First Order Clause

Balazs Toth

March 20, 2025

Abstract

This entry provides reusable theories that lift properties of first-order (ground and nonground) terms to atoms, literals, and clauses. These properties include substitutions, orders, entailment, and typing. The sessions `AFP/First_Order_Terms` and `AFP/Abstract_Substitution` are the basis of this entry.

Contents

theory *Ground-Term-Extra*
imports *Regular-Tree-Relations.Ground-Terms*
begin

lemma *gterm-is-fun*: *is-Fun* (*term-of-gterm t*)
by(*cases t*) *simp*

no-notation *subst-compose* (**infixl** $\circ_s$ *75*)
no-notation *subst-apply-term* (**infixl** $\cdot$ *67*)

end
theory *Ground-Context*
imports *Ground-Term-Extra*
begin

type-synonym *'f ground-context* = (*'f*, *'f gterm*) *actxt*

abbreviation (*input*) *GHole* ($\langle\Box_G\rangle$) **where**
$\Box_G \equiv \Box$

abbreviation *ctxt-apply-gterm* ($\langle\text{-}\langle\text{-}\rangle_G\rangle$ [*1000*, *0*] *1000*) **where**
$C\langle s\rangle_G \equiv GFun\langle C;s\rangle$

lemma *le-size-gctxt*: *size t* $\leq$ *size* ($c\langle t\rangle_G$)
by (*induction c*) *simp-all*

lemma *lt-size-gctxt*: $c \neq \Box \Longrightarrow$ *size t* $<$ *size* $c\langle t\rangle_G$
by (*induction c*) *force+*

lemma *gctxt-ident-iff-eq-GHole*[*simp*]: $c\langle t\rangle_G = t \longleftrightarrow c = \Box$
proof (*rule iffI*)
assume $c\langle t\rangle_G = t$

hence *size* ($c\langle t\rangle_G$) = *size t*
by *argo*

thus $c = \Box$
using *lt-size-gctxt*[*of c t*]
by *linarith*
next

show $c = \Box \Longrightarrow c\langle t \rangle_G = t$
by *simp*
qed

end
theory *Multiset-Extra*
imports
HOL−Library.Multiset
HOL−Library.Multiset-Order
Nested-Multisets-Ordinals.Multiset-More
Abstract-Substitution.Natural-Magma-Functor
begin

lemma *exists-multiset* [*intro*]: $\exists M.\ x \in \#\ M$
by (*meson union-single-eq-member*)

global-interpretation *muliset-magma*: *natural-magma-with-empty* **where**
to-set = *set-mset* **and** *plus* = (+) **and** *wrap* = $\lambda l.\ \{\#l\#\}$ **and** *add* = *add-mset*
and *empty* = $\{\#\}$
by *unfold-locales simp-all*

global-interpretation *multiset-functor*: *finite-natural-functor* **where**
map = *image-mset* **and** *to-set* = *set-mset*
by *unfold-locales auto*

global-interpretation *multiset-functor*: *natural-functor-conversion* **where**
map = *image-mset* **and** *to-set* = *set-mset* **and** *map-to* = *image-mset* **and**
map-from = *image-mset* **and**
map′ = *image-mset* **and** *to-set′* = *set-mset*
by *unfold-locales simp-all*

global-interpretation *muliset-functor*: *natural-magma-functor* **where**
map = *image-mset* **and** *to-set* = *set-mset* **and** *plus* = (+) **and** *wrap* = $\lambda l.\ \{\#l\#\}$
and *add* = *add-mset*
by *unfold-locales simp-all*

lemma *one-le-countE*:
assumes $1 \leq$ *count M x*
obtains *M′* **where** *M* = *add-mset x M′*
using *assms* **by** (*meson count-greater-eq-one-iff multi-member-split*)

lemma *two-le-countE*:
assumes $2 \leq$ *count M x*
obtains *M′* **where** *M* = *add-mset x* (*add-mset x M′*)
using *assms*
by (*metis Suc-1 Suc-eq-plus1-left Suc-leD add.right-neutral count-add-mset multi-member-split*
not-in-iff not-less-eq-eq)

lemma *three-le-countE*:

assumes $3 \leq$ *count* M x
obtains M' **where** $M =$ *add-mset* x (*add-mset* x (*add-mset* x M'))
using *assms*
by (*metis One-nat-def Suc-1 Suc-leD add-le-cancel-left count-add-mset numeral-3-eq-3 plus-1-eq-Suc two-le-countE*)

lemma *one-step-implies-multp$_{HO}$-strong*:
fixes A B J K :: - *multiset*
defines $J \equiv B - A$ **and** $K \equiv A - B$
assumes $J \neq \{\#\}$ **and** $\forall k \in\# K.\ \exists x \in\# J.\ R\ k\ x$
shows *multp$_{HO}$* R A B
unfolding *multp$_{HO}$-def*
proof (*intro conjI allI impI*)
show $A \neq B$
using *assms*
by *force*
next
fix y
assume *count* B $y <$ *count* A y

then show $\exists x.\ R\ y\ x \wedge$ *count* A $x <$ *count* B x
using *assms*
by (*metis in-diff-count*)
qed

lemma *Uniq-antimono*: $Q \leq P \Longrightarrow$ *Uniq* $Q \geq$ *Uniq* P
unfolding *le-fun-def le-bool-def*
by (*rule impI*) (*simp only*: *Uniq-I Uniq-D*)

lemma *Uniq-antimono'*: $(\bigwedge x.\ Q\ x \Longrightarrow P\ x) \Longrightarrow$ *Uniq* $P \Longrightarrow$ *Uniq* Q
by (*fact Uniq-antimono*[*unfolded le-fun-def le-bool-def*, *rule-format*])

lemma *multp-singleton-right*[*simp*]:
assumes *transp* R
shows *multp* R M $\{\#x\#\} \longleftrightarrow (\forall y \in\# M.\ R\ y\ x)$
proof (*rule iffI*)
show $\forall y \in\# M.\ R\ y\ x \Longrightarrow$ *multp* R M $\{\#x\#\}$
using *one-step-implies-multp*[*of* $\{\#x\#\}$ - R $\{\#\}$, *simplified*] .
next
show *multp* R M $\{\#x\#\} \Longrightarrow \forall y \in\# M.\ R\ y\ x$
using *multp-implies-one-step*[*OF* ‹*transp* R›]
by (*smt* (*verit*, *del-insts*) *add-0 set-mset-add-mset-insert set-mset-empty single-is-union singletonD*)
qed

lemma *multp-singleton-left*[*simp*]:
assumes *transp* R

shows *multp R* {#*x*#} *M* ⟷ ({#*x*#} ⊂# *M* ∨ (∃ *y* ∈# *M*. *R x y*))
proof (*rule iffI*)
show {#*x*#} ⊂# *M* ∨ (∃ *y*∈#*M*. *R x y*) ⟹ *multp R* {#*x*#} *M*
proof (*elim disjE bexE*)
show {#*x*#} ⊂# *M* ⟹ *multp R* {#*x*#} *M*
by (*simp add*: *subset-implies-multp*)
next
show ⋀*y*. *y* ∈# *M* ⟹ *R x y* ⟹ *multp R* {#*x*#} *M*
using *one-step-implies-multp*[*of M* {#*x*#} *R* {#}, *simplified*] **by** *force*
qed
next
show *multp R* {#*x*#} *M* ⟹ {#*x*#} ⊂# *M* ∨ (∃ *y*∈#*M*. *R x y*)
using *multp-implies-one-step*[*OF* ‹*transp R*›, *of* {#*x*#} *M*]
by (*metis* (*no-types*, *opaque-lifting*) *add-cancel-right-left subset-mset.gr-zeroI*
subset-mset.less-add-same-cancel2 union-commute union-is-single union-single-eq-member)
qed

lemma *multp-singleton-singleton*[*simp*]: *transp R* ⟹ *multp R* {#*x*#} {#*y*#} ⟷
R x y
using *multp-singleton-right*[*of R* {#*x*#} *y*] **by** *simp*

lemma *multp-subset-supersetI*: *transp R* ⟹ *multp R A B* ⟹ *C* ⊆# *A* ⟹ *B*
⊆# *D* ⟹ *multp R C D*
by (*metis subset-implies-multp subset-mset.antisym-conv2 transpE transp-multp*)

lemma *multp-double-doubleI*:
assumes *transp R multp R A B*
shows *multp R* (*A* + *A*) (*B* + *B*)
using *multp-repeat-mset-repeat-msetI*[*OF* ‹*transp R*› ‹*multp R A B*›, *of 2*]
by (*simp add*: *numeral-Bit0*)

lemma *multp-implies-one-step-strong*:
fixes *A B I J K* :: - *multiset*
assumes *transp R* **and** *asymp R* **and** *multp R A B*
defines *J* ≡ *B* − *A* **and** *K* ≡ *A* − *B*
shows *J* ≠ {#} **and** ∀ *k* ∈# *K*. ∃ *x* ∈# *J*. *R k x*
proof −
from *assms* **have** *multp*$_{HO}$ *R A B*
by (*simp add*: *multp-eq-multp*$_{HO}$)

thus *J* ≠ {#} **and** ∀ *k* ∈# *K*. ∃ *x* ∈# *J*. *R k x*
using *multp*$_{HO}$*-implies-one-step-strong*[*OF* ‹*multp*$_{HO}$ *R A B*›]
by (*simp-all add*: *J-def K-def*)
qed

lemma *multp-double-doubleD*:
assumes *transp R* **and** *asymp R* **and** *multp R* (*A* + *A*) (*B* + *B*)
shows *multp R A B*
proof −

```
  from assms have
    B + B − (A + A) ≠ {#} and
    ∀k∈#A + A − (B + B). ∃x∈#B + B − (A + A). R k x
    using multp-implies-one-step-strong[OF assms] by simp-all

  have multp R (A ∩# B + (A − B)) (A ∩# B + (B − A))
  proof (rule one-step-implies-multp[of B − A A − B R A ∩# B])
    show B − A ≠ {#}
      using ‹B + B − (A + A) ≠ {#}›
      by (meson Diff-eq-empty-iff-mset mset-subset-eq-mono-add)
  next
    show ∀k∈#A − B. ∃j∈#B − A. R k j
    proof (intro ballI)
      fix x assume x ∈# A − B
      hence x ∈# A + A − (B + B)
        by (simp add: in-diff-count)
      then obtain y where y ∈# B + B − (A + A) and R x y
        using ‹∀k∈#A + A − (B + B). ∃x∈#B + B − (A + A). R k x› by auto
      then show ∃j∈#B − A. R x j
        by (auto simp add: in-diff-count)
    qed
  qed

  moreover have A = A ∩# B + (A − B)
    by (simp add: inter-mset-def)

  moreover have B = A ∩# B + (B − A)
   by (metis diff-intersect-right-idem subset-mset.add-diff-inverse subset-mset.inf.cobounded2)

  ultimately show ?thesis
    by argo
qed

lemma multp-double-double:
  transp R ⟹ asymp R ⟹ multp R (A + A) (B + B) ⟷ multp R A B
  using multp-double-doubleD multp-double-doubleI by metis

lemma multp-doubleton-doubleton[simp]:
  transp R ⟹ asymp R ⟹ multp R {#x, x#} {#y, y#} ⟷ R x y
  using multp-double-double[of R {#x#} {#y#}, simplified] by simp

lemma multp-single-doubleI: M ≠ {#} ⟹ multp R M (M + M)
  using one-step-implies-multp[of M {#} - M, simplified] by simp

lemma mult1-implies-one-step-strong:
  assumes trans r and asym r and (A, B) ∈ mult1 r
  shows B − A ≠ {#} and ∀k ∈# A − B. ∃j ∈# B − A. (k, j) ∈ r
proof −
  from ‹(A, B) ∈ mult1 r› obtain b B′ A′ where
```

B-def: $B = \mathit{add\text{-}mset}\ b\ B'$ **and**
A-def: $A = B' + A'$ **and**
$\forall\, a.\ a \in\# A' \longrightarrow (a,\ b) \in r$
unfolding *mult1-def* **by** *auto*

have $b \notin\# A'$
by (*meson* ‹$\forall\, a.\ a \in\# A' \longrightarrow (a,\ b) \in r$› *assms*(2) *asym-onD iso-tuple-UNIV-I*)
then have $b \in\# B - A$
by (*simp add*: *A-def B-def*)
thus $B - A \neq \{\#\}$
by *auto*

show $\forall\, k \in\# A - B.\ \exists\, j \in\# B - A.\ (k,\ j) \in r$
by (*metis A-def B-def* ‹$\forall\, a.\ a \in\# A' \longrightarrow (a,\ b) \in r$› ‹$b \in\# B - A$› ‹$b \notin\# A'$›
add-diff-cancel-left'
add-mset-add-single diff-diff-add-mset diff-single-trivial)
qed

lemma *asymp-multp*:
assumes *asymp R* **and** *transp R*
shows *asymp* (*multp R*)
using *asymp-multp*$_{HO}$[*OF assms*]
unfolding *multp-eq-multp*$_{HO}$[*OF assms*]**.**

lemma *multp-doubleton-singleton*: *transp R* $\Longrightarrow$ *multp R* {# *x*, *x* #} {# *y* #} $\longleftrightarrow$ *R x y*
by (*cases* $x = y$) *auto*

lemma *image-mset-remove1-mset*:
assumes *inj f*
shows *remove1-mset* (*f a*) (*image-mset f X*) = *image-mset f* (*remove1-mset a X*)
using *image-mset-remove1-mset-if*
unfolding *image-mset-remove1-mset-if inj-image-mem-iff*[*OF assms*, *symmetric*]
by *simp*

lemma *multp*$_{DM}$*-map-strong*:
assumes
f-mono: *monotone-on* (*set-mset* (*M1* + *M2*)) *R S f* **and**
M1-lt-M2: *multp*$_{DM}$ *R M1 M2*
shows *multp*$_{DM}$ *S* (*image-mset f M1*) (*image-mset f M2*)
proof −
obtain *Y X* **where**
$Y \neq \{\#\}$ **and** $Y \subseteq\# M2$ **and** *M1-eq*: $M1 = M2 - Y + X$ **and**
ex-y: $\forall\, x.\ x \in\# X \longrightarrow (\exists\, y.\ y \in\# Y \wedge R\ x\ y)$
using *M1-lt-M2*[*unfolded multp*$_{DM}$*-def Let-def mset-map*] **by** *blast*

let *?fY* = *image-mset f Y*

```
  let ?fX = image-mset f X

  show ?thesis
    unfolding multp_DM-def
  proof (intro exI conjI)
    show image-mset f Y ≠ {#}
      using ‹Y ≠ {#}› unfolding image-mset-is-empty-iff .
  next
    show image-mset f Y ⊆# image-mset f M2
      using ‹Y ⊆# M2› image-mset-subseteq-mono by metis
  next
    show image-mset f M1 = image-mset f M2 − ?fY + ?fX
      using M1-eq[THEN arg-cong, of image-mset f] ‹Y ⊆# M2›
      by (metis image-mset-Diff image-mset-union)
  next
    obtain g where y: ∀x. x ∈# X ⟶ g x ∈# Y ∧ R x (g x)
      using ex-y by moura

    show ∀fx. fx ∈# ?fX ⟶ (∃fy. fy ∈# ?fY ∧ S fx fy)
    proof (intro allI impI)
      fix x' assume x' ∈# ?fX
      then obtain x where x': x' = f x and x-in: x ∈# X
        by auto
      hence y-in: g x ∈# Y and y-gt: R x (g x)
        using y[rule-format, OF x-in] by blast+

      moreover have X ⊆# M1
        using M1-eq by simp

      ultimately have f (g x) ∈# ?fY ∧ S (f x)(f (g x))
        using f-mono[THEN monotone-onD, of x g x] ‹Y ⊆# M2› ‹X ⊆# M1›
x-in
        by (metis imageI in-image-mset mset-subset-eqD union-iff)
      thus ∃fy. fy ∈# ?fY ∧ S x' fy
        unfolding x' by auto
    qed
  qed
qed

lemma multp-map-strong:
  assumes
    transp: transp R and
    f-mono: monotone-on (set-mset (M1 + M2)) R S f and
    M1-lt-M2: multp R M1 M2
  shows multp S (image-mset f M1) (image-mset f M2)
  using monotone-on-multp-multp-image-mset[THEN monotone-onD, OF f-mono
transp - - M1-lt-M2]
  by simp
```

lemma *multp$_{HO}$-add-mset*:
assumes *asymp R transp R R x y multp$_{HO}$ R X Y*
shows *multp$_{HO}$ R* (*add-mset x X*) (*add-mset y Y*)
unfolding *multp$_{HO}$-def*
proof(*intro allI conjI impI*)
show *add-mset x X* ≠ *add-mset y Y*
using *assms*(*1*, *3*, *4*)
unfolding *multp$_{HO}$-def*
by (*metis asympD count-add-mset lessI less-not-refl*)
next
fix *x′*
assume *count-x′*: *count* (*add-mset y Y*) *x′* < *count* (*add-mset x X*) *x′*
show ∃ *y′*. *R x′ y′* ∧ *count* (*add-mset x X*) *y′* < *count* (*add-mset y Y*) *y′*
proof(*cases x′* = *x*)
case *True*
then show *?thesis*
using *assms*
unfolding *multp$_{HO}$-def*
by (*metis count-add-mset irreflpD irreflp-on-if-asymp-on not-less-eq transpE*)
next
case *x′-neq-x*: *False*
show *?thesis*
proof(*cases y* = *x′*)
case *True*
then show *?thesis*
using *assms*(*1*, *3*, *4*) *count-x′ x′-neq-x*
unfolding *multp$_{HO}$-def count-add-mset*
by (*smt* (*verit*) *Suc-lessD asympD*)
next
case *False*
then show *?thesis*
using *assms count-x′ x′-neq-x*
unfolding *multp$_{HO}$-def count-add-mset*
by (*smt* (*verit*, *del-insts*) *irreflpD irreflp-on-if-asymp-on not-less-eq transpE*)
qed
qed
qed

lemma *multp-add-mset*:
assumes *asymp R transp R R x y multp R X Y*
shows *multp R* (*add-mset x X*) (*add-mset y Y*)
using *multp$_{HO}$-add-mset*[*OF assms*(*1*−*3*)] *assms*(*4*)
unfolding *multp-eq-multp$_{HO}$*[*OF assms*(*1*, *2*)]
by *simp*

lemma *multp-add-mset′*:
assumes *R x y*
shows *multp R* (*add-mset x X*) (*add-mset y X*)

using *assms*
by (*metis add-mset-add-single empty-iff insert-iff one-step-implies-multp set-mset-add-mset-insert set-mset-empty*)

lemma *multp-add-mset-reflclp*:
assumes *asymp R transp R R x y* (*multp R*)$^{==}$ *X Y*
shows *multp R* (*add-mset x X*) (*add-mset y Y*)
using
assms(*4*)
multp-add-mset'[*of R*, *OF assms*(*3*)]
multp-add-mset[*OF assms*(*1*−*3*)]
by *blast*

lemma *multp-add-same* [*simp*]:
assumes *asymp R transp R*
shows *multp R* (*add-mset x X*) (*add-mset x Y*) ⟷ *multp R X Y*
by (*meson assms asymp-on-subset irreflp-on-if-asymp-on multp-cancel-add-mset top-greatest*)

lemma *inj-mset-plus-same*: *inj* (λ*X* :: *'a multiset* . *X* + *X*)
proof(*unfold inj-def*, *intro allI impI*)
fix *X Y* :: *'a multiset*
assume *X* + *X* = *Y* + *Y*

then show *X* = *Y*
proof(*induction X arbitrary*: *Y*)
case *empty*
then show *?case*
by *simp*
next
case (*add x X*)
then show *?case*
by (*metis diff-single-eq-union diff-union-single-conv single-subset-iff subset-mset.add-diff-assoc2 union-iff union-single-eq-member*)
qed
qed

lemma *multp-image-lesseq-if-all-lesseq*:
assumes
asymp: *asymp R* **and**
transp: *transp R* **and**
all-lesseq: ∀ *x*∈#*X*. *R*$^{==}$ (*f x*) (*g x*)
shows (*multp R*)$^{==}$ (*image-mset f X*) (*image-mset g X*)
using *assms*
by(*induction X*) (*auto simp*: *multp-add-mset multp-add-mset'*)

lemma *multp-image-less-if-all-lesseq-ex-less*:
  assumes
    asymp: *asymp* R **and**
    transp: *transp* R **and**
    all-less-eq: $\forall x{\in}\#X.\ R^{==}\ (f\ x)\ (g\ x)$ **and**
    ex-less: $\exists x{\in}\#X.\ R\ (f\ x)\ (g\ x)$
  shows *multp* R $\{\#\ f\ x.\ x \in\#\ X\ \#\}$ $\{\#\ g\ x.\ x \in\#\ X\ \#\}$
  using *all-less-eq ex-less*
proof(*induction* X)
  case *empty*
  then show *?case*
    by *simp*
next
  case (*add* x X)

  show *?case*
  proof(*cases* $\exists x{\in}\#X.\ R\ (f\ x)\ (g\ x)$)
    case *True*

    then have $\forall x{\in}\#X.\ R^{==}\ (f\ x)\ (g\ x)$ $\exists x{\in}\#X.\ R\ (f\ x)\ (g\ x)$
      using *add.prems*
      by *auto*

    then have *multp* R (*image-mset* f X) (*image-mset* g X)
      using *add.IH*
      by *blast*

    then show *?thesis*
      using *add.prems*(*1*) *multp-add-mset*[*OF asymp transp*] *multp-add-same*[*OF asymp transp*]
      by *auto*
  next
    case *False*

    then have $R\ (f\ x)\ (g\ x)$
      using *add.prems*(*2*) **by** *fastforce*

    moreover have $\forall x{\in}\#X.\ f\ x = g\ x$
      using *False add.prems*(*1*) **by** *auto*

    ultimately show *?thesis*
      by (*metis image-mset-add-mset multiset.map-cong0 multp-add-mset'*)
  qed
qed

lemma *not-reflp-multp*$_{DM}$: $\neg$ *reflp* (*multp*$_{DM}$ R)
  unfolding *multp*$_{DM}$*-def reflp-def*
  by *force*

lemma *not-less-empty-multp$_{DM}$*: ¬ *multp$_{DM}$ R X* {#}
by (*simp add*: *multp$_{DM}$-def*)

lemma *not-reflp-multp$_{HO}$*: ¬ *reflp* (*multp$_{HO}$ R*)
unfolding *multp$_{HO}$-def reflp-def*
by *simp*

lemma *not-less-empty-multp$_{HO}$*: ¬ *multp$_{HO}$ R X* {#}
by (*simp add*: *multp$_{HO}$-def*)

lemma *not-refl-mult*: ¬ *refl* (*mult R*)
unfolding *refl-on-def mult-def*
by (*meson UNIV-I not-less-empty trancl.cases*)

lemma *not-less-empty-mult*: (*X*, {#}) ∉ *mult R*
by (*metis mult-def not-less-empty tranclD2*)

lemma *empty-less-mult*: *X* ≠ {#} ⟹ ({#}, *X*) ∈ *mult R*
using *subset-implies-mult*
by *force*

lemma *not-reflp-multp*: ¬ *reflp* (*multp R*)
using *not-refl-mult*
unfolding *multp-def reflp-refl-eq*
by *blast*

lemma *empty-less-multp*: *X* ≠ {#} ⟹ *multp R* {#} *X*
by (*simp add*: *subset-implies-multp subset-mset.not-eq-extremum*)

lemma *not-less-empty-multp*: ¬ *multp R X* {#}
using *not-less-empty-mult*
unfolding *multp-def*
by *blast*

end
theory *Uprod-Extra*
imports
HOL−Library.Uprod
Multiset-Extra
Abstract-Substitution.Natural-Functor
begin

abbreviation *upair* **where**
upair ≡ λ(*x*, *y*). *Upair x y*

lemma *Upair-sym*: *Upair x y* = *Upair y x*
by (*metis Upair-inject*)

lemma *upair-in-sym* [*simp*]:

```
  assumes sym I
  shows Upair a b ∈ upair ‘ I ⟷ (a, b) ∈ I ∧ (b, a) ∈ I
  using assms
  by (auto dest: symD)

lemma ex-ordered-Upair:
  assumes tot: totalp-on (set-uprod p) R
  shows ∃ x y. p = Upair x y ∧ R^== x y
proof −
  obtain x y where p = Upair x y
    by (metis uprod-exhaust)

  show ?thesis
  proof (cases R^== x y)
    case True
    show ?thesis
    proof (intro exI conjI)
      show p = Upair x y
        using ‹p = Upair x y› .
    next
      show R^== x y
        using True by simp
    qed
  next
    case False
    then show ?thesis
    proof (intro exI conjI)
      show p = Upair y x
        using ‹p = Upair x y› by simp
    next
      from tot have R y x
        using False
        by (simp add: ‹p = Upair x y› totalp-on-def)
      thus R^== y x
        by simp
    qed
  qed
qed

definition mset-uprod :: 'a uprod ⇒ 'a multiset where
  mset-uprod = case-uprod (Abs-commute (λx y. {#x, y#}))

lemma Abs-commute-inverse-mset [simp]:
  apply-commute (Abs-commute (λx y. {#x, y#})) = (λx y. {#x, y#})
  by (simp add: Abs-commute-inverse)

lemma set-mset-mset-uprod [simp]: set-mset (mset-uprod up) = set-uprod up
  by (simp add: mset-uprod-def case-uprod.rep-eq set-uprod.rep-eq case-prod-beta)
```

lemma *mset-uprod-Upair* [*simp*]: *mset-uprod* (*Upair x y*) = {#*x*, *y*#}
by (*simp add*: *mset-uprod-def*)

lemma *map-uprod-inverse*: (⋀*x*. *f* (*g x*) = *x*) ⟹ (⋀*y*. *map-uprod f* (*map-uprod g y*) = *y*)
by (*simp add*: *uprod.map-comp uprod.map-ident-strong*)

lemma *mset-uprod-image-mset*: *mset-uprod* (*map-uprod f p*) = *image-mset f* (*mset-uprod p*)
proof−
obtain *x y* **where** [*simp*]: *p* = *Upair x y*
using *uprod-exhaust* **by** *blast*

have *mset-uprod* (*map-uprod f p*) = {# *f x*, *f y* #}
by *simp*

then show *mset-uprod* (*map-uprod f p*) = *image-mset f* (*mset-uprod p*)
by *simp*
qed

lemma *ball-set-uprod* [*simp*]: (∀ *t*∈*set-uprod* (*Upair t*$_1$ *t*$_2$). *P t*) ⟷ *P t*$_1$ ∧ *P t*$_2$
by *auto*

lemma *inj-mset-uprod*: *inj mset-uprod*
proof(*unfold inj-def*, *intro allI impI*)
fix *a b* :: *'a uprod*
assume *mset-uprod a* = *mset-uprod b*
then show *a* = *b*
by(*cases a*; *cases b*)(*auto simp*: *add-mset-eq-add-mset*)
qed

lemma *mset-uprod-plus-neq*: *mset-uprod a* ≠ *mset-uprod b* + *mset-uprod b*
by(*cases a*; *cases b*)(*auto simp*: *add-mset-eq-add-mset*)

lemma *set-uprod-not-empty*: *set-uprod a* ≠ {}
by(*cases a*) *simp*

lemma *exists-uprod* [*intro*]: ∃ *a*. *x* ∈ *set-uprod a*
by (*metis insertI1 set-uprod-simps*)

global-interpretation *uprod-functor*: *finite-natural-functor* **where** *map* = *map-uprod* **and** *to-set* = *set-uprod*
by
unfold-locales
(*auto simp*: *uprod.map-comp uprod.map-ident uprod.set-map intro*: *uprod.map-cong*)

global-interpretation *uprod-functor*: *natural-functor-conversion* **where**
map = *map-uprod* **and** *to-set* = *set-uprod* **and** *map-to* = *map-uprod* **and** *map-from* = *map-uprod* **and**

map′ = map-uprod **and** *to-set′ = set-uprod*
by *unfold-locales* (*auto simp*: *uprod.set-map uprod.map-comp*)

end
theory *Ground-Clause*
imports
Saturation-Framework-Extensions.Clausal-Calculus
Ground-Term-Extra
Ground-Context
Uprod-Extra
begin

type-synonym *'f gatom = 'f gterm uprod*

end
theory *Typing*
imports *Main*
begin

locale *typing =*
fixes *welltyped* :: *'expr ⇒ 'ty ⇒ bool*
assumes *right-unique*: *right-unique welltyped*
begin

lemmas *right-uniqueD* [*dest*] = *right-uniqueD*[*OF right-unique*]

end

locale *base-typing = typing*
begin

abbreviation *is-welltyped* **where**
is-welltyped expr ≡ ∃ *τ*. *welltyped expr τ*

end

locale *typing-lifting = sub*: *typing* **where** *welltyped = sub-welltyped*
for *sub-welltyped* :: *'sub ⇒ 'ty ⇒ bool* +
fixes *to-set* :: *'expr ⇒ 'sub set*
begin

definition *is-welltyped* **where**
is-welltyped expr ≡ ∃ *τ*. ∀ *sub* ∈ *to-set expr*. *sub-welltyped sub τ*

abbreviation *welltyped* **where**
welltyped expr (-::*unit*) ≡ *is-welltyped expr*

sublocale *typing* **where** *welltyped = welltyped*
by *unfold-locales* (*auto simp*: *right-unique-def*)

end

locale *typing-lifting′* =
  fixes
    sub-welltyped :: *′extra* ⇒ *′sub* ⇒ *′ty′* ⇒ *bool* **and**
    to-set :: *′expr* ⇒ *′sub set*
  assumes *lifting*: ⋀*extra*. *typing-lifting* (*sub-welltyped extra*)
begin

sublocale *typing-lifting* **where**
  sub-welltyped = *sub-welltyped* $\mathcal{V}$ **and** *to-set* = *to-set*
  by (*rule lifting*)

end

end
theory *Natural-Magma-Typing-Lifting*
  imports
    Abstract-Substitution.Natural-Magma
    Typing
begin

locale *natural-magma-is-typed-lifting* =
  natural-magma **where** *to-set* = *to-set* +
  typing-lifting **where** *to-set* = *to-set* **and** *sub-welltyped* = *sub-welltyped*
  for *to-set* :: *′expr* ⇒ *′sub set* **and** *sub-welltyped* :: *′sub* ⇒ *unit* ⇒ *bool*
begin

abbreviation (*input*) *sub-is-welltyped* **where**
  sub-is-welltyped expr ≡ *sub-welltyped expr* ()

lemma *add* [*simp*]: *is-welltyped* (*add sub M*) ⟷ (*sub-is-welltyped sub* ∧ *is-welltyped M*)
  unfolding *is-welltyped-def*
  by *auto*

lemma *plus* [*simp*]:
  is-welltyped (*plus M M′*) ⟷ *is-welltyped M* ∧ *is-welltyped M′*
  unfolding *is-welltyped-def*
  by *auto*

end

locale *natural-magma-with-empty-is-typed-lifting* =
  natural-magma-is-typed-lifting + *natural-magma-with-empty*
begin

lemma *empty* [*intro*]: *is-welltyped empty*

unfolding *is-welltyped-def*
by *simp*

end

locale *natural-magma-typing-lifting* =
welltyped: *natural-magma-is-typed-lifting* **where** *sub-welltyped* = *sub-welltyped* +
typing-lifting +
natural-magma

locale *natural-magma-with-empty-typing-lifting* =
natural-magma-typing-lifting +
welltyped: *natural-magma-with-empty-is-typed-lifting* **where** *sub-welltyped* = *sub-welltyped*

end
theory *Multiset-Typing-Lifting*
imports
Natural-Magma-Typing-Lifting
Multiset-Extra
Abstract-Substitution.Functional-Substitution-Lifting
begin

locale *mulitset-typing-lifting* =
typing-lifting **where**
to-set = *set-mset* **and** *sub-welltyped* = *sub-welltyped*
for *sub-welltyped* :: *'sub* ⇒ *unit* ⇒ *bool*
begin

sublocale *natural-magma-with-empty-typing-lifting* **where**
to-set = *set-mset* **and** *plus* = (+) **and** *wrap* = λ*l*. {#*l*#} **and** *add* = *add-mset*
and *empty* = {#}
by *unfold-locales*

end

end
theory *Clausal-Calculus-Extra*
imports
Saturation-Framework-Extensions.Clausal-Calculus
Uprod-Extra
begin

lemma *literal-cases*: ⟦$\mathcal{P}$ ∈ {*Pos*, *Neg*}; $\mathcal{P}$ = *Pos* ⟹ *P*; $\mathcal{P}$ = *Neg* ⟹ *P*⟧ ⟹ *P*
by *blast*

lemma *map-literal-inverse*:
(⋀*x*. *f* (*g x*) = *x*) ⟹ (⋀*l*. *map-literal f* (*map-literal g l*) = *l*)
by (*simp add*: *literal.map-comp literal.map-ident-strong*)

lemma *map-literal-comp*:
map-literal f (*map-literal g l*) = *map-literal* (λ*a*. *f* (*g a*)) *l*
using *literal.map-comp*
unfolding *comp-def*.

