow \sigma^* = \sigma \rangle$

— Ground substitutions have no effect and are therefore fixed-points substitutions.

The converse is not true.

**proof** —

**assume**  $asm : \langle is\text{-ground-subst } \sigma \rangle$

**have**  $eq : \langle \sigma^1 = \sigma^2 \rangle$

**apply** ( $simp$  only:  $one\text{-add-one[symmetric]$ ,  
 $simp$ ,  $intro$   $subst\text{-ext-tmI'}$   $allI$ ,  $simp$ )

**subgoal for**  $x$

**unfolding**  $iter\text{-rcomp-eq-iter-rcomp}'$

**by** ( $cases$   $\langle \mathcal{V} x \in \text{dom } \sigma \rangle$ ,

$simp$   $add$ :  $rcomp\text{-subst-simp}$ ,

$metis$

$ground\text{-subst-grounds[OF } asm, \text{ THEN } ground\text{-imp-subst-iden]$

$subst\text{-app.rep-eq}$ ,

$simp$   $add$ :  $dom\text{-def } rcomp\text{-subst-simp}$ ).

**note**  $least\text{-le1} = \text{Least-le[of } \langle \lambda n. \sigma^n = \sigma^{n+1} \rangle 1, \text{ unfolded } nat\text{-1-add-1, OF this]$

**have**  $neq : \langle \sigma^0 \neq \sigma^1 \rangle$

**using**  $asm$   $non\text{-ground-}\mathcal{V}'$

**unfolding**  $iter\text{-rcomp-eq-iter-rcomp}'$

**by**  $auto$

**have**  $\langle (LEAST n. \sigma^n = \sigma^{n+1}) \neq 0 \rangle$

**proof** ( $rule$   $ccontr$ )

**assume**  $\langle \neg (LEAST n. \sigma^n = \sigma^{n+1}) \neq 0 \rangle$

**hence**  $\langle (LEAST n. \sigma^n = \sigma^{n+1}) = 0 \rangle$  **by**  $blast$

**from**

$LeastI$ [ $of$   $\langle \lambda n. \sigma^n = \sigma^{n+1} \rangle 1, \text{ simplified } one\text{-add-one, OF } eq,$   
 $simplified$   $add\text{-0 this}$ ]

$neq$

**show**  $False$  **by**  $blast$

**qed**

**moreover have**  $\langle (LEAST x. \sigma^x = \sigma^{x+1}) = 0 \vee (LEAST x. \sigma^x = \sigma^{x+1}) = 1 \rangle$

**using**  $least\text{-le1}$  **by**  $linarith$

**ultimately have**  $\langle (LEAST n. \sigma^n = \sigma^{n+1}) = 1 \rangle$

**by**  $sat$

**then show**  $\langle \sigma^* = \sigma \rangle$

**unfolding**  $fp\text{-subst-def } iter\text{-rcomp-eq-iter-rcomp}'$

**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  $\sigma$ , we denote the fact that  $x$  is substituted by  $y$  after one application of  $\sigma$ , i.e. that  $\text{sub } y (x \cdot \sigma)$ , 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*:  $\langle x \in \text{dom } \sigma \implies \text{sub } x (t \cdot \sigma) \implies \exists y \in \text{dom } \sigma. \text{sub } x (y \cdot \sigma) \rangle$   
 — If  $x \rightarrow_\sigma t$  for  $x \in \text{dom } \sigma$  and a term  $t$ , then there is a variable  $y \in \text{dom } \sigma$  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  $x \rightarrow_\sigma t$ .

$\langle x \in \text{dom } \sigma \implies \forall z \in \text{dom } \sigma. \neg \text{sub } x (z \cdot \sigma) \implies \forall t. \neg \text{sub } x (t \cdot \sigma) \rangle$   
**using** *dom-sub-subst* **by** *blast*

**lemma** *dom-sub-subst-iter*:

$\langle x \in \text{dom } \sigma \implies \forall z \in \text{dom } \sigma. \neg \text{sub } x (z \cdot \sigma^n) \implies \neg \text{sub } x (t \cdot \sigma^n) \rangle$

**by** (*induction*  $n$  *arbitrary: x*  $\sigma$ ,  
*simp*, *use sub-refl* **in** *blast*,  
*smt* (*verit*, *ccfv-threshold*)  
*One-nat-def*  
*V'-id-tm*  
*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**  $\langle x \in \text{dom } \sigma \rangle \langle \forall y \in \text{dom } \sigma. \text{sub } y \ t \longrightarrow \neg \text{sub } x \ (y \cdot \sigma) \rangle$

**shows** *not-sub-subst-if*:  $\langle \neg \text{sub } x \ (t \cdot \sigma) \rangle$

— For  $x$  in the domain and any term  $t$ , if there is no variable  $y$  occurring in  $t$  such that  $x \rightarrow_{\sigma} y$ , then  $x \not\rightarrow_{\sigma} t$ .

