

Abstract Substitutions as Monoid Actions

Martin Desharnais

April 11, 2026

Abstract

This entry provides a small, reusable, theory that specifies the abstract concept of substitution as monoid action. Both the substitution type and the object type are kept abstract. The theory provides multiple useful definitions and lemmas. Two example usages are provided for first order terms: one for terms from the AFP/First_Order_Terms session and one for terms from the Isabelle/HOL-ex session.

Contents

1	General Results on Groups	1
2	Monoid	2
3	Semigroup Action	2
4	Monoid Action	3
5	Group Action	4
6	Assumption-free Substitution	5
7	Basic Substitution	8
7.1	Substitution Composition	10
7.2	Substitution Identity	10
7.3	Generalization	10
7.4	Substituting on Ground Expressions	11
7.5	Instances of Ground Expressions	11
7.6	Unifier of Ground Expressions	11
7.7	Ground Substitutions	12
7.8	IMGU is Idempotent and an MGU	13
7.9	IMGU can be used before unification	13
7.10	Groundings Idempotence	13
7.11	Instances of Substitution	13
7.12	Instances of Renamed Expressions	14

8	Substitutions on variables	17
8.1	Properties of substitutions	19
9	Substitutions on base expressions	32
10	Properties of substitutions on base expressions	33
11	Lifting of substitutions using natural functors	48
11.1	Lifting of properties	51
12	Lifting of based substitutions	61
12.1	Lifting of properties	62
13	Substitutions for first order terms	65
13.1	Interpretations for first order terms	65
13.2	Compatibility with First_Order_Term	69
13.3	Interpretations for IMGUs	71
13.4	Additional lemmas	72
14	Substitutions for first order terms as binary tree	72
14.1	Substitution monoid	73
14.2	Transfer definitions and lemmas from HOL-ex.Unification . .	73
14.3	Base Substitution	75
14.4	Substitution Properties	76
14.5	Compatibility with HOL-ex.Unification	79
14.6	Interpretations for IMGUs	80

```

theory Monoid-Action
  imports Main
begin

```

1 General Results on Groups

```

lemma (in monoid) right-inverse-idem:
  fixes inv
  assumes right-inverse:  $\bigwedge a. a * inv\ a = \mathbf{1}$ 
  shows  $\bigwedge a. inv\ (inv\ a) = a$ 
  by (metis assoc right-inverse right-neutral)

```

```

lemma (in monoid) left-inverse-if-right-inverse:
  fixes inv
  assumes
    right-inverse:  $\bigwedge a. a * inv\ a = \mathbf{1}$ 
  shows  $inv\ a * a = \mathbf{1}$ 
  by (metis right-inverse-idem right-inverse)

```

```

lemma (in monoid) group-wrt-right-inverse:
  fixes inv

```

```

assumes right-inverse:  $\bigwedge a. a * \text{inv } a = \mathbf{1}$ 
shows group (*) 1 inv
proof unfold-locales
  show  $\bigwedge a. \mathbf{1} * a = a$ 
    by simp
next
  show  $\bigwedge a. \text{inv } a * a = \mathbf{1}$ 
    by (metis left-inverse-if-right-inverse right-inverse)
qed

```

2 Monoid

definition (*in monoid*) *is-left-invertible* **where**
is-left-invertible $a \iff (\exists a\text{-inv}. a\text{-inv} * a = \mathbf{1})$

definition (*in monoid*) *is-right-invertible* **where**
is-right-invertible $a \iff (\exists a\text{-inv}. a * a\text{-inv} = \mathbf{1})$

definition (*in monoid*) *left-inverse* **where**
is-left-invertible $a \implies \text{left-inverse } a = (\text{SOME } a\text{-inv}. a\text{-inv} * a = \mathbf{1})$

definition (*in monoid*) *right-inverse* **where**
is-right-invertible $a \implies \text{right-inverse } a = (\text{SOME } a\text{-inv}. a * a\text{-inv} = \mathbf{1})$

lemma (*in monoid*) *comp-left-inverse* [*simp*]:
is-left-invertible $a \implies \text{left-inverse } a * a = \mathbf{1}$
by (*auto simp: is-left-invertible-def left-inverse-def intro: someI-ex*)

lemma (*in monoid*) *comp-right-inverse* [*simp*]:
is-right-invertible $a \implies a * \text{right-inverse } a = \mathbf{1}$
by (*auto simp: is-right-invertible-def right-inverse-def intro: someI-ex*)

lemma (*in monoid*) *neutral-is-left-invertible* [*simp*]:
is-left-invertible $\mathbf{1}$
by (*simp add: is-left-invertible-def*)

lemma (*in monoid*) *neutral-is-right-invertible* [*simp*]:
is-right-invertible $\mathbf{1}$
by (*simp add: is-right-invertible-def*)

3 Semigroup Action

We define both left and right semigroup actions. Left semigroup actions seem to be prevalent in algebra, but right semigroup actions directly uses the usual notation of term/atom/literal/clause substitution.

locale *left-semigroup-action* = *semigroup* +
fixes *action* :: 'a \Rightarrow 'b \Rightarrow 'b (**infix** $\langle \cdot \rangle$ 70)

assumes *action-compatibility*[simp]: $\bigwedge a b x. (a * b) \cdot x = a \cdot (b \cdot x)$

locale *right-semigroup-action* = *semigroup* +
fixes *action* :: 'b \Rightarrow 'a \Rightarrow 'b (**infix** $\langle \cdot \rangle$ 70)
assumes *action-compatibility*[simp]: $\bigwedge x a b. x \cdot (a * b) = (x \cdot a) \cdot b$

We then instantiate the right action in the context of the left action in order to get access to any lemma proven in the context of the other locale. We do analogously in the context of the right locale.

sublocale *left-semigroup-action* \subseteq *right-semigroup-action* **where**
f = $\lambda x y. f y x$ **and** *action* = $\lambda x y. action y x$

proof *unfold-locales*

show $\bigwedge a b c. c * (b * a) = c * b * a$
by (*simp only: assoc*)

next

show $\bigwedge x a b. (b * a) \cdot x = b \cdot (a \cdot x)$
by *simp*

qed

sublocale *right-semigroup-action* \subseteq *left-semigroup-action* **where**
f = $\lambda x y. f y x$ **and** *action* = $\lambda x y. action y x$

proof *unfold-locales*

show $\bigwedge a b c. c * (b * a) = c * b * a$
by (*simp only: assoc*)

next

show $\bigwedge a b x. x \cdot (b * a) = (x \cdot b) \cdot a$
by *simp*

qed

lemma (**in** *right-semigroup-action*) *lifting-semigroup-action-to-set*:
right-semigroup-action (*) ($\lambda X a. (\lambda x. action x a) \text{ ` } X$)

proof *unfold-locales*

show $\bigwedge x a b. (\lambda x. x \cdot (a * b)) \text{ ` } x = (\lambda x. x \cdot b) \text{ ` } (\lambda x. x \cdot a) \text{ ` } x$
by (*simp add: image-comp*)

qed

lemma (**in** *right-semigroup-action*) *lifting-semigroup-action-to-list*:
right-semigroup-action (*) ($\lambda xs a. map (\lambda x. action x a) xs$)

proof *unfold-locales*

show $\bigwedge x a b. map (\lambda x. x \cdot (a * b)) x = map (\lambda x. x \cdot b) (map (\lambda x. x \cdot a) x)$
by (*simp add: image-comp*)

qed

4 Monoid Action

locale *left-monoid-action* = *monoid* +
fixes *action* :: 'a \Rightarrow 'b \Rightarrow 'b (**infix** $\langle \cdot \rangle$ 70)
assumes

monoid-action-compatibility: $\bigwedge a b x. (a * b) \cdot x = a \cdot (b \cdot x)$ **and**
action-neutral[simp]: $\bigwedge x. \mathbf{1} \cdot x = x$

locale *right-monoid-action* = *monoid* +
fixes *action* :: 'b \Rightarrow 'a \Rightarrow 'b (**infix** $\langle \cdot \rangle$ 70)
assumes

monoid-action-compatibility: $\bigwedge x a b. x \cdot (a * b) = (x \cdot a) \cdot b$ **and**
action-neutral[simp]: $\bigwedge x. x \cdot \mathbf{1} = x$

sublocale *left-monoid-action* \subseteq *left-semigroup-action*
by *unfold-locale* (*fact monoid-action-compatibility*)

sublocale *right-monoid-action* \subseteq *right-semigroup-action*
by *unfold-locale* (*fact monoid-action-compatibility*)

sublocale *left-monoid-action* \subseteq *right*: *right-monoid-action* **where**
f = $\lambda x y. f y x$ **and** *action* = $\lambda x y. \text{action } y x$
by *unfold-locale* *simp-all*

sublocale *right-monoid-action* \subseteq *left*: *left-monoid-action* **where**
f = $\lambda x y. f y x$ **and** *action* = $\lambda x y. \text{action } y x$
by *unfold-locale* *simp-all*

lemma (**in** *right-monoid-action*) *lifting-monoid-action-to-set*:
right-monoid-action (*) **1** ($\lambda X a. (\lambda x. \text{action } x a) \text{ ' } X$)

proof *unfold-locale*

show $\bigwedge x a b. (\lambda x. x \cdot (a * b)) \text{ ' } x = (\lambda x. x \cdot b) \text{ ' } (\lambda x. x \cdot a) \text{ ' } x$
by (*simp add: image-comp*)

next

show $\bigwedge x. (\lambda x. x \cdot \mathbf{1}) \text{ ' } x = x$
by *simp*

qed

lemma (**in** *right-monoid-action*) *lifting-monoid-action-to-list*:

right-monoid-action (*) **1** ($\lambda xs a. \text{map } (\lambda x. \text{action } x a) xs$)

proof *unfold-locale*

show $\bigwedge x a b. \text{map } (\lambda x. x \cdot (a * b)) x = \text{map } (\lambda x. x \cdot b) (\text{map } (\lambda x. x \cdot a) x)$
by *simp*

next

show $\bigwedge x. \text{map } (\lambda x. x \cdot \mathbf{1}) x = x$
by *simp*

qed

5 Group Action

locale *left-group-action* = *group* +
fixes *action* :: 'a \Rightarrow 'b \Rightarrow 'b (**infix** $\langle \cdot \rangle$ 70)
assumes

group-action-compatibility: $\bigwedge a b x. (a * b) \cdot x = a \cdot (b \cdot x)$ **and**

group-action-neutral: $\bigwedge x. \mathbf{1} \cdot x = x$

locale *right-group-action* = *group* +
fixes *action* :: 'b \Rightarrow 'a \Rightarrow 'b (**infixl** $\langle \cdot \rangle$ 70)
assumes
group-action-compatibility: $\bigwedge x a b. x \cdot (a * b) = (x \cdot a) \cdot b$ **and**
group-action-neutral: $\bigwedge x. x \cdot \mathbf{1} = x$

sublocale *left-group-action* \subseteq *left-monoid-action*
by *unfold-locale* (*fact group-action-compatibility group-action-neutral*)+

sublocale *right-group-action* \subseteq *right-monoid-action*
by *unfold-locale* (*fact group-action-compatibility group-action-neutral*)+

sublocale *left-group-action* \subseteq *right: right-group-action* **where**
f = $\lambda x y. f y x$ **and** *action* = $\lambda x y. action y x$
by *unfold-locale simp-all*

sublocale *right-group-action* \subseteq *left: left-group-action* **where**
f = $\lambda x y. f y x$ **and** *action* = $\lambda x y. action y x$
by *unfold-locale simp-all*

end

theory *Abstract-Substitution*
imports *Monoid-Action*
begin

abbreviation *set-prod* **where**
set-prod $\equiv \lambda(t, t'). \{t, t'\}$

6 Assumption-free Substitution

locale *abstract-substitution-ops* =
fixes
subst :: 'x \Rightarrow 's \Rightarrow 'x (**infixl** $\langle \cdot \rangle$ 67) **and**
id-subst :: 's **and**
comp-subst :: 's \Rightarrow 's \Rightarrow 's (**infixl** $\langle \odot \rangle$ 67) **and**
is-ground :: 'x \Rightarrow *bool*

begin

definition *subst-set* :: 'x *set* \Rightarrow 's \Rightarrow 'x *set* **where**
subst-set X $\sigma = (\lambda x. subst x \sigma) ` X$

definition *subst-list* :: 'x *list* \Rightarrow 's \Rightarrow 'x *list* **where**
subst-list xs $\sigma = map (\lambda x. subst x \sigma) xs$

definition *is-ground-set* :: 'x *set* \Rightarrow *bool* **where**
is-ground-set X $\longleftrightarrow (\forall x \in X. is-ground x)$

definition *is-ground-subst* :: 's ⇒ bool **where**
is-ground-subst $\gamma \longleftrightarrow (\forall x. \text{is-ground } (x \cdot \gamma))$

definition *generalizes* :: 'x ⇒ 'x ⇒ bool **where**
generalizes $x \ y \longleftrightarrow (\exists \sigma. x \cdot \sigma = y)$

definition *specializes* :: 'x ⇒ 'x ⇒ bool **where**
specializes $x \ y \equiv \text{generalizes } y \ x$

definition *strictly-generalizes* :: 'x ⇒ 'x ⇒ bool **where**
strictly-generalizes $x \ y \longleftrightarrow \text{generalizes } x \ y \wedge \neg \text{generalizes } y \ x$

definition *strictly-specializes* :: 'x ⇒ 'x ⇒ bool **where**
strictly-specializes $x \ y \equiv \text{strictly-generalizes } y \ x$

definition *instances* :: 'x ⇒ 'x set **where**
instances $x = \{y. \text{generalizes } x \ y\}$

definition *instances-set* :: 'x set ⇒ 'x set **where**
instances-set $X = (\bigcup x \in X. \text{instances } x)$

definition *ground-instances* :: 'x ⇒ 'x set **where**
ground-instances $x = \{x_G \in \text{instances } x. \text{is-ground } x_G\}$

definition *ground-instances-set* :: 'x set ⇒ 'x set **where**
ground-instances-set $X = \{x_G \in \text{instances-set } X. \text{is-ground } x_G\}$

lemma *ground-instances-set-eq-Union-ground-instances*:
ground-instances-set $X = (\bigcup x \in X. \text{ground-instances } x)$
unfolding *ground-instances-set-def* *ground-instances-def*
unfolding *instances-set-def*
by *auto*

lemma *ground-instances-eq-Collect-subst-grounding*:
ground-instances $x = \{x \cdot \gamma \mid \gamma. \text{is-ground } (x \cdot \gamma)\}$
by (*auto simp: ground-instances-def instances-def generalizes-def*)

lemma *mem-ground-instancesE[elim]*:
fixes $x \ x_G :: 'x$
assumes $x_G \in \text{ground-instances } x$
obtains $\gamma :: 's$ **where** $x_G = x \cdot \gamma$ **and** *is-ground* $(x \cdot \gamma)$
using *assms*
unfolding *ground-instances-eq-Collect-subst-grounding mem-Collect-eq*
by *iprover*

lemma *mem-ground-instances-setE[elim]*:
fixes $x_G :: 'x$ **and** $X :: 'x \text{ set}$
assumes $x_G \in \text{ground-instances-set } X$
obtains $x :: 'x$ **and** $\gamma :: 's$ **where** $x \in X$ **and** $x_G = x \cdot \gamma$ **and** *is-ground* $(x \cdot \gamma)$

using *assms*
unfolding *ground-instances-set-eq-Union-ground-instances*
by *blast*

definition *is-unifier* :: 's ⇒ 'x set ⇒ bool **where**
is-unifier v X ↔ X = {} ∨ card (subst-set X v) = 1

definition *is-unifier-set* :: 's ⇒ 'x set set ⇒ bool **where**
is-unifier-set v XX ↔ (∀ X ∈ XX. *is-unifier* v X)

definition *is-mgu* :: 's ⇒ 'x set set ⇒ bool **where**
is-mgu μ XX ↔ *is-unifier-set* μ XX ∧ (∀ v. *is-unifier-set* v XX → (∃ σ. μ ⊙ σ = v))

definition *is-imgu* :: 's ⇒ 'x set set ⇒ bool **where**
is-imgu μ XX ↔ *is-unifier-set* μ XX ∧ (∀ τ. *is-unifier-set* τ XX → μ ⊙ τ = τ)

lemma *is-unifier-iff*: *is-unifier* σ X ↔ (∀ x ∈ X. ∀ y ∈ X. x · σ = y · σ)
proof (*rule iffI*)

show *is-unifier* σ X ⇒ (∀ x ∈ X. ∀ y ∈ X. x · σ = y · σ)
unfolding *is-unifier-def subst-set-def*
by (*smt (verit, best) all-not-in-conv card-1-singletonE imageI singletonD*)

next

show (∀ x ∈ X. ∀ y ∈ X. x · σ = y · σ) ⇒ *is-unifier* σ X
unfolding *is-unifier-def subst-set-def*
by (*smt (verit, del-insts) ex-in-conv image-iff is-singletonI' is-singleton-altdef*)

qed

lemma *is-unifier-singleton[simp]*: *is-unifier* v {x}
by (*simp add: is-unifier-iff*)

lemma *is-unifier-empty[simp]*: *is-unifier* σ {}
by (*simp add: is-unifier-iff*)

lemma *is-unifier-set-empty[simp]*: *is-unifier-set* σ {}
by (*simp add: is-unifier-set-def*)

lemma *is-unifier-set-insert*:
is-unifier-set σ (insert X XX) ↔ *is-unifier* σ X ∧ *is-unifier-set* σ XX
by (*simp add: is-unifier-set-def*)

lemma *is-unifier-set-insert-singleton[simp]*:
is-unifier-set σ (insert {x} XX) ↔ *is-unifier-set* σ XX
by (*simp add: is-unifier-set-def*)

lemma *is-mgu-insert-singleton[simp]*: *is-mgu* μ (insert {x} XX) ↔ *is-mgu* μ XX

by (simp add: is-mgu-def)

lemma *is-imgu-insert-singleton*[simp]: $is\text{-imgu } \mu (insert \{x\} XX) \longleftrightarrow is\text{-imgu } \mu XX$
by (simp add: is-imgu-def)

lemma *subst-set-empty*[simp]: $subst\text{-set } \{\} \sigma = \{\}$
by (simp only: subst-set-def image-empty)

lemma *subst-set-insert*[simp]: $subst\text{-set } (insert x X) \sigma = insert (x \cdot \sigma) (subst\text{-set } X \sigma)$
by (simp only: subst-set-def image-insert)

lemma *subst-set-union*[simp]: $subst\text{-set } (X1 \cup X2) \sigma = subst\text{-set } X1 \sigma \cup subst\text{-set } X2 \sigma$
by (simp only: subst-set-def image-Un)

lemma *subst-list-Nil*[simp]: $subst\text{-list } [] \sigma = []$
by (simp only: subst-list-def list.map)

lemma *subst-list-insert*[simp]: $subst\text{-list } (x \# xs) \sigma = (x \cdot \sigma) \# (subst\text{-list } xs \sigma)$
by (simp only: subst-list-def list.map)

lemma *subst-list-append*[simp]: $subst\text{-list } (xs_1 @ xs_2) \sigma = subst\text{-list } xs_1 \sigma @ subst\text{-list } xs_2 \sigma$
by (simp only: subst-list-def map-append)

lemma *is-unifier-set-union*:
 $is\text{-unifier-set } v (XX_1 \cup XX_2) \longleftrightarrow is\text{-unifier-set } v XX_1 \wedge is\text{-unifier-set } v XX_2$
by (auto simp add: is-unifier-set-def)

lemma *is-unifier-subset*: $is\text{-unifier } v A \implies B \subseteq A \implies is\text{-unifier } v B$
unfolding *is-unifier-iff*
by *auto*

lemma *is-ground-set-subset*: $is\text{-ground-set } A \implies B \subseteq A \implies is\text{-ground-set } B$
by (auto simp: is-ground-set-def)

lemma *is-ground-set-ground-instances*[simp]: $is\text{-ground-set } (ground\text{-instances } x)$
by (simp add: ground-instances-def is-ground-set-def)

lemma *is-ground-set-ground-instances-set*[simp]: $is\text{-ground-set } (ground\text{-instances-set } x)$
by (simp add: ground-instances-set-def is-ground-set-def)

end

7 Basic Substitution

locale *abstract-substitution-monoid* = *monoid comp-subst id-subst*
for
 comp-subst :: 's \Rightarrow 's \Rightarrow 's **and**
 id-subst :: 's
begin

abbreviation *is-renaming* **where**
 is-renaming \equiv *is-right-invertible*

lemmas *is-renaming-def* = *is-right-invertible-def*

abbreviation *renaming-inverse* **where**
 renaming-inverse \equiv *right-inverse*

lemmas *renaming-inverse-def* = *right-inverse-def*

lemmas *is-renaming-id-subst* = *neutral-is-right-invertible*

definition *is-idem* :: 's \Rightarrow *bool* **where**
 is-idem a \longleftrightarrow *comp-subst* a a = a

lemma *is-idem-id-subst* [*simp*]: *is-idem id-subst*
by (*simp add: is-idem-def*)

lemma *exists-renaming* [*intro*]: $\exists \rho. \textit{is-renaming } \rho$
using *neutral-is-right-invertible*
by *blast*

lemma *exists-idem* [*intro*]: $\exists \sigma. \textit{is-idem } \sigma$
using *is-idem-id-subst*
by *blast*

end

locale *abstract-substitution* =
 abstract-substitution-monoid comp-subst id-subst +
 comp-subst: right-monoid-action comp-subst id-subst subst +
 abstract-substitution-ops subst id-subst comp-subst is-ground
for
 comp-subst :: 's \Rightarrow 's \Rightarrow 's (**infixl** $\langle \odot \rangle$ 70) **and**
 id-subst :: 's **and**
 subst :: 'x \Rightarrow 's \Rightarrow 'x (**infixl** $\langle \cdot \rangle$ 69) **and**

— Predicate identifying the fixed elements w.r.t. the monoid action
is-ground :: 'x \Rightarrow *bool* +

assumes

all-subst-ident-if-ground: is-ground x \implies ($\forall \sigma. x \cdot \sigma = x$)

begin

sublocale *comp-subst-set: right-monoid-action comp-subst id-subst subst-set*
using *comp-subst.lifting-monoid-action-to-set* **unfolding** *subst-set-def* .

sublocale *comp-subst-list: right-monoid-action comp-subst id-subst subst-list*
using *comp-subst.lifting-monoid-action-to-list* **unfolding** *subst-list-def* .