lemma *literals-distinct* [*simp*]: *Pos* ≠ *Neg Neg* ≠ *Pos*
by (*metis literal.distinct*(*1*))+

primrec *mset-lit* :: *'a uprod literal* ⇒ *'a multiset* **where**
mset-lit (*Pos a*) = *mset-uprod a* |
mset-lit (*Neg a*) = *mset-uprod a* + *mset-uprod a*

lemma *mset-lit-image-mset*: *mset-lit* (*map-literal* (*map-uprod f*) *l*) = *image-mset f* (*mset-lit l*)
by(*induction l*) (*simp-all add*: *mset-uprod-image-mset*)

lemma *uprod-mem-image-iff-prod-mem*[*simp*]:
assumes *sym I*
shows (*Upair t t'*) ∈ (λ(*t*$_1$, *t*$_2$). *Upair t*$_1$ *t*$_2$) ‘ *I* ⟷ (*t*, *t'*) ∈ *I*
using ‹*sym I*›[*THEN symD*] **by** *auto*

lemma *true-lit-uprod-iff-true-lit-prod*[*simp*]:
assumes *sym I*
shows
upair ‘ *I* ⊨*l Pos* (*Upair t t'*) ⟷ *I* ⊨*l Pos* (*t*, *t'*)
upair ‘ *I* ⊨*l Neg* (*Upair t t'*) ⟷ *I* ⊨*l Neg* (*t*, *t'*)
unfolding *true-lit-simps uprod-mem-image-iff-prod-mem*[*OF* ‹*sym I*›]
by *simp-all*

abbreviation *Pos-Upair* (**infix** ≈ *66*) **where**
Pos-Upair t t' ≡ *Pos* (*Upair t t'*)

abbreviation *Neg-Upair* (**infix** !≈ *66*) **where**
Neg-Upair t t' ≡ *Neg* (*Upair t t'*)

lemma *exists-literal-for-atom* [*intro*]: ∃ *l*. *a* ∈ *set-literal l*
by (*meson literal.set-intros*(*1*))

lemma *exists-literal-for-term* [*intro*]: ∃ *l*. *t* ∈# *mset-lit l*
by (*metis exists-uprod mset-lit.simps*(*1*) *set-mset-mset-uprod*)

lemma *finite-set-literal* [*intro*]: *finite* (*set-literal l*)
unfolding *set-literal-atm-of*
by *simp*

lemma *map-literal-map-uprod-cong*:
assumes ⋀*t*. *t* ∈# *mset-lit l* ⟹ *f t* = *g t*
shows *map-literal* (*map-uprod f*) *l* = *map-literal* (*map-uprod g*) *l*
using *assms*

```
  by(cases l)(auto cong: uprod.map-cong0)

lemma set-mset-set-uprod: set-mset (mset-lit l) = set-uprod (atm-of l)
  by(cases l) simp-all

lemma mset-lit-set-literal: t ∈# mset-lit l ⟷ t ∈ ⋃(set-uprod ‘ set-literal l)
  unfolding set-literal-atm-of
  by(simp add: set-mset-set-uprod)

lemma inj-mset-lit: inj mset-lit
proof(unfold inj-def, intro allI impI)
  fix l l' :: 'a uprod literal
  assume mset-lit: mset-lit l = mset-lit l'

  show l = l'
  proof(cases l)
    case l: (Pos a)
    show ?thesis
    proof(cases l')
      case l': (Pos a')

      show ?thesis
        using mset-lit inj-mset-uprod
        unfolding l l' inj-def
        by auto
    next
      case l': (Neg a')

      show ?thesis
        using mset-lit mset-uprod-plus-neq
        unfolding l l'
        by auto
    qed
  next
    case l: (Neg a)
    then show ?thesis
     proof(cases l')
      case l': (Pos a')

      show ?thesis
        using mset-lit mset-uprod-plus-neq
        unfolding l l'
        by (metis mset-lit.simps)
    next
      case l': (Neg a')

      show ?thesis
        using mset-lit inj-mset-plus-same inj-mset-uprod
        unfolding l l' inj-def
```

```
      by auto
    qed
  qed
qed

global-interpretation literal-functor: finite-natural-functor where
  map = map-literal and to-set = set-literal
  by
    unfold-locales
   (auto simp: literal.map-comp literal.map-ident literal.set-map intro: literal.map-cong)

global-interpretation literal-functor: natural-functor-conversion where
   map = map-literal and to-set = set-literal and map-to = map-literal and
map-from = map-literal and
  map' = map-literal and to-set' = set-literal
  by unfold-locales
    (auto simp: literal.set-map literal.map-comp)

abbreviation uprod-literal-to-set where uprod-literal-to-set l ≡ set-mset (mset-lit
l)

abbreviation map-uprod-literal where map-uprod-literal f ≡ map-literal (map-uprod
f)

global-interpretation uprod-literal-functor: finite-natural-functor where
  map = map-uprod-literal and to-set = uprod-literal-to-set
 by unfold-locales (auto simp: mset-lit-image-mset intro: map-literal-map-uprod-cong)

global-interpretation uprod-literal-functor: natural-functor-conversion where
  map = map-uprod-literal and to-set = uprod-literal-to-set and map-to = map-uprod-literal
and
  map-from = map-uprod-literal and map' = map-uprod-literal and to-set' = up-
rod-literal-to-set
  by unfold-locales (auto simp: mset-lit-image-mset)

lemma exists-inference [intro]: ∃ ι. f ∈ set-inference ι
  by (metis inference.set-intros(2))

lemma finite-set-inference [intro]: finite (set-inference ι)
  by (metis inference.exhaust inference.set List.finite-set finite.simps finite-Un)

global-interpretation inference-functor: finite-natural-functor where
  map = map-inference and to-set = set-inference
  by
    unfold-locales
     (auto simp: inference.map-comp inference.map-ident inference.set-map intro:
inference.map-cong)

global-interpretation inference-functor: natural-functor-conversion where
```

map = *map-inference* **and** *to-set* = *set-inference* **and** *map-to* = *map-inference*
and
map-from = *map-inference* **and** *map'* = *map-inference* **and** *to-set'* = *set-inference*
by *unfold-locales*
(*auto simp*: *inference.set-map inference.map-comp*)

end
theory *Clause-Typing*
imports
Multiset-Typing-Lifting

Clausal-Calculus-Extra
Multiset-Extra
Uprod-Extra
begin

locale *clause-typing* = *term*: *typing term-welltyped*
for *term-welltyped*
begin

sublocale *atom*: *typing-lifting* **where**
sub-welltyped = *term-welltyped* **and** *to-set* = *set-uprod*
by *unfold-locales*

lemma *atom-is-welltyped-iff* [*simp*]:
atom.is-welltyped (*Upair t t'*) ⟷ (∃ τ. *term-welltyped t* τ ∧ *term-welltyped t'* τ)
unfolding *atom.is-welltyped-def*
by *simp*

sublocale *literal*: *typing-lifting* **where**
sub-welltyped = *atom.welltyped* **and** *to-set* = *set-literal*
by *unfold-locales*

lemma *literal-is-welltyped-iff* [*simp*]:
literal.is-welltyped (*t* ≈ *t'*) ⟷ *atom.is-welltyped* (*Upair t t'*)
literal.is-welltyped (*t* !≈ *t'*) ⟷ *atom.is-welltyped* (*Upair t t'*)
unfolding *literal.is-welltyped-def*
by *simp-all*

lemma *literal-is-welltyped-iff-atm-of*:
literal.is-welltyped l ⟷ *atom.is-welltyped* (*atm-of l*)
unfolding *literal.is-welltyped-def*
by (*simp add*: *set-literal-atm-of*)

sublocale *clause*: *mulitset-typing-lifting* **where**
sub-welltyped = *literal.welltyped*
by *unfold-locales*

end

end
theory *Context-Extra*
imports *First-Order-Terms.Subterm-and-Context*
begin

no-notation *subst-compose* (**infixl** $\circ_s$ *75*)
no-notation *subst-apply-term* (**infixl** $\cdot$ *67*)

end
theory *Term-Typing*
imports *Typing Context-Extra*
begin

type-synonym $('f, 'ty)$ *fun-types* $= 'f \Rightarrow nat \Rightarrow 'ty\ list \times 'ty$

locale *context-compatible-typing* =
fixes *Fun welltyped*
assumes
context-compatible [*intro*]:
$\bigwedge t\ t'\ c\ \tau\ \tau'.$
welltyped $t\ \tau' \Longrightarrow$
welltyped $t'\ \tau' \Longrightarrow$
welltyped $(Fun\langle c;\ t\rangle)\ \tau \Longrightarrow$
welltyped $(Fun\langle c;\ t'\rangle)\ \tau$

locale *subterm-typing* =
fixes *Fun welltyped*
assumes
subterm': $\bigwedge f\ ts\ \tau.$ *welltyped* $(Fun\ f\ ts)\ \tau \Longrightarrow \forall\, t{\in}set\ ts.\ \exists \tau'.$ *welltyped* $t\ \tau'$
begin

lemma *subterm*: *welltyped* $(Fun\langle c;\ t\rangle)\ \tau \Longrightarrow \exists \tau.$ *welltyped* $t\ \tau$
proof(*induction c arbitrary*: τ)
case *Hole*
then show *?case*
by *auto*
next
case (*More f ss1 c ss2*)

then have *welltyped* $(Fun\ f\ (ss1\ @\ Fun\langle c;t\rangle\ \#\ ss2))\ \tau$
by *simp*

then have $\exists \tau.$ *welltyped* $(Fun\langle c;t\rangle)\ \tau$
using *subterm'*
by *simp*

then obtain τ' **where** *welltyped* $(Fun\langle c;t\rangle)\ \tau'$
by *blast*

```
  then show ?case
    using More.IH
    by simp
qed

end

locale term-typing =
  base-typing +
  welltyped: context-compatible-typing where welltyped = welltyped +
  welltyped: subterm-typing where welltyped = welltyped

end
theory Ground-Typing
  imports
    Ground-Clause
    Clause-Typing
    Term-Typing
begin

inductive welltyped for ℱ where
  GFun: ℱ f (length ts) = (τs, τ) ⟹ list-all2 (welltyped ℱ) ts τs ⟹ welltyped
ℱ (GFun f ts) τ

lemma right-unique-welltyped[iff]: right-unique (welltyped ℱ)
proof (rule right-uniqueI)
  fix t τ₁ τ₂
  assume welltyped ℱ t τ₁ and welltyped ℱ t τ₂
  thus τ₁ = τ₂
    by (auto elim!: welltyped.cases)
qed

lemma welltyped-context-compatible:
  assumes
    t-type: welltyped ℱ t τ' and
    t'-type: welltyped ℱ t' τ' and
    c-type: welltyped ℱ c⟨t⟩_G τ
  shows welltyped ℱ c⟨t'⟩_G τ
proof −
  from c-type show welltyped ℱ c⟨t'⟩_G τ
  proof (induction c arbitrary: τ)
    case Hole
    thus ?case
      using t-type t'-type
      using right-unique-welltyped[THEN right-uniqueD]
      by fastforce
  next
    case (More f ss1 c ss2)
```

```
    have welltyped F (GFun f (ss1 @ c⟨t⟩_G # ss2)) τ
      using More.prems
      by simp

    hence welltyped F (GFun f (ss1 @ c⟨t'⟩_G # ss2)) τ
    proof (cases F GFun f (ss1 @ c⟨t⟩_G # ss2) τ rule: welltyped.cases)
      case (GFun τs)
      show ?thesis
      proof (rule welltyped.GFun)
        show F f (length (ss1 @ c⟨t'⟩_G # ss2)) = (τs, τ)
          using GFun(1)
          by simp
      next
        show list-all2 (welltyped F) (ss1 @ c⟨t'⟩_G # ss2) τs
          using GFun(2) More.IH
         by (smt (verit, del-insts) list-all2-Cons1 list-all2-append1 list-all2-lengthD)
      qed
    qed

    thus ?case
      by simp
  qed
qed

lemma welltyped-subterms:
  fixes f ts τ
  assumes welltyped F (GFun f ts) τ
  shows ∀ t∈set ts. ∃τ'. welltyped F t τ'
  using assms
  by (metis gterm.inject in-set-conv-nth list-all2-conv-all-nth welltyped.simps)

global-interpretation term: term-typing where
  welltyped = welltyped F and Fun = GFun
  for F :: ('f, 'ty) fun-types
  by unfold-locales
    (auto intro: welltyped-context-compatible welltyped-subterms[rule-format])

global-interpretation clause-typing where
  term-welltyped = welltyped F
  for F :: ('f, 'ty) fun-types
  by unfold-locales

end
theory Nonground-Term
 imports
    Abstract-Substitution.Substitution-First-Order-Term
    Abstract-Substitution.Functional-Substitution-Lifting
    Ground-Term-Extra
```

begin

no-notation *subst-compose* (**infixl** $\circ_s$ *75*)
notation *subst-compose* (**infixl** $\odot$ *75*)

no-notation *subst-apply-term* (**infixl** $\cdot$ *67*)
notation *subst-apply-term* (**infixl** $\cdot t$ *67*)

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

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

1 Nonground Terms and Substitutions

type-synonym *'f ground-term* = *'f gterm*

1.1 Unified naming

locale *vars-def* =
  fixes *vars-def* :: *'expr* ⇒ *'var*
begin

abbreviation *vars* ≡ *vars-def*

end

locale *grounding-def* =
  fixes
    to-ground-def :: *'expr* ⇒ *'expr*$_G$ **and**
    from-ground-def :: *'expr*$_G$ ⇒ *'expr*
begin

abbreviation *to-ground* ≡ *to-ground-def*

abbreviation *from-ground* ≡ *from-ground-def*

end

1.2 Term

locale *nonground-term-properties* =
  base-functional-substitution +
  finite-variables +
  all-subst-ident-iff-ground

locale *term-grounding* =
  variables-in-base-imgu **where** *base-vars* = *vars* **and** *base-subst* = *subst* +
  grounding

locale *nonground-term*
begin

sublocale *vars-def* **where** *vars-def* = *vars-term* **.**

sublocale *grounding-def* **where**
to-ground-def = *gterm-of-term* **and** *from-ground-def* = *term-of-gterm* **.**

lemma *infinite-terms* [*intro*]: *infinite* (*UNIV* :: ('*f*, '*v*) *term set*)
proof−
have *infinite* (*UNIV* :: ('*f*, '*v*) *term list set*)
using *infinite-UNIV-listI***.**

then have ⋀*f* :: '*f*. *infinite* ((*Fun f*) ' (*UNIV* :: ('*f*, '*v*) *term list set*))
by (*meson finite-imageD injI term.inject*(*2*))

then show *infinite* (*UNIV* :: ('*f*, '*v*) *term set*)
using *infinite-super top-greatest* **by** *blast*
qed

sublocale *nonground-term-properties* **where**
subst = (·*t*) **and** *id-subst* = *Var* **and** *comp-subst* = (⊙) **and**
vars = *vars* :: ('*f*, '*v*) *term* ⇒ '*v set*
proof *unfold-locales*
fix *t* :: ('*f*, '*v*) *term* **and** *σ τ* :: ('*f*, '*v*) *subst*
assume ⋀*x*. *x* ∈ *vars t* ⟹ *σ x* = *τ x*
then show *t* ·*t σ* = *t* ·*t τ*
by(*rule term-subst-eq*)
next
fix *t* :: ('*f*, '*v*) *term*
show *finite* (*vars t*)
by *simp*
next
fix *t* :: ('*f*, '*v*) *term*
show (*vars t* = {}) ⟷ (∀*σ*. *t* ·*t σ* = *t*)
using *is-ground-trm-iff-ident-forall-subst***.**
next
fix *t* :: ('*f*, '*v*) *term* **and** *ts* :: ('*f*, '*v*) *term set*

assume *finite ts vars t* ≠ {}
then show ∃*σ*. *t* ·*t σ* ≠ *t* ∧ *t* ·*t σ* ∉ *ts*
proof(*induction t arbitrary*: *ts*)
case (*Var x*)

obtain *t*′ **where** *t*′: *t*′ ∉ *ts is-Fun t*′
using *Var.prems*(*1*) *finite-list* **by** *blast*

define *σ* :: ('*f*, '*v*) *subst* **where** ⋀*x*. *σ x* = *t*′

```
    have Var x ·t σ ≠ Var x
      using t'
      unfolding σ-def
      by auto

    moreover have Var x ·t σ ∉ ts
      using t'
      unfolding σ-def
      by simp

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

    obtain a where a: a ∈ set args and a-vars: vars a ≠ {}
      using Fun.prems
      by fastforce

    then obtain σ where
      σ: a ·t σ ≠ a and
      a-σ-not-in-args: a ·t σ ∉ ⋃ (set ‘ term.args ‘ ts)
      by (metis Fun.IH Fun.prems(1) List.finite-set finite-UN finite-imageI)

    then have Fun f args ·t σ ≠ Fun f args
     by (metis a subsetI term.set-intros(4) term-subst.comp-subst.left.action-neutral
         vars-term-subset-subst-eq)

    moreover have Fun f args ·t σ ∉ ts
      using a a-σ-not-in-args
      by auto

    ultimately show ?case
      using Fun
      by blast
  qed
next
  fix t :: ('f, 'v) term and ϱ :: ('f, 'v) subst

  show vars (t ·t ϱ) = ⋃ (vars ‘ ϱ ‘ vars t)
    using vars-term-subst.
next
  show ∃ t. vars t = {}
    using vars-term-of-gterm
    by metis
next
  fix x :: 'v
  show vars (Var x) = {x}
```

```
    by simp
next
  fix σ σ' :: ('f, 'v) subst and x
  show (σ ⊙ σ') x = σ x ·t σ'
    unfolding subst-compose-def ..
qed

sublocale renaming-variables where
  vars = vars :: ('f, 'v) term ⇒ 'v set and subst = (·t) and id-subst = Var and
  comp-subst = (⊙)
proof unfold-locales
  fix ϱ :: ('f, 'v) subst

  show term-subst.is-renaming ϱ ⟷ inj ϱ ∧ (∀ x. ∃ x'. ϱ x = Var x')
    using term-subst-is-renaming-iff
    unfolding is-Var-def.
next
  fix ϱ :: ('f, 'v) subst and t
  assume ϱ: term-subst.is-renaming ϱ
  show vars (t ·t ϱ) = rename ϱ ` vars t
  proof(induction t)
    case (Var x)
    have ϱ x = Var (rename ϱ x)
      using ϱ
      unfolding rename-def[OF ϱ] term-subst-is-renaming-iff is-Var-def
      by (meson someI-ex)

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

sublocale term-grounding where
  subst = (·t) and id-subst = Var and comp-subst = (⊙) and
  vars = vars :: ('f, 'v) term ⇒ 'v set and from-ground = from-ground and
  to-ground = to-ground
proof unfold-locales
   fix t :: ('f, 'v) term and μ :: ('f, 'v) subst and unifications

  assume imgu:
    term-subst.is-imgu μ unifications
    ∀ unification∈unifications. finite unification
    finite unifications

  show vars (t ·t μ) ⊆ vars t ∪ ⋃ (vars ` ⋃ unifications)
```

using *range-vars-subset-if-is-imgu*[*OF imgu*] *vars-term-subst-apply-term-subset*
by *fastforce*
next
{
fix t :: ('*f*, '*v*) *term*
assume *t-is-ground*: *is-ground t*

have ∃ *g*. *from-ground g* = *t*
proof(*intro exI*)

from *t-is-ground*
show *from-ground* (*to-ground t*) = *t*
by(*induction t*)(*simp-all add*: *map-idI*)

qed
}

then show {*t* :: ('*f*, '*v*) *term*. *is-ground t*} = *range from-ground*
by *fastforce*
next
fix t_G :: ('*f*) *ground-term*
show *to-ground* (*from-ground* t_G) = t_G
by *simp*
qed

lemma *term-context-ground-iff-term-is-ground* [*simp*]: *Term-Context.ground t* = *is-ground t*
by(*induction t*) *simp-all*

declare *Term-Context.ground-vars-term-empty* [*simp del*]

lemma *obtain-ground-fun*:
assumes *is-ground t*
obtains *f ts* **where** *t* = *Fun f ts*
using *assms*
by(*cases t*) *auto*

end

1.3 Setup for lifting from terms

locale *lifting* =
based-functional-substitution-lifting +
all-subst-ident-iff-ground-lifting +
grounding-lifting +
renaming-variables-lifting +
variables-in-base-imgu-lifting

locale *term-based-lifting* =

term: *nonground-term* +
lifting **where**
comp-subst = (⊙) **and** *id-subst* = *Var* **and** *base-subst* = (·*t*) **and** *base-vars* = *term.vars*

end
theory *Nonground-Context*
imports
Nonground-Term
Ground-Context
begin

2 Nonground Contexts and Substitutions

type-synonym ('*f*, '*v*) *context* = ('*f*, '*v*) *ctxt*

abbreviation *subst-apply-ctxt* ::
('*f*, '*v*) *context* ⇒ ('*f*, '*v*) *subst* ⇒ ('*f*, '*v*) *context* (**infixl** ·$_c$ *67*) **where**
subst-apply-ctxt ≡ *subst-apply-actxt*

global-interpretation *context*: *finite-natural-functor* **where**
map = *map-args-actxt* **and** *to-set* = *set2-actxt*
proof *unfold-locales*
fix *t* :: '*t*

show ∃ *c*. *t* ∈ *set2-actxt c*
by (*metis actxt.set-intros*(*5*) *list.set-intros*(*1*))
next
fix *c* :: ('*f*, '*t*) *actxt*

show *finite* (*set2-actxt c*)
by(*induction c*) *auto*
qed (*auto*
simp: *actxt.set-map*(*2*) *actxt.map-comp fun.map-ident actxt.map-ident-strong*
cong: *actxt.map-cong*)

global-interpretation *context*: *natural-functor-conversion* **where**
map = *map-args-actxt* **and** *to-set* = *set2-actxt* **and** *map-to* = *map-args-actxt* **and**
map-from = *map-args-actxt* **and** *map'* = *map-args-actxt* **and** *to-set'* = *set2-actxt*
by *unfold-locales*
(*auto simp*: *actxt.set-map*(*2*) *actxt.map-comp cong*: *actxt.map-cong*)

locale *nonground-context* =
term: *nonground-term*
begin

sublocale *term-based-lifting* **where**

sub-subst = $(\cdot t)$ **and** *sub-vars* = *term.vars* **and**
to-set = *set2-actxt* :: $('f, 'v)$ *context* $\Rightarrow$ $('f, 'v)$ *term set* **and** *map* = *map-args-actxt*
and
sub-to-ground = *term.to-ground* **and** *sub-from-ground* = *term.from-ground* **and**
to-ground-map = *map-args-actxt* **and** *from-ground-map* = *map-args-actxt* **and**
ground-map = *map-args-actxt* **and** *to-set-ground* = *set2-actxt*
rewrites
$\bigwedge c\ \sigma.$ *subst* $c\ \sigma = c \cdot_c \sigma$ **and**
$\bigwedge c.$ *vars* c = *vars-ctxt* c
proof *unfold-locales*
interpret *term-based-lifting* **where**
sub-vars = *term.vars* **and** *sub-subst* = $(\cdot t)$ **and** *map* = *map-args-actxt* **and**
to-set = *set2-actxt* **and**
sub-to-ground = *term.to-ground* **and** *sub-from-ground* = *term.from-ground* **and**
ground-map = *map-args-actxt* **and** *to-ground-map* = *map-args-actxt* **and**
from-ground-map = *map-args-actxt* **and** *to-set-ground* = *set2-actxt*
by *unfold-locales*

fix c :: $('f, 'v)$ *context*
show *vars* c = *vars-ctxt* c
by(*induction* c) (*auto simp*: *vars-def*)

fix σ
show *subst* $c\ \sigma = c \cdot_c \sigma$
unfolding *subst-def*
by *blast*
qed

lemma *ground-ctxt-iff-context-is-ground* [*simp*]: *ground-ctxt* $c \longleftrightarrow$ *is-ground* c
by(*induction* c) *simp-all*

lemma *term-to-ground-context-to-ground* [*simp*]:
shows *term.to-ground* $c\langle t\rangle$ = (*to-ground* c)$\langle$*term.to-ground* $t\rangle_G$
unfolding *to-ground-def*
by(*induction* c) *simp-all*

lemma *term-from-ground-context-from-ground* [*simp*]:
term.from-ground $c_G\langle t_G\rangle_G$ = (*from-ground* c_G)$\langle$*term.from-ground* $t_G\rangle$
unfolding *from-ground-def*
by(*induction* c_G) *simp-all*

lemma *term-from-ground-context-to-ground*:
assumes *is-ground* c
shows *term.from-ground* (*to-ground* c)$\langle t_G\rangle_G = c\langle$*term.from-ground* $t_G\rangle$
unfolding *to-ground-def*
by (*metis assms term-from-ground-context-from-ground to-ground-def to-ground-inverse*)

lemmas *safe-unfolds* =
eval-ctxt

term-to-ground-context-to-ground
term-from-ground-context-from-ground

lemma *composed-context-is-ground* [*simp*]:
is-ground $(c \circ_c c') \longleftrightarrow$ *is-ground* $c \wedge$ *is-ground* c'
by(*induction c*) *auto*

lemma *ground-context-subst*:
assumes
$c_G = (c \cdot t_c \sigma) \circ_c c'$
shows
$c_G = c \circ_c c' \cdot t_c \sigma$
using *assms*
by(*induction c*) *simp-all*

lemma *from-ground-hole* [*simp*]: *from-ground* $c_G = \Box \longleftrightarrow c_G = \Box$
by(*cases* c_G) (*simp-all add*: *from-ground-def*)

lemma *hole-simps* [*simp*]: *from-ground* $\Box = \Box$ *to-ground* $\Box = \Box$
by (*auto simp*: *to-ground-def*)

lemma *term-with-context-is-ground* [*simp*]:
term.is-ground $c\langle t \rangle \longleftrightarrow$ *is-ground* $c \wedge$ *term.is-ground* t
by *simp*

lemma *map-args-actxt-compose* [*simp*]:
map-args-actxt f $(c \circ_c c') =$ *map-args-actxt* f $c \circ_c$ *map-args-actxt* f c'
by(*induction c*) *auto*

lemma *from-ground-compose* [*simp*]: *from-ground* $(c \circ_c c') =$ *from-ground* $c \circ_c$ *from-ground* c'
unfolding *from-ground-def*
by *simp*

lemma *to-ground-compose* [*simp*]: *to-ground* $(c \circ_c c') =$ *to-ground* $c \circ_c$ *to-ground* c'
unfolding *to-ground-def*
by *simp*

end

locale *nonground-term-with-context* =
term: *nonground-term* +
context: *nonground-context*

end
theory *Multiset-Grounding-Lifting*

imports
HOL−Library.Multiset
Abstract-Substitution.Functional-Substitution-Lifting
begin

locale *multiset-grounding-lifting* =
functional-substitution-lifting **where** *to-set* = *set-mset* **and** *map* = *image-mset*
+
grounding-lifting **where**
to-set = *set-mset* **and** *map* = *image-mset* **and** *to-ground-map* = *image-mset* **and**
from-ground-map = *image-mset* **and** *ground-map* = *image-mset* **and** *to-set-ground*
= *set-mset*
begin

sublocale *natural-magma-with-empty-grounding-lifting* **where**
plus = (+) **and** *wrap* = λ*l*. {#*l*#} **and** *plus-ground* = (+) **and** *wrap-ground* =
λ*l*. {#*l*#} **and**
empty = {#} **and** *empty-ground* = {#} **and** *to-set* = *set-mset* **and** *map* =
image-mset **and**
to-ground-map = *image-mset* **and** *from-ground-map* = *image-mset* **and** *ground-map*
= *image-mset* **and**
to-set-ground = *set-mset* **and** *add* = *add-mset* **and** *add-ground* = *add-mset*
by *unfold-locales* (*simp-all add*: *to-ground-def from-ground-def*)

sublocale *natural-magma-functor-functional-substitution-lifting* **where**
plus = (+) **and** *wrap* = λ*l*. {#*l*#} **and** *to-set* = *set-mset* **and** *map* = *image-mset*
and *add* = *add-mset*
by *unfold-locales simp-all*

end

end
theory *Nonground-Clause*
imports
Ground-Clause
Nonground-Term
Nonground-Context
Clausal-Calculus-Extra
Multiset-Extra
Multiset-Grounding-Lifting
begin