**using** *assms*

**by** (*induction t*)

((*smt (verit, ccfv-threshold)*  
*hd.case-eq-if*  
*subst.simps(1)*  
*V'-id-tm*  
*dom-def*  
*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*:

$\langle x \in \text{dom } \sigma \implies \text{sub } x \ (t \cdot \sigma^{n+1}) \implies$

$\exists y \ z. y \in \text{dom } \sigma \wedge z \in \text{dom } \sigma \wedge \text{sub } x \ (z \cdot \sigma) \wedge \text{sub } z \ (y \cdot \sigma^n) \rangle$

— If  $x \rightarrow_{\sigma}^{n+1} t$ , then there are variables  $y$  and  $z$  such that  $y \rightarrow_{\sigma}^n z \rightarrow_{\sigma} x$ .

**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 V'-neutral-rcomp V'-id-tm*

**using** *dom-sub-subst sub-refl* **by** *blast*

**next**

**case** (*Suc n*)

**then obtain**  $t'$  **where**  $t'$ -def:  $\langle \text{sub } t' \ (t \cdot \sigma^{n+1}) \rangle \langle \text{sub } x \ (t' \cdot \sigma) \rangle$

**by** (*metis Suc-eq-plus1 iter-rcomp-Suc-right rcomp-subst-simp sub-refl*)

**have**  $\langle \exists w \in \text{dom } \sigma. \text{sub } w \ t' \wedge \text{sub } x \ (w \cdot \sigma) \rangle$

**using** *not-sub-subst-if Suc.prem(1) t'-def(2)* **by** *blast*

**then obtain**  $w$  **where**  $w$ -def:  $\langle w \in \text{dom } \sigma \rangle \langle \text{sub } w \ t' \rangle \langle \text{sub } x \ (w \cdot \sigma) \rangle$

**by** *blast*

**then obtain**  $y$  **where**  $\langle y \in \text{dom } \sigma \rangle \langle \text{sub } w (y \cdot \sigma^{n+1}) \rangle$   
**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*:

$\langle \exists x \in \text{dom } \sigma. \text{sub } z (x \cdot \sigma^{n+1}) \rangle \longleftrightarrow$   
 $\langle \exists x y. x \in \text{dom } \sigma \wedge y \in \text{dom } \sigma \wedge \text{sub } z (y \cdot \sigma) \wedge \text{sub } y (x \cdot \sigma^n) \rangle$  **if**  $\langle z \in \text{dom } \sigma \rangle$   
**using** *that*

**proof** (*intro iffI*)

**show**  $\langle z \in \text{dom } \sigma \implies \exists x \in \text{dom } \sigma. \text{sub } z (x \cdot \sigma^n + 1) \implies$   
 $\exists x y. x \in \text{dom } \sigma \wedge y \in \text{dom } \sigma \wedge \text{sub } z (y \cdot \sigma) \wedge \text{sub } y (x \cdot \sigma^n) \rangle$

**proof-**

**assume**  $\langle \exists x \in \text{dom } \sigma. \text{sub } z (x \cdot \sigma^n + 1) \rangle \langle z \in \text{dom } \sigma \rangle$   
**then obtain**  $x$  **where**  $\langle x \in \text{dom } \sigma \rangle \langle \text{sub } z (x \cdot \sigma^{n+1}) \rangle$   
**by** *blast*

**then obtain**  $y$  **where**  $\langle y \in \text{dom } \sigma \rangle \langle \text{sub } z (y \cdot \sigma) \rangle \langle \text{sub } y (x \cdot \sigma^n) \rangle$   
**by** (*metis Suc-eq-plus1*  $\langle z \in \text{dom } \sigma \rangle$  *iter-rcomp-Suc-right*  
*not-sub-subst-if rcomp-subst-simp*)