7.1 Substitution Composition

lemmas *subst-comp-subst = comp-subst.action-compatibility*

lemmas *subst-set-comp-subst = comp-subst-set.action-compatibility*

lemmas *subst-list-comp-subst = comp-subst-list.action-compatibility*

7.2 Substitution Identity

lemmas *subst-id-subst = comp-subst.action-neutral*

lemmas *subst-set-id-subst = comp-subst-set.action-neutral*

lemmas *subst-list-id-subst = comp-subst-list.action-neutral*

lemma *is-mgu-id-subst-empty[simp]: is-mgu id-subst {}*
by (*simp add: is-mgu-def*)

lemma *is-imgu-id-subst-empty[simp]: is-imgu id-subst {}*
by (*simp add: is-imgu-def*)

lemma *is-unifier-id-subst: is-unifier id-subst X \longleftrightarrow X = {} \vee card X = 1*
unfolding *is-unifier-def*
by *auto*

lemma *is-unifier-set-id-subst: is-unifier-set id-subst XX \longleftrightarrow ($\forall X \in XX. X = \{\}$ \vee card X = 1)*
by (*simp add: is-unifier-set-def is-unifier-id-subst*)

lemma *is-mgu-id-subst: is-mgu id-subst XX \longleftrightarrow ($\forall X \in XX. X = \{\}$ \vee card X = 1)*
by (*simp add: is-mgu-def is-unifier-set-id-subst*)

lemma *is-imgu-id-subst: is-imgu id-subst XX \longleftrightarrow ($\forall X \in XX. X = \{\}$ \vee card X = 1)*
by (*simp add: is-imgu-def is-unifier-set-id-subst*)

7.3 Generalization

sublocale *generalizes: preorder generalizes strictly-generalizes*

proof *unfold-locales*

show $\bigwedge x y. \text{strictly-generalizes } x y = (\text{generalizes } x y \wedge \neg \text{generalizes } y x)$

unfolding *strictly-generalizes-def generalizes-def* **by** *blast*

next

show $\bigwedge x. \text{generalizes } x x$

unfolding *generalizes-def* **using** *subst-id-subst* **by** *metis*
next
show $\bigwedge x y z. \text{generalizes } x y \implies \text{generalizes } y z \implies \text{generalizes } x z$
unfolding *generalizes-def* **using** *subst-comp-subst* **by** *metis*
qed

lemma *generalizes-antisym-if*:
assumes $\bigwedge \sigma_1 \sigma_2 x. x \cdot (\sigma_1 \odot \sigma_2) = x \implies x \cdot \sigma_1 = x$
shows $\bigwedge x y. \text{generalizes } x y \implies \text{generalizes } y x \implies x = y$
using *assms*
by (*metis generalizes-def subst-comp-subst*)

lemma *order-generalizes-if*:
assumes $\bigwedge \sigma_1 \sigma_2 x. x \cdot (\sigma_1 \odot \sigma_2) = x \implies x \cdot \sigma_1 = x$
shows *class.order generalizes strictly-generalizes*
proof *unfold-locales*
show $\bigwedge x y. \text{generalizes } x y \implies \text{generalizes } y x \implies x = y$
using *generalizes-antisym-if assms* **by** *iprover*
qed

7.4 Substituting on Ground Expressions

lemma *subst-ident-if-ground[simp]*: *is-ground* $x \implies x \cdot \sigma = x$
using *all-subst-ident-if-ground* **by** *simp*

lemma *subst-set-ident-if-ground[simp]*: *is-ground-set* $X \implies \text{subst-set } X \sigma = X$
unfolding *is-ground-set-def subst-set-def* **by** *simp*

7.5 Instances of Ground Expressions

lemma *instances-ident-if-ground[simp]*: *is-ground* $x \implies \text{instances } x = \{x\}$
unfolding *instances-def generalizes-def* **by** *simp*

lemma *instances-set-ident-if-ground[simp]*: *is-ground-set* $X \implies \text{instances-set } X = X$
unfolding *instances-set-def is-ground-set-def* **by** *simp*

lemma *ground-instances-ident-if-ground[simp]*: *is-ground* $x \implies \text{ground-instances } x = \{x\}$
unfolding *ground-instances-def* **by** *auto*

lemma *ground-instances-set-ident-if-ground[simp]*: *is-ground-set* $X \implies \text{ground-instances-set } X = X$
unfolding *is-ground-set-def ground-instances-set-eq-Union-ground-instances* **by** *simp*

7.6 Unifier of Ground Expressions

lemma *ground-eq-ground-if-unifiable*:
assumes *is-unifier* $v \{t_1, t_2\}$ **and** *is-ground* t_1 **and** *is-ground* t_2

shows $t_1 = t_2$
using *assms*
by (*simp add: card-Suc-eq is-unifier-def le-Suc-eq subst-set-def*)

corollary *ground-eq-ground-if-mgu*:
assumes *is-mgu* μ $\{\{t_1, t_2\}\}$ *is-ground* t_1 *is-ground* t_2
shows $t_1 = t_2$
using *assms ground-eq-ground-if-unifiable*
unfolding *is-mgu-def is-unifier-set-insert*
by *blast*

corollary *ground-eq-ground-if-imgu*:
assumes *is-imgu* μ $\{\{t_1, t_2\}\}$ *is-ground* t_1 *is-ground* t_2
shows $t_1 = t_2$
using *assms ground-eq-ground-if-unifiable*
unfolding *is-imgu-def is-unifier-set-insert*
by *blast*

lemma *ball-eq-constant-if-unifier*:
assumes $x \in X$ **and** *is-unifier* v X **and** *is-ground-set* X
shows $\forall y \in X. y = x$
using *assms*
by (*simp add: is-ground-set-def is-unifier-iff*)

lemma *is-mgu-unifies*:
assumes *is-mgu* μ XX
shows $\forall X \in XX. \forall t \in X. \forall t' \in X. t \cdot \mu = t' \cdot \mu$
using *assms is-unifier-iff*
unfolding *is-mgu-def is-unifier-set-def*
by *blast*

corollary *is-mgu-unifies-pair*:
assumes *is-mgu* μ $\{\{t, t'\}\}$
shows $t \cdot \mu = t' \cdot \mu$
using *is-mgu-unifies[OF assms]*
by (*meson insert-iff*)

lemmas *subst-mgu-eq-subst-mgu = is-mgu-unifies-pair*

lemma *is-imgu-unifies*:
assumes *is-imgu* μ XX
shows $\forall X \in XX. \forall t \in X. \forall t' \in X. t \cdot \mu = t' \cdot \mu$
using *assms is-unifier-iff*
unfolding *is-imgu-def is-unifier-set-def*
by *blast*

corollary *is-imgu-unifies-pair*:
assumes *is-imgu* μ $\{\{t, t'\}\}$
shows $t \cdot \mu = t' \cdot \mu$

using *is-imgu-unifies*[*OF assms*]
 by (*metis insertCI*)

lemmas *subst-imgu-eq-subst-imgu = is-imgu-unifies-pair*

7.7 Ground Substitutions

lemma *is-ground-subst-comp-left*: *is-ground-subst* $\sigma \implies$ *is-ground-subst* $(\sigma \odot \tau)$
 by (*simp add: is-ground-subst-def*)

lemma *is-ground-subst-comp-right*: *is-ground-subst* $\tau \implies$ *is-ground-subst* $(\sigma \odot \tau)$
 by (*simp add: is-ground-subst-def*)

lemma *is-ground-subst-is-ground*:
 assumes *is-ground-subst* γ
 shows *is-ground* $(t \cdot \gamma)$
 using *assms is-ground-subst-def* by *blast*

7.8 IMGU is Idempotent and an MGU

lemma *is-imgu-iff-is-idem-and-is-mgu*: *is-imgu* μ *XX* \longleftrightarrow *is-idem* $\mu \wedge$ *is-mgu* μ *XX*
 by (*auto simp add: is-imgu-def is-idem-def is-mgu-def simp flip: assoc*)

7.9 IMGU can be used before unification

lemma *subst-imgu-subst-unifier*:
 assumes *unif*: *is-unifier* v *X* and *imgu*: *is-imgu* μ $\{X\}$ and $x \in X$
 shows $x \cdot \mu \cdot v = x \cdot v$

proof –

have $x \cdot \mu \cdot v = x \cdot (\mu \odot v)$
 by *simp*

also have $\dots = x \cdot v$
 using *imgu unif* by (*simp add: is-imgu-def is-unifier-set-def*)

finally show *?thesis* .

qed

7.10 Groundings Idempotence

lemma *image-ground-instances-ground-instances*:
ground-instances ‘ *ground-instances* $x = (\lambda x. \{x\})$ ‘ *ground-instances* x

proof (*rule image-cong*)

show $\bigwedge x_G. x_G \in$ *ground-instances* $x \implies$ *ground-instances* $x_G = \{x_G\}$
 using *ground-instances-ident-if-ground* *ground-instances-def* by *auto*

qed *simp*

lemma *grounding-of-set-grounding-of-set-idem*[*simp*]:
ground-instances-set (*ground-instances-set* X) = *ground-instances-set* X

unfolding *ground-instances-set-eq-Union-ground-instances UN-UN-flatten*
unfolding *image-ground-instances-ground-instances*
by *simp*

7.11 Instances of Substitution

lemma *instances-subst*:
 $instances (x \cdot \sigma) \subseteq instances x$
proof (*rule subsetI*)
fix x_σ **assume** $x_\sigma \in instances (x \cdot \sigma)$
thus $x_\sigma \in instances x$
by (*metis CollectD CollectI generalizes-def instances-def subst-comp-subst*)
qed

lemma *instances-set-subst-set*:
 $instances-set (subst-set X \sigma) \subseteq instances-set X$
unfolding *instances-set-def subst-set-def*
using *instances-subst* **by** *auto*

lemma *ground-instances-subst*:
 $ground-instances (x \cdot \sigma) \subseteq ground-instances x$
unfolding *ground-instances-def*
using *instances-subst* **by** *auto*

lemma *ground-instances-set-subst-set*:
 $ground-instances-set (subst-set X \sigma) \subseteq ground-instances-set X$
unfolding *ground-instances-set-def*
using *instances-set-subst-set* **by** *auto*

7.12 Instances of Renamed Expressions

lemma *instances-subst-ident-if-renaming[simp]*:
 $is-renaming \varrho \implies instances (x \cdot \varrho) = instances x$
by (*metis instances-subst is-renaming-def subset-antisym subst-comp-subst subst-id-subst*)

lemma *instances-set-subst-set-ident-if-renaming[simp]*:
 $is-renaming \varrho \implies instances-set (subst-set X \varrho) = instances-set X$
by (*simp add: instances-set-def subst-set-def*)

lemma *ground-instances-subst-ident-if-renaming[simp]*:
 $is-renaming \varrho \implies ground-instances (x \cdot \varrho) = ground-instances x$
by (*simp add: ground-instances-def*)

lemma *ground-instances-set-subst-set-ident-if-renaming[simp]*:
 $is-renaming \varrho \implies ground-instances-set (subst-set X \varrho) = ground-instances-set X$
by (*simp add: ground-instances-set-def*)

end

end

```

theory Option-Extra
  imports Main
begin

abbreviation get-or :: 'a option  $\Rightarrow$  'a  $\Rightarrow$  'a where
   $\bigwedge a$  default. get-or a default  $\equiv$  case a of None  $\Rightarrow$  default | Some a  $\Rightarrow$  a

end
theory List-Extra
  imports Main
begin

definition set-list :: 'a set  $\Rightarrow$  'a list where
  set-list S  $\equiv$  SOME xs. set xs = S  $\wedge$  distinct xs

lemma set-list-empty [simp]: set-list {} = []
  unfolding set-list-def
  by auto

lemma set-list-singleton [simp]: set-list {x} = [x]
proof (unfold set-list-def, rule some1-equality, intro ex-ex1I)

  show  $\bigwedge xs y.$   $\llbracket$ set xs = {x}  $\wedge$  distinct xs; set y = {x}  $\wedge$  distinct y $\rrbracket \Longrightarrow xs = y$ 
  by (metis distinct-card replicate-eqI singleton-iff)
qed (auto simp: finite-distinct-list)

lemma set-list-set [simp]:
  assumes finite S
  shows set (set-list S) = S
  unfolding set-list-def
  by (rule someI2-ex[OF finite-distinct-list[OF assms]]) argo

lemma set-list-distinct [simp]:
  assumes finite S
  shows distinct (set-list S)
  unfolding set-list-def
  by (rule someI2-ex[OF finite-distinct-list[OF assms]]) argo

end
theory Finite-Map-Extra
  imports
    HOL-Library.Finite-Map
    List-Extra
begin

definition fmap-dom-list :: ('k, 'v) fmap  $\Rightarrow$  'k list where
  fmap-dom-list m  $\equiv$  SOME xs. set xs = fmdom' m  $\wedge$  distinct xs

lemma fmap-dom-list-exists [intro]:  $\exists xs.$  set xs = fmdom' m  $\wedge$  distinct xs

```

by (induction m) (auto simp: finite-distinct-list)

lemma set-fmap-dom-list [simp]: set (fmap-dom-list m) = fmdom' m
unfolding fmap-dom-list-def
by (rule someI2-ex[OF fmap-dom-list-exists]) argo

lemma distinct-fmap-dom-list [simp]: distinct (fmap-dom-list m)
unfolding fmap-dom-list-def
by (rule someI2-ex[OF fmap-dom-list-exists]) argo

lemma fmap-dom-list-empty [simp]: fmap-dom-list fmempty = []
by (metis set-fmap-dom-list fmdom'-empty set-empty)

definition fmap-list :: ('k, 'v) fmap \Rightarrow ('k \times 'v) list **where**
fmap-list m \equiv map ($\lambda k. (k, \text{the } (\text{fmlookup } m \ k))$) (fmap-dom-list m)

lemma fmap-list-empty [simp]: fmap-list fmempty = []
unfolding fmap-list-def
by simp

lemma set-fst-fmap-list [simp]: set (map fst (fmap-list m)) = fmdom' m
unfolding fmap-list-def
by simp

lemma distinct-fst-fmap-list [simp]: distinct (map fst (fmap-list m))
unfolding fmap-list-def
by (simp add: map-idI)

lemma fmap-list-mem-iff: $(k, v) \in \text{set } (\text{fmap-list } m) \iff \text{fmlookup } m \ k = \text{Some } v$
proof (rule iffI)
assume $(k, v) \in \text{set } (\text{fmap-list } m)$

then show fmlookup m k = Some v
unfolding fmap-list-def
by (metis dom-fmlookup graph-eq-to-snd-dom in-graphD list.set-map set-fmap-dom-list)

next
assume fmlookup m k = Some v

then show $(k, v) \in \text{set } (\text{fmap-list } m)$
unfolding fmap-list-def
using set-fmap-dom-list distinct-fmap-dom-list
by fastforce

qed

definition fmap-of-set :: 'k set \Rightarrow ('k \Rightarrow 'v) \Rightarrow ('k, 'v) fmap **where**
fmap-of-set X f \equiv fold ($\lambda x \ m. \text{fmupd } x \ (f \ x) \ m$) (set-list X) fmempty

lemma fmap-of-set-empty [simp]: fmap-of-set {} f = fmempty
unfolding fmap-of-set-def

```

    by simp

lemma fmllookup-fold-fmupd-notin [simp]:
  assumes  $x \notin \text{set } xs$ 
  shows  $\text{fmllookup } (\text{fold } (\lambda k m. \text{fmupd } k (f k) m) xs m) x = \text{fmllookup } m x$ 
  using assms
  by (induction xs arbitrary: m) auto

lemma fmllookup-fold-fmupd-in [simp]:
  assumes  $\text{distinct } xs$   $x \in \text{set } xs$ 
  shows  $\text{fmllookup } (\text{fold } (\lambda k m. \text{fmupd } k (f k) m) xs m) x = \text{Some } (f x)$ 
  using assms
  by (induction xs arbitrary: m) auto

lemma fmllookup-fmap-of-set-notin [simp]:
  assumes  $\text{finite } X$   $x \notin X$ 
  shows  $\text{fmllookup } (\text{fmap-of-set } X f) x = \text{None}$ 
  using assms
  unfolding fmap-of-set-def
  by auto

lemma fmllookup-fmap-of-set-in [simp]:
  assumes  $\text{finite } X$  and  $x \in X$ 
  shows  $\text{fmllookup } (\text{fmap-of-set } X f) x = \text{Some } (f x)$ 
  using assms
  unfolding fmap-of-set-def
  by auto

end
theory Fun-Extra
  imports Main
begin

definition bij-partition where
  bij-partition  $S T \equiv \text{SOME } h. \text{bij-betw } h (T - S) (S - T)$ 

lemma bij-partition:
  assumes  $\text{finite } S$   $\text{finite } T$   $\text{card } S = \text{card } T$ 
  shows  $\text{bij-betw } (\text{bij-partition } S T) (T - S) (S - T)$ 
  using assms
  unfolding bij-partition-def
  by (metis add-diff-cancel-left' card-add-diff-finite finite-Diff finite-same-card-bij someI)

end
theory Substitution
  imports
    Abstract-Substitution
    Option-Extra

```

Finite-Map-Extra
Fun-Extra

begin

8 Substitutions on variables

locale *substitution* = *abstract-substitution* **where**
subst = *subst*

for

subst :: 'expr \Rightarrow 'subst \Rightarrow 'expr (**infixl** · 69) **and**
apply-subst :: 'v \Rightarrow 'subst \Rightarrow 'base (**infixl** · v 69) **and**
vars :: 'expr \Rightarrow 'v set +

assumes *no-vars-if-is-ground* [*intro*]: $\bigwedge expr. is_ground\ expr \Longrightarrow vars\ expr = \{\}$
begin

abbreviation *exists-nonground* **where**

exists-nonground $\equiv \exists expr. \neg is_ground\ expr$

definition *vars-set* :: 'expr set \Rightarrow 'v set **where**

vars-set *exprs* $\equiv \bigcup expr \in exprs. vars\ expr$

lemma *subst-cannot-unground*:

assumes $\neg is_ground\ (expr \cdot \sigma)$

shows $\neg is_ground\ expr$

using *assms*

by *force*

abbreviation (*input*) *var-subst* **where**

var-subst $\sigma\ x \equiv x \cdot v\ \sigma$

abbreviation (*input*) *expr-subst* **where**

expr-subst $\sigma\ expr \equiv expr \cdot \sigma$

definition *subst-domain* :: 'subst \Rightarrow 'v set **where**

subst-domain $\sigma = \{x. x \cdot v\ \sigma \neq x \cdot v\ id_subst\}$

abbreviation *subst-range* :: 'subst \Rightarrow 'base set **where**

subst-range $\sigma \equiv var_subst\ \sigma\ `subst_domain\ \sigma$

lemma *subst-inv*:

assumes $\sigma \odot \sigma_inv = id_subst$

shows $expr \cdot \sigma \cdot \sigma_inv = expr$

using *assms*

by (*metis subst-comp-subst subst-id-subst*)

definition *rename* **where**

is-renaming $\rho \Longrightarrow rename\ \rho\ x \equiv SOME\ x'. x \cdot v\ \rho = x' \cdot v\ id_subst$

lemma *is-unifier-two-iff* [*simp*]: $is_unifier\ v\ \{expr, expr'\} \longleftrightarrow expr \cdot v = expr' \cdot$

```

v
  unfolding is-unifier-def
  using insertCI
  by fastforce

lemma is-unifier-set-two-iff [simp]: is-unifier-set v {{expr, expr'}}  $\longleftrightarrow$  expr · v
= expr' · v
  unfolding is-unifier-set-def
  by simp

lemma obtain-imgu-absorption:
  assumes is-unifier-set v XX is-imgu  $\mu$  XX
  obtains  $\sigma$  where v =  $\mu \odot \sigma$ 
  using assms
  unfolding is-imgu-def
  by metis

end