3 Nonground Clauses and Substitutions

type-synonym *'f ground-atom* = *'f gatom*
type-synonym (*'f*, *'v*) *atom* = (*'f*, *'v*) *term uprod*

locale *term-based-multiset-lifting* =
term-based-lifting **where**

map = *image-mset* **and** *to-set* = *set-mset* **and** *to-ground-map* = *image-mset* **and**
from-ground-map = *image-mset* **and** *ground-map* = *image-mset* **and** *to-set-ground*
= *set-mset*
begin

sublocale *multiset-grounding-lifting* **where**
id-subst = *Var* **and** *comp-subst* = (⊙)
by *unfold-locales*

end

locale *nonground-clause* = *nonground-term-with-context*
begin

3.1 Nonground Atoms

sublocale *atom*: *term-based-lifting* **where**
sub-subst = (·*t*) **and** *sub-vars* = *term.vars* **and** *map* = *map-uprod* **and** *to-set* =
set-uprod **and**
sub-to-ground = *term.to-ground* **and** *sub-from-ground* = *term.from-ground* **and**
to-ground-map = *map-uprod* **and** *from-ground-map* = *map-uprod* **and** *ground-map*
= *map-uprod* **and**
to-set-ground = *set-uprod*
by *unfold-locales*

notation *atom.subst* (**infixl** ·*a* 67)

lemma *vars-atom* [*simp*]: *atom.vars* (*Upair* t_1 t_2) = *term.vars* t_1 ∪ *term.vars* t_2
by (*simp-all add*: *atom.vars-def*)

lemma *subst-atom* [*simp*]:
Upair t_1 t_2 ·*a* σ = *Upair* (t_1 ·*t* σ) (t_2 ·*t* σ)
unfolding *atom.subst-def*
by *simp-all*

lemma *atom-from-ground-term-from-ground* [*simp*]:
atom.from-ground (*Upair* t_{G1} t_{G2}) = *Upair* (*term.from-ground* t_{G1}) (*term.from-ground*
t_{G2})
by (*simp add*: *atom.from-ground-def*)

lemma *atom-to-ground-term-to-ground* [*simp*]:
atom.to-ground (*Upair* t_1 t_2) = *Upair* (*term.to-ground* t_1) (*term.to-ground* t_2)
by (*simp add*: *atom.to-ground-def*)

lemma *atom-is-ground-term-is-ground* [*simp*]:
atom.is-ground (*Upair* t_1 t_2) ⟷ *term.is-ground* t_1 ∧ *term.is-ground* t_2
by *simp*

lemma *obtain-from-atom-subst*:

assumes *Upair* $t_1{}'\ t_2{}' = a \cdot a\ \sigma$
obtains $t_1\ t_2$
where $a = Upair\ t_1\ t_2\ t_1{}' = t_1 \cdot t\ \sigma\ t_2{}' = t_2 \cdot t\ \sigma$
using *assms*
unfolding *atom.subst-def*
by(*cases a*) *force*

3.2 Nonground Literals

sublocale *literal*: *term-based-lifting* **where**
sub-subst = *atom.subst* **and** *sub-vars* = *atom.vars* **and** *map* = *map-literal* **and**
to-set = *set-literal* **and** *sub-to-ground* = *atom.to-ground* **and**
sub-from-ground = *atom.from-ground* **and** *to-ground-map* = *map-literal* **and**
from-ground-map = *map-literal* **and** *ground-map* = *map-literal* **and** *to-set-ground* = *set-literal*
by *unfold-locales*

notation *literal.subst* (**infixl** $\cdot l$ *66*)

lemma *vars-literal* [*simp*]:
literal.vars (*Pos a*) = *atom.vars a*
literal.vars (*Neg a*) = *atom.vars a*
literal.vars ((*if b then Pos else Neg*) *a*) = *atom.vars a*
by (*simp-all add*: *literal.vars-def*)

lemma *subst-literal* [*simp*]:
Pos a $\cdot l\ \sigma$ = *Pos* (*a* $\cdot a\ \sigma$)
Neg a $\cdot l\ \sigma$ = *Neg* (*a* $\cdot a\ \sigma$)
atm-of (*l* $\cdot l\ \sigma$) = *atm-of l* $\cdot a\ \sigma$
unfolding *literal.subst-def*
using *literal.map-sel*
by *auto*

lemma *subst-literal-if* [*simp*]:
(*if b then Pos else Neg*) *a* $\cdot l\ \varrho$ = (*if b then Pos else Neg*) (*a* $\cdot a\ \varrho$)
by *simp*

lemma *subst-polarity-stable*:
shows
subst-neg-stable [*simp*]: *is-neg* (*l* $\cdot l\ \sigma$) $\longleftrightarrow$ *is-neg l* **and**
subst-pos-stable [*simp*]: *is-pos* (*l* $\cdot l\ \sigma$) $\longleftrightarrow$ *is-pos l*
by (*simp-all add*: *literal.subst-def*)

declare *literal.discI* [*intro*]

lemma *literal-from-ground-atom-from-ground* [*simp*]:
literal.from-ground (*Neg* a_G) = *Neg* (*atom.from-ground* a_G)
literal.from-ground (*Pos* a_G) = *Pos* (*atom.from-ground* a_G)
by (*simp-all add*: *literal.from-ground-def*)

lemma *literal-from-ground-polarity-stable* [*simp*]:
shows
neg-literal-from-ground-stable: *is-neg* (*literal.from-ground* l_G) $\longleftrightarrow$ *is-neg* l_G **and**
pos-literal-from-ground-stable: *is-pos* (*literal.from-ground* l_G) $\longleftrightarrow$ *is-pos* l_G
by (*simp-all add*: *literal.from-ground-def*)

lemma *literal-to-ground-atom-to-ground* [*simp*]:
literal.to-ground (*Pos a*) = *Pos* (*atom.to-ground a*)
literal.to-ground (*Neg a*) = *Neg* (*atom.to-ground a*)
by (*simp-all add*: *literal.to-ground-def*)

lemma *literal-is-ground-atom-is-ground* [*intro*]:
literal.is-ground l $\longleftrightarrow$ *atom.is-ground* (*atm-of l*)
by (*simp add*: *literal.vars-def set-literal-atm-of*)

lemma *obtain-from-pos-literal-subst*:
assumes $l \cdot l \ \sigma = t_1' \approx t_2'$
obtains t_1 t_2
where $l = t_1 \approx t_2$ $t_1' = t_1 \cdot t \ \sigma$ $t_2' = t_2 \cdot t \ \sigma$
using *assms obtain-from-atom-subst subst-pos-stable*
by (*metis is-pos-def literal.sel*(*1*) *subst-literal*(*3*))

lemma *obtain-from-neg-literal-subst*:
assumes $l \cdot l \ \sigma = t_1' \ !\approx \ t_2'$
obtains t_1 t_2
where $l = t_1 \ !\approx \ t_2$ $t_1 \cdot t \ \sigma = t_1'$ $t_2 \cdot t \ \sigma = t_2'$
using *assms obtain-from-atom-subst subst-neg-stable*
by (*metis literal.collapse*(*2*) *literal.disc*(*2*) *literal.sel*(*2*) *subst-literal*(*3*))

lemmas *obtain-from-literal-subst* = *obtain-from-pos-literal-subst obtain-from-neg-literal-subst*

3.3 Nonground Literals - Alternative

lemma *uprod-literal-subst-eq-literal-subst*: *map-uprod-literal* ($\lambda t.\ t \cdot t \ \sigma$) $l = l \cdot l \ \sigma$
unfolding *atom.subst-def literal.subst-def*
by *auto*

lemma *uprod-literal-vars-eq-literal-vars*: $\bigcup$ (*term.vars* ‘ *uprod-literal-to-set l*) = *literal.vars l*
unfolding *literal.vars-def atom.vars-def*
by(*cases l*) *simp-all*

lemma *uprod-literal-from-ground-eq-literal-from-ground*:
map-uprod-literal term.from-ground l_G = *literal.from-ground* l_G
unfolding *literal.from-ground-def atom.from-ground-def* **..**

lemma *uprod-literal-to-ground-eq-literal-to-ground*:
map-uprod-literal term.to-ground l = *literal.to-ground l*

unfolding *literal.to-ground-def atom.to-ground-def* **..**

sublocale *uprod-literal*: *term-based-lifting* **where**
sub-subst = $(\cdot t)$ **and** *sub-vars* = *term.vars* **and** *map* = *map-uprod-literal* **and**
to-set = *uprod-literal-to-set* **and** *sub-to-ground* = *term.to-ground* **and**
sub-from-ground = *term.from-ground* **and** *to-ground-map* = *map-uprod-literal*
and
from-ground-map = *map-uprod-literal* **and** *ground-map* = *map-uprod-literal* **and**
to-set-ground = *uprod-literal-to-set*
rewrites
uprod-literal-subst [*simp*]: $\bigwedge l\ \sigma$. *uprod-literal.subst l* σ = *literal.subst l* σ **and**
uprod-literal-vars [*simp*]: $\bigwedge l$. *uprod-literal.vars l* = *literal.vars l* **and**
uprod-literal-from-ground [*simp*]: $\bigwedge l_G$. *uprod-literal.from-ground* l_G = *literal.from-ground* l_G **and**
uprod-literal-to-ground [*simp*]:$\bigwedge l$. *uprod-literal.to-ground l* = *literal.to-ground l*
proof *unfold-locales*

interpret *term-based-lifting* **where**
sub-vars = *term.vars* **and** *sub-subst* = $(\cdot t)$ **and** *map* = *map-uprod-literal* **and**
to-set = *uprod-literal-to-set* **and** *sub-to-ground* = *term.to-ground* **and**
sub-from-ground = *term.from-ground* **and** *to-ground-map* = *map-uprod-literal*
and
from-ground-map = *map-uprod-literal* **and** *ground-map* = *map-uprod-literal* **and**
to-set-ground = *uprod-literal-to-set*
by *unfold-locales*

fix *l* :: $('f, 'v)$ *atom literal* **and** σ

show *subst l* σ = *l* $\cdot l$ σ
unfolding *subst-def literal.subst-def atom.subst-def*
by *simp*

show *vars l* = *literal.vars l*
unfolding *atom.vars-def vars-def literal.vars-def*
by(*cases l*) *simp-all*

fix l_G:: *'f ground-atom literal*
show *from-ground* l_G = *literal.from-ground* l_G
unfolding *from-ground-def literal.from-ground-def atom.from-ground-def***..**

fix *l* :: $('f, 'v)$ *atom literal*
show *to-ground l* = *literal.to-ground l*
unfolding *to-ground-def literal.to-ground-def atom.to-ground-def***..**
qed

lemma *mset-literal-from-ground*:
mset-lit (*literal.from-ground l*) = *image-mset term.from-ground* (*mset-lit l*)
by (*simp add*: *uprod-literal.from-ground-def mset-lit-image-mset*)

3.4 Nonground Clauses

sublocale *clause*: *term-based-multiset-lifting* **where**
sub-subst = *literal.subst* **and** *sub-vars* = *literal.vars* **and** *sub-to-ground* = *literal.to-ground* **and**
sub-from-ground = *literal.from-ground*
by *unfold-locales*

notation *clause.subst* (**infixl** · *67*)

lemmas *clause-submset-vars-clause-subset* [*intro*] =
clause.to-set-subset-vars-subset[*OF set-mset-mono*]

lemmas *sub-ground-clause* = *clause.to-set-subset-is-ground*[*OF set-mset-mono*]

lemma *subst-clause-remove1-mset* [*simp*]:
assumes $l \in\# C$
shows *remove1-mset* $l\ C \cdot \sigma$ = *remove1-mset* $(l \cdot l\ \sigma)\ (C \cdot \sigma)$
unfolding *clause.subst-def image-mset-remove1-mset-if*
using *assms*
by *simp*

lemma *clause-from-ground-remove1-mset* [*simp*]:
clause.from-ground (*remove1-mset* $l_G\ C_G$) =
remove1-mset (*literal.from-ground* l_G) (*clause.from-ground* C_G)
unfolding *clause.from-ground-def image-mset-remove1-mset*[*OF literal.inj-from-ground*]**..**

lemmas *clause-safe-unfolds* =
atom-to-ground-term-to-ground
literal-to-ground-atom-to-ground
atom-from-ground-term-from-ground
literal-from-ground-atom-from-ground
literal-from-ground-polarity-stable
subst-atom
subst-literal
vars-atom
vars-literal

end

end
theory *Selection-Function*
imports *Ordered-Resolution-Prover.Clausal-Logic*
begin

locale *selection-function* =
fixes *select* :: *'a clause* ⇒ *'a clause*
assumes
select-subset: $\bigwedge C$. *select* $C \subseteq\# C$ **and**
select-negative-literals: $\bigwedge C\ l$. $l \in\#$ *select* $C \Longrightarrow$ *is-neg* l

end
theory *Nonground-Selection-Function*
imports
Nonground-Clause
Selection-Function
begin

type-synonym *'f ground-select* = *'f ground-atom clause* ⇒ *'f ground-atom clause*
type-synonym *('f, 'v) select* = *('f, 'v) atom clause* ⇒ *('f, 'v) atom clause*

context *nonground-clause*
begin

definition *is-select-grounding* :: *('f, 'v) select* ⇒ *'f ground-select* ⇒ *bool* **where**
is-select-grounding select select$_G$ ≡ ∀ C_G. ∃ C γ.
clause.is-ground $(C \cdot \gamma)$ ∧
C_G = *clause.to-ground* $(C \cdot \gamma)$ ∧
select$_G$ C_G = *clause.to-ground* $((select\ C) \cdot \gamma)$

end

locale *nonground-selection-function* =
nonground-clause +
selection-function select
for *select* :: *('f, 'v) atom clause* ⇒ *('f, 'v) atom clause*
begin

abbreviation *is-grounding* :: *'f ground-select* ⇒ *bool* **where**
is-grounding select$_G$ ≡ *is-select-grounding select select*$_G$

definition *select*$_{Gs}$ **where**
select$_{Gs}$ = { *select*$_G$. *is-grounding select*$_G$ }

definition *select*$_G$*-simple* **where**
select$_G$*-simple* C = *clause.to-ground* (*select* (*clause.from-ground* C))

lemma *select*$_G$*-simple*: *is-grounding select*$_G$*-simple*
unfolding *is-select-grounding-def select*$_G$*-simple-def*
by (*metis clause.from-ground-inverse clause.ground-is-ground clause.subst-id-subst*)

lemma *select-is-ground*:
assumes *clause.is-ground C*
shows *clause.is-ground* (*select C*)
using *select-subset sub-ground-clause assms*
by *metis*

lemma *is-ground-in-selection*:
assumes *l* ∈# *select* (*clause.from-ground C*)

shows *literal.is-ground l*
using *assms clause.sub-in-ground-is-ground select-subset*
by *blast*

lemma *ground-literal-in-selection*:
assumes *clause.is-ground C* l_G $\in\#$ *clause.to-ground C*
shows *literal.from-ground* l_G $\in\#$ *C*
using *assms*
by (*metis clause.to-ground-inverse clause.ground-sub-in-ground*)

lemma *select-ground-subst*:
assumes *clause.is-ground* $(C \cdot \gamma)$
shows *clause.is-ground* (*select C* $\cdot$ γ)
using *assms*
by (*metis image-mset-subseteq-mono select-subset sub-ground-clause clause.subst-def*)

lemma *select-neg-subst*:
assumes *l* $\in\#$ *select C* $\cdot$ γ
shows *is-neg l*
using *assms subst-neg-stable select-negative-literals*
unfolding *clause.subst-def*
by *blast*

lemma *select-vars-subset*: $\bigwedge C$. *clause.vars* (*select C*) $\subseteq$ *clause.vars C*
by (*simp add*: *clause-submset-vars-clause-subset select-subset*)

end

end
theory *Collect-Extra*
imports *Main*
begin

lemma *Collect-if-eq*: $\{x.$ *if b x then P x else Q x* $\} = \{x.$ *b x* $\wedge$ *P x* $\} \cup \{x. \neg b\ x \wedge Q\ x\}$
by *auto*

lemma *Collect-not-mem-conj-eq*: $\{x.\ x \notin X \wedge P\ x\} = \{x.\ P\ x\} - X$
by *auto*

end
theory *Infinite-Variables-Per-Type*
imports
HOL−Library.Countable-Set
HOL−Cardinals.Cardinals
Fresh-Identifiers.Fresh
Collect-Extra
begin

lemma *infinite-prods*:
  assumes *infinite* (*UNIV* :: *'a set*)
  shows *infinite* {*p* :: *'a* × *'a*. *fst p* = *x*}
proof −
  have {*p* :: *'a* × *'a* . *fst p* = *x*} = {*x*} × *UNIV*
    by *auto*

  then show *?thesis*
    using *finite-cartesian-productD2 assms*
    by *auto*
qed

lemma *surj-infinite-set*: *surj g* ⟹ *infinite* {*x*. *f x* = *τ*} ⟹ *infinite* {*x*. *f* (*g x*) = *τ*}
  by (*smt* (*verit*) *UNIV-I finite-imageI image-iff mem-Collect-eq rev-finite-subset subset-eq*)

definition *infinite-variables-per-type-on* :: *'var set* ⇒ (*'var* ⇒ *'ty*) ⇒ *bool* **where**
  infinite-variables-per-type-on X 𝒱 ≡ ∀ *τ* ∈ 𝒱 ‘ *X*. *infinite* {*x*. 𝒱 *x* = *τ*}

abbreviation *infinite-variables-per-type* :: (*'var* ⇒ *'ty*) ⇒ *bool* **where**
  infinite-variables-per-type ≡ *infinite-variables-per-type-on UNIV*

lemma *obtain-type-preserving-inj*:
  fixes 𝒱 :: *'v* ⇒ *'ty*
  assumes
    finite-X: *finite X* **and**
    𝒱: *infinite-variables-per-type* 𝒱
  obtains *f* :: *'v* ⇒ *'v* **where**
    inj f
    X ∩ *f* ‘ *Y* = {}
    ∀ *x* ∈ *Y*. 𝒱 (*f x*) = 𝒱 *x*
proof (*rule that*)

  {
    fix *τ*
    assume *τ* ∈ *range* 𝒱

    then have |{*x*. 𝒱 *x* = *τ*}| =*o* |{*x*. 𝒱 *x* = *τ* } − *X*|
      using 𝒱 *finite-X card-of-infinite-diff-finite ordIso-symmetric*
      unfolding *infinite-variables-per-type-on-def*
      by *blast*

    then have |{*x*. 𝒱 *x* = *τ*}| =*o* |{*x*. 𝒱 *x* = *τ* ∧ *x* ∉ *X*}|
      using *set-diff-eq*[*of* - *X*]
      by *auto*

    then have ∃ *g*. *bij-betw g* {*x*. 𝒱 *x* = *τ*} {*x*. 𝒱 *x* = *τ* ∧ *x* ∉ *X*}
      using *card-of-ordIso someI*

```
    by blast
  }
  note exists-g = this

  define get-g where
    ⋀τ. get-g τ ≡ SOME g. bij-betw g {x. 𝒱 x = τ} {x. 𝒱 x = τ ∧ x ∉ X}

  define f where
    ⋀x. f x ≡ get-g (𝒱 x) x

  {
    fix y

    have ⋀g. bij-betw g {x. 𝒱 x = 𝒱 y} {x. 𝒱 x = 𝒱 y ∧ x ∉ X} ⟹ g y ∈ {x. 𝒱
x = 𝒱 y ∧ x ∉ X}
      using exists-g bij-betwE
      by blast

    then have f y ∈ {x. 𝒱 x = 𝒱 y ∧ x ∉ X}
      using exists-g get-g-def
      unfolding f-def
      by (metis (no-types, lifting) ext rangeI verit-sko-ex')
  }

  then show X ∩ f ‘ Y = {}  ∀y∈Y. 𝒱 (f y) = 𝒱 y
    unfolding f-def
    by auto

  show inj f
  proof (unfold inj-def, intro allI impI)
    fix x y
    assume f x = f y

    then show x = y
      using get-g-def f-def exists-g
      unfolding some-eq-ex[symmetric]
      by (smt (verit) bij-betw-iff-bijections mem-Collect-eq rangeI)
  qed
qed

lemma obtain-type-preserving-injs:
  fixes 𝒱₁ 𝒱₂ :: 'v ⇒ 'ty
  assumes
    finite-X: finite X and
    𝒱₂: infinite-variables-per-type 𝒱₂
  obtains f f' :: 'v ⇒ 'v where
    inj f inj f'
    f ‘ X ∩ f' ‘ Y = {}
    ∀x ∈ X. 𝒱₁ (f x) = 𝒱₁ x
```

```
    ∀x ∈ Y. 𝒱₂ (f' x) = 𝒱₂ x
proof −

  obtain f' where f':
    inj f'
    X ∩ f' ‘ Y = {}
    ∀x ∈ Y. 𝒱₂ (f' x) = 𝒱₂ x
    using obtain-type-preserving-inj[OF assms] .

  show ?thesis
    by (rule that[of id f']) (auto simp: f')
qed

lemma obtain-type-preserving-injs':
  fixes 𝒱₁ 𝒱₂ :: 'v ⇒ 'ty
  assumes
    finite-Y: finite Y and
    𝒱₁: infinite-variables-per-type 𝒱₁
  obtains f f' :: 'v ⇒ 'v where
    inj f inj f'
    f ‘ X ∩ f' ‘ Y = {}
    ∀x ∈ X. 𝒱₁ (f x) = 𝒱₁ x
    ∀x ∈ Y. 𝒱₂ (f' x) = 𝒱₂ x
  using obtain-type-preserving-injs[OF assms]
  by (metis inf-commute)

lemma obtain-infinite-variables-per-type-on:
  assumes
    infinite-UNIV: infinite (UNIV :: 'v set) and
    finite-Y: finite Y and
    finite-Z: finite Z and
    disjoint: Y ∩ Z = {}
  obtains 𝒱 :: 'v ⇒ 'ty
  where infinite-variables-per-type-on X 𝒱 ∀x ∈ Y. 𝒱 x = 𝒱' x ∀x ∈ Z. 𝒱 x =
𝒱'' x
proof (cases X = {})
  case True
  define 𝒱 where ⋀x. 𝒱 x ≡ if x ∈ Y then 𝒱' x else 𝒱'' x

  show ?thesis
  proof (rule that[unfolded True])

    show ∀x∈Y. 𝒱 x = 𝒱' x
      unfolding 𝒱-def
      by simp
  next

    show ∀x∈Z. 𝒱 x = 𝒱'' x
      using disjoint
```

```
    unfolding 𝒱-def
    by auto
  qed (auto simp: infinite-variables-per-type-on-def)
next
  case False

  obtain g :: 'v ⇒ 'v × 'v where bij-g: bij g
    using Times-same-infinite-bij-betw-types bij-betw-inv infinite-UNIV
    by blast

  define f :: 'v ⇒ 'v where
    ⋀x. f x ≡ if x ∈ Y ∪ Z then x else fst (g x)

  define 𝒱 where ⋀x. 𝒱 x ≡ if x ∈ Y then 𝒱' x else 𝒱'' x

  {
    fix y

    have {x. fst (g x) = y} = inv g ‘ {p. fst p = y}
     by (smt (verit, ccfv-SIG) Collect-cong bij-g bij-image-Collect-eq bij-imp-bij-inv
inv-inv-eq)

    then have infinite {x. fst (g x) = y}
      using infinite-prods[OF infinite-UNIV]
      by (metis bij-g bij-is-surj finite-imageI image-f-inv-f)

    then have infinite {x. x ∉ Y ∪ Z ∧ fst (g x) = y}
      using finite-Y finite-Z
      unfolding Collect-not-mem-conj-eq
      by simp

    then have infinite {x. f x = y}
      unfolding f-def if-distrib if-distribR Collect-if-eq
      by blast
  }

  then have 𝒱-X: ∀ y ∈ 𝒱 ‘ f ‘ X. infinite {x. 𝒱 (f x) = y}
    by (smt (verit, best) Collect-mono imageE rev-finite-subset)

  show ?thesis
  proof (rule that)
    show infinite-variables-per-type-on X (𝒱 ∘ f)
      using 𝒱-X
      unfolding infinite-variables-per-type-on-def comp-def
      by (metis image-image)
  next
    show ∀ x∈Y. (𝒱 ∘ f) x = 𝒱' x
      unfolding f-def 𝒱-def
      by auto
```

next
show $\forall x{\in}Z.\ (\mathcal{V} \circ f)\ x = \mathcal{V}''\ x$
using *disjoint*
unfolding *f-def* $\mathcal{V}$*-def*
by *auto*
qed
qed

lemma *obtain-infinite-variables-per-type-on'*:
assumes *infinite-UNIV*: *infinite* (*UNIV* :: *'v set*) **and** *finite-Y*: *finite Y*
obtains $\mathcal{V}$ :: *'v* $\Rightarrow$ *'ty*
where *infinite-variables-per-type-on X* $\mathcal{V}$ $\forall x \in Y.\ \mathcal{V}\ x = \mathcal{V}'\ x$
using *obtain-infinite-variables-per-type-on*[*OF infinite-UNIV finite-Y*, *of* {}]
by *auto*

lemma *obtain-infinite-variables-per-type-on''*:
assumes *finite Y*
obtains $\mathcal{V}$ :: *'v* :: *infinite* $\Rightarrow$ *'ty*
where *infinite-variables-per-type-on X* $\mathcal{V}$ $\forall x \in Y.\ \mathcal{V}\ x = \mathcal{V}'\ x$
using *obtain-infinite-variables-per-type-on'*[*OF infinite-UNIV assms*]**.**

lemma *infinite-variables-per-type-on-subset*:
$X \subseteq Y \Longrightarrow$ *infinite-variables-per-type-on Y* $\mathcal{V} \Longrightarrow$ *infinite-variables-per-type-on X* $\mathcal{V}$
unfolding *infinite-variables-per-type-on-def*
by *blast*

definition *infinite-variables-for-all-types* :: (*'v* $\Rightarrow$ *'ty*) $\Rightarrow$ *bool* **where**
infinite-variables-for-all-types $\mathcal{V} \equiv \forall \tau.$ *infinite* $\{x.\ \mathcal{V}\ x = \tau\}$

lemma *exists-infinite-variables-for-all-types*:
assumes |*UNIV* :: *'ty set*| $\leq o$ |*UNIV* :: (*'v* :: *infinite*) *set*|
shows $\exists \mathcal{V}$ :: *'v* $\Rightarrow$ *'ty.* *infinite-variables-for-all-types* $\mathcal{V}$
proof−
obtain *g* :: *'v* $\Rightarrow$ *'v* $\times$ *'v* **where** *bij-g*: *bij g*
using *Times-same-infinite-bij-betw-types bij-betw-inv infinite-UNIV*
by *blast*

define *f* :: *'v* $\Rightarrow$ *'v* **where**
$\bigwedge x.\ f\ x \equiv fst\ (g\ x)$

{
fix *y*

have $\{x.\ fst\ (g\ x) = y\} = inv\ g\ `\ \{p.\ fst\ p = y\}$
by (*smt* (*verit*, *ccfv-SIG*) *Collect-cong bij-g bij-image-Collect-eq bij-imp-bij-inv inv-inv-eq*)

then have *infinite* $\{x.\ f\ x = y\}$

unfolding *f-def*
using *infinite-prods*[*OF infinite-UNIV*]
by (*metis bij-g bij-is-surj finite-imageI image-f-inv-f*)
}

moreover obtain $f' :: 'v \Rightarrow 'ty$ **where** *surj* f'
using *assms*
by (*metis card-of-ordLeq2 empty-not-UNIV*)

ultimately have $\bigwedge y.$ *infinite* $\{x.\ f'\ (f\ x) = y\}$
by (*smt* (*verit*, *ccfv-SIG*) *Collect-mono finite-subset surjD*)

then show *?thesis*
unfolding *infinite-variables-for-all-types-def*
by *meson*
qed

lemma *obtain-infinite-variables-for-all-types*:
assumes $|UNIV :: 'ty\ set| \leq_o |UNIV :: 'v\ set|$
obtains $\mathcal{V} :: 'v :: infinite \Rightarrow 'ty$ **where** *infinite-variables-for-all-types* $\mathcal{V}$
using *exists-infinite-variables-for-all-types*[*OF assms*]
by *blast*

lemma *infinite-variables-per-type-if-infinite-variables-for-all-types*:
infinite-variables-for-all-types $\mathcal{V} \Longrightarrow$ *infinite-variables-per-type* $\mathcal{V}$
unfolding *infinite-variables-for-all-types-def infinite-variables-per-type-on-def*
by *blast*

end
theory *Typed-Functional-Substitution*
imports
Typing
Abstract-Substitution.Functional-Substitution
Infinite-Variables-Per-Type
begin

type-synonym $('v, 'ty)$ *var-types* $= 'v \Rightarrow 'ty$

locale *base-typed-functional-substitution* =
base-functional-substitution **where** *id-subst* = *id-subst*
for
id-subst $:: 'v \Rightarrow 'base$ **and**
welltyped $:: ('v, 'ty)$ *var-types* $\Rightarrow 'base \Rightarrow 'ty \Rightarrow$ *bool* +
assumes
base-typing: $\bigwedge \mathcal{V}.$ *Typing.typing* (*welltyped* $\mathcal{V}$) **and**
typed-id-subst [*intro*]: $\bigwedge \mathcal{V}\ x.$ *welltyped* $\mathcal{V}$ (*id-subst* x) ($\mathcal{V}\ x$)
begin

sublocale *typing welltyped* $\mathcal{V}$

using *base-typing* **.**

abbreviation *is-welltyped-on* :: *'v set* ⇒ (*'v*, *'ty*) *var-types* ⇒ (*'v* ⇒ *'base*) ⇒ *bool*
where
is-welltyped-on X $\mathcal{V}$ $\sigma \equiv \forall x \in X$. *welltyped* $\mathcal{V}$ (σ *x*) ($\mathcal{V}$ *x*)

lemma *subst-update*:
assumes *welltyped* $\mathcal{V}$ (*id-subst var*) τ *welltyped* $\mathcal{V}$ *update* τ *is-welltyped-on X* $\mathcal{V}$ γ
shows *is-welltyped-on X* $\mathcal{V}$ (γ(*var* := *update*))
using *assms typed-id-subst*
by *fastforce*

lemma *is-typed-on-subset*:
assumes *is-welltyped-on Y* $\mathcal{V}$ σ *X* ⊆ *Y*
shows *is-welltyped-on X* $\mathcal{V}$ σ
using *assms*
by *blast*

lemma *is-typed-id-subst* [*intro*]: *is-welltyped-on X* $\mathcal{V}$ *id-subst*
by *auto*

end

locale *typed-functional-substitution* =
base: *base-typed-functional-substitution* **where**
subst = *base-subst* **and** *vars* = *base-vars* **and** *welltyped* = *base-welltyped* +
based-functional-substitution **where** *vars* = *vars*
for
base-welltyped :: (*'v*, *'ty*) *var-types* ⇒ *'base* ⇒ *'ty* ⇒ *bool* **and**
vars :: *'expr* ⇒ *'v set* **and**
welltyped :: (*'v*, *'ty*) *var-types* ⇒ *'expr* ⇒ *'ty'* ⇒ *bool* +
assumes
base-typing0: $\bigwedge\mathcal{V}$. *Typing.typing* (*welltyped* $\mathcal{V}$)
begin

sublocale *typing welltyped* $\mathcal{V}$
using *base-typing0* **.**

abbreviation *is-welltyped-ground-instance* **where**
is-welltyped-ground-instance expr $\mathcal{V}$ $\gamma \equiv$
is-ground (*expr* · γ) ∧
(∃τ. *welltyped* $\mathcal{V}$ *expr* τ) ∧
base.is-welltyped-on (*vars expr*) $\mathcal{V}$ γ ∧
infinite-variables-per-type $\mathcal{V}$

end

locale *inhabited-typed-functional-substitution* =
typed-functional-substitution +
assumes *types-inhabited*: $\bigwedge \mathcal{V}\ \tau.\ \exists\, b.\ base.is\text{-}ground\ b \wedge base\text{-}welltyped\ \mathcal{V}\ b\ \tau$
begin

lemma *ground-subst-extension*:
assumes
grounding: *is-ground* (*expr* · γ) **and**
γ-*is-typed-on*: *base.is-welltyped-on* (*vars expr*) $\mathcal{V}$ γ
obtains γ'
where
base.is-ground-subst γ'
base.is-welltyped-on UNIV $\mathcal{V}$ γ'
$\forall\, x \in vars\ expr.\ \gamma\ x = \gamma'\ x$
proof (*rule that*)

define γ' **where**
$\bigwedge x.\ \gamma'\ x \equiv$
if $x \in vars\ expr$
then $\gamma\ x$
else SOME base. base.is-ground base $\wedge$ *base-welltyped* $\mathcal{V}$ *base* ($\mathcal{V}\ x$)

show *base.is-ground-subst* γ'
proof(*unfold base.is-ground-subst-def*, *intro allI*)
fix *b*

