First Order Clause

Balazs Toth

March 10, 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** · *67*)

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

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

abbreviation (*input*) *GHole* (‹$\Box_G$›) **where**
  $\Box_G \equiv \Box$

abbreviation *ctxt-apply-gterm* (‹-⟨-⟩$_G$› [*1000*, *0*] *1000*) **where**
  $C\langle s\rangle_G \equiv$ *GFun*⟨*C*;*s*⟩

lemma *le-size-gctxt*: *size t* ≤ *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*]: $\exists M.\ x \in$ *set-mset 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 ≤ 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 ≡ B − A and K ≡ A − B
  assumes J ≠ {#} and ∀ k ∈# K. ∃ x ∈# J. R k x
  shows multp_HO R A B
  unfolding multp_HO-def
proof (intro conjI allI impI)
  show A ≠ B
    using assms
    by force
next
  fix y
  assume count B y < count A y

  then show ∃ x. R y x ∧ count A x < count B x
    using assms
    by (metis in-diff-count)
qed

lemma Uniq-antimono: Q ≤ P ⟹ Uniq Q ≥ Uniq P
  unfolding le-fun-def le-bool-def
  by (rule impI) (simp only: Uniq-I Uniq-D)

lemma Uniq-antimono': (⋀x. Q x ⟹ P x) ⟹ Uniq P ⟹ 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#} ⟷ (∀ y ∈# M. R y x)
proof (rule iffI)
  show ∀ y ∈# M. R y x ⟹ multp R M {#x#}
    using one-step-implies-multp[of {#x#} - R {#}, simplified] .
next
  show multp R M {#x#} ⟹ ∀ y∈#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 sin-
gle-is-union
       singletonD)
qed

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

shows *multp* R {#x#} $M \longleftrightarrow$ ({#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 $J \equiv B - A$ **and** $K \equiv A - B$
shows $J \neq$ {#} **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 $J \neq$ {#} **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: $B = add\text{-}mset\ b\ B'$ **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: $\forall x.\ x \in\# X \longrightarrow (\exists y.\ y \in\# Y \land R\ x\ y)$
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* $\neq$ *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 $\exists y'$. *R* x' y' $\wedge$ *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 (λX :: '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*$_{DM}$: ¬ *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*: ($\bigwedge$*x*. *f* (*g x*) = *x*) $\Longrightarrow$ ($\bigwedge$*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*]: ($\forall$ *t*$\in$*set-uprod* (*Upair* t_1 t_2). *P t*) $\longleftrightarrow$ *P* t_1 $\wedge$ *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* $\neq$ *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* $\neq$ {}
  by(*cases a*) *simp*

lemma *exists-uprod* [*intro*]: $\exists$ *a*. *x* $\in$ *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 *predicate-typed* =
fixes *typed* :: *′expr* ⇒ *′ty* ⇒ *bool*
assumes *right-unique*: *right-unique typed*
begin

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

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

end

definition *uniform-typed-lifting* **where**
uniform-typed-lifting to-set sub-typed expr ≡ ∃ τ. ∀ *sub* ∈ *to-set expr*. *sub-typed sub* τ

definition *is-typed-lifting* **where**
is-typed-lifting to-set sub-is-typed expr ≡ ∀ *sub* ∈ *to-set expr*. *sub-is-typed sub*

locale *typing* =
fixes *is-typed is-welltyped*
assumes *is-typed-if-is-welltyped*:
⋀*expr*. *is-welltyped expr* ⟹ *is-typed expr*

locale *explicit-typing* =
typed: *predicate-typed* **where** *typed* = *typed* +
welltyped: *predicate-typed* **where** *typed* = *welltyped*
for *typed welltyped* :: *′expr* ⇒ *′ty* ⇒ *bool* +
assumes *typed-if-welltyped*: ⋀*expr* τ. *welltyped expr* τ ⟹ *typed expr* τ
begin

abbreviation *is-typed* **where**
is-typed ≡ *typed.is-typed*

abbreviation *is-welltyped* **where**
is-welltyped ≡ *welltyped.is-typed*

sublocale *typing* **where** *is-typed* = *is-typed* **and** *is-welltyped* = *is-welltyped*
using *typed-if-welltyped*
by *unfold-locales auto*

lemma *typed-welltyped-same-type*:
assumes *typed expr* τ *welltyped expr* τ'
shows $\tau = \tau'$
using *assms typed-if-welltyped*
by *blast*

end

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

abbreviation *is-typed* **where**
is-typed ≡ *uniform-typed-lifting to-set sub-typed*

lemmas *is-typed-def* = *uniform-typed-lifting-def*[*of to-set sub-typed*]

abbreviation *is-welltyped* **where**
is-welltyped ≡ *uniform-typed-lifting to-set sub-welltyped*

lemmas *is-welltyped-def* = *uniform-typed-lifting-def*[*of to-set sub-welltyped*]

sublocale *typing* **where** *is-typed* = *is-typed* **and** *is-welltyped* = *is-welltyped*
proof *unfold-locales*
fix *expr*
assume *is-welltyped expr*
then show *is-typed expr*
using *sub.typed-if-welltyped*
unfolding *is-typed-def is-welltyped-def*
by *auto*
qed

end

locale *typing-lifting* =
sub: *typing* **where** *is-typed* = *sub-is-typed* **and** *is-welltyped* = *sub-is-welltyped*

for *sub-is-typed sub-is-welltyped* :: *'sub* ⇒ *bool* +
fixes
  to-set :: *'expr* ⇒ *'sub set*
begin

abbreviation *is-typed* **where**
  is-typed ≡ *is-typed-lifting to-set sub-is-typed*

lemmas *is-typed-def* = *is-typed-lifting-def*[*of to-set sub-is-typed*]

abbreviation *is-welltyped* **where**
  is-welltyped ≡ *is-typed-lifting to-set sub-is-welltyped*

lemmas *is-welltyped-def* = *is-typed-lifting-def*[*of to-set sub-is-welltyped*]

sublocale *typing* **where** *is-typed* = *is-typed* **and** *is-welltyped* = *is-welltyped*
proof *unfold-locales*
  fix *expr*
  assume *is-welltyped expr*
  then show *is-typed expr*
    using *sub.is-typed-if-is-welltyped*
    unfolding *is-typed-def is-welltyped-def*
    by *simp*
qed

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*
  for *to-set* :: *'expr* ⇒ *'sub set* +
  fixes *sub-is-typed* :: *'sub* ⇒ *bool*
begin

abbreviation (*input*) *is-typed* **where**
  is-typed ≡ *is-typed-lifting to-set sub-is-typed*

lemma *add* [*simp*]:
  is-typed (*add sub M*) ⟷ *sub-is-typed sub* ∧ *is-typed M*
  using *to-set-add*
  unfolding *is-typed-lifting-def*
  by *auto*

lemma *plus* [*simp*]:

is-typed (*plus M M'*) ⟷ *is-typed M* ∧ *is-typed M'*
unfolding *is-typed-lifting-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-typed empty*
by (*simp add*: *is-typed-lifting-def*)

end

locale *natural-magma-typing-lifting* = *typing-lifting* + *natural-magma*
begin

sublocale *is-typed*: *natural-magma-is-typed-lifting* **where** *sub-is-typed* = *sub-is-typed*
by *unfold-locales*

sublocale *is-welltyped*: *natural-magma-is-typed-lifting* **where** *sub-is-typed* = *sub-is-welltyped*
by *unfold-locales*

end

locale *natural-magma-with-empty-typing-lifting* =
natural-magma-typing-lifting + *natural-magma-with-empty*
begin

sublocale *is-typed*: *natural-magma-with-empty-is-typed-lifting* **where** *sub-is-typed*
= *sub-is-typed*
by *unfold-locales*

sublocale *is-welltyped*: *natural-magma-with-empty-is-typed-lifting* **where**
sub-is-typed = *sub-is-welltyped*
by *unfold-locales*

end

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*

begin

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

end

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

lemma *literal-cases*: $[\![\mathcal{P} \in \{Pos, Neg\}; \mathcal{P} = Pos \Longrightarrow P; \mathcal{P} = Neg \Longrightarrow P]\!] \Longrightarrow P$
by *blast*

lemma *map-literal-inverse*:
$(\bigwedge x.\ f\ (g\ x) = x) \Longrightarrow (\bigwedge l.\ map\text{-}literal\ f\ (map\text{-}literal\ g\ l) = l)$
by (*simp add*: *literal.map-comp literal.map-ident-strong*)

lemma *map-literal-comp*:
map-literal f (*map-literal g l*) = *map-literal* ($\lambda a.\ f\ (g\ a)$) *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'*) ∈ ($\lambda(t_1,\ t_2).\ Upair\ t_1\ t_2$) ‘ *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′* = *uprod-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: *explicit-typing term-typed term-welltyped*
for *term-typed term-welltyped*
begin