```

8.1 Properties of substitutions

```

locale subst-update-def =
  fixes subst-update :: 'subst  $\Rightarrow$  'v  $\Rightarrow$  'base  $\Rightarrow$  'subst
begin

definition subst-updates :: 'subst  $\Rightarrow$  ('v, 'base) fmap  $\Rightarrow$  'subst ( $\langle$ - $\rangle$ ) [1000, 0] 71)
where
  subst-updates  $\sigma$  m = fold ( $\lambda(x, b)$   $\sigma$ . subst-update  $\sigma$  x b) (fmap-list m)  $\sigma$ 

end

locale subst-update =
  substitution where vars = vars and apply-subst = apply-subst +
  subst-update-def where subst-update = subst-update
for
  vars :: 'expr  $\Rightarrow$  'v set and
  apply-subst :: 'v  $\Rightarrow$  'subst  $\Rightarrow$  'base (infixl ·v 69) and
  subst-update :: 'subst  $\Rightarrow$  'v  $\Rightarrow$  'base  $\Rightarrow$  'subst ( $\langle$ - $\rangle$  := -) [1000, 0, 50] 71) +
  assumes
    subst-update-var [simp]:
      — The precondition of the assumption ensures noop substitutions
       $\bigwedge x u \sigma$ . exists-nonground  $\implies$  x ·v  $\sigma[x := u]$  = u
       $\bigwedge x y u \sigma$ . x  $\neq$  y  $\implies$  x ·v  $\sigma[y := u]$  = x ·v  $\sigma$  and
    subst-update [simp]:
       $\bigwedge x$  expr u  $\sigma$ . x  $\notin$  vars expr  $\implies$  expr ·  $\sigma[x := u]$  = expr ·  $\sigma$ 
       $\bigwedge \sigma$  x.  $\sigma[x := x \cdot v \sigma]$  =  $\sigma$  and
    subst-update-twice [simp]:
       $\bigwedge \sigma$  x a b. ( $\sigma[x := a]$ )[x := b] =  $\sigma[x := b]$ 
begin

```

```

lemma subst-updates-empty [simp]:  $\sigma[\![fmempty]\!] = \sigma$ 
  unfolding subst-updates-def
  by auto

lemma fold-redundant-updates-var [simp]:
  assumes  $x \notin \text{set } (\text{map } \text{fst } us)$ 
  shows  $x \cdot v \text{ fold } (\lambda(x, b) \sigma. \sigma[x := b]) us \sigma = x \cdot v \sigma$ 
  using assms
  by (induction us arbitrary: \sigma) (simp-all add: split-beta)

lemma fold-updates-var [simp]:
  assumes
    exists-nonground: exists-nonground and
    distinct-updates: distinct (map fst us) and
    update-in-updates: (x, b) \in set us
  shows  $x \cdot v \text{ fold } (\lambda(x, b) \sigma. \sigma[x := b]) us \sigma = b$ 
  using distinct-updates update-in-updates
  proof (induction us arbitrary: \sigma)
    case Nil

    then show ?case
      by simp
  next
    case (Cons u us)

    then show ?case
      using exists-nonground fold-redundant-updates-var
      by (cases u = (x, b)) auto
  qed

lemma fold-redundant-updates [simp]:
  assumes  $\bigwedge x b. (x, b) \in \text{set } us \implies x \notin \text{vars } \text{expr} \vee b = x \cdot v \sigma \text{ distinct } (\text{map } \text{fst } us)$ 
  shows  $\text{expr} \cdot \text{fold } (\lambda(x, b) \sigma. \sigma[x := b]) us \sigma = \text{expr} \cdot \sigma$ 
  using assms
  proof (induction us arbitrary: \sigma)
    case Nil

    then show ?case
      by simp
  next
    case (Cons u us)

    then show ?case
      by (smt (verit, best) distinct.simps(2) fold.simps(2) list.set-intros(1) list.simps(9))

    prod.collapse set-subset-Cons split-beta subsetD subst-update subst-update-var(2)
  qed

```

lemma *subst-updates-var*:

assumes *exists-nonground*: *exists-nonground*

shows $x \cdot v \sigma[u] = \text{get-or } (\text{fmlookup } u \ x) \ (x \cdot v \ \sigma)$

proof (*cases fmlookup u x*)

case *None*

have $x \cdot v \ \text{fold } (\lambda(x, b) \ \sigma. \ \sigma[x := b]) \ (\text{fmap-list } u) \ \sigma = x \cdot v \ \sigma$

proof (*rule fold-redundant-updates-var*)

show $x \notin \text{set } (\text{map } \text{fst } (\text{fmap-list } u))$

unfolding *set-fst-fmap-list*

by (*simp add: None fmdom'-notI*)

qed

then show *?thesis*

unfolding *None subst-updates-def*

by *simp*

next

case (*Some b*)

then show *?thesis*

unfolding *subst-updates-def*

by (*simp add: exists-nonground fmap-list-mem-iff*)

qed

lemma *redundant-subst-updates [simp]*:

assumes $\bigwedge x. x \in \text{vars } \text{expr} \implies \text{fmlookup } u \ x = \text{None} \vee \text{fmlookup } u \ x = \text{Some } (x \cdot v \ \sigma)$

shows $\text{expr} \cdot \sigma[u] = \text{expr} \cdot \sigma$

proof (*unfold subst-updates-def, rule fold-redundant-updates*)

fix $x \ b$

assume $(x, b) \in \text{set } (\text{fmap-list } u)$

then show $x \notin \text{vars } \text{expr} \vee b = x \cdot v \ \sigma$

unfolding *fmap-list-mem-iff*

using *assms*

by *fastforce*

next

show *distinct (map fst (fmap-list u))*

by *simp*

qed

lemma *redundant-subst-updates-vars-set [simp]*:

assumes *exists-nonground* $\bigwedge x. x \in X \implies \text{fmlookup } u \ x = \text{None} \vee \text{fmlookup } u \ x = \text{Some } (x \cdot v \ \sigma)$

shows $(\lambda x. x \cdot v \ \sigma[u]) \ 'X = (\lambda x. x \cdot v \ \sigma) \ 'X$

using *assms(2) subst-updates-var[OF assms(1)]*

by force

lemma *redundant-subst-updates-vars-image* [simp]:

assumes $\bigwedge x. x \in \bigcup (\text{vars } 'X) \implies \text{fmlookup } u \ x = \text{None} \vee \text{fmlookup } u \ x = \text{Some } (x \cdot v \ \sigma)$

shows $(\lambda \text{expr}. \text{expr} \cdot \sigma \llbracket u \rrbracket) 'X = (\lambda \text{expr}. \text{expr} \cdot \sigma) 'X$

using *assms redundant-subst-updates*

by (*meson UN-I image-cong*)

lemma *subst-updates-fmap-of-set* [simp]:

assumes *exists-nonground* $x \in X$ *finite* X

shows $x \cdot v \ \sigma \llbracket \text{fmap-of-set } X \ f \rrbracket = f \ x$

using *assms subst-updates-var*

by *simp*

lemma *redundant-subst-updates-fmap-of-set* [simp]:

assumes *exists-nonground* $x \notin X$ *finite* X

shows $x \cdot v \ \sigma \llbracket \text{fmap-of-set } X \ f \rrbracket = x \cdot v \ \sigma$

using *assms subst-updates-var*

by *simp*

definition *renaming-of-bij* **where**

renaming-of-bij $f \ S \ T \equiv$

$\text{id-subst} \llbracket \text{fmap-of-set } (S \cup T) \ (\lambda x. (\text{if } x \in S \text{ then } f \ x \text{ else } \text{bij-partition } S \ T \ x)) \cdot v \ \text{id-subst} \rrbracket$

definition *renaming-of-bij-inv* **where**

renaming-of-bij-inv $f \ S \ T \equiv$

$\text{id-subst} \llbracket \text{fmap-of-set } (S \cup T) \$

$(\lambda x. (\text{if } x \in T \text{ then } \text{inv-into } S \ f \ x \text{ else } \text{inv-into } (T - S) \ (\text{bij-partition } S \ T) \ x)) \cdot v \ \text{id-subst} \rrbracket$

lemma *redundant-renaming-of-bij*:

assumes *exists-nonground* *finite* S *bij-betw* $f \ S \ T$ $x \notin S \cup T$

shows $x \cdot v \ \text{renaming-of-bij } f \ S \ T = x \cdot v \ \text{id-subst}$

unfolding *renaming-of-bij-def*

using *assms*

by (*simp add: bij-betw-finite*)

lemma *renaming-of-bij-on-S*:

assumes *exists-nonground* *finite* S *bij-betw* $f \ S \ T$ $x \in S$

shows $x \cdot v \ \text{renaming-of-bij } f \ S \ T = f \ x \cdot v \ \text{id-subst}$

unfolding *renaming-of-bij-def*

using *assms*

by (*simp add: bij-betw-finite*)

end

locale *all-subst-ident-iff-ground* =

substitution +
assumes *all-subst-ident-iff-ground*: $\bigwedge expr. is_ground\ expr \longleftrightarrow (\forall \sigma. expr \cdot \sigma = expr)$

locale *exists-non-ident-subst* =
substitution **where** *vars* = *vars*
for *vars* :: 'expr \Rightarrow 'v set +
assumes
exists-non-ident-subst:
 $\bigwedge expr\ S. finite\ S \Longrightarrow \neg is_ground\ expr \Longrightarrow \exists \sigma. expr \cdot \sigma \neq expr \wedge expr \cdot \sigma \notin S$
begin

lemma *infinite-exprs*:
assumes *exists-nonground*
shows *infinite* (*UNIV* :: 'expr set)
using *assms exists-non-ident-subst*
by *auto*

end

locale *finite-variables* = *substitution* **where** *vars* = *vars*
for *vars* :: 'expr \Rightarrow 'v set +
assumes *finite-vars* [*intro*]: $\bigwedge expr. finite\ (vars\ expr)$
begin

abbreviation *finite-vars* :: 'expr \Rightarrow 'v fset **where**
finite-vars *expr* $\equiv Abs_fset\ (vars\ expr)$

lemma *fset-finite-vars* [*simp*]: *fset* (*finite-vars* *expr*) = *vars* *expr*
using *Abs-fset-inverse* *finite-vars*
by *blast*

end

locale *infinite-variables* = *substitution* **where** *vars* = *vars*
for *vars* :: 'expr \Rightarrow 'v set +
assumes *infinite-vars* [*intro*]: *infinite* (*UNIV* :: 'v set)

locale *renaming-variables* = *substitution* +
assumes
— The precondition of the assumption ensures noop substitutions
is-renaming-imp:
 $\bigwedge \varrho. exists_nonground \Longrightarrow$
 $is_renaming\ \varrho \Longrightarrow inj\ (var_subst\ \varrho) \wedge (\forall x. \exists x'. x \cdot v\ \varrho = x' \cdot v\ id_subst)$
and
rename-variables: $\bigwedge expr\ \varrho. is_renaming\ \varrho \Longrightarrow vars\ (expr \cdot \varrho) = rename\ \varrho\ (vars\ expr)$

begin

lemma *renaming-range-id-subst*:

assumes *exists-nonground*: *exists-nonground* **and** ϱ : *is-renaming* ϱ
shows $x \cdot v \varrho \in \text{range } (\text{var-subst } \text{id-subst})$
using *is-renaming-imp*[*OF exists-nonground* ϱ]
by *auto*

lemma *obtain-renamed-variable*:

assumes *exists-nonground* *is-renaming* ϱ
obtains x' **where** $x \cdot v \varrho = x' \cdot v \text{id-subst}$
using *renaming-range-id-subst*[*OF assms*]
by *auto*

lemma *id-subst-rewrite* [*simp*]:

assumes *exists-nonground* **and** ϱ : *is-renaming* ϱ
shows $\text{rename } \varrho \ x \cdot v \text{id-subst} = x \cdot v \varrho$
unfolding *rename-def*[*OF* ϱ]
using *obtain-renamed-variable*[*OF assms*]
by (*metis* (*mono-tags*, *lifting*) *someI*)

lemma *rename-variables-id-subst*:

assumes *is-renaming* ϱ
shows $\text{var-subst } \text{id-subst } \text{'vars } (\text{expr } \cdot \varrho) = \text{var-subst } \varrho \text{' } (\text{vars } \text{expr})$
using *rename-variables*[*OF assms*] *id-subst-rewrite*[*OF - assms*]
by (*smt* (*verit*, *best*) *empty-is-image* *image-cong* *image-image* *no-vars-if-is-ground*)

lemma *surj-inv-renaming*:

assumes *exists-nonground*: *exists-nonground* **and** ϱ : *is-renaming* ϱ
shows *surj* ($\lambda x. \text{inv } (\text{var-subst } \varrho) (x \cdot v \text{id-subst})$)
using *inv-f-f* *is-renaming-imp*[*OF exists-nonground* ϱ]
unfolding *surj-def*
by *metis*

lemma *renaming-range*:

assumes ϱ : *is-renaming* ϱ **and** x : $x \in \text{vars } (\text{expr } \cdot \varrho)$
shows $x \cdot v \text{id-subst} \in \text{range } (\text{var-subst } \varrho)$
using *rename-variables-id-subst*[*OF* ϱ] x
by *fastforce*

lemma *renaming-inv-into*:

assumes *is-renaming* ϱ $x \in \text{vars } (\text{expr } \cdot \varrho)$
shows $\text{inv } (\text{var-subst } \varrho) (x \cdot v \text{id-subst}) \cdot v \varrho = x \cdot v \text{id-subst}$
using *f-inv-into-f*[*OF renaming-range*[*OF assms*]].

lemma *inv-renaming*:

assumes *exists-nonground*: *exists-nonground* **and** ϱ : *is-renaming* ϱ
shows $\text{inv } (\text{var-subst } \varrho) (x \cdot v \varrho) = x$
using *is-renaming-imp*[*OF exists-nonground* ϱ]

by (*simp add: inv-into-f-eq*)

lemma *renaming-inv-in-vars*:
assumes ϱ : *is-renaming* ϱ **and** $x: x \in \text{vars } (\text{expr} \cdot \varrho)$
shows $\text{inv } (\text{var-subst } \varrho) (x \cdot v \text{ id-subst}) \in \text{vars expr}$
using *assms rename-variables-id-subst*[*OF* ϱ]
by (*metis no-vars-if-is-ground imageI image-inv-f-f insert-not-empty is-renaming-imp mk-disjoint-insert*)

lemma *inj-id-subst*:
assumes *exists-nonground*
shows *inj* (*var-subst id-subst*)
using *is-renaming-id-subst is-renaming-imp*[*OF* *assms*]
by *blast*

end

locale *grounding = substitution where vars = vars and id-subst = id-subst*
for $\text{vars} :: 'expr \Rightarrow 'var \text{ set}$ **and** $\text{id-subst} :: 'subst +$
fixes $\text{to-ground} :: 'expr \Rightarrow 'expr_G$ **and** $\text{from-ground} :: 'expr_G \Rightarrow 'expr$
assumes
 $\text{range-from-ground-iff-is-ground}: \{ \text{expr}. \text{is-ground expr} \} = \text{range from-ground}$
and
 $\text{from-ground-inverse [simp]}: \bigwedge \text{expr}_G. \text{to-ground } (\text{from-ground } \text{expr}_G) = \text{expr}_G$
begin

definition *ground-instances'* $:: 'expr \Rightarrow 'expr_G \text{ set}$ **where**
 $\text{ground-instances'} \text{ expr} = \{ \text{to-ground } (\text{expr} \cdot \gamma) \mid \gamma. \text{is-ground } (\text{expr} \cdot \gamma) \}$

lemma *ground-instances'-eq-ground-instances*:
 $\text{ground-instances'} \text{ expr} = (\text{to-ground } \text{'ground-instances expr})$
unfolding *ground-instances'-def ground-instances-def generalizes-def instances-def*

by *blast*

lemma *to-ground-from-ground-id* [*simp*]: $\text{to-ground} \circ \text{from-ground} = \text{id}$
using *from-ground-inverse*
by *auto*

lemma *surj-to-ground*: *surj to-ground*
using *from-ground-inverse*
by (*metis surj-def*)

lemma *inj-from-ground*: *inj-on from-ground domain_G*
by (*metis from-ground-inverse inj-on-inverseI*)

lemma *inj-on-to-ground*: *inj-on to-ground (from-ground ' domain_G)*
unfolding *inj-on-def*
by *simp*

lemma *bij-betw-to-ground*: *bij-betw to-ground (from-ground ‘ domain_G) domain_G*
by (*smt (verit, best) bij-betwI' from-ground-inverse image-iff*)

lemma *bij-betw-from-ground*: *bij-betw from-ground domain_G (from-ground ‘ domain_G)*
by (*simp add: bij-betw-def inj-from-ground*)

lemma *ground-is-ground [simp, intro]*: *is-ground (from-ground expr_G)*
using *range-from-ground-iff-is-ground*
by *blast*

lemma *is-ground-iff-range-from-ground*: *is-ground expr \longleftrightarrow expr \in range from-ground*
using *range-from-ground-iff-is-ground*
by *auto*

lemma *to-ground-inverse [simp]*:
assumes *is-ground expr*
shows *from-ground (to-ground expr) = expr*
using *inj-on-to-ground from-ground-inverse is-ground-iff-range-from-ground assms*
unfolding *inj-on-def*
by *blast*

corollary *obtain-grounding*:
assumes *is-ground expr*
obtains *expr_G where from-ground expr_G = expr*
using *to-ground-inverse assms*
by *blast*

lemma *from-ground-eq [simp]*:
from-ground expr = from-ground expr' \longleftrightarrow expr = expr'
by (*metis from-ground-inverse*)

lemma *to-ground-eq [simp]*:
assumes *is-ground expr is-ground expr'*
shows *to-ground expr = to-ground expr' \longleftrightarrow expr = expr'*
using *assms obtain-grounding*
by *fastforce*

end

locale *exists-ground = substitution +*
assumes *exists-ground: \exists expr. is-ground expr*

locale *exists-ground-subst = substitution +*
assumes *exists-ground-subst: $\exists \gamma. is-ground-subst \gamma$*
begin

lemma *obtain-ground-subst*:

```

obtains  $\gamma$  where is-ground-subst  $\gamma$ 
using exists-ground-subst
by metis

sublocale exists-ground
proof unfold-locales
  fix expr

  obtain  $\gamma$  where  $\gamma$ : is-ground-subst  $\gamma$ 
    using obtain-ground-subst .

  show  $\exists$  expr. is-ground expr
  proof (rule exI)
    show is-ground (expr ·  $\gamma$ )
      using  $\gamma$  is-ground-subst-is-ground
      by blast
    qed
  qed

lemma ground-subst-extension:
  assumes is-ground (expr ·  $\gamma$ )
  obtains  $\gamma'$ 
  where expr ·  $\gamma$  = expr ·  $\gamma'$  and is-ground-subst  $\gamma'$ 
  using obtain-ground-subst assms
  by (metis all-subst-ident-if-ground is-ground-subst-comp-right subst-comp-subst)

end

locale exists-imgu = substitution +
  assumes
     $\bigwedge v$  expr expr'. exists-nonground  $\implies$  expr · v = expr' · v  $\implies$   $\exists \mu$ . is-imgu  $\mu$ 
     $\{\{expr, expr'\}\}$ 
  begin

  lemma exists-imgu:
    assumes expr · v = expr' · v
    shows  $\exists \mu$ . is-imgu  $\mu$   $\{\{expr, expr'\}\}$ 
  proof (cases exists-nonground)
    case True

    then show ?thesis
      by (metis assms exists-imgu-axioms exists-imgu-axioms-def exists-imgu-def)
  next
    case False

    then have expr = expr'
      using assms
      by simp