{
fix *x*

have *base.is-ground* ($\gamma'\ x$)
proof(*cases* $x \in vars\ expr$)
case *True*

then show *?thesis*
unfolding γ'*-def*
using *variable-grounding*[*OF grounding*]
by *auto*
next
case *False*

then show *?thesis*
unfolding γ'*-def*
by (*smt* (*verit*) *types-inhabited tfl-some*)
qed
}

then show *base.is-ground* (*base-subst b* γ')

```
    using base.is-grounding-iff-vars-grounded
    by auto
  qed

  show base.is-welltyped-on UNIV 𝒱 γ'
    unfolding γ'-def
    using γ-is-typed-on types-inhabited
    by (simp add: verit-sko-ex-indirect)

  show ∀ x ∈ vars expr. γ x = γ' x
    by (simp add: γ'-def)
qed

lemma grounding-extension:
  assumes
    grounding: is-ground (expr · γ) and
    γ-is-typed-on: base.is-welltyped-on (vars expr) 𝒱 γ
  obtains γ'
  where
    is-ground (expr' · γ')
    base.is-welltyped-on (vars expr') 𝒱 γ'
    ∀ x ∈ vars expr. γ x = γ' x
  using ground-subst-extension[OF grounding γ-is-typed-on]
  unfolding base.is-ground-subst-def is-grounding-iff-vars-grounded
  by (metis UNIV-I base.comp-subst-iff base.left-neutral)

end

sublocale base-typed-functional-substitution ⊆ typed-functional-substitution where
  base-subst = subst and base-vars = vars and base-welltyped = welltyped
  by unfold-locales

locale typed-grounding-functional-substitution =
  typed-functional-substitution + grounding
begin

definition welltyped-ground-instances where
  welltyped-ground-instances typed-expr =
    { to-ground (fst typed-expr · γ) | γ.
      is-welltyped-ground-instance (fst typed-expr) (snd typed-expr) γ }

lemma typed-ground-instances-ground-instances':
  welltyped-ground-instances (expr, 𝒱) ⊆ ground-instances' expr
  unfolding welltyped-ground-instances-def ground-instances'-def
  by auto

end

locale typed-subst-stability = typed-functional-substitution +
```

assumes
subst-stability [*simp*]: $\bigwedge \mathcal{V}$ *expr* σ τ.
base.is-welltyped-on (*vars expr*) $\mathcal{V}$ $\sigma \Longrightarrow$ *welltyped* $\mathcal{V}$ (*expr* $\cdot$ σ) $\tau \longleftrightarrow$ *welltyped* $\mathcal{V}$ *expr* τ
begin

lemma *subst-stability-UNIV* [*simp*]:
base.is-welltyped-on UNIV $\mathcal{V}$ $\sigma \Longrightarrow$ *welltyped* $\mathcal{V}$ (*expr* $\cdot$ σ) $\tau \longleftrightarrow$ *welltyped* $\mathcal{V}$ *expr* τ
by *simp*

end

locale *base-typed-subst-stability* =
base-typed-functional-substitution +
typed-subst-stability **where** *base-subst* = *subst* **and** *base-vars* = *vars* **and** *base-welltyped* = *welltyped*
begin

lemma *typed-subst-compose* [*intro*]:
assumes
is-welltyped-on X $\mathcal{V}$ σ
is-welltyped-on ($\bigcup$(*vars* ‘ σ ‘ X)) $\mathcal{V}$ σ'
shows *is-welltyped-on* X $\mathcal{V}$ ($\sigma \odot \sigma'$)
using *assms*
unfolding *comp-subst-iff*
by *auto*

lemma *typed-subst-compose-UNIV* [*intro*]:
assumes
is-welltyped-on UNIV $\mathcal{V}$ σ
is-welltyped-on UNIV $\mathcal{V}$ σ'
shows *is-welltyped-on UNIV* $\mathcal{V}$ ($\sigma \odot \sigma'$)
using *assms*
unfolding *comp-subst-iff*
by *auto*

end

locale *replaceable-*$\mathcal{V}$ = *typed-functional-substitution* +
assumes *replace-*$\mathcal{V}$:
$\bigwedge$*expr* $\mathcal{V}$ $\mathcal{V}'$ τ. $\forall$ $x \in$ *vars expr*. $\mathcal{V}$ x = $\mathcal{V}'$ $x \Longrightarrow$ *welltyped* $\mathcal{V}$ *expr* $\tau \Longrightarrow$ *welltyped* $\mathcal{V}'$ *expr* τ
begin

lemma *replace-*$\mathcal{V}$*-iff*:
assumes $\forall$ $x \in$ *vars expr*. $\mathcal{V}$ x = $\mathcal{V}'$ x
shows *welltyped* $\mathcal{V}$ *expr* $\tau \longleftrightarrow$ *welltyped* $\mathcal{V}'$ *expr* τ
using *assms*

by (*metis replace-*$\mathcal{V}$)

lemma *is-ground-typed*:
assumes *is-ground expr*
shows *welltyped* $\mathcal{V}$ *expr* $\tau \longleftrightarrow$ *welltyped* $\mathcal{V}'$ *expr* τ
using *replace-*$\mathcal{V}$*-iff assms*
by *blast*

end

locale *typed-renaming* =
typed-functional-substitution +
renaming-variables +
assumes
typed-renaming [*simp*]:
$\bigwedge \mathcal{V}\ \mathcal{V}'$ *expr* $\varrho\ \tau$. *base.is-renaming* $\varrho \Longrightarrow$
$\forall x \in$ *vars expr*. $\mathcal{V}\ x = \mathcal{V}'$ (*rename* $\varrho\ x$) $\Longrightarrow$
welltyped $\mathcal{V}'$ (*expr* $\cdot\ \varrho$) $\tau \longleftrightarrow$ *welltyped* $\mathcal{V}$ *expr* τ

locale *base-typed-renaming* =
base-typed-functional-substitution **where**
welltyped = *welltyped* +
typed-renaming **where**
base-subst = *subst* **and** *base-vars* = *vars* **and** *base-welltyped* = *welltyped* **and**
welltyped = *welltyped* +
replaceable-$\mathcal{V}$ **where**
base-subst = *subst* **and** *base-vars* = *vars* **and** *base-welltyped* = *welltyped* **and**
welltyped = *welltyped*
for *welltyped* :: ('*v*, '*ty*) *var-types* $\Rightarrow$ '*expr* $\Rightarrow$ '*ty* $\Rightarrow$ *bool*
begin

lemma *renaming-ground-subst*:
assumes
is-renaming ϱ
is-welltyped-on $(\bigcup (vars\ `\ \varrho\ `\ X))\ \mathcal{V}'\ \gamma$
is-welltyped-on $X\ \mathcal{V}\ \varrho$
is-ground-subst γ
$\forall x \in X$. $\mathcal{V}\ x = \mathcal{V}'$ (*rename* $\varrho\ x$)
shows *is-welltyped-on* $X\ \mathcal{V}\ (\varrho \odot \gamma)$
proof(*intro ballI*)
fix x
assume *x-in-X*: $x \in X$

then have *welltyped* $\mathcal{V}$ $(\varrho\ x)$ $(\mathcal{V}\ x)$
by (*simp add*: *assms*(*3*))

define y **where** $y \equiv$ (*rename* $\varrho\ x$)

have $y \in \bigcup (vars\ `\ \varrho\ `\ X)$

```
    using x-in-X
    unfolding y-def
    by (metis UN-iff assms(1) id-subst-rename image-eqI singletonI vars-id-subst)

  moreover then have welltyped 𝒱 (γ y) (𝒱' y)
    using replace-𝒱
   by (metis assms(2,4) left-neutral emptyE is-ground-subst-is-ground comp-subst-iff)

  ultimately have welltyped 𝒱 (γ y) (𝒱 x)
    unfolding y-def
    using assms(5) x-in-X
    by fastforce

  moreover have γ y = (ϱ ⊙ γ) x
    unfolding y-def
    by (metis assms(1) comp-subst-iff id-subst-rename left-neutral)

  ultimately show welltyped 𝒱 ((ϱ ⊙ γ) x) (𝒱 x)
    by argo
qed

lemma obtain-typed-renaming:
  fixes 𝒱 :: 'v ⇒ 'ty
  assumes
    finite X
    infinite-variables-per-type 𝒱
  obtains ϱ :: 'v ⇒ 'expr where
    is-renaming ϱ
    id-subst ` X ∩ ϱ ` Y = {}
    is-welltyped-on Y 𝒱 ϱ
proof−

  obtain renaming :: 'v ⇒ 'v where
    inj: inj renaming and
    rename-apart: X ∩ renaming ` Y = {} and
    preserve-type: ∀ x ∈ Y. 𝒱 (renaming x) = 𝒱 x
    using obtain-type-preserving-inj[OF assms].

  define ϱ :: 'v ⇒ 'expr where
    ⋀x. ϱ x ≡ id-subst (renaming x)

  show ?thesis
  proof (rule that)

    show is-renaming ϱ
      using inj inj-id-subst
      unfolding ϱ-def is-renaming-iff inj-def
      by blast
  next
```

```
    show id-subst ‘ X ∩ ϱ ‘ Y = {}
      using rename-apart inj-id-subst
      unfolding ϱ-def inj-def
      by blast
  next

    show is-welltyped-on Y 𝒱 ϱ
      using preserve-type
      unfolding ϱ-def
      by (metis typed-id-subst)
  qed
qed

lemma obtain-typed-renamings:
  fixes 𝒱1 𝒱2 :: 'v ⇒ 'ty
  assumes
    finite X
    infinite-variables-per-type 𝒱2
  obtains ϱ1 ϱ2 :: 'v ⇒ 'expr where
    is-renaming ϱ1
    is-renaming ϱ2
    ϱ1 ‘ X ∩ ϱ2 ‘ Y = {}
    is-welltyped-on X 𝒱1 ϱ1
    is-welltyped-on Y 𝒱2 ϱ2
  using obtain-typed-renaming[OF assms] is-renaming-id-subst typed-id-subst
  by metis

lemma obtain-typed-renamings':
  fixes 𝒱1 𝒱2 :: 'v ⇒ 'ty
  assumes
    finite Y
    infinite-variables-per-type 𝒱1
  obtains ϱ1 ϱ2 :: 'v ⇒ 'expr where
    is-renaming ϱ1
    is-renaming ϱ2
    ϱ1 ‘ X ∩ ϱ2 ‘ Y = {}
    is-welltyped-on X 𝒱1 ϱ1
    is-welltyped-on Y 𝒱2 ϱ2
  using obtain-typed-renamings[OF assms]
  by (metis inf-commute)

lemma renaming-subst-compose:
  assumes
    is-renaming ϱ
    is-welltyped-on X 𝒱 (ϱ ⊙ σ)
    is-welltyped-on X 𝒱 ϱ
  shows is-welltyped-on (⋃ (vars ‘ ϱ ‘ X)) 𝒱 σ
   using assms
```

unfolding *is-renaming-iff*
by (*smt* (*verit*) *UN-E comp-subst-iff image-iff is-typed-id-subst left-neutral right-uniqueD singletonD vars-id-subst*)

end

lemma (**in** *renaming-variables*) *obtain-merged-*$\mathcal{V}$:
assumes
ϱ_1: *is-renaming* ϱ_1 **and**
ϱ_2: *is-renaming* ϱ_2 **and**
rename-apart: *vars* (*expr* $\cdot$ ϱ_1) $\cap$ *vars* (*expr'* $\cdot$ ϱ_2) = {} **and**
finite-vars: *finite* (*vars expr*) *finite* (*vars expr'*) **and**
infinite-UNIV: *infinite* (*UNIV* :: *'a set*)
obtains $\mathcal{V}_3$ **where**
infinite-variables-per-type-on X $\mathcal{V}_3$
$\forall\, x{\in}vars\ expr.\ \mathcal{V}_1\ x = \mathcal{V}_3\ (rename\ \varrho_1\ x)$
$\forall\, x{\in}vars\ expr'.\ \mathcal{V}_2\ x = \mathcal{V}_3\ (rename\ \varrho_2\ x)$
proof−

have *finite*: *finite* (*vars* (*expr* $\cdot$ ϱ_1)) *finite* (*vars* (*expr'* $\cdot$ ϱ_2))
using *finite-vars*
by (*simp-all add*: ϱ_1 ϱ_2 *rename-variables*)

obtain $\mathcal{V}_3$ **where**
$\mathcal{V}_3$: *infinite-variables-per-type-on* X $\mathcal{V}_3$ **and**
$\mathcal{V}_3$-$\mathcal{V}_1$: $\forall\, x{\in}vars\ (expr \cdot \varrho_1).\ \mathcal{V}_3\ x = \mathcal{V}_1\ (inv\ \varrho_1\ (id\text{-}subst\ x))$ **and**
$\mathcal{V}_3$-$\mathcal{V}_2$: $\forall\, x{\in}vars\ (expr' \cdot \varrho_2).\ \mathcal{V}_3\ x = \mathcal{V}_2\ (inv\ \varrho_2\ (id\text{-}subst\ x))$
using *obtain-infinite-variables-per-type-on*[*OF infinite-UNIV finite rename-apart*]**.**

show *?thesis*
proof (*rule that*[*OF* $\mathcal{V}_3$])

show $\forall\, x{\in}vars\ expr.\ \mathcal{V}_1\ x = \mathcal{V}_3\ (rename\ \varrho_1\ x)$
using $\mathcal{V}_3$-$\mathcal{V}_1$ ϱ_1 *inv-renaming rename-variables*
by *auto*
next

show $\forall\, x{\in}vars\ expr'.\ \mathcal{V}_2\ x = \mathcal{V}_3\ (rename\ \varrho_2\ x)$
using $\mathcal{V}_3$-$\mathcal{V}_2$ ϱ_2 *inv-renaming rename-variables*
by *auto*
qed
qed

lemma (**in** *renaming-variables*) *obtain-merged-*$\mathcal{V}$*-infinite-variables-for-all-types*:
assumes
ϱ_1: *is-renaming* ϱ_1 **and**
ϱ_2: *is-renaming* ϱ_2 **and**
rename-apart: *vars* (*expr* $\cdot$ ϱ_1) $\cap$ *vars* (*expr'* $\cdot$ ϱ_2) = {} **and**
$\mathcal{V}_2$: *infinite-variables-for-all-types* $\mathcal{V}_2$ **and**

```
    finite-vars: finite (vars expr)
  obtains 𝒱₃ where
    ∀x∈vars expr. 𝒱₁ x = 𝒱₃ (rename ϱ₁ x)
    ∀x∈vars expr'. 𝒱₂ x = 𝒱₃ (rename ϱ₂ x)
    infinite-variables-for-all-types 𝒱₃
proof (rule that)

  define 𝒱₃ where
    ⋀x. 𝒱₃ x ≡
        if x ∈ vars (expr · ϱ₁)
        then 𝒱₁ (inv ϱ₁ (id-subst x))
        else 𝒱₂ (inv ϱ₂ (id-subst x))

  show ∀x∈vars expr. 𝒱₁ x = 𝒱₃ (rename ϱ₁ x)
  proof (intro ballI)
    fix x
    assume x∈vars expr

    then have rename ϱ₁ x ∈ vars (expr · ϱ₁)
      using rename-variables[OF ϱ₁]
      by blast

    then show 𝒱₁ x = 𝒱₃ (rename ϱ₁ x)
      unfolding 𝒱₃-def
      by (simp add: ϱ₁ inv-renaming)
  qed

  show ∀x∈vars expr'. 𝒱₂ x = 𝒱₃ (rename ϱ₂ x)
  proof (intro ballI)
    fix x
    assume x∈ vars expr'

    then have rename ϱ₂ x ∈ vars (expr' · ϱ₂)
      using rename-variables[OF ϱ₂]
      by blast

    then show 𝒱₂ x = 𝒱₃ (rename ϱ₂ x)
      unfolding 𝒱₃-def
      using ϱ₂ inv-renaming rename-apart
      by (metis (mono-tags, lifting) disjoint-iff id-subst-rename)
  qed

  have finite {x. x ∈ vars (expr · ϱ₁)}
    using finite-vars
    by (simp add: ϱ₁ rename-variables)

  moreover {
    fix τ
```

have *infinite* $\{x.\ \mathcal{V}_2\ (inv\ \varrho_2\ (id\text{-}subst\ x)) = \tau\}$
proof(*rule surj-infinite-set*[*OF surj-inv-renaming*, *OF* ϱ_2])

show *infinite* $\{x.\ \mathcal{V}_2\ x = \tau\}$
using $\mathcal{V}_2$
unfolding *infinite-variables-for-all-types-def*
by *blast*
qed
}

ultimately show *infinite-variables-for-all-types* $\mathcal{V}_3$
unfolding *infinite-variables-for-all-types-def* $\mathcal{V}_3$*-def if-distrib if-distribR Collect-if-eq*
Collect-not-mem-conj-eq
by *auto*
qed

lemma (**in** *renaming-variables*) *obtain-merged-*$\mathcal{V}'$*-infinite-variables-for-all-types*:
assumes
ϱ_1: *is-renaming* ϱ_1 **and**
ϱ_2: *is-renaming* ϱ_2 **and**
rename-apart: *vars* $(expr \cdot \varrho_1) \cap vars\ (expr' \cdot \varrho_2) = \{\}$ **and**
$\mathcal{V}_1$: *infinite-variables-for-all-types* $\mathcal{V}_1$ **and**
finite-vars: *finite* $(vars\ expr')$
obtains $\mathcal{V}_3$ **where**
$\forall\, x{\in}vars\ expr.\ \mathcal{V}_1\ x = \mathcal{V}_3\ (rename\ \varrho_1\ x)$
$\forall\, x{\in}vars\ expr'.\ \mathcal{V}_2\ x = \mathcal{V}_3\ (rename\ \varrho_2\ x)$
infinite-variables-for-all-types $\mathcal{V}_3$
using *obtain-merged-*$\mathcal{V}$*-infinite-variables-for-all-types*[*OF* ϱ_2 ϱ_1 - $\mathcal{V}_1$ *finite-vars*] *rename-apart*
by (*metis disjoint-iff*)

locale *based-typed-renaming* =
base: *base-typed-renaming* **where**
subst = *base-subst* **and** *vars* = *base-vars* :: *'base* ⇒ *'v set* **and**
welltyped = *base-welltyped* :: (*'v* ⇒ *'ty*) ⇒ *'base* ⇒ *'ty* ⇒ *bool* +
typed-renaming
begin

lemma *renaming-grounding*:
assumes
renaming: *base.is-renaming* ϱ **and**
ϱ-γ*-is-welltyped*: *base.is-welltyped-on* $(vars\ expr)\ \mathcal{V}\ (\varrho \odot \gamma)$ **and**
grounding: *is-ground* $(expr \cdot \varrho \odot \gamma)$ **and**
$\mathcal{V}$-$\mathcal{V}'$: $\forall\, x \in vars\ expr.\ \mathcal{V}\ x = \mathcal{V}'\ (rename\ \varrho\ x)$
shows *base.is-welltyped-on* $(vars\ (expr \cdot \varrho))\ \mathcal{V}'\ \gamma$
proof(*intro ballI*)
fix *x*

define y **where** $y \equiv inv\ \varrho\ (id\text{-}subst\ x)$

assume *x-in-expr*: $x \in vars\ (expr \cdot \varrho)$

then have *y-in-vars*: $y \in vars\ expr$
using *base.renaming-inv-in-vars*[*OF renaming*] *base.vars-id-subst*
unfolding *y-def base.vars-subst-vars vars-subst*
by *fastforce*

then have *base.is-ground* $(base\text{-}subst\ (id\text{-}subst\ y)\ (\varrho \odot \gamma))$
using *variable-grounding*[*OF grounding y-in-vars*]
by (*metis base.comp-subst-iff base.left-neutral*)

moreover have *base-welltyped* $\mathcal{V}\ (base\text{-}subst\ (id\text{-}subst\ y)\ (\varrho \odot \gamma))\ (\mathcal{V}\ y)$
using ϱ-γ-*is-welltyped y-in-vars*
unfolding *y-def*
by (*metis base.comp-subst-iff base.left-neutral*)

ultimately have *base-welltyped* $\mathcal{V}'\ (base\text{-}subst\ (id\text{-}subst\ y)\ (\varrho \odot \gamma))\ (\mathcal{V}\ y)$
by (*meson base.is-ground-typed*)

moreover have $base\text{-}subst\ (id\text{-}subst\ y)\ (\varrho \odot \gamma) = \gamma\ x$
using *x-in-expr base.renaming-inv-into*[*OF renaming*] *base.left-neutral*
unfolding *y-def vars-subst base.comp-subst-iff*
by (*metis* (*no-types*, *lifting*) *UN-E f-inv-into-f*)

ultimately show *base-welltyped* $\mathcal{V}'\ (\gamma\ x)\ (\mathcal{V}'\ x)$
using $\mathcal{V}$-$\mathcal{V}'$[*rule-format*]
by (*metis base.right-uniqueD base.typed-id-subst id-subst-rename renaming renaming-inv-into*
x-in-expr y-def y-in-vars)
qed

lemma *obtain-merged-grounding*:
fixes $\mathcal{V}_1\ \mathcal{V}_2 :: 'v \Rightarrow 'ty$
assumes
base.is-welltyped-on $(vars\ expr)\ \mathcal{V}_1\ \gamma_1$
base.is-welltyped-on $(vars\ expr')\ \mathcal{V}_2\ \gamma_2$
is-ground $(expr \cdot \gamma_1)$
is-ground $(expr' \cdot \gamma_2)$ **and**
$\mathcal{V}_2$: *infinite-variables-per-type* $\mathcal{V}_2$ **and**
finite-vars: *finite* $(vars\ expr)$
obtains $\varrho_1\ \varrho_2\ \gamma$ **where**
base.is-renaming ϱ_1
base.is-renaming ϱ_2
$vars\ (expr \cdot \varrho_1) \cap vars\ (expr' \cdot \varrho_2) = \{\}$
base.is-welltyped-on $(vars\ expr)\ \mathcal{V}_1\ \varrho_1$
base.is-welltyped-on $(vars\ expr')\ \mathcal{V}_2\ \varrho_2$
$\forall x \in vars\ expr.\ \gamma_1\ x = (\varrho_1 \odot \gamma)\ x$

$\forall x \in vars\ expr'.\ \gamma_2\ x = (\varrho_2 \odot \gamma)\ x$
proof −

obtain $\varrho_1\ \varrho_2$ **where**
ϱ_1: *base.is-renaming* ϱ_1 **and**
ϱ_2: *base.is-renaming* ϱ_2 **and**
rename-apart: ϱ_1 ‘ $(vars\ expr) \cap \varrho_2$ ‘ $(vars\ expr') = \{\}$ **and**
ϱ_1*-is-welltyped*: *base.is-welltyped-on* $(vars\ expr)\ \mathcal{V}_1\ \varrho_1$ **and**
ϱ_2*-is-welltyped*: *base.is-welltyped-on* $(vars\ expr')\ \mathcal{V}_2\ \varrho_2$
using *base.obtain-typed-renamings*[*OF finite-vars* $\mathcal{V}_2$]**.**

have *rename-apart*: $vars\ (expr \cdot \varrho_1) \cap vars\ (expr' \cdot \varrho_2) = \{\}$
using *rename-apart rename-variables-id-subst*[*OF* ϱ_1] *rename-variables-id-subst*[*OF* ϱ_2]
by *blast*

from $\varrho_1\ \varrho_2$ **obtain** ϱ_1*-inv* ϱ_2*-inv* **where**
ϱ_1*-inv*: $\varrho_1 \odot \varrho_1$*-inv* $=$ *id-subst* **and**
ϱ_2*-inv*: $\varrho_2 \odot \varrho_2$*-inv* $=$ *id-subst*
unfolding *base.is-renaming-def*
by *blast*

define γ **where**
$\bigwedge x.\ \gamma\ x \equiv$
if $x \in vars\ (expr \cdot \varrho_1)$
then $(\varrho_1$*-inv* $\odot\ \gamma_1)\ x$
else $(\varrho_2$*-inv* $\odot\ \gamma_2)\ x$

show *?thesis*
proof(*rule that*[*OF* $\varrho_1\ \varrho_2$ *rename-apart* ϱ_1*-is-welltyped* ϱ_2*-is-welltyped*])

have $\forall x \in vars\ expr.\ \ \gamma_1\ x = (\varrho_1 \odot \gamma)\ x$
proof(*intro ballI*)
fix x
assume *x-in-vars*: $x \in vars\ expr$

obtain y **where** y: $\varrho_1\ x = id\text{-}subst\ y$
using *obtain-renamed-variable*[*OF* ϱ_1]**.**

then have $y \in vars\ (expr \cdot \varrho_1)$
using *x-in-vars* ϱ_1 *rename-variables-id-subst*
by (*metis base.inj-id-subst image-eqI inj-image-mem-iff*)

then have $\gamma\ y = base\text{-}subst\ (\varrho_1$*-inv* $y)\ \gamma_1$
unfolding γ*-def*
using *base.comp-subst-iff*
by *presburger*

then show $\gamma_1\ x = (\varrho_1 \odot \gamma)\ x$

```
      by (metis ϱ1-inv base.comp-subst-iff base.left-neutral y)
    qed

    then show ∀x ∈ vars expr. γ1 x = (ϱ1 ⊙ γ) x
      by auto

  next

    have ∀x∈ vars expr'. γ2 x = (ϱ2 ⊙ γ) x
    proof(intro ballI)
      fix x
      assume x-in-vars: x ∈ vars expr'

      obtain y where y: ϱ2 x = id-subst y
        using obtain-renamed-variable[OF ϱ2].

      then have y ∈ vars (expr' · ϱ2)
        using x-in-vars ϱ2 rename-variables-id-subst
        by (metis base.inj-id-subst imageI inj-image-mem-iff)

      then have γ y = base-subst (ϱ2-inv y) γ2
        unfolding γ-def
        using base.comp-subst-iff rename-apart
        by auto

      then show γ2 x = (ϱ2 ⊙ γ) x
        by (metis ϱ2-inv base.comp-subst-iff base.left-neutral y)
    qed

    then show ∀x ∈ vars expr'. γ2 x = (ϱ2 ⊙ γ) x
      by auto
  qed
qed

lemma obtain-merged-grounding':
  fixes V1 V2 :: 'v ⇒ 'ty
  assumes
    typed-γ1: base.is-welltyped-on (vars expr) V1 γ1 and
    typed-γ2: base.is-welltyped-on (vars expr') V2 γ2 and
    expr-grounding: is-ground (expr · γ1) and
    expr'-grounding: is-ground (expr' · γ2) and
    V1: infinite-variables-per-type V1 and
    finite-vars: finite (vars expr')
  obtains ϱ1 ϱ2 γ  where
    base.is-renaming ϱ1
    base.is-renaming ϱ2
    vars (expr · ϱ1) ∩ vars (expr' · ϱ2) = {}
    base.is-welltyped-on (vars expr) V1 ϱ1
    base.is-welltyped-on (vars expr') V2 ϱ2
```