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

lemma *atom-is-typed-iff* [*simp*]:
atom.is-typed (*Upair t t'*) $\longleftrightarrow$ ($\exists\tau$. *term-typed t* τ $\wedge$ *term-typed t'* τ)
unfolding *atom.is-typed-def*
by *auto*

lemma *atom-is-welltyped-iff* [*simp*]:
atom.is-welltyped (*Upair t t'*) $\longleftrightarrow$ ($\exists\tau$. *term-welltyped t* τ $\wedge$ *term-welltyped t'* τ)
unfolding *atom.is-welltyped-def*
by *auto*

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

lemma *literal-is-typed-iff* [*simp*]:
literal.is-typed ($t \approx t'$) $\longleftrightarrow$ *atom.is-typed* (*Upair t t'*)
literal.is-typed (t !$\approx$ t') $\longleftrightarrow$ *atom.is-typed* (*Upair t t'*)
unfolding *literal.is-typed-def*
by (*simp-all add*: *set-literal-atm-of*)

lemma *literal-is-welltyped-iff* [*simp*]:
literal.is-welltyped ($t \approx t'$) $\longleftrightarrow$ *atom.is-welltyped* (*Upair t t'*)
literal.is-welltyped (t !$\approx$ t') $\longleftrightarrow$ *atom.is-welltyped* (*Upair t t'*)
unfolding *literal.is-welltyped-def*
by *simp-all*

lemma *literal-is-typed-iff-atm-of*: *literal.is-typed l* $\longleftrightarrow$ *atom.is-typed* (*atm-of l*)
unfolding *literal.is-typed-def*
by (*simp add*: *set-literal-atm-of*)

lemma *literal-is-welltyped-iff-atm-of*:
literal.is-welltyped l $\longleftrightarrow$ *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-is-typed = *literal.is-typed* **and**
sub-is-welltyped = *literal.is-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** · *67*)

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

type-synonym (*'f*, *'ty*) *fun-types* = *'f* ⇒ *nat* ⇒ *'ty list* × *'ty*

locale *context-compatible-typing* =
  fixes *Fun typed*
  assumes
   context-compatible [*intro*]:
    $\bigwedge t\ t'\ c\ \tau\ \tau'.$
      typed t τ' ⟹
      typed t' τ' ⟹
      typed (*Fun*⟨*c*; *t*⟩) *τ* ⟹
      typed (*Fun*⟨*c*; *t'*⟩) *τ*

locale *subterm-typing* =
  fixes *Fun typed*
  assumes
    subterm': $\bigwedge$*f ts τ*. *typed* (*Fun f ts*) *τ* ⟹ ∀ *t*∈*set ts*. ∃ *τ'*. *typed t τ'*
begin

lemma *subterm*: *typed* (*Fun*⟨*c*; *t*⟩) *τ* ⟹ ∃ *τ*. *typed t τ*
proof(*induction c arbitrary*: *τ*)
  case *Hole*
  then show *?case*
    by *auto*
next
  case (*More f ss1 c ss2*)

 then have *typed* (*Fun f* (*ss1* @ *Fun*⟨*c*;*t*⟩ # *ss2*)) *τ*
    by *simp*

  then have ∃ *τ*. *typed* (*Fun*⟨*c*;*t*⟩) *τ*
    using *subterm'*
    by *simp*

  then obtain *τ'* **where** *typed* (*Fun*⟨*c*;*t*⟩) *τ'*
    by *blast*

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

end

locale *term-typing* =
  explicit-typing +
  typed: *context-compatible-typing* **where** *typed* = *typed* +
  welltyped: *context-compatible-typing* **where** *typed* = *welltyped* +
  welltyped: *subterm-typing* **where** *typed* = *welltyped* +
assumes *all-terms-are-typed*: $\bigwedge t$. *is-typed t*
begin

sublocale *typed*: *subterm-typing*
  by *unfold-locales* (*auto intro*: *all-terms-are-typed*)

end

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

inductive *typed* **for** $\mathcal{F}$ **where**
  GFun: $\mathcal{F}$ *f* (*length ts*) = (τs, τ) $\Longrightarrow$ *typed* $\mathcal{F}$ (*GFun f ts*) τ

inductive *welltyped* **for** $\mathcal{F}$ **where**
  GFun: $\mathcal{F}$ *f* (*length ts*) = (τs, τ) $\Longrightarrow$ *list-all2* (*welltyped* $\mathcal{F}$) *ts* τs $\Longrightarrow$ *welltyped* $\mathcal{F}$ (*GFun f ts*) τ

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

abbreviation *typed* **where** *typed* $\equiv$ *Ground-Typing.typed* $\mathcal{F}$
abbreviation *welltyped* **where** *welltyped* $\equiv$ *Ground-Typing.welltyped* $\mathcal{F}$

sublocale *explicit-typing* **where** *typed* = *typed* **and** *welltyped* = *welltyped*
proof *unfold-locales*