```

```

then show ?thesis
  by (metis insert-absorb2 is-imgu-id-subst-empty is-imgu-insert-singleton)
qed

lemma obtains-imgu:
  assumes  $expr \cdot v = expr' \cdot v$ 
  obtains  $\mu$  where  $is\_imgu \ \mu \ \{\{expr, expr'\}\}$ 
  using exists-imgu[OF assms]
  by metis

lemma exists-imgu-set:
  assumes
    finite-X: finite X and
    unifier: is-unifier v X
  shows  $\exists \mu. is\_imgu \ \mu \ \{X\}$ 
  using finite-X unifier
proof (cases X)
  case emptyI

    then show ?thesis
      using is-imgu-id-subst
      by blast
  next
    case (insertI X' x)

      then have  $X \neq \{\}$ 
        by simp

      with finite-X show ?thesis
        using unifier
      proof (induction X rule: finite-ne-induct)
      case (singleton x)

        then show ?case
          using is-imgu-id-subst-empty is-imgu-insert-singleton
          by blast
      next
        case (insert expr X')

          then obtain  $\mu$  where  $\mu: is\_imgu \ \mu \ \{X'\}$ 
            by (meson finite-insert is-unifier-subset subset-insertI)

          then have  $card \ (subst\text{-}set \ X' \ \mu) = 1$ 
            by (simp add: is-imgu-def is-unifier-def subst-set-def insert is-unifier-set-insert
le-Suc-eq)

          then obtain  $expr'$  where  $X'\text{-}\mu: subst\text{-}set \ X' \ \mu = \{expr'\}$ 
            using card-1-singletonE

```

by *blast*

then have $expr'$: $\bigwedge expr. expr \in X' \implies expr \cdot \mu = expr'$
 unfolding *subst-set-def*
 by *auto*

have μ -absorbs- τ : $\bigwedge expr. expr \cdot \mu \cdot v = expr \cdot v$
 using μ *insert.premis insert.hyps(1)*
 unfolding *is-imagu-def is-unifier-set-def*
 by (*metis is-unifier-subset comp-subst.left.monoid-action-compatibility singletonD subset-insertI*)

obtain μ' where μ' : *is-imagu* $\mu' \{\{expr \cdot \mu, expr'\}\}$
 proof (*rule obtains-imagu*)

obtain $expr''$ where $expr'' \in X'$
 using *insert.hyps(2)*
 by *auto*

moreover then have $expr'$: $expr' = expr'' \cdot \mu$
 using $expr'$
 by *presburger*

ultimately show $expr \cdot \mu \cdot v = expr' \cdot v$
 using μ -absorbs- τ
 unfolding $expr'$
 by (*metis insert.premis insertCI is-unifier-iff*)

qed

define μ'' where $\mu'' = \mu \odot \mu'$

show *?case*
 proof (*rule exI*)

show *is-imagu* $\mu'' \{\text{insert } expr \ X'\}$
 proof (*unfold is-imagu-def, intro conjI allI impI*)

show *is-unifier-set* $\mu'' \{\text{insert } expr \ X'\}$
 using μ'
 unfolding μ'' -def
 by (*simp add: expr' is-unifier-iff is-unifier-set-insert subst-imagu-eq-subst-imagu*)

next
 fix τ
 assume *is-unifier-set* $\tau \{\text{insert } expr \ X'\}$

moreover then have *is-unifier-set* $\tau \{\{expr \cdot \mu, expr'\}\}$
 using μ
 unfolding *is-imagu-def is-unifier-set-insert*
 by (*metis X'- μ is-unifier-def is-unifier-subset subst-set-insert empty-iff*)

```

insertCI
  subset-insertI subst-set-comp-subst)

  ultimately show  $\mu'' \odot \tau = \tau$ 
  using  $\mu \mu'$ 
  unfolding  $\mu''$ -def is-ingu-def is-unifier-set-insert
  by (metis is-unifier-subset assoc subset-insertI)
qed
qed
qed
qed

lemma exists-ingu-sets:
  assumes
    finite-XX: finite XX and
    finite-X:  $\forall X \in XX. \text{finite } X$  and
    unifier: is-unifier-set  $v \ XX$ 
  shows  $\exists \mu. \text{is-ingu } \mu \ XX$ 
using finite-XX finite-X unifier
proof (induction XX rule: finite-induct)
  case empty

  then show ?case
  by (metis is-ingu-id-subst-empty)
next
  case (insert X XX)

  obtain  $\mu$  where  $\mu: \text{is-ingu } \mu \ XX$ 
  using insert.IH insert.prem1 is-unifier-set-insert
  by force

  define  $X\text{-}\mu$  where  $X\text{-}\mu = \text{subst-set } X \ \mu$ 

  then obtain  $\mu'$  where  $\mu': \text{is-ingu } \mu' \ \{X\text{-}\mu\}$ 
  proof –
  have finite  $X\text{-}\mu$ 
  unfolding  $X\text{-}\mu$ -def subst-set-def
  using insert.prem1(1)
  by simp

  moreover have  $\mu \odot v = v$ 
  using  $\mu$  insert.prem2(2)
  unfolding is-ingu-def is-unifier-set-def
  by blast

  then have is-unifier  $v \ X\text{-}\mu$ 
  using insert.prem2(2)
  unfolding is-unifier-set-def is-unifier-iff  $X\text{-}\mu$ -def subst-set-def
  by (smt (verit, ccfv-threshold) comp-subst.left.monoid-action-compatibility)

```

```

image-iff
  insertCI)

  ultimately show ?thesis
    using that exists-imgu-set
    by blast
qed

define  $\mu''$  where  $\mu'' = \mu \odot \mu'$ 

show ?case
proof (unfold is-imgu-def, intro exI conjI allI impI)

  show is-unifier-set  $\mu''$  (insert X XX)
    using  $\mu \mu'$  insert.prem1
    unfolding  $\mu''$ -def is-imgu-def X- $\mu$ -def is-unifier-iff is-unifier-set-def
    by (metis comp-subst.left.monoid-action-compatibility insert-absorb insert-iff
      subst-set-insert)

  next
  fix  $\tau$ 
  assume is-unifier-set  $\tau$  (insert X XX)

  then show  $\mu'' \odot \tau = \tau$ 
    using  $\mu \mu'$ 
    unfolding  $\mu''$ -def X- $\mu$ -def is-imgu-def is-unifier-set-insert is-unifier-def
    by (metis abstract-substitution-ops.subst-set-empty comp-subst.left.assoc
      is-unifier-set-empty subst-set-comp-subst)
  qed
qed

end

locale subst-updates-compat =
  subst-update +
  assumes subst-updates-compat:
     $\bigwedge \text{expr } \sigma. \forall x \in \text{vars expr}. \text{fmlookup } u \ x = \text{Some } (x \cdot v \ \sigma) \implies \text{expr} \cdot \text{id-subst}[u] = \text{expr} \cdot \sigma$ 

locale subst-eq =
  substitution +
  assumes subst-eq:  $\bigwedge \text{expr } \sigma \ \tau. (\bigwedge x. x \in \text{vars expr} \implies x \cdot v \ \sigma = x \cdot v \ \tau) \implies \text{expr} \cdot \sigma = \text{expr} \cdot \tau$ 
begin

lemma subset-subst-eq:
  assumes  $\forall x \in \text{vars } C. x \cdot v \ \sigma_1 = x \cdot v \ \sigma_2$  vars  $D \subseteq \text{vars } C$ 
  shows  $D \cdot \sigma_1 = D \cdot \sigma_2$ 

```

```

using assms
by (meson subset-iff subst-eq)

end

locale is-ground-if-no-vars = substitution +
  assumes is-ground-if-no-vars:  $\bigwedge expr. vars\ expr = \{\} \implies is-ground\ expr$ 
begin

lemma is-ground-iff-no-vars:  $is-ground\ expr \longleftrightarrow vars\ expr = \{\}$ 
  by (metis is-ground-if-no-vars no-vars-if-is-ground)

end

end
theory Based-Substitution
  imports Substitution
begin

```

9 Substitutions on base expressions

```

locale base-substitution = substitution where vars = vars and apply-subst = ap-
ply-subst
  for vars :: 'base  $\Rightarrow$  'v set and apply-subst :: 'v  $\Rightarrow$  'subst  $\Rightarrow$  'base (infixl <·v> 69)
  +
  assumes
    — The precondition of the assumption ensures noop substitutions
    vars-id-subst:  $\bigwedge x. exists-nonground \implies vars\ (x \cdot v\ id-subst) = \{x\}$  and
    comp-subst-iff:  $\bigwedge \sigma\ \sigma'\ x. x \cdot v\ \sigma \odot \sigma' = subst\ (x \cdot v\ \sigma)\ \sigma'$  and
    base-vars-subst:  $\bigwedge expr\ \sigma. vars\ (expr \cdot \sigma) = \bigcup (vars\ \text{'var-subst}\ \sigma\ \text{'vars}\ expr)$ 
  and
    base-vars-grounded-if-is-grounding:
     $\bigwedge expr\ \gamma. is-ground\ (expr \cdot \gamma) \implies \forall x \in vars\ expr. is-ground\ (x \cdot v\ \gamma)$ 

locale based-substitution =
  substitution where vars = vars and apply-subst = apply-subst :: 'v  $\Rightarrow$  'subst  $\Rightarrow$ 
'base +
  base: base-substitution where
  subst = base-subst and vars = base-vars and is-ground = base-is-ground
for
  base-subst :: 'base  $\Rightarrow$  'subst  $\Rightarrow$  'base and
  base-vars :: 'base  $\Rightarrow$  'v set and
  vars :: 'expr  $\Rightarrow$  'v set and
  base-is-ground +
assumes
  exists-nonground-iff-base-exists-nonground [simp]:  $exists-nonground \longleftrightarrow base.exists-nonground$ 
and
  vars-subst:  $\bigwedge expr\ \varrho. vars\ (expr \cdot \varrho) = \bigcup (base-vars\ \text{'var-subst}\ \varrho\ \text{'vars}\ expr)$ 
and

```

vars-grounded-if-is-grounding:
 $\bigwedge expr \ \gamma. \ is_ground \ (expr \cdot \gamma) \implies \forall x \in vars \ expr. \ base_is_ground \ (x \cdot v \ \gamma)$
begin

lemma *id-subst-subst* [*simp*]: $base_subst \ (x \cdot v \ id_subst) \ \sigma = x \cdot v \ \sigma$
by (*metis base.comp-subst-iff left-neutral*)

lemma *variable-grounding:*
assumes *is-ground* ($expr \cdot \gamma$) $x \in vars \ expr$
shows *base-is-ground* ($x \cdot v \ \gamma$)
using *assms vars-grounded-if-is-grounding*
by *blast*

definition *range-vars* :: $'subst \Rightarrow 'v \ set$ **where**
 $range_vars \ \sigma = \bigcup (base_vars \ ' \ subst_range \ \sigma)$

lemma *vars-id-subst-subset*: $base_vars \ (x \cdot v \ id_subst) \subseteq \{x\}$
using *base.vars-id-subst base.no-vars-if-is-ground*
by *blast*

lemma *vars-subst-subset*: $vars \ (expr \cdot \sigma) \subseteq (vars \ expr - subst_domain \ \sigma) \cup range_vars \ \sigma$
unfolding *subst-domain-def range-vars-def vars-subst subset-eq*
using *base.vars-id-subst*
by (*smt (verit, del-insts) DiffI UN-iff UN-simps(10) UnCI base.no-vars-if-is-ground empty-iff mem-Collect-eq singleton-iff*)

end

context *base-substitution*
begin

sublocale *based-substitution*
where $base_subst = subst$ **and** $base_vars = vars$ **and** $base_is_ground = is_ground$
by *unfold-locales (simp-all add: base-vars-grounded-if-is-grounding base-vars-subst)*

declare *exists-nonground-iff-base-exists-nonground* [*simp del*]

end

hide-fact *base-substitution.base-vars-subst*
hide-fact *base-substitution.base-vars-grounded-if-is-grounding*

10 Properties of substitutions on base expressions

locale *based-subst-update* =
based-substitution +
subst-update +

```

assumes ground-subst-update-in-vars:
   $\bigwedge \text{update } \text{expr } \gamma \ x. \text{ base-is-ground } \text{update} \implies \text{is-ground } (\text{expr} \cdot \gamma) \implies x \in \text{vars } \text{expr} \implies$ 
   $\text{is-ground } (\text{expr} \cdot \gamma[x := \text{update}])$ 
begin

lemma vars-id-subst-update:  $\text{vars } (\text{expr} \cdot \text{id-subst}[x := b]) \subseteq \text{vars } \text{expr} \cup \text{base-vars } b$ 
proof (cases exists-nonground)
  case True

    then show ?thesis
      unfolding vars-subst
      using subst-update base.vars-id-subst
      by (smt (verit, del-insts) SUP-least UnCI base.no-vars-if-is-ground imageE singleton-iff subset-eq subst-update-var)
    next
      case False

    then show ?thesis
      using no-vars-if-is-ground
      by blast
  qed

lemma ground-subst-update [simp]:
  assumes base-is-ground update is-ground (expr · γ)
  shows is-ground (expr · γ[x := update])
  using assms
proof (cases x ∈ vars expr)
  case True

    show ?thesis
      using ground-subst-update-in-vars[OF assms True] .
  next
    case False

    then show ?thesis
      by (simp add: assms(2))
  qed

end

locale create-renaming = based-subst-update where
  apply-subst = apply-subst for
  apply-subst :: 'v ⇒ 'subst ⇒ 'base (infixl ·v 69) +
  assumes id-fold-subst-comp-ext:
   $\bigwedge us \ us'. \text{ exists-nonground } \implies$ 
   $(\bigwedge x. x \cdot v \text{ fold } (\lambda(x, b) \sigma. \sigma[x := b]) \text{ us } \text{id-subst} \odot \text{fold } (\lambda(x, b) \sigma. \sigma[x := b])$ 

```

$us' \text{ id-subst}$
 $= x \cdot v \text{ id-subst} \implies$
 $\text{fold } (\lambda(x, b) \sigma. \sigma[x := b]) \text{ us id-subst} \odot \text{fold } (\lambda(x, b) \sigma. \sigma[x := b]) \text{ us' id-subst}$
 $= \text{id-subst}$
begin

lemma *create-renaming*:

assumes *exists-nonground*: *exists-nonground* **and** *finite-S*: *finite S* **and** *bij*: *bij-betw f S T*

shows *is-renaming* (*renaming-of-bij f S T*)

proof (*unfold is-renaming-def*, *intro exI*)

have *finite-T*: *finite T* **and** *finite-S-T*: *finite (S \cup T)*

using *bij bij-betw-finite finite-S*

by *auto*

{
fix *x*

have $x \cdot v \text{ renaming-of-bij } f \ S \ T \odot \text{ renaming-of-bij-inv } f \ S \ T = x \cdot v \text{ id-subst}$

proof (*cases x \in S \cup T*)

case *False*

then show *?thesis*

unfolding *renaming-of-bij-def renaming-of-bij-inv-def base.comp-subst-iff*

using *exists-nonground finite-S-T*

by *auto*

next

case *x-in-S-T: True*

show *?thesis*

proof (*cases x \in S*)

case *x-in-S: True*

moreover then have $f \ x \in T$

using *bij*

by (*simp add: bij-betwE*)

moreover have *inv-into S f (f x) = x*

by (*metis x-in-S bij bij-betw-inv-into-left*)

ultimately show *?thesis*

unfolding *renaming-of-bij-def renaming-of-bij-inv-def base.comp-subst-iff*

using *subst-updates-fmap-of-set exists-nonground finite-S-T*

by *auto*

next

case *x-notin-S: False*

moreover then have $x \in T$

```

using x-in-S-T
by simp

moreover then have
  bij-partition S T x ∈ (S - T)
  inv-into (T \ S) (bij-partition S T) (bij-partition S T x) = x
using
  x-notin-S
  bij-partition[OF finite-S finite-T bij-betw-same-card[OF bij]]
  bij-betwE
  bij-betw-inv-into-left
by fastforce+

ultimately show ?thesis
unfolding renaming-of-bij-def renaming-of-bij-inv-def base.comp-subst-iff
using subst-updates-fmap-of-set exists-nonground finite-S-T
by auto
qed
qed
}

then show renaming-of-bij f S T ⊙ renaming-of-bij-inv f S T = id-subst
unfolding renaming-of-bij-def renaming-of-bij-inv-def subst-updates-def
by (rule id-fold-subst-comp-ext[OF exists-nonground])
qed

end

locale variables-in-base-imgu = based-substitution +
assumes variables-in-base-imgu:
   $\bigwedge expr \mu XX. base.is-imgu \mu XX \implies finite\ XX \implies \forall X \in XX. finite\ X \implies vars\ (expr \cdot \mu) \subseteq vars\ expr \cup (\bigcup (base-vars\ ' \bigcup XX))$ 

locale range-vars-subset-if-is-imgu = base-substitution +
assumes range-vars-subset-if-is-imgu:
   $\bigwedge \mu XX. is-imgu \mu XX \implies \forall X \in XX. finite\ X \implies finite\ XX \implies range-vars\ \mu \subseteq (\bigcup expr \in \bigcup XX. vars\ expr)$ 
begin

sublocale variables-in-base-imgu where
  base-subst = subst and base-vars = vars and base-is-ground = is-ground
using range-vars-subset-if-is-imgu vars-subst-subset
by unfold-locales fastforce

end

locale base-variables-in-base-imgu =

```

base-substitution **where** $vars = vars :: 'expr \Rightarrow 'v set +$
subst-update +
finite-variables +
infinite-variables +
assumes
not-back-to-id-subst: $\bigwedge expr \sigma. \exists x. expr \cdot \sigma = x \cdot v \text{ id-subst} \implies \exists x. expr = x \cdot v$
id-subst
begin

lemma *imgu-subst-domain-subset*:
assumes *imgu*: *is-imgu* μXX
shows *subst-domain* $\mu \subseteq \bigcup (vars \cup XX)$
proof (*intro Set.subsetI*)
fix x
assume $x \in \text{subst-domain } \mu$

then have $x \cdot \mu: x \cdot v \mu \neq x \cdot v \text{ id-subst}$
unfolding *subst-domain-def*
by *auto*

show $x \in \bigcup (vars \cup XX)$
proof (*rule ccontr*)
assume $x: x \notin \bigcup (vars \cup XX)$

define τ **where**
 $\tau \equiv \mu[x := x \cdot v \text{ id-subst}]$

have $x \cdot v \mu \odot \tau \neq x \cdot v \tau$
proof (*cases* $\exists y. x \cdot v \mu = y \cdot v \text{ id-subst}$)
case *True*

then obtain y **where** $y: x \cdot v \mu = y \cdot v \text{ id-subst}$
by *auto*

then have $x \neq y$
using $x \cdot \mu$
by *blast*

moreover have $y \cdot v \mu \neq x \cdot v \text{ id-subst}$
proof (*rule notI*)
assume $y \cdot v \mu = x \cdot v \text{ id-subst}$

then show *False*
using *imgu* $x \cdot \mu$
unfolding *is-imgu-def*
by (*metis comp-subst-iff id-subst-subst*)
qed

ultimately show *?thesis*

```

      unfolding comp-subst-iff  $y$   $\tau$ -def
      by (metis all-subst-ident-if-ground id-subst-subst subst-update-var)
next
case False

then show ?thesis
  unfolding  $\tau$ -def comp-subst-iff
  using not-back-to-id-subst
  by (metis all-subst-ident-if-ground id-subst-subst subst-update-var(1))
qed

then have  $\mu \odot \tau \neq \tau$ 
  by metis

moreover have is-unifier-set  $\tau$  XX
  using imgu is-imgu-def
  unfolding  $\tau$ -def is-unifier-set-def is-unifier-def subst-set-def
  using  $x$ 
  by auto

ultimately show False
  using imgu
  unfolding is-imgu-def
  by auto
qed
qed

lemma imgu-subst-domain-finite:
  assumes imgu: is-imgu  $\mu$  XX and finite-X:  $\forall X \in XX. \text{finite } X$  and finite-XX:
finite XX
  shows finite (subst-domain  $\mu$ )
  using imgu-subst-domain-subset[OF imgu] finite-XX finite-X finite-vars
  by (simp add: finite-subset)

lemma imgu-range-vars-finite:
  assumes imgu: is-imgu  $\mu$  XX and finite-X:  $\forall X \in XX. \text{finite } X$  and finite-XX:
finite XX
  shows finite (range-vars  $\mu$ )
  using imgu-subst-domain-finite[OF assms] finite-vars
  unfolding range-vars-def
  by blast

lemma imgu-range-vars-vars-subset:
  assumes imgu: is-imgu  $\mu$  XX and finite-X:  $\forall X \in XX. \text{finite } X$  and finite-XX:
finite XX
  shows  $\bigcup (\text{vars } \text{' } \text{expr-subst } \mu \text{' } \bigcup XX) \subseteq \bigcup (\text{vars } \text{' } \bigcup XX)$ 
proof (intro Set.subsetI)
  fix  $y$ 
  assume  $y: y \in \bigcup (\text{vars } \text{' } \text{expr-subst } \mu \text{' } \bigcup XX)$ 