$\forall x \in vars\ expr.\ \gamma_1\ x = (\varrho_1 \odot \gamma)\ x$
$\forall x \in vars\ expr'.\ \gamma_2\ x = (\varrho_2 \odot \gamma)\ x$
using *obtain-merged-grounding*[*OF typed-*γ_2 *typed-*γ_1 *expr'-grounding expr-grounding* $\mathcal{V}_1$ *finite-vars*]
by (*smt* (*verit*, *ccfv-threshold*) *inf-commute*)

end

sublocale *base-typed-renaming* $\subseteq$
based-typed-renaming **where** *base-vars* = *vars* **and** *base-subst* = *subst* **and** *base-welltyped* = *welltyped*
by *unfold-locales*

locale *functional-substitution-typing* =
welltyped: *base-typed-functional-substitution* **where** *welltyped* = *welltyped*
for
welltyped :: ('*var*, '*ty*) *var-types* $\Rightarrow$ '*expr* $\Rightarrow$ '*ty* $\Rightarrow$ *bool*
begin

abbreviation *is-welltyped-on* **where**
is-welltyped-on $\equiv$ *welltyped.is-welltyped-on*

abbreviation *is-welltyped* **where**
is-welltyped $\equiv$ *is-welltyped-on UNIV*

lemmas *welltyped-id-subst* = *welltyped.typed-id-subst*

lemmas *is-welltyped-id-subst* = *welltyped.is-typed-id-subst*

lemmas *is-welltyped-on-subset* = *welltyped.is-typed-on-subset*

lemmas *is-welltyped-subst-update* = *welltyped.subst-update*

end

end
theory *Typed-Functional-Substitution-Lifting*
imports
Typed-Functional-Substitution
Abstract-Substitution.Functional-Substitution-Lifting
begin

lemma *ext-equiv*: $(\bigwedge x.\ f\ x \equiv g\ x) \Longrightarrow f \equiv g$
by *presburger*

locale *typed-functional-substitution-lifting* =
sub: *typed-functional-substitution* **where**
vars = *sub-vars* **and** *subst* = *sub-subst* **and** *welltyped* = *sub-welltyped* **and**

base-vars = *base-vars* +
    based-functional-substitution-lifting **where** *to-set* = *to-set* **and** *base-vars* = *base-vars*
for
  sub-welltyped :: ('*v*, '*ty*) *var-types* ⇒ '*sub* ⇒ '*ty*' ⇒ *bool* **and**
  to-set :: '*expr* ⇒ '*sub set* **and**
  base-vars :: '*base* ⇒ '*v set*
begin

sublocale *typing-lifting* **where** *sub-welltyped* = *sub-welltyped* $\mathcal{V}$
  by *unfold-locales*

sublocale *typed-functional-substitution* **where**
  vars = *vars* **and** *subst* = *subst* **and** *welltyped* = *welltyped*
  by *unfold-locales*

end

locale *typed-grounding-functional-substitution-lifting* =
  typed-functional-substitution-lifting +
  grounding-lifting
begin

sublocale *typed-grounding-functional-substitution* **where**
   vars = *vars* **and** *subst* = *subst* **and** *welltyped* = *welltyped* **and** *to-ground* = *to-ground* **and**
  from-ground = *from-ground*
  by *unfold-locales*

end

locale *inhabited-typed-functional-substitution-lifting* =
  typed-functional-substitution-lifting +
  sub: *inhabited-typed-functional-substitution* **where**
  vars = *sub-vars* **and** *subst* = *sub-subst* **and** *welltyped* = *sub-welltyped*
begin

sublocale *inhabited-typed-functional-substitution* **where**
  vars = *vars* **and** *subst* = *subst* **and** *welltyped* = *welltyped*
  by *unfold-locales* (*simp add*: *sub.types-inhabited*)

end

locale *typed-subst-stability-lifting* =
  typed-functional-substitution-lifting +
  sub: *typed-subst-stability* **where**
    welltyped = *sub-welltyped* **and** *vars* = *sub-vars* **and** *subst* = *sub-subst*

begin

sublocale *typed-subst-stability* **where** *welltyped* = *welltyped* **and** *subst* = *subst* **and** *vars* = *vars*
by *unfold-locales* (*auto simp add*: *vars-def to-set-image is-welltyped-def*)

end

locale *replaceable-$\mathcal{V}$-lifting* =
typed-functional-substitution-lifting +
sub: *replaceable-$\mathcal{V}$* **where** *welltyped* = *sub-welltyped* **and** *vars* = *sub-vars* **and** *subst* = *sub-subst*
begin

sublocale *replaceable-$\mathcal{V}$* **where**
subst = *subst* **and** *vars* = *vars* **and** *welltyped* = *welltyped*
by *unfold-locales* (*metis SUP-upper2 sub.replace-$\mathcal{V}$ subset-eq vars-def is-welltyped-def*)

end

locale *typed-renaming-lifting* =
typed-functional-substitution-lifting **where**
base-welltyped = *base-welltyped* :: (*'v* ⇒ *'ty*) ⇒ *'base* ⇒ *'ty* ⇒ *bool* +
renaming-variables-lifting +
sub: *based-typed-renaming* **where**
subst = *sub-subst* **and** *vars* = *sub-vars* **and** *welltyped* = *sub-welltyped*
begin

sublocale *based-typed-renaming* **where**
subst = *subst* **and** *vars* = *vars* **and** *welltyped* = *welltyped*
by *unfold-locales* (*force simp*: *vars-def subst-def is-welltyped-def*)

end

end
theory *Nonground-Term-Typing*
imports
Term-Typing
Typed-Functional-Substitution
Nonground-Term
begin

locale *base-typed-properties* =
base-typed-renaming +
base-typed-subst-stability +
replaceable-$\mathcal{V}$ **where**
base-subst = *subst* **and** *base-vars* = *vars* **and** *base-welltyped* = *welltyped* +
typed-grounding-functional-substitution **where**

base-subst = *subst* **and** *base-vars* = *vars* **and** *base-welltyped* = *welltyped*

locale *base-typing-properties* =
welltyped: *base-typed-properties* **where** *welltyped* = *welltyped*
for *welltyped* :: ('*var*, '*ty*) *var-types* ⇒ '*base* ⇒ '*ty* ⇒ *bool*

locale *base-inhabited-typing-properties* =
base-typing-properties +
welltyped: *inhabited-typed-functional-substitution* **where**
welltyped = *welltyped* **and** *base-subst* = *subst* **and** *base-vars* = *vars* **and** *base-welltyped* = *welltyped*

locale *nonground-term-typing* =
term: *nonground-term* +
fixes $\mathcal{F}$ :: ('*f*, '*ty*) *fun-types*
begin

inductive *welltyped* :: ('*v*, '*ty*) *var-types* ⇒ ('*f*,'*v*) *term* ⇒ '*ty* ⇒ *bool*
for $\mathcal{V}$ **where**
Var: $\mathcal{V}$ *x* = τ ⟹ *welltyped* $\mathcal{V}$ (*Var x*) τ
| *Fun*: $\mathcal{F}$ *f* (*length ts*) = (τ*s*, τ) ⟹ *list-all2* (*welltyped* $\mathcal{V}$) *ts* τ*s* ⟹ *welltyped* $\mathcal{V}$ (*Fun f ts*) τ

sublocale *term*: *base-typing* **where**
welltyped = *welltyped* $\mathcal{V}$ **for** $\mathcal{V}$
proof *unfold-locales*
show *right-unique* (*welltyped* $\mathcal{V}$)
proof (*rule right-uniqueI*)
fix *t* τ_1 τ_2
assume *welltyped* $\mathcal{V}$ *t* τ_1 **and** *welltyped* $\mathcal{V}$ *t* τ_2
thus τ_1 = τ_2
by (*auto elim*!: *welltyped.cases*)
qed
qed

sublocale *term*: *term-typing* **where**
welltyped = *welltyped* $\mathcal{V}$ **and** *Fun* = *Fun* **for** $\mathcal{V}$
proof *unfold-locales*
fix *t t'* *c* τ τ'

assume
t-type: *welltyped* $\mathcal{V}$ *t* τ' **and**
t'-type: *welltyped* $\mathcal{V}$ *t'* τ' **and**
c-type: *welltyped* $\mathcal{V}$ *c*⟨*t*⟩ τ

from *c-type* **show** *welltyped* $\mathcal{V}$ *c*⟨*t'*⟩ τ
proof (*induction c arbitrary*: τ)
case *Hole*
then show *?case*

```
    using t-type t'-type
    by auto
  next
    case (More f ss1 c ss2)

    have welltyped 𝒱 (Fun f (ss1 @ c⟨t⟩ # ss2)) τ
      using More.prems
      by simp

    hence welltyped 𝒱 (Fun f (ss1 @ c⟨t'⟩ # ss2)) τ
    proof (cases 𝒱 Fun f (ss1 @ c⟨t⟩ # ss2) τ rule: welltyped.cases)
      case (Fun τs)

      show ?thesis
      proof (rule welltyped.Fun)
        show ℱ f (length (ss1 @ c⟨t'⟩ # ss2)) = (τs, τ)
          using Fun
          by simp
      next
        show list-all2 (welltyped 𝒱) (ss1 @ c⟨t'⟩ # ss2) τs
          using ‹list-all2 (welltyped 𝒱) (ss1 @ c⟨t⟩ # ss2) τs›
          using More.IH
         by (smt (verit, del-insts) list-all2-Cons1 list-all2-append1 list-all2-lengthD)
      qed
    qed

    thus ?case
      by simp
  qed
next
  fix f ts τ
  assume welltyped 𝒱 (Fun f ts) τ
  then show ∀ t∈set ts. ∃ τ'. welltyped 𝒱 t τ'
    by (cases rule: welltyped.cases) (metis in-set-conv-nth list-all2-conv-all-nth)
qed

sublocale term: base-typing-properties where
  id-subst = Var :: 'v ⇒ ('f, 'v) term and comp-subst = (⊙) and subst = (·t) and
  vars = term.vars and welltyped = welltyped and to-ground = term.to-ground
and
  from-ground = term.from-ground
proof(unfold-locales; (intro welltyped.Var refl)?)
  fix τ and 𝒱 and t :: ('f, 'v) term and σ

  assume is-welltyped-on: ∀ x ∈ term.vars t. welltyped 𝒱 (σ x) (𝒱 x)

  show welltyped 𝒱 (t ·t σ) τ ⟷ welltyped 𝒱 t τ
  proof (rule iffI)
```

```
    assume welltyped 𝒱 t τ

    then show welltyped 𝒱 (t ·t σ) τ
      using is-welltyped-on
      by (induction rule: welltyped.induct)
        (auto simp: list.rel-mono-strong list-all2-map1 welltyped.simps)
  next

    assume welltyped 𝒱 (t ·t σ) τ

    then show welltyped 𝒱 t τ
      using is-welltyped-on
    proof (induction t ·t σ τ arbitrary: t rule: welltyped.induct)
      case (Var x τ)

      then obtain x' where t: t = Var x'
        by (metis subst-apply-eq-Var)

      have welltyped 𝒱 t (𝒱 x')
        unfolding t
        by (simp add: welltyped.Var)

      moreover have welltyped 𝒱 t (𝒱 x)
        using Var
        unfolding t
        by (simp add: welltyped.simps)

      ultimately have 𝒱-x': τ = 𝒱 x'
        using Var.hyps
        by blast

      show ?case
        unfolding t 𝒱-x'
        by (simp add: welltyped.Var)
    next
      case (Fun f τs τ ts)

      then show ?case
      by (cases t) (simp-all add: list.rel-mono-strong list-all2-map1 welltyped.simps)
    qed
  qed
next
  fix t :: ('f, 'v) term and 𝒱 𝒱' τ

  assume welltyped 𝒱 t τ ∀x∈term.vars t. 𝒱 x = 𝒱' x

  then show welltyped 𝒱' t τ
   by (induction rule: welltyped.induct) (simp-all add: welltyped.simps list.rel-mono-strong)
next
```

fix $\mathcal{V}$ $\mathcal{V}'$ :: $('v, 'ty)$ *var-types* **and** t :: $('f, 'v)$ *term* **and** ϱ :: $('f, 'v)$ *subst* **and** τ

assume
renaming: *term-subst.is-renaming* ϱ **and**
$\mathcal{V}$: $\forall x{\in}$*term.vars* t. $\mathcal{V}\ x = \mathcal{V}'$ (*term.rename* $\varrho\ x$)

then show *welltyped* $\mathcal{V}'$ ($t \cdot t\ \varrho$) $\tau \longleftrightarrow$ *welltyped* $\mathcal{V}\ t\ \tau$
proof(*intro iffI*)

assume *welltyped* $\mathcal{V}'$ ($t \cdot t\ \varrho$) τ

with $\mathcal{V}$ **show** *welltyped* $\mathcal{V}\ t\ \tau$
proof(*induction t arbitrary*: τ)
case (*Var x*)

then have $\mathcal{V}'$ (*term.rename* $\varrho\ x$) = τ
using *renaming term.id-subst-rename*[*OF renaming*]
by (*metis eval-term.simps*(*1*) *nonground-term-typing.Var*
term.right-uniqueD)

then have $\mathcal{V}\ x = \tau$
by (*simp add*: *Var.prems*(*1*))

then show *?case*
by(*rule welltyped.Var*)
next
case (*Fun f ts*)

then have *welltyped* $\mathcal{V}'$ (*Fun f* (*map* ($\lambda s.\ s \cdot t\ \varrho$) *ts*)) τ
by *auto*

then obtain τs **where** τs:
list-all2 (*welltyped* $\mathcal{V}'$) (*map* ($\lambda s.\ s \cdot t\ \varrho$) *ts*) τs
$\mathcal{F}\ f$ (*length* (*map* ($\lambda s.\ s \cdot t\ \varrho$) *ts*)) = ($\tau s$, τ)
using *welltyped.simps*
by *blast*

then have $\mathcal{F}$: $\mathcal{F}\ f$ (*length ts*) = (τs, τ)
by *simp*

show *?case*
proof(*rule welltyped.Fun*[*OF* $\mathcal{F}$])

show *list-all2* (*welltyped* $\mathcal{V}$) *ts* τs
using τs(*1*) *Fun.IH*
by (*smt* (*verit*, *ccfv-SIG*) *Fun.prems*(*1*) *eval-term.simps*(*2*) *in-set-conv-nth*
length-map
list-all2-conv-all-nth nth-map term.set-intros(*4*))
qed

```
    qed
  next
    assume welltyped 𝒱 t τ
    then show welltyped 𝒱' (t ·t ϱ) τ
      using 𝒱
    proof(induction rule: welltyped.induct)
      case (Var x τ)

      then have 𝒱' (term.rename ϱ x) = τ
        by simp

      then show ?case
        using term.id-subst-rename[OF renaming]
        by (metis eval-term.simps(1) welltyped.Var)
    next
      case (Fun f ts τs τ)

      have list-all2 (welltyped 𝒱') (map (λs. s ·t ϱ) ts) τs
        using Fun
        by (auto simp: list.rel-mono-strong list-all2-map1)

      then show ?case
        by (simp add: Fun.hyps welltyped.simps)
    qed
  qed
qed

sublocale functional-substitution-typing where
  id-subst = Var :: 'v ⇒ ('f, 'v) term and comp-subst = (⊙) and subst = (·t) and
  vars = term.vars and welltyped = welltyped
  by unfold-locales

end

locale nonground-term-inhabited-typing =
  nonground-term-typing where 𝓕 = 𝓕 for 𝓕 :: ('f, 'ty) fun-types +
  assumes types-inhabited: ⋀τ. ∃f. 𝓕 f 0 = ([], τ)
begin

sublocale base-inhabited-typing-properties where
  id-subst = Var :: 'v ⇒ ('f, 'v) term and comp-subst = (⊙) and subst = (·t) and
  vars = term.vars and welltyped = welltyped and to-ground = term.to-ground
and
  from-ground = term.from-ground
proof unfold-locales
  fix 𝒱 :: ('v, 'ty) var-types and τ

  obtain f where f: 𝓕 f 0 = ([], τ)
    using types-inhabited
```

by *blast*

show $\exists\, t.$ *term.is-ground* $t \wedge$ *welltyped* $\mathcal{V}$ t τ
proof(*rule exI*[*of* - *Fun f* []], *intro conjI welltyped.Fun*)

show *term.is-ground* (*Fun f* [])
by *simp*
next

show $\mathcal{F}$ *f* (*length* []) = ([], τ)
using *f*
by *simp*
next

show *list-all2* (*welltyped* $\mathcal{V}$) [] []
by *simp*
qed
qed

end

end
theory *Nonground-Typing*
imports
Clause-Typing
Typed-Functional-Substitution-Lifting
Nonground-Term-Typing
Nonground-Clause
begin

type-synonym ('*f*, '*v*, '*ty*) *typed-clause* = ('*f*, '*v*) *atom clause* × ('*v*, '*ty*) *var-types*

locale *nonground-typed-lifting* =
typed-subst-stability-lifting +
replaceable-$\mathcal{V}$*-lifting* +
typed-renaming-lifting +
typed-grounding-functional-substitution-lifting

locale *nonground-typing-lifting* =
is-welltyped: *nonground-typed-lifting* **where**
sub-welltyped = *sub-welltyped* **and** *base-welltyped* = *base-welltyped*
for *base-welltyped sub-welltyped*
begin

abbreviation *is-welltyped-ground-instance* ≡ *is-welltyped.is-welltyped-ground-instance*

abbreviation *welltyped-ground-instances* ≡ *is-welltyped.welltyped-ground-instances*

lemmas *welltyped-ground-instances-def* = *is-welltyped.welltyped-ground-instances-def*

end

locale *nonground-inhabited-typing-lifting* =
nonground-typing-lifting +
welltyped: *inhabited-typed-functional-substitution-lifting* **where**
sub-welltyped = *sub-welltyped* **and** *base-welltyped* = *base-welltyped*

locale *term-based-nonground-typing-lifting* =
term: *nonground-term* +
nonground-typing-lifting **where**
id-subst = *Var* **and** *comp-subst* = (⊙) **and** *base-subst* = (·*t*) **and** *base-vars* = *term.vars*

locale *term-based-nonground-inhabited-typing-lifting* =
term: *nonground-term* +
nonground-inhabited-typing-lifting **where**
id-subst = *Var* **and** *comp-subst* = (⊙) **and** *base-subst* = (·*t*) **and** *base-vars* = *term.vars*

locale *nonground-typing* =
nonground-clause +
nonground-term-typing $\mathcal{F}$
for $\mathcal{F}$ :: (*'f*, *'ty*) *fun-types*
begin

sublocale *clause-typing welltyped* $\mathcal{V}$
by *unfold-locales*

sublocale *atom*: *term-based-nonground-typing-lifting* **where**
base-welltyped = *welltyped* **and** *map* = *map-uprod* **and** *to-set* = *set-uprod* **and**
sub-welltyped = *welltyped* **and** *sub-to-ground* = *term.to-ground* **and**
sub-from-ground = *term.from-ground* **and** *to-ground-map* = *map-uprod* **and**
from-ground-map = *map-uprod* **and** *ground-map* = *map-uprod* **and** *to-set-ground* = *set-uprod* **and**
sub-subst = (·*t*) **and** *sub-vars* = *term.vars*
by *unfold-locales*

sublocale *literal*: *term-based-nonground-typing-lifting* **where**
base-welltyped = *welltyped* **and** *sub-vars* = *atom.vars* **and** *sub-subst* = (·*a*) **and**
map = *map-literal* **and** *to-set* = *set-literal* **and** *sub-welltyped* = *atom.welltyped* **and**
sub-to-ground = *atom.to-ground* **and** *sub-from-ground* = *atom.from-ground* **and**
to-ground-map = *map-literal* **and** *from-ground-map* = *map-literal* **and** *ground-map* = *map-literal* **and**
to-set-ground = *set-literal*
by *unfold-locales*

sublocale *clause*: *term-based-nonground-typing-lifting* **where**

base-welltyped = *welltyped* **and** *sub-vars* = *literal.vars* **and** *sub-subst* = $(\cdot l)$ **and**
map = *image-mset* **and** *to-set* = *set-mset* **and** *sub-welltyped* = *literal.welltyped*
and
sub-to-ground = *literal.to-ground* **and** *sub-from-ground* = *literal.from-ground* **and**
to-ground-map = *image-mset* **and** *from-ground-map* = *image-mset* **and** *ground-map*
= *image-mset* **and**
to-set-ground = *set-mset*
by *unfold-locales*

end

locale *nonground-inhabited-typing* =
nonground-typing $\mathcal{F}$ +
nonground-term-inhabited-typing $\mathcal{F}$
for $\mathcal{F}$ :: $('f, 'ty)$ *fun-types*
begin

sublocale *atom*: *term-based-nonground-inhabited-typing-lifting* **where**
base-welltyped = *welltyped* **and** *map* = *map-uprod* **and** *to-set* = *set-uprod* **and**
sub-welltyped = *welltyped* **and** *sub-to-ground* = *term.to-ground* **and**
sub-from-ground = *term.from-ground* **and** *to-ground-map* = *map-uprod* **and**
from-ground-map = *map-uprod* **and** *ground-map* = *map-uprod* **and** *to-set-ground*
= *set-uprod* **and**
sub-subst = $(\cdot t)$ **and** *sub-vars* = *term.vars*
by *unfold-locales*

sublocale *literal*: *term-based-nonground-inhabited-typing-lifting* **where**
base-welltyped = *welltyped* **and** *sub-vars* = *atom.vars* **and**
sub-subst = $(\cdot a)$ **and** *map* = *map-literal* **and** *to-set* = *set-literal* **and**
sub-welltyped = *atom.welltyped* **and** *sub-to-ground* = *atom.to-ground* **and**
sub-from-ground = *atom.from-ground* **and** *to-ground-map* = *map-literal* **and**
from-ground-map = *map-literal* **and** *ground-map* = *map-literal* **and** *to-set-ground*
= *set-literal*
by *unfold-locales*

sublocale *clause*: *term-based-nonground-inhabited-typing-lifting* **where**
base-welltyped = *welltyped* **and**
sub-vars = *literal.vars* **and** *sub-subst* = $(\cdot l)$ **and** *map* = *image-mset* **and** *to-set*
= *set-mset* **and**
sub-welltyped = *literal.welltyped* **and**
sub-to-ground = *literal.to-ground* **and** *sub-from-ground* = *literal.from-ground* **and**
to-ground-map = *image-mset* **and** *from-ground-map* = *image-mset* **and** *ground-map*
= *image-mset* **and**
to-set-ground = *set-mset*
by *unfold-locales*

end

end

theory *HOL-Extra*
imports *Main*
begin

lemmas *UniqI* = *Uniq-I*

lemma *Uniq-prodI*:
assumes $\bigwedge x1\ y1\ x2\ y2.\ P\ x1\ y1 \Longrightarrow P\ x2\ y2 \Longrightarrow (x1,\ y1) = (x2,\ y2)$
shows $\exists_{\leq 1}(x,\ y).\ P\ x\ y$
using *assms*
by (*metis UniqI case-prodE*)

lemma *Uniq-implies-ex1*: $\exists_{\leq 1} x.\ P\ x \Longrightarrow P\ y \Longrightarrow \exists! x.\ P\ x$
by (*iprover intro*: *ex1I dest*: *Uniq-D*)

lemma *Uniq-antimono*: $Q \leq P \Longrightarrow Uniq\ Q \geq Uniq\ P$
unfolding *le-fun-def le-bool-def*
by (*rule impI*) (*simp only*: *Uniq-I Uniq-D*)

lemma *Uniq-antimono'*: $(\bigwedge x.\ Q\ x \Longrightarrow P\ x) \Longrightarrow Uniq\ P \Longrightarrow Uniq\ Q$
by (*fact Uniq-antimono*[*unfolded le-fun-def le-bool-def*, *rule-format*])

lemma *Collect-eq-if-Uniq*: $(\exists_{\leq 1} x.\ P\ x) \Longrightarrow \{x.\ P\ x\} = \{\} \vee (\exists x.\ \{x.\ P\ x\} = \{x\})$
using *Uniq-D* **by** *fastforce*

lemma *Collect-eq-if-Uniq-prod*:
$(\exists_{\leq 1}(x,\ y).\ P\ x\ y) \Longrightarrow \{(x,\ y).\ P\ x\ y\} = \{\} \vee (\exists x\ y.\ \{(x,\ y).\ P\ x\ y\} = \{(x,\ y)\})$
using *Collect-eq-if-Uniq* **by** *fastforce*

lemma *Ball-Ex-comm*:
$(\forall x \in X.\ \exists f.\ P\ (f\ x)\ x) \Longrightarrow (\exists f.\ \forall x \in X.\ P\ (f\ x)\ x)$
$(\exists f.\ \forall x \in X.\ P\ (f\ x)\ x) \Longrightarrow (\forall x \in X.\ \exists f.\ P\ (f\ x)\ x)$
by *meson+*

lemma *set-map-id*:
assumes $x \in set\ X\ f\ x \notin set\ X\ map\ f\ X = X$
shows *False*
using *assms*
by(*induction X*) *auto*

lemma *Ball-singleton*: $(\forall x \in \{x\}.\ P\ x) \longleftrightarrow P\ x$
by *simp*

end
theory *Grounded-Selection-Function*
imports
Nonground-Selection-Function
Nonground-Typing
HOL-Extra

begin

context *nonground-typing*
begin

abbreviation *select-subst-stability-on-clause* **where**
select-subst-stability-on-clause select select$_G$ C_G C $\mathcal{V}$ γ ≡
$C \cdot \gamma$ = *clause.from-ground* C_G ∧
select$_G$ C_G = *clause.to-ground* ((*select* C) · γ) ∧
clause.is-welltyped-ground-instance C $\mathcal{V}$ γ

abbreviation *select-subst-stability-on* **where**
select-subst-stability-on select select$_G$ N ≡
∀ C_G ∈ ⋃ (*clause.welltyped-ground-instances* ‘ N). ∃ (C, $\mathcal{V}$) ∈ N. ∃ γ.
select-subst-stability-on-clause select select$_G$ C_G C $\mathcal{V}$ γ

lemma *obtain-subst-stable-on-select-grounding*:
fixes *select* :: ('*f*, '*v*) *select*
obtains *select$_G$* **where**
select-subst-stability-on select select$_G$ N
is-select-grounding select select$_G$
proof−
let *?N_G* = ⋃(*clause.welltyped-ground-instances* ‘ N)

{
fix C $\mathcal{V}$ γ
assume
(C, $\mathcal{V}$) ∈ N
clause.is-welltyped-ground-instance C $\mathcal{V}$ γ

then have
∃ γ'. ∃ (C', $\mathcal{V}'$)∈N. ∃ *select$_G$*.
select-subst-stability-on-clause select select$_G$ (*clause.to-ground* ($C \cdot \gamma$)) C' $\mathcal{V}'$ γ'
by(*intro exI*[*of* - γ], *intro bexI*[*of* - (C, $\mathcal{V}$)]) *auto*
}

then have
∀ C_G ∈ *?N_G*. ∃ γ. ∃ (C, $\mathcal{V}$) ∈ N. ∃ *select$_G$*.
select-subst-stability-on-clause select select$_G$ C_G C $\mathcal{V}$ γ
unfolding *clause.welltyped-ground-instances-def*
by *force*

then have *select$_G$-exists-for-premises*:
∀ C_G ∈ *?N_G*. ∃ *select$_G$* γ. ∃ (C, $\mathcal{V}$) ∈ N.
select-subst-stability-on-clause select select$_G$ C_G C $\mathcal{V}$ γ
by *blast*

obtain *select$_G$-on-groundings* **where**

```
    select_G-on-groundings: select-subst-stability-on select select_G-on-groundings N
    using Ball-Ex-comm(1)[OF select_G-exists-for-premises]
    unfolding prod.case-eq-if
    by fast

  define select_G where
    ⋀C_G. select_G C_G = (
        if C_G ∈ ?N_G
        then select_G-on-groundings C_G
        else clause.to-ground (select (clause.from-ground C_G))
    )

  have grounding: is-select-grounding select select_G
    using select_G-on-groundings
    unfolding is-select-grounding-def select_G-def prod.case-eq-if
   by (metis (no-types, lifting) clause.from-ground-inverse clause.ground-is-ground
        clause.subst-id-subst)

  show ?thesis
    using that[OF - grounding] select_G-on-groundings
    unfolding select_G-def
    by fastforce
qed

end

locale grounded-selection-function =
  nonground-selection-function select +
  nonground-typing F
  for
    select :: ('f, 'v :: infinite) atom clause ⇒ ('f, 'v) atom clause and
    F :: ('f, 'ty) fun-types +
fixes select_G
assumes select_G: is-select-grounding select select_G
begin

abbreviation subst-stability-on where
  subst-stability-on N ≡ select-subst-stability-on select select_G N

lemma select_G-subset: select_G C ⊆# C
  using select_G
  unfolding is-select-grounding-def
 by (metis select-subset clause.to-ground-def image-mset-subseteq-mono clause.subst-def)

lemma select_G-negative-literals:
  assumes l_G ∈# select_G C_G
  shows is-neg l_G
proof −
  obtain C γ where
```

is-ground: *clause.is-ground* $(C \cdot \gamma)$ **and**
select$_G$: *select*$_G$ C_G = *clause.to-ground* (*select* $C \cdot \gamma$)
using *select*$_G$
unfolding *is-select-grounding-def*
by *blast*

show *?thesis*
using
ground-literal-in-selection[
OF select-ground-subst[*OF is-ground*] *assms*[*unfolded select*$_G$],
THEN select-neg-subst
]
by *simp*

qed

sublocale *ground*: *selection-function select*$_G$
by *unfold-locales* (*simp-all add*: *select*$_G$*-subset select*$_G$*-negative-literals*)

end

end
theory *Term-Rewrite-System*
imports *Ground-Context*
begin

definition *compatible-with-gctxt* :: *'f gterm rel* ⇒ *bool* **where**
compatible-with-gctxt I ⟷ (∀ *t t' ctxt*. (*t*, *t'*) ∈ *I* ⟶ (*ctxt*⟨*t*⟩$_G$, *ctxt*⟨*t'*⟩$_G$) ∈ *I*)

lemma *compatible-with-gctxtD*:
compatible-with-gctxt I ⟹ (*t*, *t'*) ∈ *I* ⟹ (*ctxt*⟨*t*⟩$_G$, *ctxt*⟨*t'*⟩$_G$) ∈ *I*
by (*simp add*: *compatible-with-gctxt-def*)

lemma *compatible-with-gctxt-converse*:
assumes *compatible-with-gctxt I*
shows *compatible-with-gctxt* (I^{-1})
unfolding *compatible-with-gctxt-def*
proof (*intro allI impI*)
fix *t t' ctxt*
assume (*t*, *t'*) ∈ I^{-1}
thus (*ctxt*⟨*t*⟩$_G$, *ctxt*⟨*t'*⟩$_G$) ∈ I^{-1}
by (*simp add*: *assms compatible-with-gctxtD*)
qed

lemma *compatible-with-gctxt-symcl*:
assumes *compatible-with-gctxt I*
shows *compatible-with-gctxt* ($I^{\leftrightarrow}$)
unfolding *compatible-with-gctxt-def*
proof (*intro allI impI*)

```
  fix t t' ctxt
  assume (t, t') ∈ I^↔
  thus (ctxt⟨t⟩_G, ctxt⟨t'⟩_G) ∈ I^↔
  proof (induction ctxt arbitrary: t t')
    case Hole
    thus ?case by simp
  next
    case (More f ts1 ctxt ts2)
    thus ?case
      using assms[unfolded compatible-with-gctxt-def, rule-format]
      by blast
  qed
qed

lemma compatible-with-gctxt-rtrancl:
  assumes compatible-with-gctxt I
  shows compatible-with-gctxt (I^*)
  unfolding compatible-with-gctxt-def
proof (intro allI impI)
  fix t t' ctxt
  assume (t, t') ∈ I^*
  thus (ctxt⟨t⟩_G, ctxt⟨t'⟩_G) ∈ I^*
  proof (induction t' rule: rtrancl-induct)
    case base
    show ?case
      by simp
  next
    case (step y z)
    thus ?case
      using assms[unfolded compatible-with-gctxt-def, rule-format]
      by (meson rtrancl.rtrancl-into-rtrancl)
  qed
qed

lemma compatible-with-gctxt-relcomp:
  assumes compatible-with-gctxt I1 and compatible-with-gctxt I2
  shows compatible-with-gctxt (I1 O I2)
  unfolding compatible-with-gctxt-def
proof (intro allI impI)
  fix t t'' ctxt
  assume (t, t'') ∈ I1 O I2
  then obtain t' where (t, t') ∈ I1 and (t', t'') ∈ I2
    by auto

  have (ctxt⟨t⟩_G, ctxt⟨t'⟩_G) ∈ I1
    using ‹(t, t') ∈ I1› assms(1) compatible-with-gctxtD by blast
  moreover have (ctxt⟨t'⟩_G, ctxt⟨t''⟩_G) ∈ I2
    using ‹(t', t'') ∈ I2› assms(2) compatible-with-gctxtD by blast
  ultimately show (ctxt⟨t⟩_G, ctxt⟨t''⟩_G) ∈ I1 O I2
```

by *auto*
qed

lemma *compatible-with-gctxt-join*:
assumes *compatible-with-gctxt I*
shows *compatible-with-gctxt* $(I^{\downarrow})$
using *assms*
by (*simp-all add*: *join-def compatible-with-gctxt-relcomp compatible-with-gctxt-rtrancl compatible-with-gctxt-converse*)

lemma *compatible-with-gctxt-conversion*:
assumes *compatible-with-gctxt I*
shows *compatible-with-gctxt* $(I^{\leftrightarrow *})$
by (*simp add*: *assms compatible-with-gctxt-rtrancl compatible-with-gctxt-symcl conversion-def*)

definition *rewrite-inside-gctxt* :: *'f gterm rel* ⇒ *'f gterm rel* **where**
rewrite-inside-gctxt R = $\{(ctxt\langle t1\rangle_G,\ ctxt\langle t2\rangle_G) \mid ctxt\ t1\ t2.\ (t1,\ t2) \in R\}$

lemma *mem-rewrite-inside-gctxt-if-mem-rewrite-rules*[*intro*]:
$(l,\ r) \in R \Longrightarrow (l,\ r) \in$ *rewrite-inside-gctxt R*
by (*metis* (*mono-tags*, *lifting*) *intp-actxt.simps*(*1*) *mem-Collect-eq rewrite-inside-gctxt-def*)

lemma *ctxt-mem-rewrite-inside-gctxt-if-mem-rewrite-rules*[*intro*]:
$(l,\ r) \in R \Longrightarrow (ctxt\langle l\rangle_G,\ ctxt\langle r\rangle_G) \in$ *rewrite-inside-gctxt R*
by (*auto simp*: *rewrite-inside-gctxt-def*)

lemma *rewrite-inside-gctxt-mono*: $R \subseteq S \Longrightarrow$ *rewrite-inside-gctxt* $R \subseteq$ *rewrite-inside-gctxt S*
by (*auto simp add*: *rewrite-inside-gctxt-def*)

lemma *rewrite-inside-gctxt-union*:
rewrite-inside-gctxt $(R \cup S)$ = *rewrite-inside-gctxt R* ∪ *rewrite-inside-gctxt S*
by (*auto simp add*: *rewrite-inside-gctxt-def*)

lemma *rewrite-inside-gctxt-insert*:
rewrite-inside-gctxt (*insert r R*) = *rewrite-inside-gctxt* $\{r\}$ ∪ *rewrite-inside-gctxt R*
using *rewrite-inside-gctxt-union*[*of* $\{r\}$ *R*, *simplified*] .