**then show**  $\langle \exists x y. x \in \text{dom } \sigma \wedge y \in \text{dom } \sigma \wedge \text{sub } z (y \cdot \sigma) \wedge \text{sub } y (x \cdot \sigma^n) \rangle$   
**using**  $\langle x \in \text{dom } \sigma \rangle$  **by** *blast*

**qed**

**show**  $\langle z \in \text{dom } \sigma \implies$

$\exists x y. x \in \text{dom } \sigma \wedge y \in \text{dom } \sigma \wedge \text{sub } z (y \cdot \sigma) \wedge \text{sub } y (x \cdot \sigma^n) \implies$   
 $\exists x \in \text{dom } \sigma. \text{sub } z (x \cdot \sigma^n + 1) \rangle$

**proof** –

**assume**  $\langle \exists x y. x \in \text{dom } \sigma \wedge y \in \text{dom } \sigma \wedge \text{sub } z (y \cdot \sigma) \wedge \text{sub } y (x \cdot \sigma^n) \rangle$

**then obtain**  $x y$  **where** *xy-def*:

$\langle x \in \text{dom } \sigma \rangle \langle y \in \text{dom } \sigma \rangle \langle \text{sub } z (y \cdot \sigma) \rangle \langle \text{sub } y (x \cdot \sigma^n) \rangle$

**by** *blast*

**then have**  $\langle \text{sub } (y \cdot \sigma) (x \cdot \sigma^{n+1}) \rangle$ ,

**unfolding** *Suc-eq-plus1[symmetric]* *iter-rcomp-Suc-right rcomp-subst-simp*

**using** *sub-subst'* **by** *blast*

**with** *xy-def(3)* **have**  $\langle \text{sub } z (x \cdot \sigma^{n+1}) \rangle$ ,

**using** *sub-trans* **by** *blast*

**with** *xy-def(1)* **show**  $\langle \exists x \in \text{dom } \sigma. \text{sub } z (x \cdot \sigma^n + 1) \rangle$

**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*:  $\langle \text{is-acyclic } \sigma \implies \exists n. \sigma^n = \sigma^{n+1} \rangle$

**proof** –

**assume** *acyc*:  $\langle \text{is-acyclic } \sigma \rangle$

**define**  $S$  **where**  $\langle S \equiv \lambda n. \bigcup (\text{subterms } \langle \text{ran } \sigma^n \rangle \cap \text{dom } \sigma) \rangle$

**have** *S-alt-def*:  $\langle S n = \{z \in \text{dom } \sigma. \exists x \in \text{dom } \sigma. \text{sub } z (x \cdot \sigma^n)\} \rangle$  **if**  $\langle n > 0 \rangle$  **for**  $n$

**unfolding** *S-def ran-def subterms-def*

*acyclic-iter-dom-eq*[*OF that acyc, symmetric*]

**by** *blast*

**have** *S-σ-unfold*:

$\langle S n \cdot \sigma = \{x \cdot \sigma \mid x. x \in \text{dom } \sigma \wedge (\exists y \in \text{dom } \sigma. \text{sub } x (y \cdot \sigma^n))\} \rangle$  **if**  $\langle n > 0 \rangle$  **for**  $n$

**unfolding** *set-image-subst-collect*[*of*  $\langle S n \rangle$ , *unfolded S-def*]

*S-def subterms-def*

**unfolding** *ran-def acyclic-iter-dom-eq*[*OF that acyc, symmetric*]

**by** *blast*

**have** *sources-charac*:

$\langle n > 0 \implies S (n+1) = S n - \{z \in S n. \forall x \in S n. \neg \text{sub } z (x \cdot \sigma)\} \rangle$  **for**  $n$

**proof** (*intro subset-antisym subsetI*)

**fix**  $z$  **assume**  $\langle n > 0 \rangle$

**show**  $\langle z \in S (n+1) \implies z \in S n - \{z \in S n. \forall x \in S n. \neg \text{sub } z (x \cdot \sigma)\} \rangle$

**proof**–

**assume**  $\langle z \in S (n+1) \rangle$

**then obtain**  $x$  **where** *x-def*:  $\langle x \in \text{dom } \sigma \rangle$   $\langle \text{sub } z (x \cdot \sigma^{n+1}) \rangle$