```

then obtain x where
 $x: x \in \bigcup (\text{vars } ' \bigcup XX) \ y \in \text{vars } (x \cdot v \ \mu)$
using *vars-subst*
by *auto*

show $y \in \bigcup (\text{vars } ' \bigcup XX)$
proof (*rule ccontr*)
assume $y': y \notin \bigcup (\text{vars } ' \bigcup XX)$

then have $x\text{-neq-}y: x \neq y$
using x
by *auto*

obtain z where $z: z \notin \text{range-vars } \mu \ z \notin \text{subst-domain } \mu$
using
imgu-subst-domain-finite[*OF assms*]
imgu-range-vars-finite[*OF assms*]
infinite-vars
by (*metis Diff-iff finite-Diff2 infinite-super subsetI*)

define τ **where**
 $\tau \equiv \text{id-subst}\llbracket y := z \cdot v \ \text{id-subst} \rrbracket \odot \mu$

have $x \cdot v \ \mu \odot \tau \neq x \cdot v \ \tau$
proof –

have $\forall x. x \cdot v \ \mu = x \cdot v \ \text{id-subst} \vee z \notin \text{vars } (x \cdot v \ \mu)$
using *range-vars-def subst-domain-def z(1)*
by *auto*

then have $z \notin \text{vars } (x \cdot v \ \mu)$
using $z(1)$ $x(2)$ $x\text{-neq-}y$
unfolding *range-vars-def subst-domain-def*
by (*metis all-not-in-conv no-vars-if-is-ground singleton-iff vars-id-subst*)

moreover have $z \in \text{vars } (x \cdot v \ \mu \cdot \text{id-subst}\llbracket y := z \cdot v \ \text{id-subst} \rrbracket)$
using $x(2)$ *vars-id-subst subst-update-var(1)*
unfolding *vars-subst*
by (*metis UN-I equals0D imageI insertI1 no-vars-if-is-ground*)

then have $z \in \text{vars } (x \cdot v \ \mu \cdot \text{id-subst}\llbracket y := z \cdot v \ \text{id-subst} \rrbracket \cdot \mu)$
using $z(2)$ *vars-id-subst no-vars-if-is-ground*
unfolding *range-vars-def subst-domain-def vars-subst*
by *fastforce*

ultimately show *?thesis*
using $x\text{-neq-}y$
unfolding $\tau\text{-def comp-subst-iff}$

```

    by (metis id-subst-subst subst-comp-subst subst-update-var(2))
  qed

  then have  $\mu \odot \tau \neq \tau$ 
    by metis

  moreover have is-unifier-set  $\tau$   $XX$ 
    using imgu is-imgu-def
    unfolding  $\tau$ -def is-unifier-set-def is-unifier-def subst-set-def
    using  $y'$ 
    by auto

  ultimately show False
    using imgu
    unfolding is-imgu-def
    by auto
  qed
qed

lemma range-vars-subset-if-is-imgu:
  assumes is-imgu  $\mu$   $XX \ \forall X \in XX. \text{finite } X \text{ finite } XX$ 
  shows range-vars  $\mu \subseteq (\bigcup \text{expr} \in \bigcup XX. \text{vars } \text{expr})$ 
proof -
  have range-vars  $\mu = (\bigcup x \in \text{subst-domain } \mu. \text{vars } (x \cdot v \ \mu))$ 
    by (simp add: range-vars-def)

  also have  $\dots \subseteq (\bigcup (\text{vars } \text{' } (\lambda \text{expr}. \text{expr} \cdot \mu) \text{' } \bigcup XX))$ 
    using imgu-subst-domain-subset[OF assms(1)] subsetD vars-subst
    by fastforce

  also have  $\dots \subseteq (\bigcup \text{expr} \in \bigcup XX. \text{vars } \text{expr})$ 
    using imgu-range-vars-vars-subset[OF assms] .

  finally show ?thesis .
qed

sublocale variables-in-base-imgu where
  base-subst = subst and base-vars = vars and base-is-ground = is-ground
  using range-vars-subset-if-is-imgu vars-subst-subset
  by unfold-locales fastforce

end

locale base-exists-non-ident-subst =
  base-substitution where vars = vars +
  finite-variables where vars = vars +
  infinite-variables where vars = vars +
  all-subst-ident-iff-ground where vars = vars +
  subst-update where vars = vars +

```

```

    is-ground-if-no-vars where vars = vars
for vars :: 'expr  $\Rightarrow$  'v set
begin

sublocale exists-non-ident-subst
proof unfold-locales
  fix expr and S :: 'expr set
  assume finite-S: finite S and vars-not-empty:  $\neg$  is-ground expr

  obtain x where x: x  $\in$  vars expr
    using is-ground-iff-no-vars vars-not-empty
    by auto

  have finite (vars-set S)
    using finite-S finite-vars
    unfolding vars-set-def
    by blast

  obtain y where y: y  $\notin$  vars expr y  $\notin$  vars-set S
  proof –

    have finite (vars-set S)
      using finite-S finite-vars
      unfolding vars-set-def
      by blast

    then have finite (vars expr  $\cup$  vars-set S)
      using finite-vars
      by simp

    then show ?thesis
      using that infinite-vars finite-vars
      by (meson UnCI ex-new-if-finite)
  qed

  define  $\sigma$  where  $\sigma \equiv id\text{-subst}[x := y \cdot v\ id\text{-subst}]$ 

  show  $\exists \sigma. expr \cdot \sigma \neq expr \wedge expr \cdot \sigma \notin S$ 
  proof (intro exI conjI)

    have y-in-expr- $\sigma$ : y  $\in$  vars (expr  $\cdot$   $\sigma$ )
      unfolding  $\sigma$ -def vars-subst
      using x
      by (metis UN-iff image-eqI singletonI subst-update-var(1) vars-id-subst vars-not-empty)

    then show expr  $\cdot$   $\sigma \neq expr$ 
      using y
      by metis

```

```

    show  $expr \cdot \sigma \notin S$ 
      using  $y\text{-in-expr-}\sigma\ y(2)$ 
      unfolding  $vars\text{-set-def}$ 
      by auto
  qed
end

```

```

locale  $vars\text{-grounded-iff-is-grounding} = based\text{-substitution} +$ 
  assumes  $is\text{-grounding-if-}\mathit{vars}\text{-grounded}$ :
     $\bigwedge expr\ \gamma. \forall x \in vars\ expr. base\text{-is-ground}\ (x \cdot v\ \gamma) \implies is\text{-ground}\ (expr \cdot \gamma)$ 
begin

```

```

lemma  $vars\text{-grounded-iff-is-grounding}$ :  $(\forall x \in vars\ b. base\text{-is-ground}\ (x \cdot v\ \gamma)) \longleftrightarrow$ 
 $is\text{-ground}\ (b \cdot \gamma)$ 
  using  $is\text{-grounding-if-}\mathit{vars}\text{-grounded}\ \mathit{vars}\text{-grounded-if-is-grounding}$ 
  by  $blast$ 

```

```

end

```

```

end
theory  $Natural\text{-Magma}$ 
  imports  $Main$ 
begin

```

```

locale  $natural\text{-magma} =$ 
  fixes
     $to\text{-set} :: 'b \Rightarrow 'a\ set$  and
     $plus :: 'b \Rightarrow 'b \Rightarrow 'b$  and
     $wrap :: 'a \Rightarrow 'b$  and
     $add$ 
  defines  $\bigwedge a\ b. add\ a\ b \equiv plus\ (wrap\ a)\ b$ 
  assumes
     $to\text{-set-plus}\ [simp]: \bigwedge b\ b'. to\text{-set}\ (plus\ b\ b') = (to\text{-set}\ b) \cup (to\text{-set}\ b')$  and
     $to\text{-set-wrap}\ [simp]: \bigwedge a. to\text{-set}\ (wrap\ a) = \{a\}$ 
begin

```

```

lemma  $to\text{-set-add}\ [simp]: to\text{-set}\ (add\ a\ b) = insert\ a\ (to\text{-set}\ b)$ 
  using  $to\text{-set-plus}\ to\text{-set-wrap}\ add\text{-def}$ 
  by  $simp$ 

```

```

end

```

```

locale  $natural\text{-magma-with-empty} = natural\text{-magma} +$ 
  fixes  $empty$ 
  assumes  $to\text{-set-empty}\ [simp]: to\text{-set}\ empty = \{\}$ 

```

```

end
theory Natural-Functor
  imports Main
begin

locale natural-functor =
  fixes
    map :: ('a ⇒ 'a) ⇒ 'b ⇒ 'b and
    to-set :: 'b ⇒ 'a set
  assumes
    map-comp [simp]:  $\bigwedge b f g. \text{map } f (\text{map } g b) = \text{map } (f \circ g) b$  and
    map-ident [simp]:  $\bigwedge b. \text{map } (\lambda x. x) b = b$  and
    map-cong0 [cong]:  $\bigwedge b f g. (\bigwedge a. a \in \text{to-set } b \implies f a = g a) \implies \text{map } f b = \text{map } g b$  and
    to-set-map [simp]:  $\bigwedge b f. \text{to-set } (\text{map } f b) = f \text{ ` to-set } b$ 
begin

lemma map-comp' [simp]:  $\bigwedge b f g. \text{map } f (\text{map } g b) = \text{map } (\lambda x. f (g x)) b$ 
  using map-comp
  by simp

lemma map-id [simp]:  $\text{map } \text{id } b = b$ 
  using map-ident
  unfolding id-def .

lemma map-cong [cong]:
  assumes  $b = b' \wedge a. a \in \text{to-set } b' \implies f a = g a$ 
  shows  $\text{map } f b = \text{map } g b'$ 
  using map-cong0 assms
  by blast

lemma map-id-cong [simp]:
  assumes  $\bigwedge a. a \in \text{to-set } b \implies f a = a$ 
  shows  $\text{map } f b = b$ 
  using assms
  by simp

lemma to-set-map-not-ident:
  assumes  $a \in \text{to-set } b \wedge f a \notin \text{to-set } b$ 
  shows  $\text{map } f b \neq b$ 
  using assms
  by (metis rev-image-eqI to-set-map)

lemma map-in-to-set:
  assumes  $\text{map } f b = b \wedge a \in \text{to-set } b$ 
  shows  $f a \in \text{to-set } b$ 
  using assms
  by (metis image-eqI to-set-map)

```

```

lemma to-set-const [simp]: to-set  $b \neq \{\}$   $\implies$  to-set (map ( $\lambda\cdot$ .  $a$ )  $b$ ) =  $\{a\}$ 
  by auto

lemma map-inverse:  $(\bigwedge x. f (g x) = x) \implies \text{map } f (\text{map } g b) = b$ 
  by simp

end

locale non-empty-natural-functor = natural-functor +
  assumes exists-non-empty:  $\exists b. \text{to-set } b \neq \{\}$ 
begin

lemma exists-functor [intro]:  $\exists b. a \in \text{to-set } b$ 
proof –

  obtain  $b$  where to-set  $b \neq \{\}$ 
    using exists-non-empty
    by blast

  then have  $a \in \text{to-set } (\text{map } (\lambda\cdot$ .  $a$ )  $b$ )
    by auto

  then show ?thesis
    by blast
qed

end

locale finite-natural-functor = natural-functor +
  assumes finite-to-set [intro]:  $\bigwedge b. \text{finite } (\text{to-set } b)$ 

locale non-empty-finite-natural-functor =
  non-empty-natural-functor + finite-natural-functor

locale natural-functor-conversion =
  natural-functor +
  functor': natural-functor where map = map' and to-set = to-set'
  for map' ::  $('b \Rightarrow 'b) \Rightarrow 'd \Rightarrow 'd$  and to-set' ::  $'d \Rightarrow 'b \text{ set} +$ 
  fixes
    map-to ::  $('a \Rightarrow 'b) \Rightarrow 'c \Rightarrow 'd$  and
    map-from ::  $('b \Rightarrow 'a) \Rightarrow 'd \Rightarrow 'c$ 
  assumes
    to-set-map-from [simp]:  $\bigwedge f d. \text{to-set } (\text{map-from } f d) = f \text{ ' to-set' } d$  and
    to-set-map-to [simp]:  $\bigwedge f c. \text{to-set' } (\text{map-to } f c) = f \text{ ' to-set } c$  and
    conversion-map-comp [simp]:  $\bigwedge c f g. \text{map-from } f (\text{map-to } g c) = \text{map } (\lambda x. f (g$ 
   $x)) c$  and
    conversion-map-comp' [simp]:  $\bigwedge d f g. \text{map-to } f (\text{map-from } g d) = \text{map' } (\lambda x. f$ 

```

(g x) d

lemma *non-empty-helper*: $x \in \text{to-set } b \implies \exists b. \text{to-set } b \neq \{\}$
by *blast*

ML ‹
local

open *BNF-Util*;
open *BNF-Def*;
open *BNF-Tactics*;
open *BNF-FP-Def-Sugar*;

in

(* *TODO: Is ignoring correct? **)
fun *bnf-name-qualified* *bnf* =
 (*case* *T-of-bnf* *bnf* *of*
 Type (*name*, -) => *SOME name*
 | - => *NONE*)

structure *Natural-Function-Ignore* :
sig
 val *add-ignore* : *string* -> *theory* -> *theory*
 val *is-ignored* : *bnf* -> *Proof.context* -> *bool*
end = *struct*

structure *Data* = *Generic-Data*
(
 type *T* = *Symtab.set*
 val *empty* = *Symtab.empty*
 val *merge* = *Symtab.merge* (*K true*)
)

fun *add-ignore* *name* *thy* =
 Context.theory-map (*Data.map* (*Symtab.insert-set* *name*)) *thy*

fun *is-ignored* *bnf* *ctxt* =
 (*case* *bnf-name-qualified* *bnf* *of*
 SOME name => *Symtab.defined* (*Data.get* (*Context.Proof* *ctxt*)) *name*
 | *NONE* => *false*)

end

(* *TODO: Improve printing (done is sometimes wrong) **)
fun *method-template* *name* *methods* = *Method.Basic* (*fn* *ctxt* => *METHOD* (*fn*
thms =>
 TRY (*EVERY1* ([

```

K (print-tac ctxt (Interpreting ^ name)),
K (Locale.intro-locales-tac {strict = false, eager = true} ctxt thms)] @
methods ctxt @
[K (print-tac ctxt done))))))

fun method-base bnf name = method-template name (fn ctxt => [
  K (print-tac ctxt map-comp),
  rtac ctxt (map-comp-of-bnf bnf RS trans),
  SELECT-GOAL (unfold-thms-tac ctxt @ {thms comp-def id-o o-id}),
  rtac ctxt refl,
  K (print-tac ctxt map-ident),
  rtac ctxt (map-ident-of-bnf bnf),
  K (print-tac ctxt map-cong),
  rtac ctxt (map-cong0-of-bnf bnf)
  THEN-ALL-NEW (Goal.assume-rule-tac ctxt ORELSE' rtac ctxt refl),
  K (print-tac ctxt set-map),
  resolve-tac ctxt (set-map-of-bnf bnf)])

fun method-non-empty (fp-sugar: fp-sugar) name =
  let
    val set-introssss = #set-introssss (#fp-bnf-sugar fp-sugar);
    val set-intros = flat (flat (flat set-introssss));
  in
    method-template name (fn ctxt =>
      [K (print-tac ctxt non-empty),
       rtac ctxt @ {thm non-empty-helper},
       resolve-tac ctxt set-intros])
  end;

fun method-finite bnf name = (case set-finite-of-bnf bnf of
  NONE => Method.Basic (K (METHOD (K no-tac)))
| SOME set-finite =>
  method-template name (fn ctxt => [
    K (print-tac ctxt finite),
    resolve-tac ctxt set-finite]))

fun interpret locale method bnf lthy =
  let
    fun interpret i map set lthy =
      let
        val index = if i <= 1 then else string-of-int i
        (* TODO: Check for name clashes (Like for inference) *)
        val name = Binding.name-of (name-of-bnf bnf) ^ -functor ^ index
        val expression = (Expression.Positional [SOME map, SOME set], [])

        val state =
          Interpretation.isar-interpretation ([ (locale, ((name, true), expression)) ], [])
      in
        lthy;
      end
  end
end

```

```

    val lthy =
      Proof.global-terminal-proof ((method name, Position.no-range), NONE)
state
    handle ERROR - => (tracing (Could not interpret ^ name) ; lthy);

    in lthy end;

    val live = live-of-bnf bnf
    and deads = deads-of-bnf bnf;

    val ((As, unsorted-Ds), -) = lthy
    |> mk-TFrees live
    ||>> mk-TFrees (length deads);
    val Ds = map2 (resort-tfree-or-tvar o Type.sort-of-atyp) deads unsorted-Ds;

    fun mk-map i =
      let
        val map-all = mk-map-of-bnf Ds As As bnf;
        fun id A = Abs (x, A, Bound 0);
        val args = map-index (fn (j, A) => if i = j then Bound 0 else id A) As;
        val term = list-comb (map-all, args);
      in Abs (x, nth As i --> nth As i, term) end

    val maps = map mk-map (0 upto live - 1);
    val sets = mk-sets-of-bnf (replicate live Ds) (replicate live As) bnf;

    in
      if Natural-Functor-Ignore.is-ignored bnf (Local-Theory.target-of lthy)
      then lthy
      else @{\fold 3} interpret (1 upto live) maps sets lthy
    end;

    val natural-functor-setup =
      let
        val description = interpret natural functor locale for BNFs;

        fun interpretation bnf = interpret @{\locale natural-functor} (method-base bnf)
        bnf;

      in bnf-interpretation description interpretation end;

    val non-empty-natural-functor-setup =
      let
        val description = interpret nonempty natural functor locale for BNFs;

        fun interpretation sugar =
          interpret @{\locale non-empty-natural-functor} (method-non-empty sugar)
          (#fp-bnf sugar);
      end;

```

```

    in fp-sugars-interpretation description (fold interpretation) end;

val finite-natural-functor-setup =
  let
    val description = interpret finite natural functor locale for BNFs;

    fun interpretation bnf = interpret @{locale finite-natural-functor} (method-finite
    bnf) bnf;

    in bnf-interpretation description interpretation end;

val natural-functor-setups =
  natural-functor-setup
  #> non-empty-natural-functor-setup
  #> finite-natural-functor-setup

fun natural-functor-ignore type-name =
  Natural-Functor-Ignore.add-ignore type-name

end
end
theory Natural-Magma-Function
  imports Natural-Magma Natural-Function
begin

locale natural-magma-functor = natural-magma + natural-functor +
assumes
  map-wrap:  $\bigwedge f a. \text{map } f (\text{wrap } a) = \text{wrap } (f a)$  and
  map-plus:  $\bigwedge f b b'. \text{map } f (\text{plus } b b') = \text{plus } (\text{map } f b) (\text{map } f b')$ 
begin

lemma map-add:  $\bigwedge f a b. \text{map } f (\text{add } a b) = \text{add } (f a) (\text{map } f b)$ 
  unfolding add-def
  using map-plus map-wrap
  by simp

end

end
theory Substitution-Lifting
  imports Substitution Natural-Magma-Function
begin

```

11 Lifting of substitutions using natural functors

```

locale substitution-lifting =
  sub: substitution where

```

```

subst = sub-subst and vars = sub-vars and apply-subst = apply-subst and
is-ground = sub-is-ground +
non-empty-natural-functor where map = map and to-set = to-set
for
sub-vars :: 'sub ⇒ 'v set and
sub-subst :: 'sub ⇒ 'subst ⇒ 'sub and
sub-is-ground :: 'sub ⇒ bool and
map :: ('sub ⇒ 'sub) ⇒ 'expr ⇒ 'expr and
to-set :: 'expr ⇒ 'sub set and
apply-subst :: 'v ⇒ 'subst ⇒ 'base (infixl ·v 69)
begin

definition vars :: 'expr ⇒ 'v set where
vars expr ≡ ⋃ (sub-vars ' to-set expr)

definition is-ground :: 'expr ⇒ bool where
is-ground expr ≡ ∀ sub ∈ to-set expr. sub-is-ground sub

notation sub-subst (infixl ·s 70)

definition subst :: 'expr ⇒ 'subst ⇒ 'expr (infixl · 70) where
expr · σ ≡ map (λsub. sub ·s σ) expr

lemma subst-in-to-set-subst [intro]:
assumes sub ∈ to-set expr
shows sub ·s σ ∈ to-set (expr · σ)
unfolding subst-def to-set-map
using assms
by simp

sublocale substitution where subst = (·) and vars = vars and is-ground =
is-ground
proof unfold-locales
fix expr σ1 σ2

show expr · (σ1 ⊙ σ2) = expr · σ1 · σ2
unfolding subst-def map-comp comp-apply sub.subst-comp-subst ..
next
fix expr

show expr · id-subst = expr
using map-ident
unfolding subst-def sub.subst-id-subst .
next
fix expr
assume is-ground expr

then show ∀ σ. expr · σ = expr
unfolding subst-def is-ground-def