lemma *converse-rewrite-steps*: $(rewrite\text{-}inside\text{-}gctxt\ R)^{-1} = rewrite\text{-}inside\text{-}gctxt\ (R^{-1})$
by (*auto simp*: *rewrite-inside-gctxt-def*)

lemma *rhs-lt-lhs-if-rule-in-rewrite-inside-gctxt*:
fixes *less-trm* :: *'f gterm* ⇒ *'f gterm* ⇒ *bool* (**infix** $\prec_t$ *50*)
assumes
rule-in: $(t1,\ t2) \in$ *rewrite-inside-gctxt R* **and**
ball-R-rhs-lt-lhs: $\bigwedge t1\ t2.\ (t1,\ t2) \in R \Longrightarrow t2 \prec_t t1$ **and**
compatible-with-gctxt: $\bigwedge t1\ t2\ ctxt.\ t2 \prec_t t1 \Longrightarrow ctxt\langle t2\rangle_G \prec_t ctxt\langle t1\rangle_G$

shows $t2 \prec_t t1$
proof −
from *rule-in* **obtain** $t1'$ $t2'$ *ctxt* **where**
$(t1', t2') \in R$ **and**
$t1 = ctxt\langle t1' \rangle_G$ **and**
$t2 = ctxt\langle t2' \rangle_G$
by (*auto simp*: *rewrite-inside-gctxt-def*)

from *ball-R-rhs-lt-lhs* **have** $t2' \prec_t t1'$
using ‹$(t1', t2') \in R$› **by** *simp*

with *compatible-with-gctxt* **have** $ctxt\langle t2' \rangle_G \prec_t ctxt\langle t1' \rangle_G$
by *metis*

thus *?thesis*
using ‹$t1 = ctxt\langle t1' \rangle_G$› ‹$t2 = ctxt\langle t2' \rangle_G$› **by** *metis*
qed

lemma *mem-rewrite-step-union-NF*:
assumes $(t, t') \in$ *rewrite-inside-gctxt* $(R1 \cup R2)$
$t \in NF$ (*rewrite-inside-gctxt* $R2$)
shows $(t, t') \in$ *rewrite-inside-gctxt* $R1$
using *assms*
unfolding *rewrite-inside-gctxt-union*
by *blast*

lemma *predicate-holds-of-mem-rewrite-inside-gctxt*:
assumes *rule-in*: $(t1, t2) \in$ *rewrite-inside-gctxt* R **and**
ball-P: $\bigwedge t1\ t2.\ (t1, t2) \in R \Longrightarrow P\ t1\ t2$ **and**
preservation: $\bigwedge t1\ t2\ ctxt\ \sigma.\ (t1, t2) \in R \Longrightarrow P\ t1\ t2 \Longrightarrow P\ ctxt\langle t1 \rangle_G\ ctxt\langle t2 \rangle_G$
shows $P\ t1\ t2$
proof −
from *rule-in* **obtain** $t1'$ $t2'$ *ctxt* σ **where**
$(t1', t2') \in R$ **and**
$t1 = ctxt\langle t1' \rangle_G$ **and**
$t2 = ctxt\langle t2' \rangle_G$
by (*auto simp*: *rewrite-inside-gctxt-def*)
thus *?thesis*
using *ball-P*[*OF* ‹$(t1', t2') \in R$›]
using *preservation*[*OF* ‹$(t1', t2') \in R$›, *of ctxt*]
by *simp*
qed

lemma *compatible-with-gctxt-rewrite-inside-gctxt*[*simp*]: *compatible-with-gctxt* (*rewrite-inside-gctxt* E)
unfolding *compatible-with-gctxt-def rewrite-inside-gctxt-def*
unfolding *mem-Collect-eq*
by (*metis Pair-inject intp-actxt-compose*)

lemma *subset-rewrite-inside-gctxt*[*simp*]: $E \subseteq$ *rewrite-inside-gctxt* E
proof (*rule Set.subsetI*)
  fix e **assume** *e-in*: $e \in E$
  moreover obtain s t **where** *e-def*: $e = (s, t)$
    by *fastforce*
  show $e \in$ *rewrite-inside-gctxt* E
    unfolding *rewrite-inside-gctxt-def*
    unfolding *mem-Collect-eq*
  proof (*intro exI conjI*)
    show $e = (\Box\langle s\rangle_G, \Box\langle t\rangle_G)$
      unfolding *e-def*
      by *simp*
  next
    show $(s, t) \in E$
      using *e-in*
      unfolding *e-def* **.**
  qed
qed

lemma *wf-converse-rewrite-inside-gctxt*:
  fixes $E :: 'f$ *gterm rel*
  assumes
    wfP-R: *wfP* R **and**
    R-compatible-with-gctxt: $\bigwedge ctxt\ t\ t'.\ R\ t\ t' \Longrightarrow R\ ctxt\langle t\rangle_G\ ctxt\langle t'\rangle_G$ **and**
    equations-subset-R: $\bigwedge x\ y.\ (x, y) \in E \Longrightarrow R\ y\ x$
  shows *wf* $((\textit{rewrite-inside-gctxt}\ E)^{-1})$
proof (*rule wf-subset*)
  from *wfP-R* **show** *wf* $\{(x, y).\ R\ x\ y\}$
    by (*simp add*: *wfp-def*)
next
  show $(\textit{rewrite-inside-gctxt}\ E)^{-1} \subseteq \{(x, y).\ R\ x\ y\}$
  proof (*rule Set.subsetI*)
    fix e **assume** $e \in (\textit{rewrite-inside-gctxt}\ E)^{-1}$
    then obtain *ctxt* s t **where** *e-def*: $e = (ctxt\langle s\rangle_G, ctxt\langle t\rangle_G)$ **and** $(t, s) \in E$
     by (*smt* (*verit*) *Pair-inject converseE mem-Collect-eq rewrite-inside-gctxt-def*)
    hence $R\ s\ t$
      using *equations-subset-R* **by** *simp*
    hence $R\ ctxt\langle s\rangle_G\ ctxt\langle t\rangle_G$
      using *R-compatible-with-gctxt* **by** *simp*
    then show $e \in \{(x, y).\ R\ x\ y\}$
      by (*simp add*: *e-def*)
  qed
qed

end
theory *Entailment-Lifting*
  imports *Abstract-Substitution.Functional-Substitution-Lifting*
begin

locale *entailment* =
  based: *based-functional-substitution* **where** *base-subst* = *base-subst* **and** *vars* = *vars* +
  base: *grounding* **where** *subst* = *base-subst* **and** *vars* = *base-vars* **and** *to-ground* = *base-to-ground* **and**
  from-ground = *base-from-ground* **for**
  vars :: $'expr \Rightarrow 'var\ set$ **and**
  base-subst :: $'base \Rightarrow ('var \Rightarrow 'base) \Rightarrow 'base$ **and**
  base-to-ground :: $'base \Rightarrow 'base_G$ **and**
  base-from-ground +
fixes *entails-def* :: $'expr \Rightarrow bool$ **and** $I :: ('base_G \times 'base_G)\ set$
assumes
  congruence: $\bigwedge expr\ \gamma\ var\ update.$
      based.base.is-ground update $\Longrightarrow$
      based.base.is-ground $(\gamma\ var) \Longrightarrow$
      $(base\text{-}to\text{-}ground\ (\gamma\ var), base\text{-}to\text{-}ground\ update) \in I \Longrightarrow$
      based.is-ground $(subst\ expr\ \gamma) \Longrightarrow$
      entails-def $(subst\ expr\ (\gamma(var := update))) \Longrightarrow$
      entails-def $(subst\ expr\ \gamma)$
begin

abbreviation *entails* $\equiv$ *entails-def*

end

locale *symmetric-entailment* = *entailment* +
  assumes *sym*: *sym I*
begin

lemma *symmetric-congruence*:
  assumes
    update-is-ground: *based.base.is-ground update* **and**
    var-grounding: *based.base.is-ground* $(\gamma\ var)$ **and**
    var-update: $(base\text{-}to\text{-}ground\ (\gamma\ var), base\text{-}to\text{-}ground\ update) \in I$ **and**
    expr-grounding: *based.is-ground* $(subst\ expr\ \gamma)$
  shows
    entails $(subst\ expr\ (\gamma(var := update))) \longleftrightarrow$ *entails* $(subst\ expr\ \gamma)$
  using *congruence*[*OF var-grounding*, *of* $\gamma(var := update)$] *assms*
 by (*metis based.ground-subst-update congruence fun-upd-same fun-upd-triv fun-upd-upd sym symD*)

end

locale *symmetric-base-entailment* =
  base-functional-substitution **where** *subst* = *subst* +
  grounding **where** *subst* = *subst* **and** *to-ground* = *to-ground* **for**
  subst :: $'base \Rightarrow ('var \Rightarrow 'base) \Rightarrow 'base$ (**infixl** $\cdot$ *70*) **and**
  to-ground :: $'base \Rightarrow 'base_G$ +
fixes $I :: ('base_G \times 'base_G)\ set$

assumes
sym: *sym I* **and**
congruence: $\bigwedge$*expr expr′ update γ var*.
is-ground update $\Longrightarrow$
is-ground (*γ var*) $\Longrightarrow$
(*to-ground* (*γ var*), *to-ground update*) $\in I \Longrightarrow$
is-ground (*expr* · *γ*) $\Longrightarrow$
(*to-ground* (*expr* · (*γ*(*var* := *update*))), *expr′*) $\in I \Longrightarrow$
(*to-ground* (*expr* · *γ*), *expr′*) $\in I$
begin

lemma *symmetric-congruence*:
assumes
update-is-ground: *is-ground update* **and**
var-grounding: *is-ground* (*γ var*) **and**
expr-grounding: *is-ground* (*expr* · *γ*) **and**
var-update: (*to-ground* (*γ var*), *to-ground update*) $\in I$
shows (*to-ground* (*expr* · (*γ*(*var* := *update*))), *expr′*) $\in I \longleftrightarrow$ (*to-ground* (*expr* · *γ*), *expr′*) $\in I$
using *assms congruence*[*OF var-grounding*, *of γ*(*var* := *update*) *var*] *congruence*
by (*metis fun-upd-same fun-upd-triv fun-upd-upd ground-subst-update sym symD*)

lemma *simultaneous-congruence*:
assumes
update-is-ground: *is-ground update* **and**
var-grounding: *is-ground* (*γ var*) **and**
var-update: (*to-ground* (*γ var*), *to-ground update*) $\in I$ **and**
expr-grounding: *is-ground* (*expr* · *γ*) *is-ground* (*expr′* · *γ*)
shows
(*to-ground* (*expr* · (*γ*(*var* := *update*))), *to-ground* (*expr′* · (*γ*(*var* := *update*)))) $\in I \longleftrightarrow$
(*to-ground* (*expr* · *γ*), *to-ground* (*expr′* · *γ*)) $\in I$
using *assms*
by (*meson sym symD symmetric-congruence*)

end

locale *entailment-lifting* =
based-functional-substitution-lifting +
finite-variables-lifting +
sub: *symmetric-entailment*
where *subst* = *sub-subst* **and** *vars* = *sub-vars* **and** *entails-def* = *sub-entails*
for *sub-entails* +
fixes
is-negated :: *'d* ⇒ *bool* **and**
empty :: *bool* **and**
connective :: *bool* ⇒ *bool* ⇒ *bool* **and**
entails-def
assumes

is-negated-subst: $\bigwedge$*expr* σ. *is-negated* (*subst expr* σ) $\longleftrightarrow$ *is-negated expr* **and**
entails-def: $\bigwedge$*expr*. *entails-def expr* $\longleftrightarrow$
(*if is-negated expr then Not else* (λx. *x*))
(*Finite-Set.fold connective empty* (*sub-entails* ‘ *to-set expr*))
begin

notation *sub-entails* (($\models_s$ -) [*50*] *50*)
notation *entails-def* (($\models$ -) [*50*] *50*)

sublocale *symmetric-entailment* **where** *subst* = *subst* **and** *vars* = *vars* **and** *entails-def* = *entails-def*
proof *unfold-locales*
fix *expr* γ *var update P*
assume
base.is-ground update
base.is-ground (γ *var*)
is-ground (*expr* $\cdot$ γ)
(*base-to-ground* (γ *var*), *base-to-ground update*) $\in$ *I*
$\models$ *expr* $\cdot$ γ(*var* := *update*)

moreover then have $\forall$ *sub* $\in$ *to-set expr*. ($\models_s$ *sub* $\cdot_s$ γ(*var* := *update*)) $\longleftrightarrow$ $\models_s$ *sub* $\cdot_s$ γ
using *sub.symmetric-congruence*[*of update* γ] *to-set-is-ground-subst*
by *blast*

ultimately show $\models$ *expr* $\cdot$ γ
unfolding *is-negated-subst entails-def*
by(*auto simp*: *image-image subst-def*)

qed (*simp-all add*: *is-grounding-iff-vars-grounded sub.sym*)

end

locale *entailment-lifting-conj* = *entailment-lifting*
where *connective* = ($\wedge$) **and** *empty* = *True*

locale *entailment-lifting-disj* = *entailment-lifting*
where *connective* = ($\vee$) **and** *empty* = *False*

end
theory *Fold-Extra*
imports *Main*
begin

lemma *comp-fun-idem-conj*: *comp-fun-idem-on X* ($\wedge$)
by *unfold-locales fastforce+*

lemma *comp-fun-idem-disj*: *comp-fun-idem-on X* ($\vee$)
by *unfold-locales fastforce+*

lemma *fold-conj-insert* [*simp*]:
Finite-Set.fold (∧) *True* (*insert b B*) ⟷ *b* ∧ *Finite-Set.fold* (∧) *True B*
using *comp-fun-idem-on.fold-insert-idem*[*OF comp-fun-idem-conj*]
by (*metis finite top-greatest*)

lemma *fold-disj-insert* [*simp*]:
Finite-Set.fold (∨) *False* (*insert b B*) ⟷ *b* ∨ *Finite-Set.fold* (∨) *False B*
using *comp-fun-idem-on.fold-insert-idem*[*OF comp-fun-idem-disj*]
by (*metis finite top-greatest*)

end
theory *Nonground-Entailment*
imports
Nonground-Context
Nonground-Clause
Term-Rewrite-System
Entailment-Lifting
Fold-Extra
begin

4 Entailment

context *nonground-term*
begin

lemma *var-in-term*:
assumes *x* ∈ *vars t*
obtains *c* **where** *t* = *c*⟨*Var x*⟩
using *assms*
proof(*induction t*)
case *Var*
then show *?case*
by (*meson supteq-Var supteq-ctxtE*)
next
case (*Fun f args*)
then obtain *t′* **where** *t′* ∈ *set args* *x* ∈ *vars t′*
by (*metis term.distinct*(*1*) *term.sel*(*4*) *term.set-cases*(*2*))

moreover then obtain *args1 args2* **where**
args1 @ [*t′*] @ *args2* = *args*
by (*metis append-Cons append-Nil split-list*)

moreover then have (*More f args1* □ *args2*)⟨*t′*⟩ = *Fun f args*
by *simp*

ultimately show *?case*
using *Fun*(*1*)
by (*meson assms supteq-ctxtE that vars-term-supteq*)

qed

lemma *vars-term-ms-count*:
assumes *is-ground t*
shows
size {#x' ∈# *vars-term-ms* $c\langle Var\ x\rangle$. $x' = x$#} = *Suc* (*size* {#x' ∈# *vars-term-ms* $c\langle t\rangle$. $x' = x$#})
by(*induction c*)(*auto simp*: *assms filter-mset-empty-conv*)

end

context *nonground-clause*
begin

lemma *not-literal-entails* [*simp*]:
¬ *upair* ‘ *I* ⊨*l Neg a* ⟷ *upair* ‘ *I* ⊨*l Pos a*
¬ *upair* ‘ *I* ⊨*l Pos a* ⟷ *upair* ‘ *I* ⊨*l Neg a*
by *auto*

lemmas *literal-entails-unfolds* =
not-literal-entails true-lit-simps

end

locale *clause-entailment* = *nonground-clause* +
fixes *I* :: (′*f gterm* × ′*f gterm*) *set*
assumes
trans: *trans I* **and**
sym: *sym I* **and**
compatible-with-gctxt: *compatible-with-gctxt I*
begin

lemma *symmetric-context-congruence*:
assumes (*t*, *t*′) ∈ *I*
shows ($c\langle t\rangle_G$, *t*′′) ∈ *I* ⟷ ($c\langle t'\rangle_G$, *t*′′) ∈ *I*
by (*meson assms compatible-with-gctxt compatible-with-gctxtD sym trans symD transE*)

lemma *symmetric-upair-context-congruence*:
assumes *Upair t t*′ ∈ *upair* ‘ *I*
shows *Upair* $c\langle t\rangle_G$ *t*′′ ∈ *upair* ‘ *I* ⟷ *Upair* $c\langle t'\rangle_G$ *t*′′ ∈ *upair* ‘ *I*
using *assms uprod-mem-image-iff-prod-mem*[*OF sym*] *symmetric-context-congruence*
by *simp*

lemma *upair-compatible-with-gctxtI* [*intro*]:
Upair t t′ ∈ *upair* ‘ *I* ⟹ *Upair* $c\langle t\rangle_G$ $c\langle t'\rangle_G$ ∈ *upair* ‘ *I*
using *compatible-with-gctxt*
unfolding *compatible-with-gctxt-def*
by (*simp add*: *sym*)

sublocale *term*: *symmetric-base-entailment* **where** *vars* = *term.vars* :: ('*f*, '*v*) *term* ⇒ '*v set* **and**
  id-subst = *Var* **and** *comp-subst* = (⊙) **and** *subst* = (·*t*) **and** *to-ground* = *term.to-ground* **and**
  from-ground = *term.from-ground*
proof *unfold-locales*
  fix γ :: ('*f*, '*v*) *subst* **and** *t t'* *update var*

  assume
    update-is-ground: *term.is-ground update* **and**
    var-grounding: *term.is-ground* (γ *var*) **and**
    var-update: (*term.to-ground* (γ *var*), *term.to-ground update*) ∈ *I* **and**
    term-grounding: *term.is-ground* (*t* ·*t* γ) **and**
    updated-term: (*term.to-ground* (*t* ·*t* γ(*var* := *update*)), *t'*) ∈ *I*

  from *term-grounding updated-term*
  show (*term.to-ground* (*t* ·*t* γ), *t'*) ∈ *I*
  proof(*induction size* (*filter-mset* (λ*var'*. *var'* = *var*) (*vars-term-ms t*)) *arbitrary*: *t*)
    case *0*

    then have *var* ∉ *term.vars t*
      by (*metis* (*mono-tags*, *lifting*) *filter-mset-empty-conv set-mset-vars-term-ms size-eq-0-iff-empty*)

    then have *t* ·*t* γ(*var* := *update*) = *t* ·*t* γ
      using *term.subst-reduntant-upd*
      by (*simp add*: *eval-with-fresh-var*)

    with *0* **show** *?case*
      by *argo*
  next
    case (*Suc n*)

    let *?context-to-ground* = *map-args-actxt term.to-ground*

    have *var* ∈ *term.vars t*
      using *Suc.hyps*(*2*)
    by (*metis* (*full-types*) *filter-mset-empty-conv nonempty-has-size set-mset-vars-term-ms zero-less-Suc*)

    then obtain *c* **where** *t* [*simp*]: *t* = *c*⟨*Var var*⟩
      by (*meson term.var-in-term*)

    have [*simp*]:
      (*?context-to-ground* (*c* ·*t*$_c$ γ))⟨*term.to-ground* (γ *var*)⟩$_G$ = *term.to-ground* (*c*⟨*Var var*⟩ ·*t* γ)
      using *Suc*

by(*induction c*) *simp-all*

have *context-update* [*simp*]:
(*?context-to-ground* (*c* ·t_c γ))⟨*term.to-ground update*⟩$_G$ = *term.to-ground* (*c*⟨*update*⟩ ·*t* γ)
using *Suc update-is-ground*
by(*induction c*) *auto*

have *n* = *size* {#*var'* ∈# *vars-term-ms c*⟨*update*⟩. *var'* = *var*#}
using *Suc term.vars-term-ms-count*[*OF update-is-ground*, *of var c*]
by *auto*

moreover have *term.is-ground* (*c*⟨*update*⟩ ·*t* γ)
using *Suc.prems update-is-ground*
by *auto*

moreover have (*term.to-ground* (*c*⟨*update*⟩ ·*t* γ(*var* := *update*)), *t'*) ∈ *I*
using *Suc.prems update-is-ground*
by *auto*

moreover have (*term.to-ground update*, *term.to-ground* (γ *var*)) ∈ *I*
using *var-update sym*
by (*metis symD*)

moreover have (*term.to-ground* (*c*⟨*update*⟩ ·*t* γ), *t'*) ∈ *I*
using *Suc calculation*
by *blast*

ultimately have ((*?context-to-ground* (*c* ·t_c γ))⟨*term.to-ground* (γ *var*)⟩$_G$, *t'*) ∈ *I*
using *symmetric-context-congruence context-update*
by *metis*

then show *?case*
by *simp*
qed
qed (*rule sym*)

sublocale *atom*: *symmetric-entailment*
where *comp-subst* = (⊙) **and** *id-subst* = *Var*
and *base-subst* = (·*t*) **and** *base-vars* = *term.vars* **and** *subst* = (·*a*) **and** *vars* = *atom.vars*
and *base-to-ground* = *term.to-ground* **and** *base-from-ground* = *term.from-ground* **and** *I* = *I*
and *entails-def* = λ*a*. *atom.to-ground a* ∈ *upair* ‘ *I*
proof *unfold-locales*
fix *a* :: ('*f*, '*v*) *atom* **and** γ *var update P*

assume *assms*:

```
    term.is-ground update
    term.is-ground (γ var)
    (term.to-ground (γ var), term.to-ground update) ∈ I
    atom.is-ground (a ·a γ)
    (atom.to-ground (a ·a γ(var := update)) ∈ upair ‘ I)

  show (atom.to-ground (a ·a γ) ∈ upair ‘ I)
  proof(cases a)
    case (Upair t t′)

    moreover have
      (term.to-ground (t′ ·t γ), term.to-ground (t ·t γ)) ∈ I ⟷
       (term.to-ground (t ·t γ), term.to-ground (t′ ·t γ)) ∈ I
      by (metis local.sym symD)

    ultimately show ?thesis
      using assms
      unfolding atom.to-ground-def atom.subst-def atom.vars-def
      by(auto simp: sym term.simultaneous-congruence)
  qed
qed (simp-all add: sym)

sublocale literal: entailment-lifting-conj
  where comp-subst = (⊙) and id-subst = Var
    and base-subst = (·t) and base-vars = term.vars and sub-subst = (·a) and
sub-vars = atom.vars
   and base-to-ground = term.to-ground and base-from-ground = term.from-ground
and I = I
    and sub-entails = atom.entails and map = map-literal and to-set = set-literal
    and is-negated = is-neg and entails-def = λl. upair ‘ I ⊨l literal.to-ground l
proof unfold-locales
  fix l :: ('f, 'v) atom literal

  show (upair ‘ I ⊨l literal.to-ground l) =
    (if is-neg l then Not else (λx. x))
      (Finite-Set.fold (∧) True ((λa. atom.to-ground a ∈ upair ‘ I) ‘ set-literal l))
    unfolding literal.vars-def literal.to-ground-def
    by(cases l)(auto)

qed auto

sublocale clause: entailment-lifting-disj
  where comp-subst = (⊙) and id-subst = Var
    and base-subst = (·t) and base-vars = term.vars
   and base-to-ground = term.to-ground and base-from-ground = term.from-ground
and I = I
   and sub-subst = (·l) and sub-vars = literal.vars and sub-entails = literal.entails
    and map = image-mset and to-set = set-mset and is-negated = λ-. False
    and entails-def = λC. upair ‘ I ⊨ clause.to-ground C
```

proof *unfold-locales*
fix $C :: ('f, 'v)$ *atom clause*

show *upair* ‘ $I \Vdash$ *clause.to-ground* $C \longleftrightarrow$
(*if False then Not else* $(\lambda x.\ x)$) (*Finite-Set.fold* $(\vee)$ *False* (*literal.entails* ‘ *set-mset* C))
unfolding *clause.to-ground-def*
by(*induction* C) *auto*

qed *auto*

lemma *literal-compatible-with-gctxtI* [*intro*]:
literal.entails $(t \approx t') \Longrightarrow$ *literal.entails* $(c\langle t\rangle \approx c\langle t'\rangle)$
by (*simp add*: *upair-compatible-with-gctxtI*)

lemma *symmetric-literal-context-congruence*:
assumes *Upair* $t\ t' \in$ *upair* ‘ I
shows
upair ‘ $I \Vdash l\ c\langle t\rangle_G \approx t'' \longleftrightarrow$ *upair* ‘ $I \Vdash l\ c\langle t'\rangle_G \approx t''$
upair ‘ $I \Vdash l\ c\langle t\rangle_G\ !\approx t'' \longleftrightarrow$ *upair* ‘ $I \Vdash l\ c\langle t'\rangle_G\ !\approx t''$
using *assms symmetric-upair-context-congruence*
by *auto*

end

end
theory *Nonground-Inference*
imports *Nonground-Clause Nonground-Typing*
begin

locale *nonground-inference* = *nonground-clause*
begin

sublocale *inference*: *term-based-lifting* **where**
sub-subst = *clause.subst* **and** *sub-vars* = *clause.vars* **and** *map* = *map-inference*
and
to-set = *set-inference* **and** *sub-to-ground* = *clause.to-ground* **and**
sub-from-ground = *clause.from-ground* **and** *to-ground-map* = *map-inference* **and**
from-ground-map = *map-inference* **and** *ground-map* = *map-inference* **and** *to-set-ground* = *set-inference*
by *unfold-locales*

notation *inference.subst* (**infixl** $\cdot\iota$ *67*)

lemma *vars-inference* [*simp*]:
inference.vars (*Infer Ps C*) = $\bigcup$ (*clause.vars* ‘ *set Ps*) $\cup$ *clause.vars C*
unfolding *inference.vars-def*
by *auto*

lemma *subst-inference* [*simp*]:
Infer Ps C $\cdot\iota$ σ = *Infer* (*map* (λ*P*. *P* $\cdot$ σ) *Ps*) (*C* $\cdot$ σ)
unfolding *inference.subst-def*
by *simp-all*

lemma *inference-from-ground-clause-from-ground* [*simp*]:
inference.from-ground (*Infer Ps C*) = *Infer* (*map clause.from-ground Ps*) (*clause.from-ground C*)
by (*simp add*: *inference.from-ground-def*)

lemma *inference-to-ground-clause-to-ground* [*simp*]:
inference.to-ground (*Infer Ps C*) = *Infer* (*map clause.to-ground Ps*) (*clause.to-ground C*)
by (*simp add*: *inference.to-ground-def*)

lemma *inference-is-ground-clause-is-ground* [*simp*]:
inference.is-ground (*Infer Ps C*) $\longleftrightarrow$ *list-all clause.is-ground Ps* $\wedge$ *clause.is-ground C*
by (*auto simp*: *Ball-set*)

end

end
theory *Restricted-Order*
imports *Main*
begin

5 Restricted Orders

locale *relation-restriction* =
fixes *R* :: $'a \Rightarrow 'a \Rightarrow$ *bool* **and** *lift* :: $'b \Rightarrow 'a$
assumes *inj-lift* [*intro*]: *inj lift*
begin

definition R_r :: $'b \Rightarrow 'b \Rightarrow$ *bool* **where**
R_r *b b'* $\equiv$ *R* (*lift b*) (*lift b'*)

end

5.1 Strict Orders

locale *strict-order* =
fixes
less :: $'a \Rightarrow 'a \Rightarrow$ *bool* (**infix** $\prec$ *50*)
assumes
transp [*intro*]: *transp* ($\prec$) **and**
asymp [*intro*]: *asymp* ($\prec$)
begin

abbreviation *less-eq* **where** *less-eq* $\equiv (\prec)^{==}$

notation *less-eq* (**infix** $\preceq$ *50*)

sublocale *order* $(\preceq)$ $(\prec)$
  by (*rule order-reflclp-if-transp-and-asymp*[*OF transp asymp*])

end

locale *strict-order-restriction* =
  strict-order +
  relation-restriction **where** $R = (\prec)$
begin

abbreviation $less_r \equiv R_r$

lemmas $less_r$*-def* = R_r*-def*

notation $less_r$ (**infix** $\prec_r$ *50*)

sublocale *restriction*: *strict-order* $(\prec_r)$
  by *unfold-locales* (*auto simp*: R_r*-def transp-def*)

abbreviation $less\text{-}eq_r \equiv$ *restriction.less-eq*
notation $less\text{-}eq_r$ (**infix** $\preceq_r$ *50*)