  show *right-unique typed*
  proof (*rule right-uniqueI*)
    fix *t* τ_1 τ_2

    assume *typed t* τ_1 **and** *typed t* τ_2

    thus $\tau_1 = \tau_2$
      by (*auto elim*!: *typed.cases*)

```
  qed
next

  show right-unique welltyped
  proof (rule right-uniqueI)
    fix t τ1 τ2

    assume welltyped t τ1 and welltyped t τ2

    thus τ1 = τ2
      by (auto elim!: welltyped.cases)
  qed
next
  fix t τ

  assume welltyped t τ

  then show typed t τ
    by (metis typed.intros welltyped.cases)
qed

sublocale term-typing where typed = typed and welltyped = welltyped and Fun
= GFun
proof unfold-locales
  fix t t' c τ τ'

  assume
    t-type: welltyped t τ' and
    t'-type: welltyped t' τ' and
    c-type: welltyped c⟨t⟩G τ

  from c-type show welltyped c⟨t'⟩G 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 (GFun f (ss1 @ c⟨t⟩G # ss2)) τ
      using More.prems
      by simp

    then have welltyped (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 $\mathcal{F}$ *f* (*length* (*ss1* @ $c\langle t'\rangle_G$ # *ss2*)) = (τs, τ)
using *GFun*(*1*)
by *simp*
next

show *list-all2 welltyped* (*ss1* @ $c\langle t'\rangle_G$ # *ss2*) τs
using ‹*list-all2 welltyped* (*ss1* @ $c\langle t\rangle_G$ # *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 *t* t' *c* τ τ'

assume *typed t* τ' *typed* t' τ' *typed* $c\langle t\rangle_G$ τ

then show *typed* $c\langle t'\rangle_G$ τ
by(*induction c arbitrary*: τ) (*auto simp*: *typed.simps*)
next
fix *f ts* τ

assume *welltyped* (*GFun f ts*) τ

then show $\forall$ *t*$\in$*set ts. is-welltyped t*
by (*metis gterm.inject in-set-conv-nth list-all2-conv-all-nth welltyped.simps*)
next
fix *t*

show *is-typed t*
by (*cases t*) (*meson surj-pair typed.intros*)
qed

end

locale *ground-typing* = *term*: *ground-term-typing*
begin

sublocale *clause-typing* **where** *term-typed* = *term.typed* **and** *term-welltyped* = *term.welltyped*
by *unfold-locales*

end

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** · *67*)
notation *subst-apply-term* (**infixl** ·*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 $\bigwedge$*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 $\bigwedge$*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': $t' \notin ts$ *is-Fun* t'
using *Var.prems*(*1*) *finite-list* **by** *blast*

define σ :: ($'f$, $'v$) *subst* **where** $\bigwedge x.\ \sigma\ x = t'$

have *Var x* $\cdot t\ \sigma \neq$ *Var x*
using t'
unfolding σ*-def*
by *auto*

moreover have *Var x* $\cdot t\ \sigma \notin ts$
using t'
unfolding σ*-def*
by *simp*

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

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

then obtain σ **where**
σ: $a \cdot t\ \sigma \neq a$ **and**
a-σ*-not-in-args*: $a \cdot t\ \sigma \notin \bigcup$ (*set* ‘ *term.args* ‘ *ts*)
by (*metis Fun.IH Fun.prems*(*1*) *List.finite-set finite-UN finite-imageI*)

then have *Fun f args* $\cdot t\ \sigma \neq$ *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* $\cdot t\ \sigma \notin ts$
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* ($t \cdot t\ \varrho$) $= \bigcup$ (*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*
$\forall$ *unification*$\in$*unifications*. *finite unification*
finite unifications

show *vars* $(t \cdot t\ \mu) \subseteq$ *vars* $t \cup \bigcup$ (*vars* ‘ $\bigcup$ *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 $\exists$ *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 = ($\odot$) **and** *id-subst* = *Var* **and** *base-subst* = ($\cdot t$) **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* $\Rightarrow$ ($'f$, $'v$) *subst* $\Rightarrow$ ($'f$, $'v$) *context* (**infixl** $\cdot t_c$ *67*) **where**
subst-apply-ctxt $\equiv$ *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 $\exists\, c.\ t \in$ *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 = (·*t*) **and** *sub-vars* = *term.vars* **and**
to-set = *set2-actxt* :: ('*f*, '*v*) *context* ⇒ ('*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
$\bigwedge c\ \sigma.\ subst\ c\ \sigma = c \cdot t_c\ \sigma$ **and**
$\bigwedge c.\ vars\ c = vars\text{-}ctxt\ c$
proof *unfold-locales*
interpret *term-based-lifting* **where**
sub-vars = *term.vars* **and** *sub-subst* = (·*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 $subst\ c\ \sigma = c \cdot t_c\ \sigma$
unfolding *subst-def*
by *blast*
qed

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

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

lemma *term-from-ground-context-from-ground* [*simp*]:
$term.from\text{-}ground\ c_G\langle t_G\rangle_G = (from\text{-}ground\ c_G)\langle term.from\text{-}ground\ t_G\rangle$
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
is-ground c_G
$c_G = (c \cdot t_c\ \sigma) \circ_c c'$
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* = $\lambda l.$ {#*l*#} **and** *plus-ground* = (+) **and** *wrap-ground* =
$\lambda l.$ {#*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* = $\lambda l.$ {#*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) $\longleftrightarrow$ *term.is-ground* t_1 $\wedge$ *term.is-ground* t_2
by *simp*

lemma *obtain-from-atom-subst*:
assumes *Upair* t_1' t_2' = a $\cdot a$ σ
obtains t_1 t_2
where a = *Upair* t_1 t_2 t_1' = t_1 $\cdot t$ σ t_2' = t_2 $\cdot t$ σ
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 $\cdot l$ σ = *Pos* (a $\cdot a$ σ)
Neg a $\cdot l$ σ = *Neg* (a $\cdot a$ σ)
atm-of (l $\cdot l$ σ) = *atm-of* l $\cdot a$ σ
unfolding *literal.subst-def*
using *literal.map-sel*
by *auto*

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

lemma *subst-polarity-stable*:
shows
subst-neg-stable [*simp*]: *is-neg* (l $\cdot l$ σ) $\longleftrightarrow$ *is-neg* l **and**
subst-pos-stable [*simp*]: *is-pos* (l $\cdot l$ σ) $\longleftrightarrow$ *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) ⟷ *is-neg* l_G **and**
pos-literal-from-ground-stable: *is-pos* (*literal.from-ground* l_G) ⟷ *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 ⟷ *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 $l \cdot l\ \sigma = t_1' \mathrel{!\approx} t_2'$
obtains $t_1\ t_2$
where $l = t_1 \mathrel{!\approx} t_2\ t_1\ \cdot t\ \sigma = t_1'\ t_2\ \cdot t\ \sigma = t_2'$
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* $\cdot l\ \sigma$
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* · σ = *remove1-mset* (*l* ·*l* σ) (*C* · σ)
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 $select_G \equiv \forall\ C_G.\ \exists\ C\ \gamma.$
    clause.is-ground $(C \cdot \gamma) \wedge$
    C_G = *clause.to-ground* $(C \cdot \gamma) \wedge$
    $select_G\ C_G$ = *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 $l \in\#$ *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* $C \cdot \gamma$)
using *assms*
by (*metis image-mset-subseteq-mono select-subset sub-ground-clause clause.subst-def*)

lemma *select-neg-subst*:
assumes $l \in\#$ *select* $C \cdot \gamma$
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* $\{p :: 'a \times 'a.\ fst\ p = x\}$
proof −
have $\{p :: 'a \times 'a\ .\ fst\ p = x\} = \{x\} \times UNIV$
by *auto*

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

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

definition *infinite-variables-per-type-on* $:: 'var\ set \Rightarrow ('var \Rightarrow 'ty) \Rightarrow bool$ **where**
infinite-variables-per-type-on $X\ \mathcal{V} \equiv \forall \tau \in \mathcal{V}\ `\ X.\ infinite\ \{x.\ \mathcal{V}\ x = \tau\}$

abbreviation *infinite-variables-per-type* $:: ('var \Rightarrow 'ty) \Rightarrow bool$ **where**
infinite-variables-per-type $\equiv$ *infinite-variables-per-type-on UNIV*

lemma *obtain-type-preserving-inj*:
fixes $\mathcal{V} :: 'v \Rightarrow 'ty$
assumes
finite-X: *finite* X **and**
$\mathcal{V}$: *infinite-variables-per-type* $\mathcal{V}$
obtains $f :: 'v \Rightarrow 'v$ **where**
inj f
$X \cap f\ `\ Y = \{\}$
$\forall x \in Y.\ \mathcal{V}\ (f\ x) = \mathcal{V}\ x$
proof (*rule that*)

{
fix τ
assume $\tau \in range\ \mathcal{V}$

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

then have $|\{x.\ \mathcal{V}\ x = \tau\}| =o\ |\{x.\ \mathcal{V}\ x = \tau \wedge x \notin X\}|$
using *set-diff-eq*[*of* - *X*]
by *auto*

then have $\exists g.\ bij\text{-}betw\ g\ \{x.\ \mathcal{V}\ x = \tau\}\ \{x.\ \mathcal{V}\ x = \tau \wedge x \notin X\}$
using *card-of-ordIso someI*
by *blast*
}
note *exists-g* = *this*

define *get-g* **where**
$\bigwedge\tau.\ get\text{-}g\ \tau \equiv SOME\ g.\ bij\text{-}betw\ g\ \{x.\ \mathcal{V}\ x = \tau\}\ \{x.\ \mathcal{V}\ x = \tau \wedge x \notin X\}$

define *f* **where**
$\bigwedge x.\ f\ x \equiv get\text{-}g\ (\mathcal{V}\ x)\ x$

{
fix *y*

have $\bigwedge g.\ bij\text{-}betw\ g\ \{x.\ \mathcal{V}\ x = \mathcal{V}\ y\}\ \{x.\ \mathcal{V}\ x = \mathcal{V}\ y \wedge x \notin X\} \Longrightarrow g\ y \in \{x.\ \mathcal{V}\ x = \mathcal{V}\ y \wedge x \notin X\}$
using *exists-g bij-betwE*
by *blast*

then have $f\ y \in \{x.\ \mathcal{V}\ x = \mathcal{V}\ y \wedge x \notin X\}$
using *exists-g get-g-def*
unfolding *f-def*
by (*metis* (*no-types*, *lifting*) *ext rangeI verit-sko-ex'*)
}

then show $X \cap f\ `\ Y = \{\}$ $\ \forall y{\in}Y.\ \mathcal{V}\ (f\ y) = \mathcal{V}\ y$
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 $\mathcal{V}_1\ \mathcal{V}_2 :: 'v \Rightarrow 'ty$

assumes
  finite-X: *finite* X **and**
  $\mathcal{V}_2$: *infinite-variables-per-type* $\mathcal{V}_2$
obtains $f\, f' :: 'v \Rightarrow 'v$ **where**
  inj f *inj* f'
  f ‘ $X \cap f'$ ‘ $Y = \{\}$
  $\forall x \in X.\ \mathcal{V}_1\ (f\ x) = \mathcal{V}_1\ x$
  $\forall x \in Y.\ \mathcal{V}_2\ (f'\ x) = \mathcal{V}_2\ x$
proof−

  obtain f' **where** f':
    inj f'
    $X \cap f'$ ‘ $Y = \{\}$
    $\forall x \in Y.\ \mathcal{V}_2\ (f'\ x) = \mathcal{V}_2\ x$
    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 $\mathcal{V}_1\ \mathcal{V}_2 :: 'v \Rightarrow 'ty$
  assumes
    finite-Y: *finite* Y **and**
    $\mathcal{V}_1$: *infinite-variables-per-type* $\mathcal{V}_1$
  obtains $f\, f' :: 'v \Rightarrow 'v$ **where**
    inj f *inj* f'
    f ‘ $X \cap f'$ ‘ $Y = \{\}$
    $\forall x \in X.\ \mathcal{V}_1\ (f\ x) = \mathcal{V}_1\ x$
    $\forall x \in Y.\ \mathcal{V}_2\ (f'\ x) = \mathcal{V}_2\ x$
  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: $Y \cap Z = \{\}$
  obtains $\mathcal{V} :: 'v \Rightarrow 'ty$
  where *infinite-variables-per-type-on* $X\ \mathcal{V}$ $\forall x \in Y.\ \mathcal{V}\ x = \mathcal{V}'\ x$ $\forall x \in Z.\ \mathcal{V}\ x =$
$\mathcal{V}''\ x$
proof (*cases* $X = \{\}$)
  case *True*
  define $\mathcal{V}$ **where** $\bigwedge x.\ \mathcal{V}\ x \equiv$ *if* $x \in Y$ *then* $\mathcal{V}'\ x$ *else* $\mathcal{V}''\ x$

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

show $\forall x \in Y.\ \mathcal{V}\ x = \mathcal{V}'\ x$
  unfolding *$\mathcal{V}$-def*
  by *simp*
next

show $\forall x \in Z.\ \mathcal{V}\ x = \mathcal{V}''\ x$
  using *disjoint*
  unfolding *$\mathcal{V}$-def*
  by *auto*
qed (*auto simp*: *infinite-variables-per-type-on-def*)
next
case *False*

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**
  $\bigwedge x.\ f\ x \equiv \mathit{if}\ x \in Y \cup Z\ \mathit{then}\ x\ \mathit{else}\ \mathit{fst}\ (g\ x)$

define $\mathcal{V}$ **where** $\bigwedge x.\ \mathcal{V}\ x \equiv \mathit{if}\ x \in Y\ \mathit{then}\ \mathcal{V}'\ x\ \mathit{else}\ \mathcal{V}''\ x$

{
  fix *y*

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

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

  then have *infinite* $\{x.\ x \notin Y \cup Z \wedge \mathit{fst}\ (g\ x) = y\}$
   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 *$\mathcal{V}$-X*: $\forall y \in \mathcal{V}\ `\ f\ `\ X.\ \mathit{infinite}\ \{x.\ \mathcal{V}\ (f\ x) = y\}$
  by (*smt* (*verit*, *best*) *Collect-mono imageE rev-finite-subset*)

show *?thesis*
proof (*rule that*)
  show *infinite-variables-per-type-on* $X\ (\mathcal{V} \circ f)$

using $\mathcal{V}$*-X*
unfolding *infinite-variables-per-type-on-def comp-def*
by (*metis image-image*)
next
show $\forall x \in Y.\ (\mathcal{V} \circ f)\ x = \mathcal{V}'\ x$
unfolding *f-def* $\mathcal{V}$*-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* ⇒ *'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* ⇒ *'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* ⇒ *'ty*) ⇒ *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*| ≤*o* |*UNIV* :: (*'v* :: *infinite*) *set*|
shows $\exists \mathcal{V}$ :: *'v* ⇒ *'ty*. *infinite-variables-for-all-types* $\mathcal{V}$
proof−
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**
$\bigwedge x.\ f\ x \equiv$ *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 $('var, 'ty)\ var\text{-}types = 'var \Rightarrow 'ty$

locale *explicitly-typed-functional-substitution* =
base-functional-substitution **where** *vars* = *vars* **and** *id-subst* = *id-subst*
for
$id\text{-}subst :: 'var \Rightarrow 'base$ **and**

vars :: *'base* ⇒ *'var set* **and**
typed :: (*'var*, *'ty*) *var-types* ⇒ *'base* ⇒ *'ty* ⇒ *bool* +
assumes
predicate-typed: $\bigwedge\mathcal{V}$. *predicate-typed* (*typed* $\mathcal{V}$) **and**
typed-id-subst [*intro*]: $\bigwedge\mathcal{V}$ *x*. *typed* $\mathcal{V}$ (*id-subst x*) ($\mathcal{V}$ *x*)
begin

sublocale *predicate-typed typed* $\mathcal{V}$
using *predicate-typed* **.**

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

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

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

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

end

locale *inhabited-explicitly-typed-functional-substitution* =
explicitly-typed-functional-substitution +
assumes *types-inhabited*: $\bigwedge\mathcal{V}$ τ. $\exists$ *b*. *is-ground b* $\wedge$ *typed* $\mathcal{V}$ *b* τ

locale *typed-functional-substitution* =
base: *explicitly-typed-functional-substitution* **where**
vars = *base-vars* **and** *subst* = *base-subst* **and** *typed* = *base-typed* +
based-functional-substitution **where** *vars* = *vars*
for
vars :: *'expr* ⇒ *'var set* **and**
is-typed :: (*'var*, *'ty*) *var-types* ⇒ *'expr* ⇒ *bool* **and**
base-typed :: (*'var*, *'ty*) *var-types* ⇒ *'base* ⇒ *'ty* ⇒ *bool*
begin

abbreviation *is-typed-ground-instance* **where**
is-typed-ground-instance expr $\mathcal{V}$ γ ≡

is-ground (*expr* · γ) ∧
is-typed $\mathcal{V}$ *expr* ∧
base.is-typed-on (*vars expr*) $\mathcal{V}$ γ ∧
infinite-variables-per-type $\mathcal{V}$

end

sublocale *explicitly-typed-functional-substitution* ⊆ *typed-functional-substitution* **where**
base-subst = *subst* **and** *base-vars* = *vars* **and** *is-typed* = *is-typed* **and**
base-typed = *typed*
by *unfold-locales*

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

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

lemma *typed-ground-instances-ground-instances*′:
typed-ground-instances (*expr*, $\mathcal{V}$) ⊆ *ground-instances*′ *expr*
unfolding *typed-ground-instances-def ground-instances*′*-def*
by *auto*

end

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

sublocale *typed-grounding-functional-substitution* **where**
base-subst = *subst* **and** *base-vars* = *vars* **and** *is-typed* = *is-typed* **and**
base-typed = *typed*
by *unfold-locales*

end

locale *inhabited-typed-functional-substitution* =
typed-functional-substitution +
base: *inhabited-explicitly-typed-functional-substitution* **where**
subst = *base-subst* **and** *vars* = *base-vars* **and** *typed* = *base-typed*
begin

lemma *ground-subst-extension*:
assumes
grounding: *is-ground* (*expr* · γ) **and**
γ*-is-typed-on*: *base.is-typed-on* (*vars expr*) $\mathcal{V}$ γ