```

```

    by simp
next
  fix expr
  assume is-ground expr

  then show vars expr = {}
    unfolding is-ground-def vars-def
    by (simp add: sub.no-vars-if-is-ground)
qed

lemma exists-nonground-iff-sub-exists-nonground: exists-nonground  $\longleftrightarrow$  sub.exists-nonground
  using exists-functor is-ground-def
  by fastforce

lemma ground-subst-iff-sub-ground-subst [simp]: is-ground-subst  $\gamma \longleftrightarrow$  sub.is-ground-subst
 $\gamma$ 
proof (unfold is-ground-subst-def sub.is-ground-subst-def, intro iffI allI)
  fix sub
  assume all-ground:  $\forall$  expr. is-ground (expr  $\cdot$   $\gamma$ )

  show sub-is-ground (sub  $\cdot_s$   $\gamma$ )
  proof (rule ccontr)
    assume sub-not-ground:  $\neg$ sub-is-ground (sub  $\cdot_s$   $\gamma$ )

    then obtain expr where sub  $\in$  to-set expr
      using exists-functor
      by blast

    then have  $\neg$ is-ground (expr  $\cdot$   $\gamma$ )
      using sub-not-ground to-set-map
      unfolding subst-def is-ground-def
      by auto

    then show False
      using all-ground
      by blast
  qed
qed
next
  fix expr
  assume  $\forall$  sub. sub-is-ground (sub  $\cdot_s$   $\gamma$ )

  then show is-ground (expr  $\cdot$   $\gamma$ )
    unfolding is-ground-def subst-def
    using to-set-map
    by simp
qed

lemma to-set-is-ground [intro]:
  assumes sub  $\in$  to-set expr is-ground expr

```

```

shows sub-is-ground sub
using assms
by (simp add: is-ground-def)

lemma to-set-is-ground-subst:
assumes  $sub \in to\ set\ expr\ is\ ground\ (expr \cdot \gamma)$ 
shows  $sub\ is\ ground\ (sub \cdot_s \gamma)$ 
using assms
by (meson subst-in-to-set-subst to-set-is-ground)

lemma subst-empty:
assumes  $to\ set\ expr' = \{\}$ 
shows  $expr \cdot \sigma = expr' \iff expr = expr'$ 
using assms map-id-cong to-set-map
unfolding subst-def
by (metis empty-iff image-is-empty)

lemma empty-is-ground:
assumes  $to\ set\ expr = \{\}$ 
shows  $is\ ground\ expr$ 
using assms
by (simp add: is-ground-def)

lemma to-set-image:  $to\ set\ (expr \cdot \sigma) = (\lambda a. a \cdot_s \sigma) \ ` \ to\ set\ expr$ 
unfolding subst-def to-set-map ..

lemma to-set-subset-vars-subset:
assumes  $to\ set\ expr \subseteq to\ set\ expr'$ 
shows  $vars\ expr \subseteq vars\ expr'$ 
using assms
unfolding vars-def
by blast

lemma to-set-subset-is-ground:
assumes  $to\ set\ expr' \subseteq to\ set\ expr\ is\ ground\ expr$ 
shows  $is\ ground\ expr'$ 
using assms to-set-subset-vars-subset is-ground-def
by auto

end

11.1 Lifting of properties

locale subst-update-lifting =
  sub: subst-update where
  vars = sub-vars :: 'sub  $\Rightarrow$  'v set and subst = sub-subst and is-ground = sub-is-ground
+
  substitution-lifting
begin

```

```

sublocale subst-update where vars = vars and subst = subst and is-ground =
is-ground
proof unfold-locales
  fix  $\sigma$  x update
  assume exists-nonground

  then show  $x \cdot v \sigma[x := update] = update$ 
    unfolding is-ground-def
    using sub.subst-update-var(1)
    by fastforce
next
  fix x expr update  $\sigma$ 
  assume  $x \notin vars$  expr

  then show  $expr \cdot \sigma[x := update] = expr \cdot \sigma$ 
    unfolding vars-def subst-def
    by auto
qed simp-all

end

locale finite-variables-lifting =
  sub: finite-variables where
    vars = sub-vars :: 'sub  $\Rightarrow$  'v set and subst = sub-subst and is-ground = sub-is-ground
  +

  finite-natural-functor where to-set = to-set :: 'expr  $\Rightarrow$  'sub set +

  substitution-lifting
begin

abbreviation to-fset :: 'expr  $\Rightarrow$  'sub fset where
  to-fset expr  $\equiv$  Abs-fset (to-set expr)

sublocale finite-variables where vars = vars and subst = subst and is-ground =
is-ground
  by unfold-locales (auto simp: vars-def finite-to-set)

lemma fset-to-fset [simp]: fset (to-fset expr) = to-set expr
  using Abs-fset-inverse finite-to-set
  by blast

lemma to-fset-map: to-fset (map f expr) = f |` to-fset expr
  using to-set-map
  by (metis fset.set-map fset-inverse fset-to-fset)

lemma to-fset-is-ground-subst:
  assumes sub  $|\in|$  to-fset expr is-ground (subst expr  $\gamma$ )

```

```

shows sub-is-ground (sub ·s  $\gamma$ )
using assms
by (simp add: to-set-is-ground-subst)

end

locale renaming-variables-lifting =
  substitution-lifting +
  sub: renaming-variables where vars = sub-vars and subst = sub-subst and
is-ground = sub-is-ground
begin

sublocale renaming-variables where subst = subst and vars = vars and is-ground
= is-ground
proof unfold-locales
  fix expr  $\varrho$ 
  assume sub.is-renaming  $\varrho$ 

  then show vars (expr ·  $\varrho$ ) = rename  $\varrho$  ' vars expr
    using sub.rename-variables
    unfolding vars-def subst-def to-set-map
    by fastforce
next
  fix  $\varrho$ 
  assume exists-nonground

  then show sub.is-renaming  $\varrho \implies$  (inj ( $\lambda x. x \cdot v \varrho$ )  $\wedge$  ( $\forall x. \exists x'. x \cdot v \varrho = x' \cdot v$ 
id-subst))
    unfolding is-ground-def
    using sub.is-renaming-imp
    by auto
qed

end

locale exists-ground-lifting =
  substitution-lifting +
  sub: exists-ground where vars = sub-vars and subst = sub-subst and is-ground
= sub-is-ground
begin

sublocale exists-ground where vars = vars and subst = subst and is-ground =
is-ground
proof unfold-locales

  show  $\exists$  expr. is-ground expr
    using sub.exists-ground empty-is-ground
    unfolding is-ground-def
    by (metis singletonD to-set-const)

```

qed

end

locale *grounding-lifting* =

substitution-lifting **where** *sub-vars* = *sub-vars* **and** *sub-subst* = *sub-subst* **and**
map = *map* +

sub: grounding **where**

vars = *sub-vars* **and** *subst* = *sub-subst* **and** *to-ground* = *sub-to-ground* **and**
from-ground = *sub-from-ground* **and** *is-ground* = *sub-is-ground* +

natural-functor-conversion **where**

map = *map* **and** *map-to* = *to-ground-map* **and** *map-from* = *from-ground-map*
and *map'* = *ground-map* **and**
to-set' = *to-set-ground*

for

sub-to-ground :: '*sub* ⇒ '*sub_G* **and**
sub-from-ground :: '*sub_G* ⇒ '*sub* **and**
sub-vars :: '*sub* ⇒ '*var set* **and**
sub-subst :: '*sub* ⇒ '*subst* ⇒ '*sub* **and**
map :: ('*sub* ⇒ '*sub*) ⇒ '*expr* ⇒ '*expr* **and**
to-ground-map :: ('*sub* ⇒ '*sub_G*) ⇒ '*expr* ⇒ '*expr_G* **and**
from-ground-map :: ('*sub_G* ⇒ '*sub*) ⇒ '*expr_G* ⇒ '*expr* **and**
ground-map :: ('*sub_G* ⇒ '*sub_G*) ⇒ '*expr_G* ⇒ '*expr_G* **and**
to-set-ground :: '*expr_G* ⇒ '*sub_G* set

begin

definition *to-ground* :: '*expr* ⇒ '*expr_G* **where**

to-ground *expr* ≡ *to-ground-map* *sub-to-ground* *expr*

definition *from-ground* :: '*expr_G* ⇒ '*expr* **where**

from-ground *expr_G* ≡ *from-ground-map* *sub-from-ground* *expr_G*

sublocale *grounding* **where**

vars = *vars* **and** *subst* = *subst* **and** *to-ground* = *to-ground* **and** *from-ground* =
from-ground **and**
is-ground = *is-ground*

proof *unfold-locales*

{
 fix *expr*

assume *is-ground* *expr*

then have $\forall sub \in to\text{-}set\ expr. sub \in range\ sub\text{-}from\text{-}ground$

by (*simp* *add: sub.is-ground-iff-range-from-ground is-ground-def*)

then have $\forall sub \in to\text{-}set\ expr. \exists sub_G. sub\text{-}from\text{-}ground\ sub_G = sub$

```

    by fast

  then have  $\exists \text{expr}_G. \text{from-ground } \text{expr}_G = \text{expr}$ 
    unfolding from-ground-def
    by (metis conversion-map-comp map-id-cong)

  then have  $\text{expr} \in \text{range from-ground}$ 
    unfolding from-ground-def
    by blast
}

then show  $\{\text{expr}. \text{is-ground expr}\} = \text{range from-ground}$ 
  by (simp add: from-ground-def is-ground-def subset-antisym subset-eq)
next
  fix  $\text{expr}_G$ 
  show  $\text{to-ground } (\text{from-ground } \text{expr}_G) = \text{expr}_G$ 
    unfolding from-ground-def to-ground-def
    by simp
qed

lemma to-set-from-ground:  $\text{to-set } (\text{from-ground } \text{expr}) = \text{sub-from-ground } \text{' } (\text{to-set-ground } \text{expr})$ 
  unfolding from-ground-def
  by simp

lemma sub-in-ground-is-ground:
  assumes  $\text{sub} \in \text{to-set } (\text{from-ground } \text{expr})$ 
  shows  $\text{sub-is-ground } \text{sub}$ 
  using assms
  by (simp add: to-set-is-ground)

lemma ground-sub-in-ground:
   $\text{sub} \in \text{to-set-ground } \text{expr} \iff \text{sub-from-ground } \text{sub} \in \text{to-set } (\text{from-ground } \text{expr})$ 
  by (simp add: inj-image-mem-iff sub.inj-from-ground to-set-from-ground)

lemma ground-sub:
   $(\forall \text{sub} \in \text{to-set } (\text{from-ground } \text{expr}_G). P \text{ sub}) \iff$ 
   $(\forall \text{sub}_G \in \text{to-set-ground } \text{expr}_G. P (\text{sub-from-ground } \text{sub}_G))$ 
  by (simp add: to-set-from-ground)

end

locale exists-non-ident-subst-lifting =
  finite-variables-lifting where  $\text{map} = \text{map} +$ 
   $\text{sub}: \text{exists-non-ident-subst}$  where  $\text{subst} = \text{sub-subst}$  and  $\text{vars} = \text{sub-vars}$  and
   $\text{is-ground} = \text{sub-is-ground}$ 
for  $\text{map} :: (\text{'sub} \Rightarrow \text{'sub}) \Rightarrow \text{'expr} \Rightarrow \text{'expr}$ 
begin

```

```

sublocale exists-non-ident-subst where subst = subst and vars = vars and is-ground
= is-ground
proof unfold-locale
  fix expr :: 'expr and S :: 'expr set

  assume finite: finite S and not-ground:  $\neg$  is-ground expr

  then have finite-subst: finite ( $\bigcup$  (to-set ' insert expr S))
    using finite-to-set
    by blast

  obtain sub where sub: sub  $\in$  to-set expr and sub-not-ground:  $\neg$  sub-is-ground
sub
    using not-ground
    unfolding is-ground-def
    by blast

  obtain  $\sigma$  where sigma-not-ident: sub  $\cdot_s$   $\sigma \neq$  sub sub  $\cdot_s$   $\sigma \notin \bigcup$  (to-set ' insert expr
S)
    using sub.exists-non-ident-subst[OF finite-subst sub-not-ground]
    by blast

  then have expr  $\cdot$   $\sigma \neq$  expr
    using sub
    unfolding subst-def
    by (simp add: to-set-map-not-ident)

  moreover have expr  $\cdot$   $\sigma \notin$  S
    using sigma-not-ident(2) sub to-set-map
    unfolding subst-def
    by auto

  ultimately show  $\exists \sigma. \textit{expr} \cdot \sigma \neq \textit{expr} \wedge \textit{expr} \cdot \sigma \notin S$ 
    by blast
qed

end

locale all-subst-ident-iff-ground-lifting =
  exists-non-ident-subst-lifting where map = map +
  sub: all-subst-ident-iff-ground where
  subst = sub-subst and vars = sub-vars and is-ground = sub-is-ground
for map :: ('sub  $\Rightarrow$  'sub)  $\Rightarrow$  'expr  $\Rightarrow$  'expr
begin

sublocale all-subst-ident-iff-ground where subst = subst and vars = vars and
is-ground = is-ground
proof unfold-locale
  fix expr

```

```

show is-ground expr  $\longleftrightarrow (\forall \sigma. \text{expr} \cdot \sigma = \text{expr})$ 
proof(rule iffI allI)
  assume is-ground expr

  then show  $\forall \sigma. \text{expr} \cdot \sigma = \text{expr}$ 
    by simp
next
  assume all-subst-ident:  $\forall \sigma. \text{expr} \cdot \sigma = \text{expr}$ 

  show is-ground expr
proof(rule ccontr)
  assume  $\neg \text{is-ground expr}$ 

  then obtain sub where sub: sub  $\in$  to-set expr  $\neg \text{sub-is-ground sub}$ 
    unfolding is-ground-def
    by blast

  then obtain  $\sigma$  where sub  $\cdot_s \sigma \neq \text{sub}$  and sub  $\cdot_s \sigma \notin$  to-set expr
    using sub.exists-non-ident-subst finite-to-set
    by blast

  then show False
    using sub subst-in-to-set-subst all-subst-ident
    by metis
  qed
qed
qed

end

locale natural-magma-substitution-lifting = substitution-lifting + natural-magma
begin

lemma vars-add [simp]:
  vars (add sub expr) = sub-vars sub  $\cup$  vars expr
  unfolding vars-def
  using to-set-add by auto

lemma vars-plus [simp]:
  vars (plus expr expr') = vars expr  $\cup$  vars expr'
  unfolding vars-def
  by simp

lemma is-ground-plus [simp]:
  is-ground (plus expr expr')  $\longleftrightarrow$  is-ground expr  $\wedge$  is-ground expr'
  unfolding is-ground-def
  by auto

```

lemma *is-ground-add* [*simp*]:
 $is_ground (add\ sub\ expr) \longleftrightarrow sub_is_ground\ sub \wedge is_ground\ expr$
unfolding *is-ground-def*
by *simp*

end

locale *natural-magma-functor-substitution-lifting* =
natural-magma-substitution-lifting + *natural-magma-functor*
begin

lemma *add-subst* [*simp*]:
 $(add\ sub\ expr) \cdot \sigma = add (sub \cdot_s \sigma) (expr \cdot \sigma)$
unfolding *subst-def*
using *map-add*
by *simp*

lemma *plus-subst* [*simp*]: $(plus\ expr\ expr') \cdot \sigma = plus (expr \cdot \sigma) (expr' \cdot \sigma)$
unfolding *subst-def*
using *map-plus*
by *simp*

end

locale *natural-magma-grounding-lifting* =
grounding-lifting +
natural-magma-substitution-lifting +
ground: natural-magma **where**
 $to_set = to_set_ground$ **and** $plus = plus_ground$ **and** $wrap = wrap_ground$ **and**
 $add = add_ground$
for *plus-ground* *wrap-ground* *add-ground* +
assumes
to-ground-plus [*simp*]:
 $\bigwedge expr\ expr'.\ to_ground (plus\ expr\ expr') = plus_ground (to_ground\ expr) (to_ground\ expr')$ **and**
 $wrap_from_ground: \bigwedge sub.\ from_ground (wrap_ground\ sub) = wrap (sub_from_ground\ sub)$ **and**
 $wrap_to_ground: \bigwedge sub.\ to_ground (wrap\ sub) = wrap_ground (sub_to_ground\ sub)$
begin

lemma *from-ground-plus* [*simp*]:
 $from_ground (plus_ground\ expr\ expr') = plus (from_ground\ expr) (from_ground\ expr')$
using *to-ground-plus*
by (*metis from-ground-inverse ground-is-ground to-ground-inverse is-ground-plus*)

lemma *from-ground-add* [*simp*]:
 $from_ground (add_ground\ sub\ expr) = add (sub_from_ground\ sub) (from_ground\ expr)$

```

unfolding ground.add-def add-def
using from-ground-plus
by (simp add: wrap-from-ground)

lemma to-ground-add [simp]:
  to-ground (add sub expr) = add-ground (sub-to-ground sub) (to-ground expr)
unfolding ground.add-def add-def
using from-ground-add wrap-to-ground
by simp

lemma ground-add:
  assumes from-ground expr = add sub expr'
  shows expr = add-ground (sub-to-ground sub) (to-ground expr')
  using assms
  by (metis from-ground-inverse to-ground-add)

end

locale natural-magma-with-empty-grounding-lifting =
  natural-magma-grounding-lifting +
  natural-magma-with-empty +
  ground: natural-magma-with-empty where
  to-set = to-set-ground and plus = plus-ground and wrap = wrap-ground and
  add = add-ground and
  empty = empty-ground
for empty-ground +
assumes to-ground-empty [simp]: to-ground empty = empty-ground
begin

lemmas empty-magma-is-ground [simp] = empty-is-ground[OF to-set-empty]

lemmas magma-subst-empty [simp] =
  subst-ident-if-ground[OF empty-magma-is-ground]
  subst-empty[OF to-set-empty]

lemma from-ground-empty [simp]: from-ground empty-ground = empty
  using to-ground-inverse[OF empty-magma-is-ground]
  by simp

lemma to-ground-empty' [simp]: to-ground expr = empty-ground  $\longleftrightarrow$  expr =
  empty
  using from-ground-empty to-ground-empty ground.to-set-empty to-ground-inverse
  unfolding to-ground-def is-ground-def
  by fastforce

lemma from-ground-empty' [simp]: from-ground expr = empty  $\longleftrightarrow$  expr = empty-ground
  using from-ground-empty from-ground-eq
  unfolding from-ground-def
  by auto