end

5.2 Wellfounded Strict Orders

locale *restricted-wellfounded-strict-order* = *strict-order* +
  fixes *restriction*
  assumes *wfp* [*intro*]: *wfp-on restriction* $(\prec)$

locale *wellfounded-strict-order* =
  restricted-wellfounded-strict-order **where** *restriction* = *UNIV*

locale *wellfounded-strict-order-restriction* =
  strict-order-restriction +
  restricted-wellfounded-strict-order **where** *restriction* = *range lift* **and** *less* = $(\prec)$
begin

sublocale *wellfounded-strict-order* $(\prec_r)$
proof *unfold-locales*
  show *wfp* $(\prec_r)$
    using *wfp-on-if-convertible-to-wfp-on*[*OF wfp*]
    unfolding R_r*-def*
    by *simp*
qed

end

5.3 Total Strict Orders

locale *restricted-total-strict-order* = *strict-order* +
  fixes *restriction*
  assumes *totalp* [*intro*]: *totalp-on restriction* ($\prec$)
begin

lemma *restricted-not-le*:
  assumes $a \in$ *restriction* $b \in$ *restriction* $\neg\ b \prec a$
  shows $a \preceq b$
  using *assms*
  by (*metis less-le local.order-refl totalp totalp-on-def*)

end

locale *total-strict-order* =
  restricted-total-strict-order **where** *restriction* = *UNIV*
begin

sublocale *linorder* ($\preceq$) ($\prec$)
  using *totalpD*
  by *unfold-locales fastforce*

end

locale *total-strict-order-restriction* =
  strict-order-restriction +
  restricted-total-strict-order **where** *restriction* = *range lift* **and** *less* = ($\prec$)
begin

sublocale *total-strict-order* ($\prec_r$)
proof *unfold-locales*
  show *totalp* ($\prec_r$)
    using *totalp inj-lift*
    unfolding R_r*-def totalp-on-def inj-def*
    by *blast*
qed

end

locale *restricted-wellfounded-total-strict-order* =
  restricted-wellfounded-strict-order + *restricted-total-strict-order*

end
theory *Context-Compatible-Order*
  imports

Ground-Context
Restricted-Order
begin

locale *restriction-restricted* =
fixes *restriction context-restriction restricted restricted-context*
assumes
restricted:
$\bigwedge t.\ t \in restriction \longleftrightarrow restricted\ t$
$\bigwedge c.\ c \in context\text{-}restriction \longleftrightarrow restricted\text{-}context\ c$

locale *restricted-context-compatibility* =
restriction-restricted +
fixes *R Fun*
assumes
context-compatible [*simp*]:
$\bigwedge c\ t_1\ t_2.$
restricted $t_1 \Longrightarrow$
restricted $t_2 \Longrightarrow$
restricted-context $c \Longrightarrow$
$R\ (Fun\langle c;t_1\rangle)\ (Fun\langle c;t_2\rangle) \longleftrightarrow R\ t_1\ t_2$

locale *context-compatibility* = *restricted-context-compatibility* **where**
restriction = *UNIV* **and** *context-restriction* = *UNIV* **and** *restricted* = λ-. *True*
and
restricted-context = λ-. *True*
begin

lemma *context-compatibility* [*simp*]: $R\ (Fun\langle c;t_1\rangle)\ (Fun\langle c;t_2\rangle) \longleftrightarrow R\ t_1\ t_2$
by *simp*

end

locale *context-compatible-restricted-order* =
restricted-total-strict-order +
restriction-restricted +
fixes *Fun*
assumes *less-context-compatible*:
$\bigwedge c\ t_1\ t_2.$
restricted $t_1 \Longrightarrow$
restricted $t_2 \Longrightarrow$
restricted-context $c \Longrightarrow$
$t_1 \prec t_2 \Longrightarrow$
$Fun\langle c;t_1\rangle \prec Fun\langle c;t_2\rangle$
begin

sublocale *restricted-context-compatibility* **where** $R = (\prec)$
using *less-context-compatible restricted*
by *unfold-locales* (*metis dual-order.asym totalp totalp-onD*)

sublocale *less-eq*: *restricted-context-compatibility* **where** $R = (\preceq)$
using *context-compatible restricted-not-le dual-order.order-iff-strict restricted*
by *unfold-locales metis*

lemma *context-less-term-lesseq*:
assumes
$restricted\ t$
$restricted\ t'$
$restricted\text{-}context\ c$
$restricted\text{-}context\ c'$
$\bigwedge t.\ restricted\ t \Longrightarrow Fun\langle c;t\rangle \prec Fun\langle c';t\rangle$
$t \preceq t'$
shows $Fun\langle c;t\rangle \prec Fun\langle c';t'\rangle$
using *assms context-compatible dual-order.strict-trans*
by *blast*

lemma *context-lesseq-term-less*:
assumes
$restricted\ t$
$restricted\ t'$
$restricted\text{-}context\ c$
$restricted\text{-}context\ c'$
$\bigwedge t.\ restricted\ t \Longrightarrow Fun\langle c;t\rangle \preceq Fun\langle c';t\rangle$
$t \prec t'$
shows $Fun\langle c;t\rangle \prec Fun\langle c';t'\rangle$
using *assms context-compatible dual-order.strict-trans1*
by *meson*

end

locale *context-compatible-order* =
total-strict-order +
fixes *Fun*
assumes *less-context-compatible*: $t_1 \prec t_2 \Longrightarrow Fun\langle c;t_1\rangle \prec Fun\langle c;t_2\rangle$
begin

sublocale *restricted*: *context-compatible-restricted-order* **where**
$restriction = UNIV$ **and** $context\text{-}restriction = UNIV$ **and** $restricted = \lambda\text{-}.\ True$
and
$restricted\text{-}context = \lambda\text{-}.\ True$
using *less-context-compatible*
by *unfold-locales simp-all*

sublocale *context-compatibility* $(\prec)$
by *unfold-locales*

sublocale *less-eq*: *context-compatibility* $(\preceq)$
by *unfold-locales*

lemma *context-less-term-lesseq*:
assumes
$\bigwedge t.\ Fun\langle c;t\rangle \prec Fun\langle c';t\rangle$
$t \preceq t'$
shows $Fun\langle c;t\rangle \prec Fun\langle c';t'\rangle$
using *assms restricted.context-less-term-lesseq*
by *blast*

lemma *context-lesseq-term-less*:
assumes
$\bigwedge t.\ Fun\langle c;t\rangle \preceq Fun\langle c';t\rangle$
$t \prec t'$
shows $Fun\langle c;t\rangle \prec Fun\langle c';t'\rangle$
using *assms restricted.context-lesseq-term-less*
by *blast*

end

end
theory *Term-Order-Notation*
imports *Main*
begin

locale *term-order-notation* =
fixes $less_t :: 't \Rightarrow 't \Rightarrow bool$
begin

notation $less_t$ (**infix** $\prec_t$ *50*)

abbreviation $less\text{-}eq_t \equiv (\prec_t)^{==}$

notation $less\text{-}eq_t$ (**infix** $\preceq_t$ *50*)

end

end
theory *Transitive-Closure-Extra*
imports *Main*
begin

lemma *reflclp-iff*: $\bigwedge R\ x\ y.\ R^{==}\ x\ y \longleftrightarrow R\ x\ y \vee x = y$
by (*metis* (*full-types*) *sup2CI sup2E*)

lemma *reflclp-refl*: $R^{==}\ x\ x$
by *simp*

lemma *transpD-strict-non-strict*:
assumes *transp R*

shows $\bigwedge x\ y\ z.\ R\ x\ y \Longrightarrow R^{==}\ y\ z \Longrightarrow R\ x\ z$
using ‹*transp R*›[*THEN transpD*] **by** *blast*

lemma *transpD-non-strict-strict*:
assumes *transp R*
shows $\bigwedge x\ y\ z.\ R^{==}\ x\ y \Longrightarrow R\ y\ z \Longrightarrow R\ x\ z$
using ‹*transp R*›[*THEN transpD*] **by** *blast*

lemma *mem-rtrancl-union-iff-mem-rtrancl-lhs*:
assumes $\bigwedge z.\ (x,\ z) \in A^* \Longrightarrow z \notin Domain\ B$
shows $(x,\ y) \in (A \cup B)^* \longleftrightarrow (x,\ y) \in A^*$
using *assms*
by (*meson Domain.DomainI in-rtrancl-UnI rtrancl-Un-separatorE*)

lemma *mem-rtrancl-union-iff-mem-rtrancl-rhs*:
assumes
$\bigwedge z.\ (x,\ z) \in B^* \Longrightarrow z \notin Domain\ A$
shows $(x,\ y) \in (A \cup B)^* \longleftrightarrow (x,\ y) \in B^*$
using *assms*
by (*metis mem-rtrancl-union-iff-mem-rtrancl-lhs sup-commute*)

end
theory *Ground-Term-Order*
imports
Ground-Context
Context-Compatible-Order
Term-Order-Notation
Transitive-Closure-Extra
begin

locale *context-compatible-ground-order* = *context-compatible-order* **where** *Fun* = *GFun*

locale *subterm-property* =
strict-order **where** *less* = $less_t$
for $less_t$ :: *'f gterm* ⇒ *'f gterm* ⇒ *bool* +
assumes
subterm-property [*simp*]: $\bigwedge t\ c.\ c \neq \Box \Longrightarrow less_t\ t\ c\langle t\rangle_G$
begin

interpretation *term-order-notation*.

lemma *less-eq-subterm-property*: $t \preceq_t c\langle t\rangle_G$
using *subterm-property*
by (*metis gctxt-ident-iff-eq-GHole reflclp-iff*)

end

locale *ground-term-order* =

wellfounded-strict-order $less_t$ +
total-strict-order $less_t$ +
context-compatible-ground-order $less_t$ +
subterm-property $less_t$
for $less_t$ :: *'f gterm* ⇒ *'f gterm* ⇒ *bool*
begin

interpretation *term-order-notation*.

end

end
theory *Grounded-Order*
imports
Restricted-Order
Abstract-Substitution.Functional-Substitution-Lifting
begin

6 Orders with ground restrictions

locale *grounded-order* =
strict-order **where** *less* = *less* +
grounding **where** *vars* = *vars*
for
less :: *'expr* ⇒ *'expr* ⇒ *bool* (**infix** ‹≺› *50*) **and**
vars :: *'expr* ⇒ *'var set*
begin

sublocale *strict-order-restriction* **where** *lift* = *from-ground*
by *unfold-locales* (*rule inj-from-ground*)

abbreviation $less_G \equiv less_r$
lemmas $less_G$-*def* = $less_r$-*def*
notation $less_G$ (**infix** $\prec_G$ *50*)

abbreviation *less-eq*$_G \equiv$ *less-eq*$_r$
notation *less-eq*$_G$ (**infix** $\preceq_G$ *50*)

lemma *to-ground-less*$_r$ [*simp*]:
assumes *is-ground e* **and** *is-ground e'*
shows *to-ground e* $\prec_G$ *to-ground e'* ⟷ *e* ≺ *e'*
by (*simp add*: *assms less*$_r$-*def*)

lemma *to-ground-less-eq*$_r$ [*simp*]:
assumes *is-ground e* **and** *is-ground e'*
shows *to-ground e* $\preceq_G$ *to-ground e'* ⟷ *e* ⪯ *e'*
using *assms obtain-grounding*
by *fastforce*

lemma *less-eq$_r$-from-ground* [*simp*]:
e$_G$ $\preceq_G$ *e$_G$'* $\longleftrightarrow$ *from-ground e$_G$* $\preceq$ *from-ground e$_G$'*
unfolding *R$_r$-def*
by (*simp add*: *inj-eq inj-lift*)

end

locale *grounded-restricted-total-strict-order* =
order: *restricted-total-strict-order* **where** *restriction* = *range from-ground* +
grounded-order
begin

sublocale *total-strict-order-restriction* **where** *lift* = *from-ground*
by *unfold-locales*

lemma *not-less-eq* [*simp*]:
assumes *is-ground expr* **and** *is-ground expr'*
shows ¬ *order.less-eq expr' expr* $\longleftrightarrow$ *expr* $\prec$ *expr'*
using *assms order.totalp order.less-le-not-le*
unfolding *totalp-on-def is-ground-iff-range-from-ground*
by *blast*

end

locale *grounded-restricted-wellfounded-strict-order* =
restricted-wellfounded-strict-order **where** *restriction* = *range from-ground* +
grounded-order
begin

sublocale *wellfounded-strict-order-restriction* **where** *lift* = *from-ground*
by *unfold-locales*

end

6.1 Ground substitution stability

locale *ground-subst-stability* = *grounding* +
fixes *R*
assumes
ground-subst-stability:
$\bigwedge$*expr$_1$ expr$_2$* γ.
is-ground (*expr$_1$* $\cdot$ γ) $\Longrightarrow$
is-ground (*expr$_2$* $\cdot$ γ) $\Longrightarrow$
R expr$_1$ expr$_2$ $\Longrightarrow$
R (*expr$_1$* $\cdot$ γ) (*expr$_2$* $\cdot$ γ)

locale *ground-subst-stable-grounded-order* =
grounded-order +

ground-subst-stability **where** $R = (\prec)$
begin

sublocale *less-eq*: *ground-subst-stability* **where** $R = (\preceq)$
using *ground-subst-stability*
by *unfold-locales blast*

lemma *ground-less-not-less-eq*:
assumes
grounding: *is-ground* $(expr_1 \cdot \gamma)$ *is-ground* $(expr_2 \cdot \gamma)$ **and**
less: $expr_1 \cdot \gamma \prec expr_2 \cdot \gamma$
shows
$\neg\ expr_2 \preceq expr_1$
using *less ground-subst-stability*[*OF grounding*(*2*, *1*)] *dual-order.asym*
by *blast*

end

6.2 Substitution update stability

locale *subst-update-stability* =
based-functional-substitution +
fixes *base-R R*
assumes
subst-update-stability:
$\bigwedge$*update x* γ *expr*.
base.is-ground update $\Longrightarrow$
base-R update $(\gamma\ x)$ $\Longrightarrow$
is-ground $(expr \cdot \gamma)$ $\Longrightarrow$
$x \in vars\ expr \Longrightarrow$
$R\ (expr \cdot \gamma(x := update))\ (expr \cdot \gamma)$

locale *base-subst-update-stability* =
base-functional-substitution +
subst-update-stability **where** *base-R* = *R* **and** *base-subst* = *subst* **and** *base-vars* = *vars*

locale *subst-update-stable-grounded-order* =
grounded-order + *subst-update-stability* **where** *R* = *less* **and** *base-R* = *base-less*
for *base-less*
begin

sublocale *less-eq*: *subst-update-stability*
where *base-R* = *base-less*$^{==}$ **and** *R* = *less*$^{==}$
using *subst-update-stability*
by *unfold-locales auto*

end

locale *base-subst-update-stable-grounded-order* =
base-subst-update-stability **where** R = *less* +
subst-update-stable-grounded-order **where**
base-less = *less* **and** *base-subst* = *subst* **and** *base-vars* = *vars*

end
theory *Multiset-Extension*
imports
Restricted-Order
Multiset-Extra
begin

7 Multiset Extensions

locale *multiset-extension* = *order*: *strict-order* +
fixes *to-mset* :: $'b \Rightarrow 'a$ *multiset*
begin

definition *multiset-extension* :: $'b \Rightarrow 'b \Rightarrow$ *bool* **where**
multiset-extension b1 b2 ≡ *multp* (≺) (*to-mset b1*) (*to-mset b2*)

notation *multiset-extension* (**infix** $\prec_m$ *50*)

sublocale *strict-order* ($\prec_m$)
proof *unfold-locales*
show *transp* ($\prec_m$)
using *transp-multp*[*OF order.transp*]
unfolding *multiset-extension-def transp-on-def*
by *blast*
next
show *asymp* ($\prec_m$)
unfolding *multiset-extension-def*
by (*simp add*: *asympD asymp-multp*$_{HO}$ *asymp-onI multp-eq-multp*$_{HO}$)
qed

notation *less-eq* (**infix** $\preceq_m$ *50*)

end

7.1 Wellfounded Multiset Extensions

locale *wellfounded-multiset-extension* =
order: *wellfounded-strict-order* +
multiset-extension
begin

sublocale *wellfounded-strict-order* ($\prec_m$)
proof *unfold-locales*

show *wfp* $(\prec_m)$
unfolding *multiset-extension-def*
using *wfp-if-convertible-to-wfp*[*OF wfp-multp*[*OF order.wfp*]]
by *meson*
qed

end

7.2 Total Multiset Extensions

locale *restricted-total-multiset-extension* =
base: *restricted-total-strict-order* +
multiset-extension +
assumes *inj-on-to-mset*: *inj-on to-mset* {*b*. *set-mset* (*to-mset b*) ⊆ *restriction*}
begin

sublocale *restricted-total-strict-order* $(\prec_m)$ {*b*. *set-mset* (*to-mset b*) ⊆ *restriction*}
proof *unfold-locales*
have *totalp-on* {*b*. *set-mset b* ⊆ *restriction*} (*multp* $(\prec)$)
using *totalp-on-multp*[*OF base.totalp base.transp*]
by *fastforce*

then show *totalp-on* {*b*. *set-mset* (*to-mset b*) ⊆ *restriction*} $(\prec_m)$
using *inj-on-to-mset*
unfolding *multiset-extension-def totalp-on-def inj-on-def*
by *auto*
qed

end

locale *total-multiset-extension* =
order: *total-strict-order* +
multiset-extension +
assumes *inj-to-mset*: *inj to-mset*
begin

sublocale *restricted-total-multiset-extension* **where** *restriction* = *UNIV*
by *unfold-locales* (*simp add*: *inj-to-mset*)

sublocale *total-strict-order* $(\prec_m)$
using *totalp*
by *unfold-locales simp*

end

locale *total-wellfounded-multiset-extension* =
wellfounded-multiset-extension + *total-multiset-extension*

end

theory *Grounded-Multiset-Extension*
imports *Grounded-Order Multiset-Extension*
begin

8 Grounded Multiset Extensions

locale *functional-substitution-multiset-extension* =
sub: *strict-order* **where** *less* = $(\prec)$ :: *'sub* ⇒ *'sub* ⇒ *bool* +
multiset-extension **where** *to-mset* = *to-mset* +
functional-substitution-lifting **where** *id-subst* = *id-subst* **and** *to-set* = *to-set*
for
to-mset :: *'expr* ⇒ *'sub multiset* **and**
id-subst :: *'var* ⇒ *'base* **and**
to-set :: *'expr* ⇒ *'sub set* +
assumes

to-mset-to-set: $\bigwedge$*expr*. *set-mset* (*to-mset expr*) = *to-set expr* **and**
to-mset-map: $\bigwedge$*f b*. *to-mset* (*map f b*) = *image-mset f* (*to-mset b*) **and**
inj-to-mset: *inj to-mset*
begin

no-notation *less-eq* (**infix** $\preceq$ *50*)
notation *sub.less-eq* (**infix** $\preceq$ *50*)

lemma *lesseq-if-all-lesseq*:
assumes $\forall$ *sub* $\in\#$ *to-mset expr*. *sub* $\cdot_s$ σ' $\preceq$ *sub* $\cdot_s$ σ
shows *expr* $\cdot$ σ' $\preceq_m$ *expr* $\cdot$ σ
using *multp-image-lesseq-if-all-lesseq*[*OF sub.asymp sub.transp assms*] *inj-to-mset*
unfolding *multiset-extension-def subst-def inj-def*
by (*auto simp*: *to-mset-map*)

lemma *less-if-all-lesseq-ex-less*:
assumes
$\forall$ *sub*$\in\#$*to-mset expr*. *sub* $\cdot_s$ σ' $\preceq$ *sub* $\cdot_s$ σ
$\exists$ *sub*$\in\#$*to-mset expr*. *sub* $\cdot_s$ σ' $\prec$ *sub* $\cdot_s$ σ
shows
expr $\cdot$ σ' $\prec_m$ *expr* $\cdot$ σ
using *multp-image-less-if-all-lesseq-ex-less*[*OF sub.asymp sub.transp assms*]
unfolding *multiset-extension-def subst-def to-mset-map*.

end

locale *grounded-multiset-extension* =
grounding-lifting **where**
id-subst = *id-subst* :: *'var* ⇒ *'base* **and** *to-set* = *to-set* :: *'expr* ⇒ *'sub set* **and**
to-set-ground = *to-set-ground* +
functional-substitution-multiset-extension **where** *to-mset* = *to-mset*
for

to-mset :: *'expr* ⇒ *'sub multiset* **and**
to-set-ground :: *'expr*$_G$ ⇒ *'sub*$_G$ *set*
begin

sublocale *strict-order-restriction* (≺$_m$) *from-ground*
by *unfold-locales* (*rule inj-from-ground*)

end

locale *total-grounded-multiset-extension* =
grounded-multiset-extension +
sub: *total-strict-order-restriction* **where** *lift* = *sub-from-ground*
begin

sublocale *total-strict-order-restriction* (≺$_m$) *from-ground*
proof *unfold-locales*
have *totalp-on* {*expr*. *set-mset expr* ⊆ *range sub-from-ground*} (*multp* (≺))
using *sub.totalp totalp-on-multp*
by *force*

then have *totalp-on* {*expr*. *set-mset* (*to-mset expr*) ⊆ *range sub-from-ground*}
(≺$_m$)
using *inj-to-mset*
unfolding *inj-def multiset-extension-def totalp-on-def*
by *blast*

then show *totalp-on* (*range from-ground*) (≺$_m$)
unfolding *multiset-extension-def totalp-on-def from-ground-def*
by (*simp add*: *image-mono to-mset-to-set*)
qed

end

locale *based-grounded-multiset-extension* =
based-functional-substitution-lifting **where** *base-vars* = *base-vars* +
grounded-multiset-extension +
base: *strict-order* **where** *less* = *base-less*
for
base-vars :: *'base* ⇒ *'var set* **and**
base-less :: *'base* ⇒ *'base* ⇒ *bool*

8.1 Ground substitution stability

locale *ground-subst-stable-total-multiset-extension* =
grounded-multiset-extension +
sub: *ground-subst-stable-grounded-order* **where**
less = *less* **and** *subst* = *sub-subst* **and** *vars* = *sub-vars* **and** *from-ground* =
sub-from-ground **and**

to-ground = *sub-to-ground*
begin

sublocale *ground-subst-stable-grounded-order* **where**
less = ($\prec_m$) **and** *subst* = *subst* **and** *vars* = *vars* **and** *from-ground* = *from-ground*
and
to-ground = *to-ground*
proof *unfold-locales*

fix $expr_1$ $expr_2$ γ

assume *grounding*: *is-ground* ($expr_1 \cdot \gamma$) *is-ground* ($expr_2 \cdot \gamma$) **and** *less*: $expr_1 \prec_m expr_2$

show $expr_1 \cdot \gamma \prec_m expr_2 \cdot \gamma$
proof(
unfold multiset-extension-def subst-def to-mset-map,
rule multp-map-strong[*OF sub.transp - less*[*unfolded multiset-extension-def*]])

show *monotone-on* (*set-mset* (*to-mset* $expr_1$ + *to-mset* $expr_2$)) ($\prec$) ($\prec$) (λ*sub*. *sub* $\cdot_s$ γ)
using *grounding monotone-onI sub.ground-subst-stability*
by (*metis* (*mono-tags*, *lifting*) *to-mset-to-set to-set-is-ground-subst union-iff*)
qed
qed

end

8.2 Substitution update stability

locale *subst-update-stable-multiset-extension* =
based-grounded-multiset-extension +
sub: *subst-update-stable-grounded-order* **where**
vars = *sub-vars* **and** *subst* = *sub-subst* **and** *to-ground* = *sub-to-ground* **and**
from-ground = *sub-from-ground*
begin

no-notation *less-eq* (**infix** $\preceq$ *50*)

sublocale *subst-update-stable-grounded-order* **where**
less = ($\prec_m$) **and** *vars* = *vars* **and** *subst* = *subst* **and** *from-ground* = *from-ground*
and
to-ground = *to-ground*
proof *unfold-locales*
fix *update* *x* γ *expr*

assume *assms*:
base.is-ground update base-less update (γ *x*) *is-ground* (*expr* $\cdot$ γ) *x* $\in$ *vars expr*

moreover then have $\forall\, sub \in\# \textit{to-mset expr}.\ sub \cdot_s \gamma(x := update) \preceq sub \cdot_s \gamma$
using
sub.subst-update-stability
sub.subst-reduntant-upd
to-mset-to-set
to-set-is-ground-subst
by *blast*

moreover have $\exists\, sub \in\# \textit{to-mset expr}.\ sub \cdot_s \gamma(x := update) \prec (sub \cdot_s \gamma)$
using *sub.subst-update-stability assms*
unfolding *vars-def subst-def to-mset-to-set*
by *fastforce*

ultimately show $expr \cdot \gamma(x := update) \prec_m expr \cdot \gamma$
using *less-if-all-lesseq-ex-less*
by *blast*
qed

end

end
theory *Maximal-Literal*
imports
Clausal-Calculus-Extra
Min-Max-Least-Greatest.Min-Max-Least-Greatest-Multiset
Restricted-Order
begin

locale *maximal-literal* = *order*: *strict-order* **where** *less* = *less*
for *less* :: *'a literal* ⇒ *'a literal* ⇒ *bool*
begin

abbreviation *is-maximal* :: *'a literal* ⇒ *'a clause* ⇒ *bool* **where**
is-maximal l C ≡ *order.is-maximal-in-mset C l*

abbreviation *is-strictly-maximal* :: *'a literal* ⇒ *'a clause* ⇒ *bool* **where**
is-strictly-maximal l C ≡ *order.is-strictly-maximal-in-mset C l*

lemmas *is-maximal-def* = *order.is-maximal-in-mset-iff*

lemmas *is-strictly-maximal-def* = *order.is-strictly-maximal-in-mset-iff*

lemmas *is-maximal-if-is-strictly-maximal* = *order.is-maximal-in-mset-if-is-strictly-maximal-in-mset*

lemma *maximal-in-clause*:
assumes *is-maximal l C*
shows *l* ∈# *C*
using *assms*

unfolding *is-maximal-def*
by(*rule conjunct1*)

lemma *strictly-maximal-in-clause*:
assumes *is-strictly-maximal l C*
shows $l \in\# C$
using *assms*
unfolding *is-strictly-maximal-def*
by(*rule conjunct1*)

lemma *is-maximal-not-empty* [*intro*]: *is-maximal l C* $\Longrightarrow$ $C \neq \{\#\}$
using *maximal-in-clause*
by *fastforce*

lemma *is-strictly-maximal-not-empty* [*intro*]: *is-strictly-maximal l C* $\Longrightarrow$ $C \neq \{\#\}$
using *strictly-maximal-in-clause*
by *fastforce*

end

end
theory *Term-Order-Lifting*
imports
Grounded-Multiset-Extension
Maximal-Literal
Term-Order-Notation
begin

locale *restricted-term-order-lifting* =
term.order: *restricted-wellfounded-total-strict-order* **where** *less* = $less_t$
for $less_t$ :: $'t \Rightarrow 't \Rightarrow bool$ +
fixes *literal-to-mset* :: $'a$ *literal* $\Rightarrow$ $'t$ *multiset*
assumes *inj-literal-to-mset*: *inj literal-to-mset*
begin

sublocale *term-order-notation*.

abbreviation *literal-order-restriction* **where**
literal-order-restriction $\equiv \{b.$ *set-mset* (*literal-to-mset b*) $\subseteq$ *restriction*$\}$

sublocale *literal.order*: *restricted-total-multiset-extension* **where**
less = $(\prec_t)$ **and** *to-mset* = *literal-to-mset*
using *inj-literal-to-mset*
by *unfold-locales* (*auto simp*: *inj-on-def*)

notation *literal.order.multiset-extension* (**infix** $\prec_l$ *50*)
notation *literal.order.less-eq* (**infix** $\preceq_l$ *50*)

lemmas $less_l$*-def* = *literal.order.multiset-extension-def*

sublocale *maximal-literal* ($\prec_l$)
  by *unfold-locales*

sublocale *clause.order*: *restricted-total-multiset-extension* **where**
  less = ($\prec_l$) **and** *to-mset* = $\lambda x.\ x$ **and** *restriction* = *literal-order-restriction*
  by *unfold-locales auto*

notation *clause.order.multiset-extension* (**infix** $\prec_c$ *50*)
notation *clause.order.less-eq* (**infix** $\preceq_c$ *50*)

lemmas $less_c$*-def* = *clause.order.multiset-extension-def*

end

locale *term-order-lifting* =
  restricted-term-order-lifting **where** *restriction* = *UNIV* +
  term.order: *wellfounded-strict-order* $less_t$ +
  term.order: *total-strict-order* $less_t$
begin

sublocale *literal.order*: *total-wellfounded-multiset-extension* **where**
  less = ($\prec_t$) **and** *to-mset* = *literal-to-mset*
  by *unfold-locales* (*simp add*: *inj-literal-to-mset*)

sublocale *clause.order*: *total-wellfounded-multiset-extension* **where**
  less = ($\prec_l$) **and** *to-mset* = $\lambda x.\ x$
  by *unfold-locales simp*

end

end
theory *Ground-Order*
  imports *Ground-Term-Order Term-Order-Lifting*
begin

locale *ground-order* =
  term.order: *ground-term-order* +
  term-order-lifting

locale *ground-order-with-equality* =
  term.order: *ground-term-order*
begin

sublocale *ground-order*
  where *literal-to-mset* = *mset-lit*
  by *unfold-locales* (*rule inj-mset-lit*)

end

end
theory *Nonground-Term-Order*
imports
Nonground-Term
Nonground-Context
Ground-Order
begin

locale *ground-context-compatible-order* =
nonground-term-with-context +
restricted-total-strict-order **where** *restriction* = *range term.from-ground* +
assumes *ground-context-compatibility*:
$\bigwedge c\ t_1\ t_2.$
term.is-ground $t_1 \Longrightarrow$
term.is-ground $t_2 \Longrightarrow$
context.is-ground $c \Longrightarrow$
$t_1 \prec t_2 \Longrightarrow$
$c\langle t_1 \rangle \prec c\langle t_2 \rangle$
begin

sublocale *context-compatible-restricted-order* **where**
restriction = *range term.from-ground* **and** *context-restriction* = *range context.from-ground*
and
Fun = *Fun* **and** *restricted* = *term.is-ground* **and** *restricted-context* = *context.is-ground*
using *ground-context-compatibility*
by *unfold-locales*
(*auto simp*: *term.is-ground-iff-range-from-ground context.is-ground-iff-range-from-ground*)

end

locale *ground-subterm-property* =
nonground-term-with-context +
fixes *R*
assumes *ground-subterm-property*:
$\bigwedge t_G\ c_G.$
term.is-ground $t_G \Longrightarrow$
context.is-ground $c_G \Longrightarrow$
$c_G \neq \Box \Longrightarrow$
$R\ t_G\ c_G\langle t_G \rangle$

locale *base-grounded-order* =
order: *base-subst-update-stable-grounded-order* +
order: *grounded-restricted-total-strict-order* +
order: *grounded-restricted-wellfounded-strict-order* +
order: *ground-subst-stable-grounded-order* +
grounding

locale *nonground-term-order* =
  nonground-term-with-context +
  order: *restricted-wellfounded-total-strict-order* **where**
  less = $less_t$ **and** *restriction* = *range term.from-ground* +
  order: *ground-subst-stability* **where** $R = less_t$ **and** *comp-subst* = $(\odot)$ **and** *subst* = $(\cdot t)$ **and**
  vars = *term.vars* **and** *id-subst* = *Var* **and** *to-ground* = *term.to-ground* **and**
  from-ground = *term.from-ground* +
  order: *ground-context-compatible-order* **where** *less* = $less_t$ +
  order: *ground-subterm-property* **where** $R = less_t$
for $less_t$ :: $('f, 'v)$ *Term.term* $\Rightarrow$ $('f, 'v)$ *Term.term* $\Rightarrow$ *bool*
begin

interpretation *term-order-notation*.