```
  obtains γ'
  where
    base.is-ground-subst γ'
    base.is-typed-on UNIV 𝒱 γ'
    ∀ x ∈ vars expr. γ x = γ' x
proof (rule that)

  define γ' where
    ⋀x. γ' x ≡
      if x ∈ vars expr
      then γ x
      else SOME base. base.is-ground base ∧ base-typed 𝒱 base (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 (γ' x)
      proof(cases x ∈ 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) base.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-typed-on UNIV 𝒱 γ'
    unfolding γ'-def
    using γ-is-typed-on base.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* $(expr \cdot \gamma)$ **and**
    γ-is-typed-on: *base.is-typed-on* $(vars\ expr)\ \mathcal{V}\ \gamma$
  obtains γ'
  where
    is-ground $(expr' \cdot \gamma')$
    base.is-typed-on $(vars\ expr')\ \mathcal{V}\ \gamma'$
    $\forall x \in vars\ expr.\ \gamma\ x = \gamma'\ x$
  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 *explicitly-typed-functional-substitution* ⊆ *typed-functional-substitution* **where**
  base-subst = *subst* **and** *base-vars* = *vars* **and** *is-typed* = *is-typed* **and**
  base-typed = *typed*
  by *unfold-locales*

locale *typed-subst-stability* = *typed-functional-substitution* +
assumes
  subst-stability [*simp*]:
    $\bigwedge\mathcal{V}\ expr\ \sigma.\ base.is\text{-}typed\text{-}on\ (vars\ expr)\ \mathcal{V}\ \sigma \Longrightarrow is\text{-}typed\ \mathcal{V}\ (expr \cdot \sigma) \longleftrightarrow$
is-typed $\mathcal{V}$ *expr*
begin