```

end

locale *exists-ground-subst-lifting* =
 substitution-lifting +
 sub: *exists-ground-subst* **where** *vars* = *sub-vars* **and** *subst* = *sub-subst* **and**
 is-ground = *sub-is-ground*
begin

sublocale *exists-ground-subst* **where** *vars* = *vars* **and** *subst* = *subst* **and** *is-ground*
= *is-ground*
proof *unfold-locales*

show $\exists \gamma. \textit{is-ground-subst } \gamma$
 by (*simp add: sub.exists-ground-subst*)
qed

end

locale *subst-updates-compat-lifting* =
 subst-update-lifting +
 sub: *subst-updates-compat* **where**
 vars = *sub-vars* **and** *subst* = *sub-subst* **and** *is-ground* = *sub-is-ground*
begin

sublocale *subst-updates-compat* **where** *vars* = *vars* **and** *subst* = *subst* **and** *is-ground*
= *is-ground*
proof *unfold-locales*
 fix *u expr* σ

assume $\forall x \in \textit{vars} \textit{expr}. \textit{fmlookup } u \ x = \textit{Some } (x \cdot v \ \sigma)$

then have $\forall \textit{sub} \in \textit{to-set } \textit{expr}. \textit{sub} \cdot_s \textit{id-subst}[[u]] = \textit{sub} \cdot_s \sigma$
 unfolding *vars-def*
 by (*simp add: sub.subst-updates-compat(1)*)

then show $\textit{expr} \cdot \textit{id-subst}[[u]] = \textit{expr} \cdot \sigma$
 unfolding *subst-def vars-def*
 by *simp*

qed

end

locale *subst-eq-lifting* =
 sub: *subst-eq* **where**
 vars = *sub-vars* :: '*sub* \Rightarrow '*v set* **and** *subst* = *sub-subst* **and** *is-ground* = *sub-is-ground*
 +
 substitution-lifting

```

begin

sublocale subst-eq where vars = vars and subst = subst and is-ground = is-ground
proof unfold-locales
  fix expr  $\sigma$   $\tau$ 
  assume  $\bigwedge x. x \in \text{vars } \text{expr} \implies x \cdot v \sigma = x \cdot v \tau$ 

  then show  $\text{expr} \cdot \sigma = \text{expr} \cdot \tau$ 
    unfolding vars-def subst-def
    using sub.subst-eq
    by (meson UN-I local.map-cong)
qed

end

locale is-ground-if-no-vars-lifting =
  substitution-lifting +
  sub: is-ground-if-no-vars where
    vars = sub-vars and is-ground = sub-is-ground and subst = sub-subst
begin

sublocale is-ground-if-no-vars where vars = vars and is-ground = is-ground and
subst = subst
  by unfold-locales (simp add: is-ground-def sub.is-ground-if-no-vars vars-def)

end

end
theory Based-Substitution-Lifting
imports
  Based-Substitution
  Substitution-Lifting
begin

```

12 Lifting of based substitutions

```

locale based-substitution-lifting =
  substitution-lifting +
  base: base-substitution where
    subst = base-subst and vars = base-vars and is-ground = base-is-ground +
    sub: based-substitution where
      subst = sub-subst and vars = sub-vars and is-ground = sub-is-ground
begin

sublocale based-substitution where subst = subst and vars = vars and is-ground
= is-ground
proof unfold-locales
  fix expr  $\varrho$ 

```

```

show vars (expr · ρ) = ⋃ (base-vars ‘ (λx. x · v ρ) ‘ vars expr)
  using sub.vars-subst
  unfolding subst-def vars-def
  by simp
next
  fix expr γ
  assume is-ground (expr · γ)

  then show ∀ x ∈ vars expr. base-is-ground (x · v γ)
    unfolding is-ground-def vars-def
    using sub.variable-grounding
    by auto
next

  show exists-nonground = base.exists-nonground
    by (metis sub.exists-nonground-iff-base-exists-nonground
      exists-nonground-iff-sub-exists-nonground)
qed

end

```

12.1 Lifting of properties

```

locale variables-in-base-imgu-lifting =
  based-substitution-lifting +
  sub: variables-in-base-imgu where
  vars = sub-vars and subst = sub-subst and is-ground = sub-is-ground
begin

sublocale variables-in-base-imgu where subst = subst and vars = vars and
is-ground = is-ground
proof unfold-locales
  fix expr μ XX
  assume imgu: base.is-imgu μ XX finite XX ∀ X ∈ XX. finite X

  show vars (expr · μ) ⊆ vars expr ∪ ⋃ (base-vars ‘ ⋃ XX)
    using sub.variables-in-base-imgu[OF imgu]
    unfolding vars-def subst-def to-set-map
    by auto
qed

end

locale based-subst-update-lifting =
  based-substitution-lifting +
  subst-update-lifting +
  sub: based-subst-update where
  vars = sub-vars and subst = sub-subst and is-ground = sub-is-ground
begin

```

```

sublocale based-subst-update where subst = subst and vars = vars and is-ground
= is-ground
proof unfold-locales
  fix update expr  $\gamma$  x
  assume base-is-ground update is-ground (expr  $\cdot$   $\gamma$ ) x  $\in$  vars expr

  then show is-ground (expr  $\cdot$   $\gamma$ [[x := update]])
    using sub.ground-subst-update-in-vars
    unfolding is-ground-def subst-def
    by auto
qed

end

locale vars-grounded-iff-is-grounding-lifting =
  based-substitution-lifting +
  sub: vars-grounded-iff-is-grounding where
  vars = sub-vars and subst = sub-subst and is-ground = sub-is-ground
begin

sublocale vars-grounded-iff-is-grounding where
  subst = subst and vars = vars and is-ground = is-ground
  using sub.is-grounding-if-vars-grounded
  by unfold-locales (simp add: vars-def is-ground-def subst-def)

end

end
theory Noop-Substitution
  imports Based-Substitution
begin

definition noop-comp-subst where
  noop-comp-subst  $\sigma$  - =  $\sigma$ 

definition noop-id-subst where
  noop-id-subst = ()

global-interpretation unit: abstract-substitution-monoid where
  comp-subst = noop-comp-subst and id-subst = noop-id-subst
  by unfold-locales auto

locale properties =
  base-substitution +
  all-subst-ident-iff-ground +
  exists-non-ident-subst +
  subst-update +
  grounding +

```

finite-variables +
renaming-variables +
exists-ground +
range-vars-subset-if-is-imgu +
exists-imgu +
 create-renaming **where** *base-subst* = *subst* **and** *base-vars* = *vars* **and** *base-is-ground*
 = *is-ground*

definition *noop-apply-subst* **where**
noop-apply-subst - - \equiv *SOME* *expr*. *True*

definition *noop-subst* **where**
noop-subst *expr* - \equiv *expr*

definition *noop-vars* **where**
noop-vars - \equiv {}

definition *noop-is-ground* **where**
noop-is-ground - \longleftrightarrow *True*

definition *noop-subst-update* **where**
noop-subst-update σ - - \equiv σ

context *abstract-substitution-monoid*
begin

sublocale *noop: base-substitution* **where**
vars = *noop-vars* **and** *apply-subst* = *noop-apply-subst* **and** *is-ground* = *noop-is-ground*
and
subst = *noop-subst*
by
unfold-locales
 (auto *simp*: *noop-vars-def* *noop-subst-update-def* *noop-apply-subst-def* *noop-subst-def*
noop-is-ground-def)

sublocale *noop: properties* **where**
vars = *noop-vars* **and** *subst-update* = *noop-subst-update* **and** *apply-subst* =
noop-apply-subst **and**
subst = *noop-subst* **and** *to-ground* = *id* **and** *from-ground* = *id* **and** *is-ground* =
noop-is-ground
by
unfold-locales
 (auto *simp*: *noop-vars-def* *noop.range-vars-def* *noop.subst-domain-def* *noop-is-ground-def*
noop-subst-update-def)

lemma *noop-subst [simp]*: *noop-subst* *expr* σ = *expr*
by (*simp add*: *noop-subst-def*)

lemma *noop-is-ground* [*simp*]: *noop-is-ground* *expr*
using *noop-is-ground-def*
by *fast*

lemma *noop-vars* [*simp*]: *noop-vars* *expr* = {}
unfolding *noop-vars-def*
by *blast*

end

end

theory *Substitution-First-Order-Term*

imports

Based-Substitution

First-Order-Terms.Unification

Regular-Tree-Relations.Ground-Terms

begin

13 Substitutions for first order terms

13.1 Interpretations for first order terms

abbreviation (*input*) *apply-subst* **where**
apply-subst $x \sigma \equiv \sigma x$

abbreviation (*input*) *is-ground* **where**
is-ground $t \equiv \text{vars-term } t = \{\}$

global-interpretation *term: base-substitution* **where**
comp-subst = (\circ_s) **and** *id-subst* = *Var* **and** *subst* = (\cdot) **and** *vars* = *vars-term*
and

apply-subst = *apply-subst* **and** *is-ground* = *is-ground*

proof *unfold-locales*

fix $x :: 'v$

show *vars-term* (*Var* x) = { x }
by *simp*

next

fix $\sigma \sigma' :: ('f, 'v) \text{ subst}$ **and** x

show $(\sigma \circ_s \sigma') x = \sigma x \cdot \sigma'$
unfolding *subst-compose-def* ..

next

fix $t :: ('f, 'v) \text{ term}$ **and** $\rho :: ('f, 'v) \text{ subst}$

show *vars-term* $(t \cdot \rho) = \bigcup (\text{vars-term } ' \rho ' \text{vars-term } t)$
using *vars-term-subst* .

next

```

fix t :: ('f, 'v) term

assume vars-term t = {}

then show  $\forall \sigma. t \cdot \sigma = t$ 
  by (simp add: ground-term-subst)
next
fix t and  $\gamma :: ('f, 'v) \text{ subst}$ 
assume is-ground (t ·  $\gamma$ )

then show  $\forall x \in \text{vars-term } t. \text{is-ground } (\gamma x)$ 
  by (meson ground-substD ground-vars-term-empty)
qed auto

lemma term-is-renaming-iff: term.is-renaming  $\varrho \iff \text{inj } \varrho \wedge (\forall x. \exists x'. \varrho x = \text{Var } x')$ 
  unfolding term.is-renaming-def
  by (metis eval-subst-def ex-inverse-of-renaming injI is-Var-def subst-apply-eq-Var term.inject(1))

locale term-properties =
  base-substitution +
  all-subst-ident-iff-ground +
  exists-non-ident-subst +
  grounding +
  finite-variables +
  renaming-variables +
  exists-ground +
  create-renaming where
  base-vars = vars and base-subst = subst and base-is-ground = is-ground

global-interpretation term: term-properties where
  comp-subst = ( $\circ_s$ ) and id-subst = Var and subst = ( $\cdot$ ) and subst-update =
  fun-upd and
  apply-subst = apply-subst and vars = vars-term and to-ground = gterm-of-term
and
  from-ground = term-of-gterm and is-ground = is-ground
proof unfold-locales
  fix t :: ('f, 'v) term

  show finite (vars-term t)
    by simp
next
fix t :: ('f, 'v) term

  show is-ground t  $\iff (\forall \sigma. t \cdot \sigma = t)$ 
  proof(induction t)
    case Var
    then show ?case

```

```

    by auto
  next
  case Fun

    moreover have  $\bigwedge xs\ x\ \sigma. \forall \sigma. \text{map } (\lambda s. s \cdot \sigma)\ xs = xs \implies x \in \text{set } xs \implies x \cdot$ 
 $\sigma = x$ 
    by (metis list.map-ident map-eq-conv)

    ultimately show ?case
      by (auto simp: map-idI)
    qed
  next
  fix  $t :: ('f, 'v)\ \text{term}$  and  $ts :: ('f, 'v)\ \text{term set}$ 

  assume finite ts  $\neg$ is-ground t

  then show  $\exists \sigma. t \cdot \sigma \neq t \wedge t \cdot \sigma \notin ts$ 
  proof (induction t arbitrary: ts)
    case (Var x)

    obtain  $t'$  where  $t': t' \notin ts$  is-Fun t'
      using Var.prem1 finite-list
      by blast

    define  $\sigma :: ('f, 'v)\ \text{subst}$  where  $\bigwedge x. \sigma\ x = t'$ 

    have  $\text{Var } x \cdot \sigma \neq \text{Var } x$ 
      using  $t'$ 
      unfolding  $\sigma$ -def
      by auto

    moreover have  $\text{Var } x \cdot \sigma \notin ts$ 
      using  $t'$ 
      unfolding  $\sigma$ -def
      by simp

    ultimately show ?case
      using Var
      by blast
  next
  case (Fun f args)

  obtain  $a$  where  $a: a \in \text{set } args$  and  $a$ -vars: vars-term a  $\neq \{\}$ 
    using Fun.prem1
    by fastforce

  then obtain  $\sigma$  where
     $\sigma: a \cdot \sigma \neq a$  and
     $a$ - $\sigma$ -not-in-args: a  $\cdot \sigma \notin \bigcup (\text{set } 'args\ 'ts)$ 

```

```

    by (metis Fun.IH Fun.prem1 List.finite-set finite-UN finite-imageI)

  then have  $Fun\ f\ args \cdot \sigma \neq Fun\ f\ args$ 
    using a map-eq-conv
    by fastforce

  moreover have  $Fun\ f\ args \cdot \sigma \notin ts$ 
    using a a- $\sigma$ -not-in-args
    by auto

  ultimately show ?case
    using Fun
    by blast
qed
next
{
  fix  $t :: ('f, 'v)\ term$ 
  assume  $t\text{-is-ground}: is-ground\ t$ 

  have  $\exists g. term-of-gterm\ g = t$ 
  proof(intro exI)

    from  $t\text{-is-ground}$ 
    show  $term-of-gterm\ (gterm-of-term\ t) = t$ 
    by (induction t) (simp-all add: map-idI)

  qed
}

then show  $\{t :: ('f, 'v)\ term. is-ground\ t\} = range\ term-of-gterm$ 
  by fastforce
next
fix  $t_G :: 'f\ gterm$ 

show  $gterm-of-term\ (term-of-gterm\ t_G) = t_G$ 
  by simp
next
fix  $\rho :: ('f, 'v)\ subst$  and  $t$ 

assume  $\rho: term.is-renaming\ \rho$ 

then show  $inj\ \rho \wedge (\forall x. \exists x'. \rho\ x = Var\ x')$ 
  unfolding term-is-renaming-iff .

show  $vars-term\ (t \cdot \rho) = term.rename\ \rho\ 'vars-term\ t$ 
proof (induction t)
  case (Var x)

  have  $\rho\ x = Var\ (term.rename\ \rho\ x)$ 

```

```

    using  $\varrho$ 
    unfolding term-is-renaming-iff term.rename-def[OF  $\varrho$ ] is-Var-def
    by (meson someI-ex)

    then show ?case
      by auto
  next
    case (Fun f ts)

    then show ?case
      by auto
  qed
next

  show  $\exists t. \text{is-ground } t$ 
    by (metis vars-term-of-gterm)
next
  fix x update and t :: ('f, 'v) term and  $\sigma$  :: ('f, 'v) subst
  assume  $x \notin \text{vars-term } t$ 

  then show  $t \cdot \sigma(x := \text{update}) = t \cdot \sigma$ 
    by (simp add: eval-with-fresh-var)
next
  fix  $\gamma$  :: ('f, 'v) subst and t u :: ('f, 'v) term and x
  assume vars-term u = {} vars-term (t ·  $\gamma$ ) = {}  $x \in \text{vars-term } t$ 

  then show vars-term (t ·  $\gamma(x := u)$ ) = {}
    by (metis fun-upd-apply ground-subst ground-vars-term-empty)
  qed auto

lemma exists-nonground-term [intro]:  $\exists t. \text{vars-term } t \neq \{\}$ 
  using term.set-intros(3)
  by fastforce

```

lemmas term-id-subst-rewrite = term.id-subst-rewrite[OF exists-nonground-term]

13.2 Compatibility with First_Order_Term

Prefer `term.subst-id-subst` to `subst-apply-term-empty`.

```
declare subst-apply-term-empty[no-atp]
```

lemmas term-context-ground-iff-term-is-ground [simp] = ground-vars-term-empty

The other direction does not hold!

lemma Term-is-renaming-if-is-renaming:

```

  assumes term.is-renaming  $\varrho$ 
  shows Term.is-renaming  $\varrho$ 
  using assms
  unfolding is-renaming-def term-is-renaming-iff

```

by (metis inj-on-subset is-VarI top-greatest)

lemma *Term-subst-domain-eq-subst-domain* [simp]: $\text{Term.subst-domain } \sigma = \text{term.subst-domain } \sigma$
by (simp add: subst-domain-def term.subst-domain-def)

lemma *Term-range-vars-eq-range-vars* [simp]: $\text{Term.range-vars } \sigma = \text{term.range-vars } \sigma$
by (simp add: range-vars-def term.range-vars-def)

lemma *Unifiers-unifiers-Times-iff-is-unifier*:
 $\mu \in \text{Unifiers.unifiers } (X \times X) \longleftrightarrow \text{term.is-unifier } \mu X$
unfolding *term.is-unifier-iff unifiers-def*
by simp

lemma *Unifiers-unifiers-Union-Times-iff-is-unifier-set*:
 $\mu \in \text{Unifiers.unifiers } (\bigcup X \in XX. X \times X) \longleftrightarrow \text{term.is-unifier-set } \mu XX$
using *Unifiers-unifiers-Times-iff-is-unifier*
unfolding *term.is-unifier-set-def unifiers-def*
by fast

lemma *Unifiers-is-mgu-Union-Times-iff-is-mgu*:
 $\text{Unifiers.is-mgu } \mu (\bigcup X \in XX. X \times X) \longleftrightarrow \text{term.is-mgu } \mu XX$
unfolding *term.is-mgu-def is-mgu-def*
using *Unifiers-unifiers-Union-Times-iff-is-unifier-set*
by (metis (mono-tags, lifting))

lemma *Unifiers-is-imgu-Union-Times-iff-is-imgu*:
 $\text{Unifiers.is-imgu } \mu (\bigcup X \in XX. X \times X) \longleftrightarrow \text{term.is-imgu } \mu XX$
unfolding *term.is-imgu-def is-imgu-def*
using *Unifiers-unifiers-Union-Times-iff-is-unifier-set*
by auto

lemma *Unifiers-is-mgu-iff-is-imgu-image-set-prod*:
fixes $\mu :: ('f, 'v) \text{ subst}$ **and** $X :: (('f, 'v) \text{ term} \times ('f, 'v) \text{ term}) \text{ set}$
shows $\text{Unifiers.is-imgu } \mu X \longleftrightarrow \text{term.is-imgu } \mu (\text{set-prod } 'X)$
proof (rule iffI)
assume $\text{is-imgu } \mu X$

moreover then have

$\forall e \in X. \text{fst } e \cdot \mu = \text{snd } e \cdot \mu$
 $\forall \tau :: ('f, 'v) \text{ subst}. (\forall e \in X. \text{fst } e \cdot \tau = \text{snd } e \cdot \tau) \longrightarrow \tau = \mu \circ_s \tau$
unfolding *is-imgu-def unifiers-def*
by auto

moreover then have

$\bigwedge \tau :: ('f, 'v) \text{ subst}. \forall e \in X. \forall t t'. e = (t, t') \longrightarrow \text{card } \{t \cdot \tau, t' \cdot \tau\} \leq \text{Suc } 0$
 $\implies \mu \circ_s \tau = \tau$
by (metis Suc-n-not-le-n card-1-singleton-iff card-Suc-eq insert-iff prod.collapse)

```

ultimately show term.is-imgu  $\mu$  (set-prod ' X)
  unfolding term.is-imgu-def term.is-unifier-set-def term.is-unifier-def
  by (auto split: prod.splits)
next
assume is-imgu: term.is-imgu  $\mu$  (set-prod ' X)

show is-imgu  $\mu$  X
proof (unfold is-imgu-def unifiers-def, intro conjI ballI)

  show  $\mu \in \{\sigma. \forall e \in X. fst\ e \cdot \sigma = snd\ e \cdot \sigma\}$ 
  using term.is-imgu-unifies[OF is-imgu]
  by fastforce
next
fix  $\tau :: ('f, 'v)\ subst$ 
assume  $\tau \in \{\sigma. \forall e \in X. fst\ e \cdot \sigma = snd\ e \cdot \sigma\}$ 

then have  $\forall e \in X. fst\ e \cdot \tau = snd\ e \cdot \tau$ 
  by blast

then show  $\tau = \mu \circ_s \tau$ 
  using is-imgu
  unfolding term.is-imgu-def term.is-unifier-set-def
  by fastforce
qed
qed

```

13.3 Interpretations for IMGUs

We could also use *base-variables-in-base-imgu*, but would then require infinite variables.

```

global-interpretation term: range-vars-subset-if-is-imgu where
  comp-subst = ( $\circ_s$ ) and id-subst = Var and subst = ( $\cdot$ ) and
  apply-subst = apply-subst and vars = vars-term and is-ground = is-ground
proof unfold-locales
fix  $\mu :: ('f, 'v)\ subst$  and XX

assume imgu: term.is-imgu  $\mu$  XX and finite:  $\forall X \in XX. finite\ X\ finite\ XX$ 

then have is-imgu: Unifiers.is-imgu  $\mu$  ( $\bigcup X \in XX. X \times X$ )
  using Unifiers.is-imgu-Union-Times-iff-is-imgu[of - XX]
  by simp

have finite-prod: finite ( $\bigcup X \in XX. X \times X$ )
  using finite
  by blast

have ( $\bigcup e \in \bigcup X \in XX. X \times X. vars-term\ (fst\ e) \cup vars-term\ (snd\ e)$ ) = ( $\bigcup t \in \bigcup XX. vars-term\ t$ )