**unfolding** *S-alt-def*[*OF zero-less-Suc*[*of*  $n$ , *unfolded Suc-eq-plus1*]]

**by** *blast*

**then have**  $\langle z \in S n \rangle$

**unfolding** *S-def ran-def*

$acyclic\text{-}iter\text{-}dom\text{-}eq[OF \langle n > 0 \rangle acyc, symmetric]$   
 $subterms\text{-}def$   
**by** ( $simp, metis IntE S\text{-}def Suc\text{-}eq\text{-}plus1 \langle z \in S (n + 1) \rangle$   
 $dom\text{-}sub\text{-}subst\text{-}iter iter\text{-}rcomp\text{-}Suc\text{-}left rcomp\text{-}subst\text{-}simp$ )  
**moreover have**  $\langle \exists x \in S n. sub z (x \cdot \sigma) \rangle$   
**using**  
 $S\text{-}alt\text{-}def[OF \langle n > 0 \rangle]$   
 $dom\text{-}sub\text{-}subst\text{-}iter\text{-}Suc[of z \sigma]$   
 $x\text{-}def(\emptyset) \langle z \in S n \rangle$   
**by fast**

**ultimately show**  $\langle z \in S n - \{z \in S n. \forall x \in S n. \neg sub z (x \cdot \sigma)\} \rangle$   
**by blast**

**qed**  
**show**  $\langle z \in S n - \{z \in S n. \forall x \in S n. \neg sub z (x \cdot \sigma)\} \implies z \in S (n + 1) \rangle$   
**proof-**  
**assume**  $\langle z \in S n - \{z \in S n. \forall x \in S n. \neg sub z (x \cdot \sigma)\} \rangle$   
**hence**  $\langle z \in S n \rangle \langle \exists y \in S n. sub z (y \cdot \sigma) \rangle$   
**by blast+**  
**then obtain**  $x y w$  **where**  
 $\langle x \in dom \sigma \rangle \langle sub z (x \cdot \sigma^n) \rangle$   
 $\langle y \in dom \sigma \rangle \langle w \in dom \sigma \rangle \langle sub z (y \cdot \sigma) \rangle \langle sub y (w \cdot \sigma^n) \rangle$   
**unfolding**  $S\text{-}def subterms\text{-}def ran\text{-}def$   
 $acyclic\text{-}iter\text{-}dom\text{-}eq[OF \langle n > 0 \rangle acyc, symmetric]$   
**by blast**  
**show**  $\langle z \in S (n + 1) \rangle$   
**unfolding**  $S\text{-}alt\text{-}def[OF zero\text{-}less\text{-}Suc[of n], unfolded Suc\text{-}eq\text{-}plus1]$   
**apply** ( $simp, intro conjI, use \langle z \in S n \rangle[unfolded S\text{-}def]$  **in**  $simp$ )  
**using**  $sub\text{-}trans[OF \langle sub z (y \cdot \sigma) \rangle$   
 $sub\text{-}subst'[OF \langle sub y (w \cdot \sigma^n) \rangle, of \sigma]] \langle w \in dom \sigma \rangle$   
**unfolding**  $iter\text{-}rcomp\text{-}Suc\text{-}right rcomp\text{-}subst\text{-}simp$   
**by** ( $intro bexI[of - w]$ )

**qed**  
**qed**

**have**  $fin\text{-}Sn: \langle finite (S n) \rangle$  **for**  $n$   
**unfolding**  $S\text{-}def$  **using**  $subst\text{-}finite\text{-}dom$   
**by blast**

**have**  $decr: \langle S n \neq \emptyset \implies S (n + 1) \subset S n \rangle$  **for**  $n$   
**proof**  
**show**  $\langle S n \neq \emptyset \implies S (n + 1) \subseteq S n \rangle$   
**proof** ( $cases \langle n > 0 \rangle$ )  
**case True**  
**assume**  $\langle S n \neq \emptyset \rangle$   
**show**  $?thesis$   
**proof** ( $intro subsetI, rule ccontr$ )  
**fix**  $x$  **assume**  $\langle x \in S (n + 1) \rangle \langle x \notin S n \rangle$   
**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  $\langle x \in \text{dom } \sigma \rangle \langle y \in \text{dom } \sigma \rangle \langle \text{sub } x (y \cdot \sigma^{n+1}) \rangle$ 
  by blast
with  $\langle x \notin S \ n \rangle$ [unfolded S-alt-def[OF True], simplified]
have  $\langle \forall z \in \text{dom } \sigma. \neg \text{sub } x (z \cdot \sigma^n) \rangle$ 
  by blast
from
  dom-sub-subst-iter[OF  $\langle x \in \text{dom } \sigma \rangle$  this, of  $\langle y \cdot \sigma \rangle$ ]
show False
  using  $\langle \text{sub } x (y \cdot \sigma^{n+1}) \rangle$ [
    folded Suc-eq-plus1,
    unfolded iter-rcomp-Suc-left rcomp-subst-simp] ..
qed
next
case False
assume  $\langle S \ n \neq \emptyset \rangle$ 
moreover have  $\langle S \ 0 = \emptyset \rangle$ 
  unfolding S-def iter-rcomp-eq-iter-rcomp'
  by (simp add: ran-V'-empty)
ultimately show ?thesis
  using False by simp
qed