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

end

locale *explicitly-typed-subst-stability* = *explicitly-typed-functional-substitution* +
assumes
  explicit-subst-stability [*simp*]:
    $\bigwedge\mathcal{V}\ expr\ \sigma\ \tau.\ is\text{-}typed\text{-}on\ (vars\ expr)\ \mathcal{V}\ \sigma \Longrightarrow typed\ \mathcal{V}\ (expr \cdot \sigma)\ \tau \longleftrightarrow typed$
$\mathcal{V}\ expr\ \tau$
begin

lemma *explicit-subst-stability-UNIV* [*simp*]:
  $\bigwedge\mathcal{V}\ expr\ \sigma.\ is\text{-}typed\text{-}on\ UNIV\ \mathcal{V}\ \sigma \Longrightarrow typed\ \mathcal{V}\ (expr \cdot \sigma)\ \tau \longleftrightarrow typed\ \mathcal{V}\ expr\ \tau$
  by *simp*

sublocale *typed-subst-stability* **where**
  base-vars = *vars* **and** *base-subst* = *subst* **and** *base-typed* = *typed* **and** *is-typed* =

is-typed
using *explicit-subst-stability*
by *unfold-locales blast*

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

lemma *typed-subst-compose-UNIV* [*intro*]:
assumes
is-typed-on UNIV $\mathcal{V}$ σ
is-typed-on UNIV $\mathcal{V}$ σ'
shows *is-typed-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* $\mathcal{V}$ $\mathcal{V}'$. $\forall x\in$ *vars expr*. $\mathcal{V}$ x = $\mathcal{V}'$ x $\Longrightarrow$ *is-typed* $\mathcal{V}$ *expr* $\Longrightarrow$ *is-typed* $\mathcal{V}'$ *expr*
begin

lemma *replace-*$\mathcal{V}$*-iff*:
assumes $\forall x\in$ *vars expr*. $\mathcal{V}$ x = $\mathcal{V}'$ x
shows *is-typed* $\mathcal{V}$ *expr* $\longleftrightarrow$ *is-typed* $\mathcal{V}'$ *expr*
using *assms*
by (*metis replace-*$\mathcal{V}$)

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

end

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

lemma *explicit-replace-$\mathcal{V}$-iff*:
  assumes $\forall x\in$ *vars expr*. $\mathcal{V}$ $x = \mathcal{V}'$ x
  shows *typed* $\mathcal{V}$ *expr* $\tau \longleftrightarrow$ *typed* $\mathcal{V}'$ *expr* τ
  using *assms*
  by (*metis explicit-replace-$\mathcal{V}$*)

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

sublocale *replaceable-$\mathcal{V}$* **where**
  base-vars = *vars* **and** *base-subst* = *subst* **and** *base-typed* = *typed* **and** *is-typed* = *is-typed*
  using *explicit-replace-$\mathcal{V}$*
  by *unfold-locales blast*

end

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

locale *explicitly-typed-renaming* =
  explicitly-typed-functional-substitution **where** *typed* = *typed* +
  renaming-variables +
  explicitly-replaceable-$\mathcal{V}$ **where** *typed* = *typed*
for *typed* :: ('*var* $\Rightarrow$ '*ty*) $\Rightarrow$ '*expr* $\Rightarrow$ '*ty* $\Rightarrow$ *bool* +
assumes
  explicit-typed-renaming [*simp*]:
  $\bigwedge\mathcal{V}\ \mathcal{V}'$ *expr* ϱ τ. *is-renaming* $\varrho \Longrightarrow$
    $\forall x \in$ *vars expr*. $\mathcal{V}$ $x = \mathcal{V}'$ (*rename* ϱ x) $\Longrightarrow$
    typed $\mathcal{V}'$ (*expr* $\cdot$ ϱ) $\tau \longleftrightarrow$ *typed* $\mathcal{V}$ *expr* τ
begin

sublocale *typed-renaming*
  where *base-vars* = *vars* **and** *base-subst* = *subst* **and** *base-typed* = *typed* **and** *is-typed* = *is-typed*
  using *explicit-typed-renaming*
  by *unfold-locales blast*

lemma *renaming-ground-subst*:
  assumes
    is-renaming ϱ

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

then have *typed* $\mathcal{V}\ (\varrho\ x)\ (\mathcal{V}\ x)$
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 *typed* $\mathcal{V}\ (\gamma\ y)\ (\mathcal{V}'\ y)$
using *explicit-replace-*$\mathcal{V}$
by (*metis assms*(*2*,*4*) *left-neutral emptyE is-ground-subst-is-ground comp-subst-iff*)

ultimately have *typed* $\mathcal{V}\ (\gamma\ y)\ (\mathcal{V}\ x)$
unfolding *y-def*
using *assms*(*5*) *x-in-X*
by *fastforce*

moreover have $\gamma\ y = (\varrho \odot \gamma)\ x$
unfolding *y-def*
by (*metis assms*(*1*) *comp-subst-iff id-subst-rename left-neutral*)

ultimately show *typed* $\mathcal{V}\ ((\varrho \odot \gamma)\ x)\ (\mathcal{V}\ x)$
by *argo*
qed

lemma *inj-id-subst*: *inj id-subst*
using *is-renaming-id-subst is-renaming-iff*
by *blast*

lemma *obtain-typed-renaming*:
fixes $\mathcal{V} :: 'var \Rightarrow 'ty$
assumes
finite X
infinite-variables-per-type $\mathcal{V}$
obtains $\varrho :: 'var \Rightarrow 'expr$ **where**
is-renaming ϱ
id-subst $`\ X \cap \varrho\ `\ Y = \{\}$
is-typed-on $Y\ \mathcal{V}\ \varrho$

proof −

obtain *renaming* :: *'var* ⇒ *'var* **where**
inj: *inj renaming* **and**
rename-apart: $X \cap$ *renaming* ‘ $Y = \{\}$ **and**
preserve-type: $\forall x \in Y.\ \mathcal{V}$ (*renaming x*) $= \mathcal{V}\ x$
using *obtain-type-preserving-inj*[*OF assms*]**.**

define ϱ :: *'var* ⇒ *'expr* **where**
$\bigwedge x.\ \varrho\ x \equiv$ *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* ‘ $X \cap \varrho$ ‘ $Y = \{\}$
using *rename-apart inj-id-subst*
unfolding *ϱ-def inj-def*
by *blast*
next

show *is-typed-on* $Y\ \mathcal{V}\ \varrho$
using *preserve-type*
unfolding *ϱ-def*
by (*metis typed-id-subst*)
qed
qed

lemma *obtain-typed-renamings*:
fixes $\mathcal{V}_1\ \mathcal{V}_2$ :: *'var* ⇒ *'ty*
assumes
finite X
infinite-variables-per-type $\mathcal{V}_2$
obtains $\varrho_1\ \varrho_2$ :: *'var* ⇒ *'expr* **where**
is-renaming ϱ_1
is-renaming ϱ_2
ϱ_1 ‘ $X \cap \varrho_2$ ‘ $Y = \{\}$
is-typed-on $X\ \mathcal{V}_1\ \varrho_1$
is-typed-on $Y\ \mathcal{V}_2\ \varrho_2$
using *obtain-typed-renaming*[*OF assms*] *is-renaming-id-subst typed-id-subst*
by *metis*

lemma *obtain-typed-renamings'*:
fixes $\mathcal{V}_1\ \mathcal{V}_2$ :: *'var* ⇒ *'ty*

assumes
finite Y
infinite-variables-per-type $\mathcal{V}_1$
obtains $\varrho_1\ \varrho_2$ :: *'var ⇒ 'expr* **where**
is-renaming ϱ_1
is-renaming ϱ_2
ϱ_1 ‘ $X \cap \varrho_2$ ‘ $Y = \{\}$
is-typed-on X $\mathcal{V}_1\ \varrho_1$
is-typed-on Y $\mathcal{V}_2\ \varrho_2$
using *obtain-typed-renamings*[*OF assms*]
by (*metis inf-commute*)