sublocale *base-grounded-order* **where**
  comp-subst = $(\odot)$ **and** *subst* = $(\cdot t)$ **and** *vars* = *term.vars* **and** *id-subst* = *Var* **and**
  to-ground = *term.to-ground* **and** *from-ground* = *term.from-ground* **and** *less* = $(\prec_t)$
proof *unfold-locales*
  fix *update* x γ **and** t :: $('f, 'v)$ *term*
  assume
    update-is-ground: *term.is-ground update* **and**
    update-less: *update* $\prec_t$ γ x **and**
    term-grounding: *term.is-ground* $(t \cdot t\ \gamma)$ **and**
    var: $x \in$ *term.vars* t

  from *term-grounding var*
  show $t \cdot t\ \gamma(x := update) \prec_t t \cdot t\ \gamma$
  proof(*induction t*)
    case *Var*
    then show *?case*
      using *update-is-ground update-less*
      by *simp*
  next
    case (*Fun f subs*)

    then have $\forall$ *sub* $\in$ *set subs*. *sub* $\cdot t\ \gamma(x := update) \preceq_t sub \cdot t\ \gamma$
      by (*metis eval-with-fresh-var is-ground-iff reflclp-iff term.set-intros*(*4*))

    moreover then have $\exists$ *sub* $\in$ *set subs*. *sub* $\cdot t\ \gamma(x := update) \prec_t sub \cdot t\ \gamma$
      using *Fun update-less*
    by (*metis* (*full-types*) *fun-upd-same term.distinct*(*1*) *term.sel*(*4*) *term.set-cases*(*2*)
        order.dual-order.strict-iff-order term-subst-eq-rev)

    ultimately show *?case*
      using *Fun*(*2*, *3*)

proof(*induction filter* (λ*sub. sub* $\cdot t$ $\gamma(x := update) \prec_t sub \cdot t \gamma$) *subs arbitrary*: *subs*)
case *Nil*
then show *?case*
unfolding *empty-filter-conv*
by *blast*
next
case *first*: (*Cons s ss*)

have *groundings* [*simp*]: *term.is-ground* (*s* $\cdot t$ $\gamma(x := update)$) *term.is-ground* (*s* $\cdot t$ γ)
using *term.ground-subst-update update-is-ground*
by (*metis* (*lifting*) *filter-eq-ConsD first.hyps*(*2*) *first.prems*(*3*) *in-set-conv-decomp is-ground-iff term.set-intros*(*4*))+

show *?case*
proof(*cases ss*)
case *Nil*
then obtain *ss1 ss2* **where** *subs*: *subs* = *ss1* @ *s* # *ss2*
using *filter-eq-ConsD*[*OF first.hyps*(*2*)[*symmetric*]]
by *blast*

have *ss1*: $\forall s \in set\ ss1.\ s \cdot t\ \gamma(x := update) = s \cdot t\ \gamma$
using *first.hyps*(*2*) *first.prems*(*1*)
unfolding *Nil subs*
by (*smt* (*verit*, *del-insts*) *Un-iff append-Cons-eq-iff filter-empty-conv filter-eq-ConsD*
set-append order.antisym-conv2)

have *ss2*: $\forall s \in set\ ss2.\ s \cdot t\ \gamma(x := update) = s \cdot t\ \gamma$
using *first.hyps*(*2*) *first.prems*(*1*)
unfolding *Nil subs*
by (*smt* (*verit*, *ccfv-SIG*) *Un-iff append-Cons-eq-iff filter-empty-conv filter-eq-ConsD*
list.set-intros(*2*) *set-append order.antisym-conv2*)

let *?c* = *More f ss1* $\square$ *ss2* $\cdot t_c$ γ

have *context.is-ground ?c*
using *subs first*(*5*)
by *auto*

moreover have *s* $\cdot t$ $\gamma(x := update) \prec_t s \cdot t\ \gamma$
using *first.hyps*(*2*)
by (*meson Cons-eq-filterD*)

ultimately have *?c*$\langle s \cdot t\ \gamma(x := update)\rangle \prec_t$ *?c*$\langle s \cdot t\ \gamma\rangle$
using *order.ground-context-compatibility groundings*
by *blast*

```
    moreover have Fun f subs ·t γ(x := update) = ?c⟨s ·t γ(x := update)⟩
      unfolding subs
      using ss1 ss2
      by simp

    moreover have Fun f subs ·t γ = ?c⟨s ·t γ⟩
      unfolding subs
      by auto

    ultimately show ?thesis
      by argo
  next
    case (Cons t' ts')

    from first(2)
    obtain ss1 ss2 where
      subs: subs = ss1 @ s # ss2 and
      ss1: ∀ s∈set ss1. ¬ s ·t γ(x := update) ≺t s ·t γ and
      less: s ·t γ(x := update) ≺t s ·t γ and
      ss: ss = filter (λterm. term ·t γ(x := update)≺t term ·t γ) ss2
      using Cons-eq-filter-iff[of s ss (λs. s ·t γ(x := update) ≺t s ·t γ)]
      by blast

    let ?subs' = ss1 @ (s ·t γ(x := update)) # ss2

    have [simp]: s ·t γ(x := update) ·t γ = s ·t γ(x := update)
      using first.prems(3) update-is-ground
      unfolding subs
      by (simp add: is-ground-iff)

    have [simp]: s ·t γ(x := update) ·t γ(x := update) = s ·t γ(x := update)
      using first.prems(3) update-is-ground
      unfolding subs
      by (simp add: is-ground-iff)

    have ss: ss = filter (λsub. sub ·t γ(x := update) ≺t sub ·t γ) ?subs'
      using ss1 ss
      by auto

    moreover have ∀ sub ∈ set ?subs'. sub ·t γ(x := update) ⪯t sub ·t γ
      using first.prems(1)
      unfolding subs
      by simp

    moreover have ex-less: ∃ sub ∈ set ?subs'. sub ·t γ(x := update) ≺t sub ·t
γ
      using ss Cons neq-Nil-conv
      by force
```

```
      moreover have subs'-grounding: term.is-ground (Fun f ?subs' ·t γ)
        using first.prems(3)
        unfolding subs
        by simp

      moreover have x ∈ term.vars (Fun f ?subs')
        by (metis ex-less eval-with-fresh-var term.set-intros(4) order.less-irrefl)

      ultimately have less-subs': Fun f ?subs' ·t γ(x := update) ≺t Fun f ?subs'
·t γ
        using first.hyps(1) first.prems(3)
        by blast

      have context-grounding: context.is-ground (More f ss1 □ ss2 ·tc γ)
        using subs'-grounding
        by auto

      have Fun f (ss1 @ s ·t γ(x := update) # ss2) ·t γ ≺t Fun f subs ·t γ
        unfolding subs
        using order.ground-context-compatibility[OF - - context-grounding less]
        by simp

      with less-subs' show ?thesis
        unfolding subs
        by simp
    qed
  qed
 qed
qed


notation order.less_G (infix ≺tG 50)
notation order.less-eq_G (infix ⪯tG 50)

sublocale restriction: ground-term-order (≺tG)
proof unfold-locales
 fix c t t'
 assume t ≺tG t'
 then show c⟨t⟩G ≺tG c⟨t'⟩G
   using order.ground-context-compatibility[OF
       term.ground-is-ground term.ground-is-ground context.ground-is-ground]
   unfolding order.less_G-def
   by simp
next
 fix t :: 'f gterm and c :: 'f ground-context
 assume c ≠ □
 then show t ≺tG c⟨t⟩G
  using order.ground-subterm-property[OF term.ground-is-ground context.ground-is-ground]
```

unfolding *order.less$_G$-def*
by *simp*
qed

end

end
theory *Nonground-Order*
imports
Nonground-Clause
Nonground-Term-Order
Term-Order-Lifting
begin

9 Nonground Order

locale *nonground-order-lifting* =
grounding-lifting +
order: *total-grounded-multiset-extension* +
order: *ground-subst-stable-total-multiset-extension* +
order: *subst-update-stable-multiset-extension*
begin

sublocale *order*: *grounded-restricted-total-strict-order* **where**
less = *order.multiset-extension* **and** *subst* = *subst* **and** *vars* = *vars* **and** *to-ground* = *to-ground* **and**
from-ground = *from-ground*
by *unfold-locales*

end

locale *nonground-term-based-order-lifting* =
term: *nonground-term* +
nonground-order-lifting **where**
id-subst = *Var* **and** *comp-subst* = $(\odot)$ **and** *base-vars* = *term.vars* **and** *base-less* = *less$_t$* **and**
base-subst = $(\cdot t)$
for *less$_t$*

locale *nonground-equality-order* =
nonground-clause +
term: *nonground-term-order*
begin

sublocale *restricted-term-order-lifting* **where**
restriction = *range term.from-ground* **and** *literal-to-mset* = *mset-lit*
by *unfold-locales* (*rule inj-mset-lit*)

notation *term.order.less*$_G$ (**infix** $\prec_{tG}$ *50*)
notation *term.order.less-eq*$_G$ (**infix** $\preceq_{tG}$ *50*)

sublocale *literal*: *nonground-term-based-order-lifting* **where**
less = *less*$_t$ **and** *sub-subst* = $(\cdot t)$ **and** *sub-vars* = *term.vars* **and** *sub-to-ground* = *term.to-ground* **and**
sub-from-ground = *term.from-ground* **and** *map* = *map-uprod-literal* **and** *to-set* = *uprod-literal-to-set* **and**
to-ground-map = *map-uprod-literal* **and** *from-ground-map* = *map-uprod-literal* **and**
ground-map = *map-uprod-literal* **and** *to-set-ground* = *uprod-literal-to-set* **and** *to-mset* = *mset-lit*
rewrites
$\bigwedge l\ \sigma$. *functional-substitution-lifting.subst* $(\cdot t)$ *map-uprod-literal l* σ = *literal.subst l* σ **and**
$\bigwedge l$. *functional-substitution-lifting.vars term.vars uprod-literal-to-set l* = *literal.vars l* **and**
$\bigwedge l_G$. *grounding-lifting.from-ground term.from-ground map-uprod-literal* l_G = *literal.from-ground* l_G **and**
$\bigwedge l$. *grounding-lifting.to-ground term.to-ground map-uprod-literal l* = *literal.to-ground l*
by *unfold-locales* (*auto simp*: *inj-mset-lit mset-lit-image-mset*)

notation *literal.order.less*$_G$ (**infix** $\prec_{lG}$ *50*)
notation *literal.order.less-eq*$_G$ (**infix** $\preceq_{lG}$ *50*)

sublocale *clause*: *nonground-term-based-order-lifting* **where**
less = $(\prec_l)$ **and** *sub-subst* = *literal.subst* **and** *sub-vars* = *literal.vars* **and**
sub-to-ground = *literal.to-ground* **and** *sub-from-ground* = *literal.from-ground* **and**
map = *image-mset* **and** *to-set* = *set-mset* **and** *to-ground-map* = *image-mset* **and**
from-ground-map = *image-mset* **and** *ground-map* = *image-mset* **and** *to-set-ground* = *set-mset* **and**
to-mset = $\lambda x.\ x$
by *unfold-locales simp-all*

notation *clause.order.less*$_G$ (**infix** $\prec_{cG}$ *50*)
notation *clause.order.less-eq*$_G$ (**infix** $\preceq_{cG}$ *50*)

lemma *obtain-maximal-literal*:
assumes
not-empty: $C \neq \{\#\}$ **and**
grounding: *clause.is-ground* $(C \cdot \gamma)$
obtains *l*
where *is-maximal l C is-maximal* $(l \cdot l\ \gamma)$ $(C \cdot \gamma)$
proof−

have *grounding-not-empty*: $C \cdot \gamma \neq \{\#\}$
using *not-empty*

```
    by simp

  obtain l where
    l-in-C: l ∈# C and
    l-grounding-is-maximal: is-maximal (l ·l γ) (C · γ)
    using
      ex-maximal-in-mset-wrt[OF
        literal.order.transp-on-less literal.order.asymp-on-less grounding-not-empty]
      maximal-in-clause
    unfolding clause.subst-def
    by (metis (mono-tags, lifting) image-iff multiset.set-map)

  show ?thesis
  proof(cases is-maximal l C)
    case True

    with l-grounding-is-maximal that
    show ?thesis
      by blast
  next
    case False
    then obtain l' where
      l'-in-C: l' ∈# C and
      l-less-l': l ≺l l'
      unfolding is-maximal-def
      using l-in-C
      by blast

    note literals-in-C = l-in-C l'-in-C
    note literals-grounding = literals-in-C[THEN clause.to-set-is-ground-subst[OF
- grounding]]

    have l ·l γ ≺l l' ·l γ
      using literal.order.ground-subst-stability[OF literals-grounding l-less-l'].

    then have False
     using
       l-grounding-is-maximal
       clause.subst-in-to-set-subst[OF l'-in-C]
     unfolding is-maximal-def
     by force

    then show ?thesis..
  qed
qed

lemma obtain-strictly-maximal-literal:
  assumes
   grounding: clause.is-ground (C · γ) and
```

```
  ground-strictly-maximal: is-strictly-maximal l_G (C · γ)
 obtains l where
  is-strictly-maximal l C l_G = l ·l γ
proof−

  have grounding-not-empty: C · γ ≠ {#}
    using is-strictly-maximal-not-empty[OF ground-strictly-maximal].

  have l_G-in-grounding: l_G ∈# C · γ
    using strictly-maximal-in-clause[OF ground-strictly-maximal].

  obtain l where
    l-in-C: l ∈# C and
    l_G [simp]: l_G = l ·l γ
    using l_G-in-grounding
    unfolding clause.subst-def
    by blast

  show ?thesis
  proof(cases is-strictly-maximal l C)
    case True
    show ?thesis
      using that[OF True l_G].
  next
    case False

    then obtain l' where
      l'-in-C: l' ∈# C − {# l #} and
      l-less-eq-l': l ⪯l l'
      unfolding is-strictly-maximal-def
      using l-in-C
      by blast

    note l-grounding =
       clause.to-set-is-ground-subst[OF l-in-C grounding]

    have l'-grounding: literal.is-ground (l' ·l γ)
      using l'-in-C grounding
      by (meson clause.to-set-is-ground-subst in-diffD)

    have l ·l γ ⪯l l' ·l γ
       using literal.order.less-eq.ground-subst-stability[OF l-grounding l'-grounding
l-less-eq-l'].

    then have False
      using clause.subst-in-to-set-subst[OF l'-in-C] ground-strictly-maximal
      unfolding is-strictly-maximal-def subst-clause-remove1-mset[OF l-in-C]
      by simp
```

```
    then show ?thesis..
  qed
qed

lemma is-maximal-if-grounding-is-maximal:
  assumes
   l-in-C: l ∈# C and
   C-grounding: clause.is-ground (C · γ) and
   l-grounding-is-maximal: is-maximal (l ·l γ) (C · γ)
 shows
   is-maximal l C
proof(rule ccontr)
  assume ¬ is-maximal l C

  then obtain l' where l-less-l': l ≺l l' and l'-in-C: l' ∈# C
    using l-in-C
    unfolding is-maximal-def
    by blast

  have l'-grounding: literal.is-ground (l' ·l γ)
    using clause.to-set-is-ground-subst[OF l'-in-C C-grounding].

  have l-grounding: literal.is-ground (l ·l γ)
    using clause.to-set-is-ground-subst[OF l-in-C C-grounding].

  have l'-γ-in-C-γ: l' ·l γ ∈# C · γ
    using clause.subst-in-to-set-subst[OF l'-in-C].

  have l ·l γ ≺l l' ·l γ
    using literal.order.ground-subst-stability[OF l-grounding l'-grounding l-less-l'].

  then have ¬ is-maximal (l ·l γ) (C · γ)
    using l'-γ-in-C-γ
    unfolding is-maximal-def literal.subst-comp-subst
    by fastforce

  then show False
    using l-grounding-is-maximal..
qed

lemma is-strictly-maximal-if-grounding-is-strictly-maximal:
  assumes
   l-in-C: l ∈# C and
   grounding: clause.is-ground (C · γ) and
   grounding-strictly-maximal: is-strictly-maximal (l ·l γ) (C · γ)
 shows
   is-strictly-maximal l C
  using
    is-maximal-if-grounding-is-maximal[OF
```

l-in-C
grounding
is-maximal-if-is-strictly-maximal[*OF grounding-strictly-maximal*]
]
grounding-strictly-maximal
unfolding
is-strictly-maximal-def is-maximal-def
subst-clause-remove1-mset[*OF l-in-C*, *symmetric*]
reflclp-iff
by (*metis in-diffD clause.subst-in-to-set-subst*)

lemma *unique-maximal-in-ground-clause*:
assumes
clause.is-ground C
is-maximal l C
is-maximal l′ C
shows
$l = l'$
using *assms clause.to-set-is-ground literal.order.not-less-eq*
unfolding *is-maximal-def reflclp-iff*
by *meson*

lemma *unique-strictly-maximal-in-ground-clause*:
assumes
clause.is-ground C
is-strictly-maximal l C
is-strictly-maximal l′ C
shows
$l = l'$
using *assms unique-maximal-in-ground-clause*
by *blast*

thm *literal.order.order.strict-iff-order*

abbreviation *ground-is-maximal* **where**
ground-is-maximal l_G $C_G \equiv$ *is-maximal* (*literal.from-ground* l_G) (*clause.from-ground* C_G)

abbreviation *ground-is-strictly-maximal* **where**
ground-is-strictly-maximal l_G $C_G \equiv$
is-strictly-maximal (*literal.from-ground* l_G) (*clause.from-ground* C_G)

sublocale *ground*: *ground-order-with-equality* **where**
$less_t = (\prec_{tG})$
rewrites
$less_{lG}$-*rewrite* [*simp*]: *multiset-extension.multiset-extension* $(\prec_{tG})$ *mset-lit* $= (\prec_{lG})$
and
$less_{cG}$-*rewrite* [*simp*]: *multiset-extension.multiset-extension* $(\prec_{lG})$ $(\lambda x.\ x) = (\prec_{cG})$

and
is-maximal-rewrite [*simp*]: $\bigwedge l_G\ C_G$. *ground.is-maximal* $l_G\ C_G \longleftrightarrow$ *ground-is-maximal* $l_G\ C_G$ **and**
is-strictly-maximal-rewrite [*simp*]:
$\bigwedge l_G\ C_G$. *ground.is-strictly-maximal* $l_G\ C_G \longleftrightarrow$ *ground-is-strictly-maximal* $l_G\ C_G$
proof *unfold-locales*

interpret *multiset-extension* $(\prec_{tG})$ *mset-lit*
by *unfold-locales*

interpret *relation-restriction*
$(\lambda b1\ b2.$ *multp* $(\prec_t)$ *(mset-lit b1)* *(mset-lit b2))* *literal.from-ground*
by *unfold-locales*

show *less*$_{lG}$*-rewrite*: $(\prec_m) = (\prec_{lG})$
unfolding *multiset-extension-def literal.order.multiset-extension-def* R_r*-def*
unfolding *term.order.less*$_G$*-def literal.from-ground-def atom.from-ground-def*
by (*metis term.inj-from-ground mset-lit-image-mset multp-image-mset-image-msetD multp-image-mset-image-msetI term.order.transp-on-less*)

fix $l_G\ C_G$
show *is-maximal-in-mset* $C_G\ l_G \longleftrightarrow$ *ground-is-maximal* $l_G\ C_G$
unfolding *is-maximal-in-mset-iff*
by (*simp add*: *clause.to-set-from-ground image-iff is-maximal-def less*$_{lG}$*-rewrite literal.order.less*$_r$*-def*)

then show *is-strictly-maximal-in-mset* $C_G\ l_G \longleftrightarrow$ *ground-is-strictly-maximal* $l_G\ C_G$
unfolding
is-strictly-maximal-def is-strictly-maximal-in-mset-iff reflclp-iff is-maximal-def is-maximal-in-mset-iff
by (*smt* (*verit*, *ccfv-SIG*) *clause.ground-sub-in-ground clause-from-ground-remove1-mset in-remove1-mset-neq*)
next

interpret *multiset-extension* $(\prec_{lG})$ $\lambda x.\ x$
by *unfold-locales*

interpret *relation-restriction multp* $(\prec_l)$ *clause.from-ground*
by *unfold-locales*

show *less*$_{cG}$*-rewrite*: $(\prec_m) = (\prec_{cG})$
unfolding *multiset-extension-def clause.order.multiset-extension-def* R_r*-def*
unfolding *literal.order.less*$_G$*-def clause.from-ground-def*
by (*metis literal.inj-from-ground literal.order.transp multp-image-mset-image-msetD multp-image-mset-image-msetI*)

qed

lemma *less$_t$-less$_l$*:
assumes $t_1 \prec_t t_2$
shows
less$_t$-less$_l$-pos: $t_1 \approx t_3 \prec_l t_2 \approx t_3$ **and**
less$_t$-less$_l$-neg: $t_1 \not\approx t_3 \prec_l t_2 \not\approx t_3$
using *assms*
unfolding *less$_l$-def*
by (*auto simp*: *multp-add-mset multp-add-mset'*)

lemma *literal-order-less-if-all-lesseq-ex-less-set*:
assumes
$\forall t \in$ *set-uprod* (*atm-of l*). $t \cdot t \sigma' \preceq_t t \cdot t \sigma$
$\exists t \in$ *set-uprod* (*atm-of l*). $t \cdot t \sigma' \prec_t t \cdot t \sigma$
shows $l \cdot l \sigma' \prec_l l \cdot l \sigma$
using *literal.order.less-if-all-lesseq-ex-less*[*OF assms*[*folded set-mset-set-uprod*]].

lemma *less$_c$-add-mset*:
assumes $l \prec_l l'$ $C \preceq_c C'$
shows *add-mset l C* $\prec_c$ *add-mset l' C'*
using *assms multp-add-mset-reflclp*[*OF literal.order.asymp literal.order.transp*]
unfolding *less$_c$-def*
by *blast*

lemmas *less$_c$-add-same* [*simp*] =
multp-add-same[*OF literal.order.asymp literal.order.transp*, *folded less$_c$-def*]

end

end
theory *Typed-Functional-Substitution-Example*
imports
Typed-Functional-Substitution
Abstract-Substitution.Functional-Substitution-Example
begin

type-synonym (*'f*, *'ty*) *fun-types* = *'f* ⇒ *'ty list* × *'ty*

Inductive predicate defining well-typed terms.

inductive *welltyped* :: (*'f*, *'ty*) *fun-types* ⇒ (*'v*, *'ty*) *var-types* ⇒ (*'f*,*'v*) *term* ⇒ *'ty* ⇒ *bool*
for $\mathcal{F}$ $\mathcal{V}$ **where**
Var: $\mathcal{V}$ $x = \tau \Longrightarrow$ *welltyped* $\mathcal{F}$ $\mathcal{V}$ (*Var x*) τ
| *Fun*: $\mathcal{F}$ $f = (\tau s, \tau) \Longrightarrow$ *list-all2* (*welltyped* $\mathcal{F}$ $\mathcal{V}$) *ts* $\tau s \Longrightarrow$ *welltyped* $\mathcal{F}$ $\mathcal{V}$ (*Fun f ts*) τ

global-interpretation *term*: *base-typing welltyped* $\mathcal{F}$ $\mathcal{V}$

proof *unfold-locales*
show *right-unique* (*welltyped* $\mathcal{F}$ $\mathcal{V}$)
proof (*rule right-uniqueI*)
fix t τ_1 τ_2
assume *welltyped* $\mathcal{F}$ $\mathcal{V}$ t τ_1 **and** *welltyped* $\mathcal{F}$ $\mathcal{V}$ t τ_2
thus $\tau_1 = \tau_2$
by (*auto elim*!: *welltyped.cases*)
qed
qed

global-interpretation *functional-substitution-typing* **where**
welltyped = *welltyped* $\mathcal{F}$ **and**
subst = *subst-apply-term* **and** *id-subst* = *Var* **and** *comp-subst* = *subst-compose*
and
vars = *vars-term* :: ('*f*, '*v*) *term* ⇒ '*v set*
for $\mathcal{F}$ :: ('*f*, '*ty*) *fun-types*
by *unfold-locales* (*simp-all add*: *welltyped.Var*)

A selection of substitution properties for typed terms.

locale *typed-term-subst-properties* =
welltyped: *base-typed-subst-stability* **where** *welltyped* = *welltyped* $\mathcal{F}$
for $\mathcal{F}$ :: ('*f*, '*ty*) *fun-types*

global-interpretation *term*: *typed-term-subst-properties* **where**
subst = *subst-apply-term* **and** *id-subst* = *Var* **and** *comp-subst* = *subst-compose*
and
vars = *vars-term* :: ('*f*, '*v*) *term* ⇒ '*v set* **and** $\mathcal{F} = \mathcal{F}$
for $\mathcal{F}$ :: '*f* ⇒ '*ty list* × '*ty*
proof (*unfold-locales*)
fix $\mathcal{V}$ :: ('*v*, '*ty*) *var-types* **and** t :: ('*f*, '*v*) *term* **and** σ τ
assume *is-welltyped-on*: ∀ x ∈ *vars-term* t. *welltyped* $\mathcal{F}$ $\mathcal{V}$ (σ x) ($\mathcal{V}$ x)

show *welltyped* $\mathcal{F}$ $\mathcal{V}$ (t · σ) τ ⟷ *welltyped* $\mathcal{F}$ $\mathcal{V}$ t τ
proof(*rule iffI*)
assume *welltyped* $\mathcal{F}$ $\mathcal{V}$ t τ
then show *welltyped* $\mathcal{F}$ $\mathcal{V}$ (t · σ) τ
using *is-welltyped-on*
by(*induction rule*: *welltyped.induct*)
(*auto simp*: *list.rel-mono-strong list-all2-map1 welltyped.simps*)
next
assume *welltyped* $\mathcal{F}$ $\mathcal{V}$ (t · σ) τ
then show *welltyped* $\mathcal{F}$ $\mathcal{V}$ t τ
using *is-welltyped-on*
proof(*induction* t · σ τ *arbitrary*: t *rule*: *welltyped.induct*)
case (*Var* x τ)

then obtain x' **where** t: t = *Var* x'
by (*metis subst-apply-eq-Var*)

have *welltyped* $\mathcal{F}$ $\mathcal{V}$ *t* ($\mathcal{V}$ *x'*)
unfolding *t*
by (*simp add*: *welltyped.Var*)

moreover have *welltyped* $\mathcal{F}$ $\mathcal{V}$ *t* ($\mathcal{V}$ *x*)
using *Var*
unfolding *t*
by (*simp add*: *welltyped.simps*)

ultimately have $\mathcal{V}$-*x'*: τ = $\mathcal{V}$ *x'*
using *Var.hyps*
by (*simp add*: *t welltyped.simps*)

show *?case*
unfolding *t* $\mathcal{V}$-*x'*
by (*simp add*: *welltyped.Var*)
next
case (*Fun f* τ*s* τ *ts*)

then show *?case*
by (*cases t*) (*simp-all add*: *list.rel-mono-strong list-all2-map1 welltyped.simps*)
qed
qed
qed

find-theorems *name*: *is-welltyped-subst-update*

Examples of generated lemmas and definitions

thm

term.right-unique
term.welltyped.subst-stability

is-welltyped-subst-update
is-welltyped-on-subset
is-welltyped-id-subst

term *term.is-welltyped*
term *is-welltyped-on*
term *is-welltyped*

end
theory *Typed-Functional-Substitution-Lifting-Example*
imports
Typed-Functional-Substitution-Lifting
Typed-Functional-Substitution-Example
Abstract-Substitution.Functional-Substitution-Lifting-Example
begin

All property locales have corresponding lifting locales

locale *nonground-typing-lifting* =
  is-welltyped: *typed-subst-stability-lifting* **where**
    sub-welltyped = *sub-welltyped* **and** *base-welltyped* = *welltyped* $\mathcal{F}$
for $\mathcal{F}$ :: $('f, 'ty)$ *fun-types* **and** *sub-welltyped* :: $('v \Rightarrow 'ty) \Rightarrow 'sub \Rightarrow 'ty' \Rightarrow$ *bool*

locale *example-typing-lifting* =
  fixes $\mathcal{F}$ :: $('f, 'ty)$ *fun-types*
begin

sublocale *equation*:
  typing-lifting' **where**
    sub-welltyped = *welltyped* $\mathcal{F}$ **and**
  to-set = *fset*
  by *unfold-locales*

sublocale *equation*:
  nonground-typing-lifting **where**
  base-vars = *vars-term* **and** *base-subst* = *subst-apply-term* **and** *map* = λf. *map-prod f f* **and**
  to-set = *set-prod* **and** *comp-subst* = *subst-compose* **and** *id-subst* = *Var* **and**
  sub-vars = *vars-term* **and** *sub-subst* = *subst-apply-term* **and**
  sub-welltyped = *welltyped* $\mathcal{F}$
  by *unfold-locales*

Lifted lemmas and definitions

thm
  equation.is-welltyped-def
  equation.is-welltyped.subst-stability

term *equation.is-welltyped*

We can lift multiple levels

sublocale *equation-set*:
  typing-lifting **where**
    sub-welltyped = *equation.is-welltyped.welltyped* $\mathcal{V}$ **and**
  to-set = *fset*
  by *unfold-locales*

sublocale *equation-set*:
  nonground-typing-lifting **where**
  base-vars = *vars-term* **and** *base-subst* = *subst-apply-term* **and** *map* = *fimage* **and**
  to-set = *fset* **and** *comp-subst* = *subst-compose* **and** *id-subst* = *Var* **and**
  sub-vars = *equation-subst.vars* **and** *sub-subst* = *equation-subst.subst* **and**
  sub-welltyped = *equation.is-welltyped.welltyped*
  by *unfold-locales*

Lifted lemmas and definitions

thm

equation-set.is-welltyped-def
equation-set.is-welltyped.subst-stability

term *equation-set.is-welltyped*

end

Interpretation with unit as type

global-interpretation *example-typing-lifting* λ-. ([], ())**.**

end