show  $\langle S \ n \neq \emptyset \implies S \ (n+1) \neq S \ n \rangle$ 
proof (cases  $n$ )
case (Suc  $m$ )
hence  $\langle n > 0 \rangle$  by simp
assume  $\langle S \ n \neq \emptyset \rangle$ 
have  $\langle \{z \in S \ n. \forall x \in S \ n. \neg \text{sub } z (x \cdot \sigma)\} \neq \emptyset \rangle$ 
proof (rule ccontr)
  assume  $\langle \neg \{z \in S \ n. \forall x \in S \ n. \neg \text{sub } z (x \cdot \sigma)\} \neq \{ \} \rangle$ 
  hence  $ch: \langle \forall x. x \in S \ n \longrightarrow (\exists y \in S \ n. \text{sub } x (y \cdot \sigma)) \rangle$ 
  by simp
  define  $E$  where  $\langle E \equiv \lambda x. \{y \in S \ n. \text{sub } x (y \cdot \sigma)\} \rangle$ 
  have  $E\text{-nemp}: \langle x \in S \ n \implies E \ x \neq \emptyset \rangle$  for  $x$ 
    unfolding E-def using  $ch$  by fast
  have  $\langle x \in S \ n \implies x \notin E \ x \rangle$  for  $x$ 
proof (rule ccontr)
  assume  $a: \langle x \in S \ n \rangle \langle \neg x \notin E \ x \rangle$ 
  from this(1) have  $\langle x \in \text{dom } \sigma \rangle$ 
    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*  $\langle S\ n \neq \emptyset \rangle$  **obtain** *x y* **where**  $\langle x \in S\ n \rangle \langle y \in S\ n \rangle \langle \text{sub } y\ (x \cdot \sigma) \rangle$   
**by** *blast*  
**with** *acyc* **have**  $\langle y \neq x \rangle$   
**unfolding** *S-def is-acyclic-def subterms-def*  
**by** *(metis IntE One-nat-def  $\mathcal{V}'$ -id-tm iter-rcomp-0  
 iter-rcomp-Suc-left mem-Collect-eq rcomp-subst-simp zero-less-one)*

**let**  $?C = \langle \text{card } (S\ n) \rangle$   
**have** *incl*:  $\langle E\ v \subseteq S\ n \rangle$  **for** *v*  
**unfolding** *E-def* **by** *blast*

**have**  $\langle k > 0 \implies$   
 $\exists \text{ seq. } k = \text{length seq} \wedge$   
 $\text{seq!}0 \in S\ n \wedge$   
 $(\forall m < \text{length seq} - 1. \text{seq!}(m+1) \in E\ (\text{seq!}m)) \wedge$   
 $\text{distinct seq} \rangle$   
**for** *k*

**proof** *(induction k rule: nat-induct-non-zero)*  
**case** *(Suc k)*  
**then obtain** *seq* **where** *seq-def*:  $\langle k = \text{length seq} \rangle \langle \text{seq!}0 \in S\ n \rangle$   
 $\langle (\forall m < k-1. \text{seq!}(m+1) \in E\ (\text{seq!}m)) \rangle \langle \text{distinct seq} \rangle$   
**by** *blast*  
**hence**  $\langle l < k \implies \text{seq!}l \in S\ n \rangle$  **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*:  $\langle v \in E\ (\text{seq!}(k-1)) \rangle$   
**using** *E-nemp Suc.hyps*  
**by** *(metis Suc-eq-plus1 Suc-pred' all-not-in-conv less-add-one)*