lemma *renaming-subst-compose*:
assumes
is-renaming ϱ
is-typed-on X $\mathcal{V}$ ($\varrho \odot \sigma$)
is-typed-on X $\mathcal{V}\ \varrho$
shows *is-typed-on* ($\bigcup$ (*vars* ‘ ϱ ‘ *X*)) $\mathcal{V}\ \sigma$
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* · ϱ_1) ∩ *vars* (*expr'* · 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 $\mathcal{V}_3$
∀ *x*∈*vars expr*. $\mathcal{V}_1\ x = \mathcal{V}_3$ (*rename* ϱ_1 *x*)
∀ *x*∈*vars expr'*. $\mathcal{V}_2\ x = \mathcal{V}_3$ (*rename* ϱ_2 *x*)
proof−

have *finite*: *finite* (*vars* (*expr* · ϱ_1)) *finite* (*vars* (*expr'* · ϱ_2))
using *finite-vars*
by (*simp-all add*: $\varrho_1\ \varrho_2$ *rename-variables*)

obtain $\mathcal{V}_3$ **where**
$\mathcal{V}_3$: *infinite-variables-per-type-on X* $\mathcal{V}_3$ **and**
$\mathcal{V}_3$-$\mathcal{V}_1$: ∀ *x*∈*vars* (*expr* · ϱ_1). $\mathcal{V}_3\ x = \mathcal{V}_1$ (*inv* ϱ_1 (*id-subst x*)) **and**
$\mathcal{V}_3$-$\mathcal{V}_2$: ∀ *x*∈*vars* (*expr'* · ϱ_2). $\mathcal{V}_3\ x = \mathcal{V}_2$ (*inv* ϱ_2 (*id-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 ∀ *x*∈*vars expr*. $\mathcal{V}_1$ *x* = $\mathcal{V}_3$ (*rename* ϱ_1 *x*)
using $\mathcal{V}_3$-$\mathcal{V}_1$ ϱ_1 *inv-renaming rename-variables*
by *auto*
next

show ∀ *x*∈*vars expr'*. $\mathcal{V}_2$ *x* = $\mathcal{V}_3$ (*rename* ϱ_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* · ϱ_1) ∩ *vars* (*expr'* · ϱ_2) = {} **and**
$\mathcal{V}_2$: *infinite-variables-for-all-types* $\mathcal{V}_2$ **and**
finite-vars: *finite* (*vars expr*)
obtains $\mathcal{V}_3$ **where**
∀ *x*∈*vars expr*. $\mathcal{V}_1$ *x* = $\mathcal{V}_3$ (*rename* ϱ_1 *x*)
∀ *x*∈*vars expr'*. $\mathcal{V}_2$ *x* = $\mathcal{V}_3$ (*rename* ϱ_2 *x*)
infinite-variables-for-all-types $\mathcal{V}_3$
proof (*rule that*)

define $\mathcal{V}_3$ **where**
⋀*x*. $\mathcal{V}_3$ *x* ≡
if x ∈ *vars* (*expr* · ϱ_1)
then $\mathcal{V}_1$ (*inv* ϱ_1 (*id-subst x*))
else $\mathcal{V}_2$ (*inv* ϱ_2 (*id-subst x*))

show ∀ *x*∈*vars expr*. $\mathcal{V}_1$ *x* = $\mathcal{V}_3$ (*rename* ϱ_1 *x*)
proof (*intro ballI*)
fix *x*
assume *x*∈*vars expr*

then have *rename* ϱ_1 *x* ∈ *vars* (*expr* · ϱ_1)
using *rename-variables*[*OF* ϱ_1]
by *blast*

then show $\mathcal{V}_1$ *x* = $\mathcal{V}_3$ (*rename* ϱ_1 *x*)
unfolding $\mathcal{V}_3$*-def*
by (*simp add*: ϱ_1 *inv-renaming*)
qed

show ∀ *x*∈*vars expr'*. $\mathcal{V}_2$ *x* = $\mathcal{V}_3$ (*rename* ϱ_2 *x*)
proof (*intro ballI*)
fix *x*

assume $x \in$ *vars expr'*

then have *rename* ϱ_2 $x \in$ *vars* (*expr'* $\cdot$ ϱ_2)
using *rename-variables*[*OF* ϱ_2]
by *blast*

then show $\mathcal{V}_2$ x = $\mathcal{V}_3$ (*rename* ϱ_2 x)
unfolding $\mathcal{V}_3$*-def*
using ϱ_2 *inv-renaming rename-apart*
by (*metis* (*mono-tags*, *lifting*) *disjoint-iff id-subst-rename*)
qed

have *finite* {*x*. $x \in$ *vars* (*expr* $\cdot$ ϱ_1)}
using *finite-vars*
by (*simp add*: ϱ_1 *rename-variables*)

moreover **{**
fix τ

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

show *infinite* {*x*. $\mathcal{V}_2$ x = τ}
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* $\cdot$ ϱ_1) $\cap$ *vars* (*expr'* $\cdot$ ϱ_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* ϱ_1 x)
$\forall x \in$ *vars expr'*. $\mathcal{V}_2$ x = $\mathcal{V}_3$ (*rename* ϱ_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: explicitly-typed-renaming where
  subst = base-subst and vars = base-vars :: 'base ⇒ 'v set and
  typed = typed :: ('v ⇒ 'ty) ⇒ 'base ⇒ 'ty ⇒ bool +
  base: explicitly-typed-functional-substitution where
  vars = base-vars and subst = base-subst +
  based-functional-substitution +
  renaming-variables
begin

lemma renaming-grounding:
  assumes
    renaming: base.is-renaming ϱ and
    ϱ-γ-is-welltyped: base.is-typed-on (vars expr) 𝒱 (ϱ ⊙ γ) and
    grounding: is-ground (expr · ϱ ⊙ γ) and
    𝒱-𝒱': ∀ x ∈ vars expr. 𝒱 x = 𝒱' (rename ϱ x)
  shows base.is-typed-on (vars (expr · ϱ)) 𝒱' γ
proof(intro ballI)
  fix x

  define y where y ≡ inv ϱ (id-subst x)

  assume x-in-expr: x ∈ vars (expr · ϱ)

  then have y-in-vars: y ∈ 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-subst (id-subst y) (ϱ ⊙ γ))
    using variable-grounding[OF grounding y-in-vars]
    by (metis base.comp-subst-iff base.left-neutral)

  moreover have typed 𝒱 (base-subst (id-subst y) (ϱ ⊙ γ)) (𝒱 y)
    using ϱ-γ-is-welltyped y-in-vars
    unfolding y-def
    by (metis base.comp-subst-iff base.left-neutral)

  ultimately have typed 𝒱' (base-subst (id-subst y) (ϱ ⊙ γ)) (𝒱 y)
    by (meson base.explicit-is-ground-typed)

  moreover have base-subst (id-subst y) (ϱ ⊙ γ) = γ 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 typed 𝒱' (γ x) (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-typed-on (*vars expr*) $\mathcal{V}_1$ γ_1
base.is-typed-on (*vars expr'*) $\mathcal{V}_2$ γ_2
is-ground (*expr* $\cdot$ γ_1)
is-ground (*expr'* $\cdot$ γ_2) **and**
$\mathcal{V}_2$: *infinite-variables-per-type* $\mathcal{V}_2$ **and**
finite-vars: *finite* (*vars expr*)
obtains ϱ_1 ϱ_2 γ **where**
base.is-renaming ϱ_1
base.is-renaming ϱ_2
vars (*expr* $\cdot$ ϱ_1) $\cap$ *vars* (*expr'* $\cdot$ ϱ_2) = {}
base.is-typed-on (*vars expr*) $\mathcal{V}_1$ ϱ_1
base.is-typed-on (*vars expr'*) $\mathcal{V}_2$ ϱ_2
$\forall x \in$ *vars expr*. γ_1 x = ($\varrho_1 \odot \gamma$) x
$\forall x \in$ *vars expr'*. γ_2 x = ($\varrho_2 \odot \gamma$) x
proof−