```

```

    by fastforce

  then show term.range-vars  $\mu \subseteq \bigcup (vars-term \cup XX)$ 
    using imgu-range-vars-subset[OF is-imgu finite-prod]
    by simp
qed

global-interpretation term: exists-imgu where
  comp-subst = ( $\circ_s$ ) and id-subst = Var and subst = ( $\cdot$ ) and
  apply-subst = apply-subst and vars = vars-term and is-ground = is-ground
proof unfold-locales
  fix v :: ('f, 'v) subst and t t'
  assume unifier:  $t \cdot v = t' \cdot v$ 

  show  $\exists \mu. term.is-imgu \mu \{\{t, t'\}\}$ 
  proof (rule exI)

    show term.is-imgu (the-mgu t t')  $\{\{t, t'\}\}$ 
    using the-mgu-is-imgu unifier
    unfolding Unifiers-is-mgu-iff-is-imgu-image-set-prod
    by auto
  qed
qed

```

13.4 Additional lemmas

```

lemmas infinite-terms [intro] = term.infinite-exprs[OF exists-nonground-term]

lemma is-renaming-iff-ex-inj-fun-on-vars: term.is-renaming  $\rho \iff (\exists f. inj f \wedge \rho = Var \circ f)$ 
  unfolding term-is-renaming-iff
  by (metis (no-types, lifting) ext comp-def inj-compose inj-on-Var inj-on-imageI2)

end

theory Substitution-HOL-ex-Unification
imports
  Based-Substitution
  HOL-ex.Unification
  Fresh-Identifiers.Fresh
begin

```

14 Substitutions for first order terms as binary tree

```

no-notation Comb (infix  $\langle \cdot \rangle$  60)

quotient-type 'v subst = ('v  $\times$  'v trm) list / ( $\doteq$ )
proof (rule equivI)

```

```

  show reflp ( $\doteq$ )
    using reflpI subst-refl by metis
next
  show symp ( $\doteq$ )
    using sympI subst-sym by metis
next
  show transp ( $\doteq$ )
    using transpI subst-trans by metis
qed

```

14.1 Substitution monoid

lift-definition $subst\text{-}comp :: 'v\ subst \Rightarrow 'v\ subst \Rightarrow 'v\ subst$ (**infixl** $\langle \odot \rangle$ 67)
 is *Unification.comp*
 using *Unification.subst-cong* .

lift-definition $subst\text{-}id :: 'v\ subst$
 is \square .

global-interpretation $subst\text{-}comp$: monoid $subst\text{-}comp$ $subst\text{-}id$

proof *unfold-locales*

fix $\sigma\ \sigma'\ \sigma'' :: 'v\ subst$

show $\sigma \odot \sigma' \odot \sigma'' = \sigma \odot (\sigma' \odot \sigma'')$
 by *transfer auto*

next

fix $\sigma :: 'v\ subst$

show $subst\text{-}id \odot \sigma = \sigma$
 by *transfer simp*

show $\sigma \odot subst\text{-}id = \sigma$
 by *transfer simp*

qed

14.2 Transfer definitions and lemmas from HOL-ex.Unification

lift-definition $subst\text{-}apply :: 'v\ trm \Rightarrow 'v\ subst \Rightarrow 'v\ trm$ (**infixl** $\langle \cdot \rangle$ 61)
 is *Unification.subst*
 using *Unification.subst-eq-dest* .

lift-definition $subst\text{-}domain :: 'v\ subst \Rightarrow 'v\ set$
 is *Unification.subst-domain*
 by (*metis (no-types, lifting) ext subst-domain-def subst-eq-def*)

lift-definition $range\text{-}vars :: 'v\ subst \Rightarrow 'v\ set$
 is *Unification.range-vars*
 by (*metis (no-types, lifting) ext range-vars-def subst-eq-def*)

lift-definition *Unifier* :: 'v subst \Rightarrow 'v trm \Rightarrow 'v trm \Rightarrow bool
is *Unification.Unifier*
unfolding *subst-eq-def Unifier-def*
by *auto*

lift-definition *IMGU* :: 'v subst \Rightarrow 'v trm \Rightarrow 'v trm \Rightarrow bool
is *Unification.IMGU*
unfolding *subst-eq-def IMGU-def*
by (*simp add: Unification.Unifier-def*)

lift-definition *unify* :: 'v trm \Rightarrow 'v trm \Rightarrow 'v subst option
is *Unification.unify* .

lemma *unify-eq-Some-if-Unifier*:
assumes *Unifier* σ t t'
shows $\exists \tau. \text{unify } t \ t' = \text{Some } \tau$

proof –

obtain *rep- σ* **where** σ -def: $\sigma = \text{abs-subst } \text{rep-}\sigma$
by *transfer simp*

from *assms* **have** *Unification.Unifier* *rep- σ* t t'
unfolding σ -def
by *transfer*

then obtain *rep- τ* **where** *Unification.unify* t $t' = \text{Some } \text{rep-}\tau$
using *Unification.unify-eq-Some-if-Unifier*
by *blast*

then have *unify* t $t' = \text{Some } (\text{abs-subst } \text{rep-}\tau)$
by (*simp add: unify.abs-eq*)

thus *?thesis*
by *blast*

qed

lemma *unify-computes-IMGU*:
assumes *unify* t $t' = \text{Some } \sigma$
shows *IMGU* σ t t'

proof –

obtain *rep- σ* **where** σ -def: $\sigma = \text{abs-subst } \text{rep-}\sigma$
by *transfer simp*

have *map-option abs-subst* (*Unification.unify* t t') = *Some* (*abs-subst rep- σ*)
using *assms* σ -def
by (*simp add: unify.abs-eq*)

then obtain *rep- σ'* **where**
Unification.unify t $t' = \text{Some } \text{rep-}\sigma'$ **and**

```

    rep-σ': abs-subst rep-σ' = abs-subst rep-σ
  by (cases Unification.unify t t') auto

  then have Unification.IMGU rep-σ' t t'
    using Unification.unify-computes-IMGU
    by blast

  hence IMGU (abs-subst rep-σ') t t'
    by transfer

  thus IMGU σ t t'
    using rep-σ' σ-def
    by simp
qed

lift-definition subst-update :: 'v × 'v trm ⇒ 'v subst ⇒ 'v subst
  is (#)
proof -
  fix update and σ σ' :: ('v × 'v trm) list
  assume subst-eq: σ ≐ σ'

  {
    fix t
    have t < update # σ = t < update # σ'
    proof (induction t)
      case Var
        then show ?case
          using subst-eq
          unfolding subst-eq-def
          by (metis assoc.simps(2) prod.exhaust subst.simps(1))
    qed simp-all
  }

  then show update # σ ≐ update # σ'
    unfolding subst-eq-def
    by satx
qed

abbreviation (input) is-ground where
  is-ground t ≡ vars-of t = {}

```

14.3 Base Substitution

```

global-interpretation term: base-substitution where
  comp-subst = subst-comp and id-subst = subst-id and subst = subst-apply and
  vars = vars-of and
  apply-subst = λx σ. subst-apply (Var x) σ and is-ground = is-ground
proof unfold-locales

```

```

fix  $t$  and  $\sigma \sigma' :: 'v \text{ subst}$ 

show  $t \cdot \sigma \odot \sigma' = t \cdot \sigma \cdot \sigma'$ 
  by transfer simp
next
fix  $t :: 'v \text{ trm}$ 

show  $t \cdot \text{subst-id} = t$ 
  by transfer simp
next
fix  $t :: 'v \text{ trm}$ 

assume  $\text{vars-of } t = \{\}$ 

then show  $\forall \sigma. t \cdot \sigma = t$ 
  by transfer (metis agreement empty-iff subst-Nil)
next
fix  $x :: 'v$ 

show  $\text{vars-of } (\text{Var } x \cdot \text{subst-id}) = \{x\}$ 
  by transfer simp
next
fix  $\sigma \sigma' :: 'v \text{ subst and } x$ 

show  $\text{Var } x \cdot \sigma \odot \sigma' = \text{Var } x \cdot \sigma \cdot \sigma'$ 
  by transfer simp
next
fix  $t$  and  $\sigma :: 'v \text{ subst}$ 

show  $\text{vars-of } (t \cdot \sigma) = \bigcup (\text{vars-of } (\lambda x. \text{Var } x \cdot \sigma) \text{ ` vars-of } t)$ 
  by transfer (simp add: vars-of-subst-conv-Union)
next
fix  $t$  and  $\gamma :: 'v \text{ subst}$ 
assume is-ground  $(t \cdot \gamma)$ 

then show  $\forall x \in \text{vars-of } t. \text{is-ground } (\text{Var } x \cdot \gamma)$ 
  by transfer (metis occs-vars-subset subset-empty subst-mono vars-iff-occseq)
qed simp

```

14.4 Substitution Properties

global-interpretation *term: create-renaming where*
comp-subst = subst-comp and id-subst = subst-id and subst = subst-apply and
vars = vars-of and
is-ground = is-ground and apply-subst = $\lambda x \sigma. \text{subst-apply } (\text{Var } x) \sigma$ and
subst-update = $\lambda \sigma x \text{update}. \text{subst-update } (x, \text{update}) \sigma$ and
base-subst = subst-apply and base-vars = vars-of and base-is-ground = is-ground
proof *unfold-locales*
fix σx **and** $\text{update} :: 'v \text{ trm}$

```

show  $Var\ x \cdot subst\text{-}update\ (x, update)\ \sigma = update$ 
  by transfer simp
next
fix  $\sigma :: 'v\ subst$  and  $x\ y :: 'v$  and  $update$ 
assume  $x \neq y$ 

then show  $Var\ x \cdot subst\text{-}update\ (y, update)\ \sigma = Var\ x \cdot \sigma$ 
  by transfer simp
next
fix  $x :: 'v$  and  $t\ update$  and  $\sigma :: 'v\ subst$ 
assume  $x \notin vars\text{-}of\ t$ 

then show  $t \cdot subst\text{-}update\ (x, update)\ \sigma = t \cdot \sigma$ 
  by transfer (simp add: repl-invariance)
next
fix  $\sigma :: 'v\ subst$  and  $x$ 

show  $subst\text{-}update\ (x, Var\ x \cdot \sigma)\ \sigma = \sigma$ 
  by transfer (simp add: agreement subst-eq-intro)
next
fix  $\sigma :: 'v\ subst$  and  $x\ a\ b$ 

show  $subst\text{-}update\ (x, b)(subst\text{-}update\ (x, a)\ \sigma) = subst\text{-}update\ (x, b)\ \sigma$ 
  by transfer (simp add: agreement subst-eq-def)
next
fix  $\gamma :: 'v\ subst$  and  $u\ t :: 'v\ trm$  and  $x$ 
assume is-ground  $u$  is-ground  $(t \cdot \gamma)$   $x \in vars\text{-}of\ t$ 

then show is-ground  $(t \cdot subst\text{-}update\ (x, u)\ \gamma)$ 
proof (induction  $t$ )
  case ( $Var\ x'$ )

    then show ?case
      by transfer simp
next
case ( $Const\ x'$ )

    then show ?case
      by transfer simp
next
case ( $Comb\ t1\ t2$ )

    then show ?case
      by transfer (fastforce simp: repl-invariance)
qed
next
fix  $us\ us' :: ('v \times 'v\ trm)\ list$ 

```

let $?upd = \lambda(x, b). \text{subst-update } (x, b)$
assume $\bigwedge x. \text{Var } x \cdot \text{fold } ?upd \text{ us } \text{subst-id} \odot \text{fold } ?upd \text{ us}' \text{ subst-id} = \text{Var } x \cdot \text{subst-id}$
then show $\text{fold } ?upd \text{ us } \text{subst-id} \odot \text{fold } ?upd \text{ us}' \text{ subst-id} = \text{subst-id}$
by transfer (use agreement in blast)
qed

global-interpretation term: finite-variables where
 $\text{comp-subst} = \text{subst-comp}$ **and** $\text{id-subst} = \text{subst-id}$ **and** $\text{subst} = \text{subst-apply}$ **and**
 $\text{vars} = \text{vars-of}$ **and**
 $\text{apply-subst} = \lambda x \sigma. \text{subst-apply } (\text{Var } x) \sigma$ **and** $\text{is-ground} = \text{is-ground}$
proof *unfold-locales*
fix $t :: 'v \text{ trm}$

show $\text{finite } (\text{vars-of } t)$
by *blast*
qed

We could also prove *all-subst-ident-iff-ground* and *exists-non-ident-subst* directly without needing infinite variables.

global-interpretation term: base-exists-non-ident-subst where
 $\text{comp-subst} = \text{subst-comp}$ **and** $\text{id-subst} = \text{subst-id} :: 'v :: \text{infinite subst}$ **and**
 $\text{subst} = \text{subst-apply}$ **and** $\text{vars} = \text{vars-of}$ **and** $\text{is-ground} = \text{is-ground}$ **and**
 $\text{subst-update} = \lambda \sigma x \text{update}. \text{subst-update } (x, \text{update}) \sigma$ **and**
 $\text{apply-subst} = \lambda x \sigma. \text{subst-apply } (\text{Var } x) \sigma$
proof *unfold-locales*
fix $t :: 'v \text{ trm}$

show $\text{is-ground } t = (\forall \sigma. t \cdot \sigma = t)$
by transfer (*metis agreement all-not-in-conv remove-var subst-Nil vars-of.simps(2)*)

next

show $\text{infinite } (\text{UNIV} :: 'v \text{ set})$
using *infinite-UNIV*
by *simp*
qed

global-interpretation term: renaming-variables where
 $\text{comp-subst} = \text{subst-comp}$ **and** $\text{id-subst} = \text{subst-id} :: 'v \text{ subst}$ **and**
 $\text{subst} = \text{subst-apply}$ **and** $\text{vars} = \text{vars-of}$ **and** $\text{is-ground} = \text{is-ground}$ **and**
 $\text{apply-subst} = \lambda x \sigma. \text{subst-apply } (\text{Var } x) \sigma$
proof (*unfold-locales, intro conjI*)
fix $\rho :: 'v \text{ subst}$ **and** t
assume $\rho: \text{term.is-renaming } \rho$

then show $\text{inj } (\lambda x. \text{Var } x \cdot \rho)$

```

unfolding inj-def term.is-renaming-def
by (metis term.subst-inv trm.inject(1))

show rename:  $\forall x. \exists x'. \text{Var } x \cdot \varrho = \text{Var } x' \cdot \text{subst-id}$ 
using  $\varrho$  term.subst-inv term.comp-subst-iff
unfolding term.is-renaming-def subst-apply.rep-eq
by (metis subst-apply-eq-Var)

show vars-of (t ·  $\varrho$ ) = term.rename  $\varrho$  ‘ vars-of t
proof (induction t)
case (Var y)

have Var y ·  $\varrho = \text{Var}$  (term.rename  $\varrho$  y)
using rename someI-ex[of  $\lambda x'. \text{Var } y \cdot \varrho = \text{Var } x' \cdot \text{subst-id}$ ]
unfolding term.rename-def[OF  $\varrho$ ]
by auto

then show ?case
by simp
next
case (Const c)

then show ?case
by auto
next
case (Comb t1 t2)

then show ?case
by (simp add: image-Un subst-apply.rep-eq)
qed
qed

```

14.5 Compatibility with HOL-ex.Unification

lemma subst-domain-eq-term-subst-domain [simp]: subst-domain $\sigma = \text{term.subst-domain } \sigma$

```

unfolding term.subst-domain-def
by transfer (simp add: Unification.subst-domain-def)

```

lemma range-vars-eq-term-range-vars [simp]: range-vars $\sigma = \text{term.range-vars } \sigma$

```

unfolding term.range-vars-def subst-domain-eq-term-subst-domain[symmetric]
by transfer (auto simp: Unification.range-vars-def Unification.subst-domain-def)

```

lemma Unifier-iff-term-is-unifier:

$\text{Unifier } \mu \text{ } t \text{ } t' \longleftrightarrow \text{term.is-unifier } \mu \{t, t'\}$

```

unfolding term.is-unifier-def term.subst-set-def
by transfer (use Unification.Unifier-def subset-singleton-iff in fastforce)

```

lemma Unifier-iff-is-unifier-set:

Unifier $\mu t t' \longleftrightarrow \text{term.is-unifier-set } \mu \{\{t, t'\}\}$
using *Unifier-iff-term-is-unifier*
unfolding *term.is-unifier-set-def*
by *auto*

lemma *IMGU-iff-term-is-mgu*:
IMGU $\mu t t' \longleftrightarrow \text{term.is-imgu } \mu \{\{t, t'\}\}$
unfolding *term.is-imgu-def Unifier-iff-is-unifier-set[symmetric]*
by *transfer (meson Unification.IMGU-def subst-sym)*

lemma *IMGU-range-vars-subset*:
assumes *IMGU* $\mu t u$
shows *range-vars* $\mu \subseteq \text{vars-of } t \cup \text{vars-of } u$
using *assms*
by *transfer (rule IMGU-range-vars-subset)*

14.6 Interpretations for IMGUs

We could also prove *range-vars-subset-if-is-imgu* without needing infinite variables.

global-interpretation *term: base-variables-in-base-imgu where*
comp-subst = subst-comp and id-subst = subst-id :: 'v :: infinite subst and
subst = subst-apply and vars = vars-of and is-ground = is-ground and
subst-update = $\lambda \sigma x$ update. subst-update (x, update) σ and
apply-subst = $\lambda x \sigma$. subst-apply (Var x) σ
proof(*unfold-locales, transfer*)
fix $t :: 'v \text{ trm}$ **and** σ

show $\exists x. t \triangleleft \sigma = \text{Var } x \triangleleft [] \implies \exists x. t = \text{Var } x \triangleleft []$
using *subst-apply-eq-Var*
by *fastforce*

qed

global-interpretation *term: exists-imgu where*
comp-subst = subst-comp and id-subst = subst-id and subst = subst-apply and
vars = vars-of and
is-ground = is-ground and apply-subst = $\lambda x \sigma$. subst-apply (Var x) σ
proof *unfold-locales*
fix v **and** $t t' :: 'v \text{ trm}$
assume $t \cdot v = t' \cdot v$

then have *Unifier* $v t t'$
by *transfer (simp add: Unification.Unifier-def)*

then obtain μ **where** *unify* $t t' = \text{Some } \mu$
using *unify-eq-Some-if-Unifier*
by *blast*

then have *IMGU*: *IMGU* $\mu t t'$

```

    by (simp add: unify-computes-IMGU)

show  $\exists \mu. \text{term.is-imgu } \mu \ \{\{t, t'\}\}$ 
proof (rule exI)

  show  $\text{term.is-imgu } \mu \ \{\{t, t'\}\}$ 
  using IMGU IMGU-iff-term-is-mgu
  by auto
qed
qed

end
theory Substitution-Lifting-Example
imports
  Based-Substitution-Lifting
  Substitution-First-Order-Term
begin

setup natural-functor-setups

Lifting of properties from term to equations (modelled as pairs)
type-synonym  $(f, 'v)$  equation =  $(f, 'v)$  term  $\times$   $(f, 'v)$  term

All property locales have corresponding lifting locales
locale lifting-term-subst-properties =
  based-substitution-lifting where
  id-subst = Var and comp-subst = subst-compose and base-subst = subst-apply-term
and
  base-vars = vars-term ::  $(f, 'v)$  term  $\Rightarrow$   $'v$  set and apply-subst =  $\lambda x \sigma. \sigma \ x$  and
  base-is-ground = is-ground +

  finite-variables-lifting where
  id-subst = Var and comp-subst = subst-compose and apply-subst =  $\lambda x \sigma. \sigma \ x$ 

global-interpretation equation-subst:
  lifting-term-subst-properties where
  sub-vars = vars-term and sub-subst = subst-apply-term and sub-is-ground =
  is-ground and
  map =  $\lambda f. \text{map-prod } f \ f$  and to-set = set-prod
  by unfold-locales fastforce+

Lifted lemmas and defintions
thm
  equation-subst.vars-id-subst-subset
  equation-subst.vars-subst-subset
  equation-subst.to-fset-is-ground-subst

  equation-subst.vars-def
  equation-subst.subst-def

```

equation-subst.is-ground-def

We can lift multiple levels

global-interpretation *equation-set-subst:*

lifting-term-subst-properties **where**

sub-vars = *equation-subst.vars* **and** *sub-subst* = *equation-subst.subst* **and**

sub-is-ground = *equation-subst.is-ground* **and** *map* = *fimage* **and** *to-set* = *fset*

by *unfold-locales blast*

Lifted lemmas and definitions

thm

equation-set-subst.vars-id-subst-subset

equation-set-subst.vars-subst-subset

equation-set-subst.to-fset-is-ground-subst

equation-set-subst.vars-def

equation-set-subst.subst-def

equation-set-subst.is-ground-def

end