**have** *seq-sub*:  
 $\langle i < \text{length seq} \implies j \leq i \implies \text{sub } (\text{seq!}j)\ ((\text{seq!}i) \cdot \sigma^{i-j}) \rangle$  **for** *i j*

**proof** *(induction i arbitrary: j)*  
**case** *(Suc l)*  
**with** *seq-def* **have**  $\langle \text{seq!}(Suc\ l) \in E\ (\text{seq!}l) \rangle$   
**by** *simp*  
**hence**  $\langle \text{sub } (\text{seq!}l)\ ((\text{seq!}(Suc\ l)) \cdot \sigma) \rangle$   
**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**  $\langle seq' \equiv seq \ @ \ [v] \rangle$   
**hence**  $\langle sub \ (seq!(k-1)) \ (v \cdot \sigma) \rangle$   
**using** *v-def unfolding E-def*  
**by** *blast*

**have**  $\langle v \in dom \ \sigma \rangle$   
**using** *incl v-def*  
**unfolding** *E-def S-def*  
**by** *fast*

**have**  $\langle v \notin set \ seq \rangle$

**proof**

**assume**  $\langle v \in set \ seq \rangle$   
**then obtain** *j* **where**  $\langle j < k \rangle \langle v = seq!j \rangle$   
**unfolding** *in-set-conv-nth seq-def(1)* **by** *fast*  
**with** *seq-sub[of  $\langle k-1 \rangle$  *j*, folded seq-def(1) this(2)] Suc.hyps*  
**have**  $\langle sub \ v \ (seq!(k-1) \cdot \sigma^{k-1-j}) \rangle$   
**by** *simp*  
**from** *sub-trans[OF*  
   *this*  
   *sub-subst'[OF  $\langle sub \ (seq!(k-1)) \ (v \cdot \sigma) \rangle$ , of  $\langle \sigma^{k-1-j} \rangle$ ,*  
   *folded rcomp-subst-simp iter-rcomp-Suc-left]]*  
**show** *False*  
**using** *acyc  $\langle v \in dom \ \sigma \rangle$*   
**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-ℳ'-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-ℳ'-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 · σk))} = ∅⟩
by blast
then have ⟨σk = σk+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: ⟨σm = σm+1⟩
by blast
define n where ⟨n ≡ LEAST n. σn = σn+1⟩
have eq: ⟨σ* = σn⟩
unfolding fp-subst-def n-def..
from m-def have ⟨σn = σn+1⟩
unfolding n-def
by (intro LeastI[of ⟨λn. σn = σn+1⟩ m])

```

```

hence ⟨ $\sigma^n = \sigma^{n+k}$ ⟩ for  $k$ 
proof (induction  $k$ )
  case (Suc  $k$ )
  show ?case
    unfolding add-Suc-right iter-rcomp-Suc-right
      Suc.IH[OF Suc.premis, symmetric]
    unfolding iter-rcomp-Suc-right[symmetric] Suc-eq-plus1
    using Suc.premis.
  qed simp
then show ?thesis
  unfolding eq iter-comp-com-add-morphism
  by (induction  $n$ ) presburger+
qed

lemma fp-subst-stable-iter: ⟨is-acyclic  $\sigma \implies n > 0 \implies (\sigma^*)^n = \sigma^*$ ⟩
  by (induction  $n$ , simp,
    unfold iter-rcomp-Suc-right,
    use fp-subst-comp-stable in fastforce)

lemma fp-subst-stable-fp: ⟨is-acyclic  $\sigma \implies (\sigma^*)^* = \sigma^*$ ⟩
proof –
  assume ⟨is-acyclic  $\sigma$ ⟩
  define  $m$  where ⟨ $m \equiv \text{LEAST } n. (\sigma^*)^n = (\sigma^*)^{n+1}$ ⟩
  show ?thesis
    unfolding fp-subst-def[of ⟨ $\sigma^*$ ⟩, folded  $m$ -def]
    using fp-subst-stable-iter[OF ⟨is-acyclic  $\sigma$ ⟩]
    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](https://isa-afp.org/entries/Lambda_Free_RPOs.html), Formal proof development.
- [3] S. Tournet, P. Fontaine, D. E. Ouraoui, and H. Barbosa. Lifting congruence closure with free variables to  $\lambda$ -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.