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

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

from ϱ_1 ϱ_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* $\cdot$ ϱ_1)
then (ϱ_1-*inv* $\odot$ γ_1) x
else (ϱ_2-*inv* $\odot$ γ_2) x

show *?thesis*
proof(*rule that*[*OF* ϱ_1 ϱ_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-subst* y
using *obtain-renamed-variable*[*OF* ϱ_1]**.**

then have $y \in$ *vars* (*expr* $\cdot$ ϱ_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-subst* (ϱ_1*-inv* y) γ_1
unfolding γ*-def*
using *base.comp-subst-iff*
by *presburger*

then show $\gamma_1\ x = (\varrho_1 \odot \gamma)\ x$
by (*metis* ϱ_1*-inv base.comp-subst-iff base.left-neutral* y)
qed

then show $\forall x \in$ *vars expr*. $\gamma_1\ x = (\varrho_1 \odot \gamma)\ x$
by *auto*

next

have $\forall x \in$ *vars expr'*. $\gamma_2\ x = (\varrho_2 \odot \gamma)\ x$
proof(*intro ballI*)
fix x
assume *x-in-vars*: $x \in$ *vars expr'*

obtain y **where** y: $\varrho_2\ x =$ *id-subst* y
using *obtain-renamed-variable*[*OF* ϱ_2]**.**

then have $y \in$ *vars* (*expr'* $\cdot$ ϱ_2)
using *x-in-vars* ϱ_2 *rename-variables-id-subst*
by (*metis base.inj-id-subst imageI inj-image-mem-iff*)

then have $\gamma\ y =$ *base-subst* (ϱ_2*-inv* y) γ_2
unfolding γ*-def*
using *base.comp-subst-iff rename-apart*
by *auto*

then show $\gamma_2\ x = (\varrho_2 \odot \gamma)\ x$
by (*metis* ϱ_2*-inv base.comp-subst-iff base.left-neutral* y)

qed

then show $\forall x \in$ *vars expr′*. $\gamma_2\ x = (\varrho_2 \odot \gamma)\ x$
by *auto*
qed
qed

lemma *obtain-merged-grounding′*:
fixes $\mathcal{V}_1\ \mathcal{V}_2$ :: *′v* ⇒ *′ty*
assumes
typed-γ_1: *base.is-typed-on* (*vars expr*) $\mathcal{V}_1\ \gamma_1$ **and**
typed-γ_2: *base.is-typed-on* (*vars expr′*) $\mathcal{V}_2\ \gamma_2$ **and**
expr-grounding: *is-ground* (*expr* · γ_1) **and**
expr′-grounding: *is-ground* (*expr′* · γ_2) **and**
$\mathcal{V}_1$: *infinite-variables-per-type* $\mathcal{V}_1$ **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* · ϱ_1) ∩ *vars* (*expr′* · ϱ_2) = {}
base.is-typed-on (*vars expr*) $\mathcal{V}_1\ \varrho_1$
base.is-typed-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′*. $\gamma_2\ x = (\varrho_2 \odot \gamma)\ x$
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 *explicitly-typed-renaming* ⊆
based-typed-renaming **where** *base-vars* = *vars* **and** *base-subst* = *subst*
by *unfold-locales*

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

locale *subst-is-typed-abbreviations* =
is-typed: *typed-functional-substitution* **where**
base-typed = *base-typed* **and** *is-typed* = *expr-is-typed* +
is-welltyped: *typed-functional-substitution* **where**
base-typed = *base-welltyped* **and** *is-typed* = *expr-is-welltyped*
for
base-typed base-welltyped :: (*′var*, *′ty*) *var-types* ⇒ *′base* ⇒ *′ty* ⇒ *bool* **and**
expr-is-typed expr-is-welltyped :: (*′var*, *′ty*) *var-types* ⇒ *′expr* ⇒ *bool*
begin

abbreviation *is-typed-on* **where**
is-typed-on ≡ *is-typed.base.is-typed-on*

abbreviation *is-welltyped-on* **where**
is-welltyped-on ≡ *is-welltyped.base.is-typed-on*

abbreviation *is-typed* **where**
is-typed ≡ *is-typed.base.is-typed-on UNIV*

abbreviation *is-welltyped* **where**
is-welltyped ≡ *is-welltyped.base.is-typed-on UNIV*

end

locale *functional-substitution-typing* =
is-typed: *typed-functional-substitution* **where**
base-typed = *base-typed* **and** *is-typed* = *is-typed* +
is-welltyped: *typed-functional-substitution* **where**
base-typed = *base-welltyped* **and** *is-typed* = *is-welltyped*
for
base-typed base-welltyped :: ('*var*, '*ty*) *var-types* ⇒ '*base* ⇒ '*ty* ⇒ *bool* **and**
is-typed is-welltyped :: ('*var*, '*ty*) *var-types* ⇒ '*expr* ⇒ *bool* +
assumes *typing*: ⋀𝒱. *typing* (*is-typed* 𝒱) (*is-welltyped* 𝒱)
begin

sublocale *base*: *typing is-typed* 𝒱 *is-welltyped* 𝒱
by (*rule typing*)

sublocale *subst*: *subst-is-typed-abbreviations*
where *expr-is-typed* = *is-typed* **and** *expr-is-welltyped* = *is-welltyped*
by *unfold-locales*

end

locale *base-functional-substitution-typing* =
typed: *explicitly-typed-functional-substitution* **where** *typed* = *typed* +
welltyped: *explicitly-typed-functional-substitution* **where** *typed* = *welltyped*
for
welltyped typed :: ('*var*, '*ty*) *var-types* ⇒ '*expr* ⇒ '*ty* ⇒ *bool* +
assumes
explicit-typing: ⋀𝒱. *explicit-typing* (*typed* 𝒱) (*welltyped* 𝒱)
begin

sublocale *base*: *explicit-typing typed* 𝒱 *welltyped* 𝒱
using *explicit-typing* **.**

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

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

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

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

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

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

sublocale *functional-substitution-typing* **where**
is-typed = *base.is-typed* **and** *is-welltyped* = *base.is-welltyped* **and** *base-typed* = *typed* **and**
base-welltyped = *welltyped* **and** *base-vars* = *vars* **and** *base-subst* = *subst*
by *unfold-locales*

sublocale *subst*: *typing subst.is-typed-on X $\mathcal{V}$ subst.is-welltyped-on X $\mathcal{V}$*
using *base.typed-if-welltyped*
by *unfold-locales blast*

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

abbreviation (*input*) *lifted-is-typed* **where**
lifted-is-typed $\mathcal{V}$ ≡ *is-typed-lifting to-set* (*sub-is-typed* $\mathcal{V}$)

lemmas *lifted-is-typed-def* = *is-typed-lifting-def*[*of to-set*, *THEN ext-equiv*, *of sub-is-typed*]

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

end

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

abbreviation (*input*) *lifted-is-typed* **where**
lifted-is-typed $\mathcal{V}$ ≡ *uniform-typed-lifting to-set* (*base-typed* $\mathcal{V}$)

lemmas *lifted-is-typed-def* = *uniform-typed-lifting-def*[*of to-set*, *THEN ext-equiv*, *of base-typed*]

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

end

locale *uniform-typed-grounding-functional-substitution-lifting* =
uniform-typed-functional-substitution-lifting +
grounding-lifting **where** *sub-subst* = *base-subst* **and** *sub-vars* = *base-vars* +
base: *explicitly-typed-grounding-functional-substitution* **where**
vars = *base-vars* **and** *subst* = *base-subst* **and** *typed* = *base-typed* **and**
to-ground = *sub-to-ground* **and** *from-ground* = *sub-from-ground*
begin

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

end

locale *typed-grounding-functional-substitution-lifting* =
typed-functional-substitution-lifting +
grounding-lifting +
sub: *typed-grounding-functional-substitution* **where**
vars = *sub-vars* **and** *subst* = *sub-subst* **and** *is-typed* = *sub-is-typed* **and**

to-ground = *sub-to-ground* **and** *from-ground* = *sub-from-ground*
begin

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

end

locale *uniform-inhabited-typed-functional-substitution-lifting* =
uniform-typed-functional-substitution-lifting +
base: *inhabited-explicitly-typed-functional-substitution* **where**
vars = *base-vars* **and** *subst* = *base-subst* **and** *typed* = *base-typed*
begin

sublocale *inhabited-typed-functional-substitution* **where**
vars = *vars* **and** *subst* = *subst* **and** *is-typed* = *lifted-is-typed*
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** *is-typed* = *sub-is-typed*
begin

sublocale *inhabited-typed-functional-substitution* **where**
vars = *vars* **and** *subst* = *subst* **and** *is-typed* = *lifted-is-typed*
by *unfold-locales*

end

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

sublocale *typed-subst-stability* **where**
is-typed = *lifted-is-typed* **and** *subst* = *subst* **and** *vars* = *vars*
proof *unfold-locales*
fix *expr* $\mathcal{V}$ σ
assume *sub.base.is-typed-on* (*vars expr*) $\mathcal{V}$ σ

then show *lifted-is-typed* $\mathcal{V}$ (*expr* $\cdot$ σ) $\longleftrightarrow$ *lifted-is-typed* $\mathcal{V}$ *expr*
unfolding *vars-def is-typed-lifting-def*

using *sub.subst-stability to-set-image*
by *fastforce*

qed

end

locale *uniform-typed-subst-stability-lifting* =
uniform-typed-functional-substitution-lifting +
base: *explicitly-typed-subst-stability* **where**
typed = *base-typed* **and** *vars* = *base-vars* **and** *subst* = *base-subst*
begin

sublocale *typed-subst-stability* **where**
is-typed = *lifted-is-typed* **and** *subst* = *subst* **and** *vars* = *vars*
proof *unfold-locales*
fix *expr* $\mathcal{V}$ σ
assume *base.is-typed-on* (*vars expr*) $\mathcal{V}$ σ

then show *lifted-is-typed* $\mathcal{V}$ (*subst expr* σ) $\longleftrightarrow$ *lifted-is-typed* $\mathcal{V}$ *expr*
unfolding *vars-def uniform-typed-lifting-def*
using *base.subst-stability to-set-image*
by *force*
qed

end

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

sublocale *replaceable-*$\mathcal{V}$ **where**
subst = *subst* **and** *vars* = *vars* **and** *is-typed* = *lifted-is-typed*
by *unfold-locales* (*auto simp*: *sub.replace-*$\mathcal{V}$ *vars-def is-typed-lifting-def*)

end

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

sublocale *replaceable-*$\mathcal{V}$ **where**
is-typed = *lifted-is-typed* **and** *subst* = *subst* **and** *vars* = *vars*
by
unfold-locales

(*auto 4 4 simp*: *vars-def uniform-typed-lifting-def intro*: *sub.explicit-replace-*$\mathcal{V}$)

end

locale *based-typed-renaming-lifting* =
based-functional-substitution-lifting +
renaming-variables-lifting +
based-typed-renaming **where** *subst* = *sub-subst* **and** *vars* = *sub-vars*
begin

sublocale *based-typed-renaming* **where** *subst* = *subst* **and** *vars* = *vars*
by *unfold-locales*

end

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

sublocale *typed-renaming* **where**
subst = *subst* **and** *vars* = *vars* **and** *is-typed* = *lifted-is-typed*
proof *unfold-locales*
fix ϱ *expr* **and** $\mathcal{V}$ $\mathcal{V}'$ :: '*v* ⇒ '*ty*
assume *sub.base.is-renaming* ϱ ∀ *x*∈*vars expr*. $\mathcal{V}$ *x* = $\mathcal{V}'$ (*rename* ϱ *x*)

then show *lifted-is-typed* $\mathcal{V}'$ (*expr* · ϱ) = *lifted-is-typed* $\mathcal{V}$ *expr*
using *sub.typed-renaming*
unfolding *vars-def subst-def is-typed-lifting-def*
by *force*
qed

end

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

sublocale *typed-renaming* **where**
is-typed = *lifted-is-typed* **and** *subst* = *subst* **and** *vars* = *vars*
proof *unfold-locales*
fix ϱ *expr* **and** $\mathcal{V}$ $\mathcal{V}'$ :: '*v* ⇒ '*ty*
assume *base.is-renaming* ϱ ∀ *x*∈*vars expr*. $\mathcal{V}$ *x* = $\mathcal{V}'$ (*rename* ϱ *x*)

then show *lifted-is-typed* $\mathcal{V}'$ (*subst expr* ϱ) = *lifted-is-typed* $\mathcal{V}$ *expr*
using *base.typed-renaming*
unfolding *vars-def subst-def uniform-typed-lifting-def*
by *force*
qed

end

end
theory *Functional-Substitution-Typing-Lifting*
imports
Functional-Substitution-Typing
Typed-Functional-Substitution-Lifting
begin

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

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

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

end

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

sublocale *uniform-typing-lifting* **where**

sub-typed = *base-typed* $\mathcal{V}$ **and** *sub-welltyped* = *base-welltyped* $\mathcal{V}$
by *unfold-locales*

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

end

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

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

locale *base-typing-properties* =
base-functional-substitution-typing +
typed: *base-typed-properties* +
welltyped: *base-typed-properties* **where** *typed* = *welltyped*

locale *base-inhabited-typing-properties* =
base-typing-properties +
typed: *inhabited-explicitly-typed-functional-substitution* +
welltyped: *inhabited-explicitly-typed-functional-substitution* **where** *typed* = *welltyped*

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

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

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*) = (*τs*, *τ*) ⟹ *list-all2* (*welltyped* $\mathcal{V}$) *ts τs* ⟹ *welltyped* $\mathcal{V}$ (*Fun f ts*) *τ*

```
sublocale term: explicit-typing typed (𝒱 :: ('v, 'ty) var-types) welltyped 𝒱
proof unfold-locales
  show right-unique (typed 𝒱)
  proof (rule right-uniqueI)
    fix t τ₁ τ₂
    assume typed 𝒱 t τ₁ and typed 𝒱 t τ₂
    thus τ₁ = τ₂
      by (auto elim!: typed.cases)
  qed
next
  show right-unique (welltyped 𝒱)
  proof (rule right-uniqueI)
    fix t τ₁ τ₂
    assume welltyped 𝒱 t τ₁ and welltyped 𝒱 t τ₂
    thus τ₁ = τ₂
      by (auto elim!: welltyped.cases)
  qed
next
  fix t τ
  assume welltyped 𝒱 t τ
  then show typed 𝒱 t τ
    by (metis (full-types) typed.simps welltyped.cases)
qed

sublocale term: term-typing where
  typed = typed (𝒱 :: 'v ⇒ 'ty) and welltyped = welltyped 𝒱 and Fun = Fun
proof unfold-locales
  fix t t' c τ τ'

  assume
    t-type: welltyped 𝒱 t τ' and
    t'-type: welltyped 𝒱 t' τ' and
    c-type: welltyped 𝒱 c⟨t⟩ τ

  from c-type show welltyped 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 t t' c τ τ'
  assume
    t-type: typed 𝒱 t τ' and
    t'-type: typed 𝒱 t' τ' and
    c-type: typed 𝒱 c⟨t⟩ τ

  from c-type show typed 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 typed 𝒱 (Fun f (ss1 @ c⟨t⟩ # ss2)) τ
      using More.prems
      by simp

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

      then show ?thesis
        by (simp add: typed.simps)
    qed
```

```
    thus ?case
      by simp
  qed
next
  fix f ts τ
  assume welltyped 𝒱 (Fun f ts) τ
  then show ∀ t∈set ts. term.is-welltyped 𝒱 t
    by (cases rule: welltyped.cases) (metis in-set-conv-nth list-all2-conv-all-nth)
next
  fix t
  show term.is-typed 𝒱 t
    by (metis term.exhaust prod.exhaust typed.simps)
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 typed = typed and to-ground
= term.to-ground and
  from-ground = term.from-ground
proof(unfold-locales; (intro typed.Var welltyped.Var refl)?)
  fix τ and 𝒱 and t :: ('f, 'v) term and σ
  assume is-typed-on: ∀ x ∈ term.vars t. typed 𝒱 (σ x) (𝒱 x)

  show typed 𝒱 (t ·t σ) τ ⟷ typed 𝒱 t τ
  proof (rule iffI)
    assume typed 𝒱 t τ

    then show typed 𝒱 (t ·t σ) τ
      using is-typed-on
      by (induction rule: typed.induct) (auto simp: typed.Fun)
  next
    assume typed 𝒱 (t ·t σ) τ

    then show typed 𝒱 t τ
      using is-typed-on
      by (induction t) (auto simp: typed.simps)
  qed
next
  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* $\mathcal{V}$ ($t \cdot t\ \sigma$) τ

then show *welltyped* $\mathcal{V}$ t τ
using *is-welltyped-on*
proof (*induction* $t \cdot t\ \sigma\ \tau$ *arbitrary*: t *rule*: *welltyped.induct*)
case (*Var* $x\ \tau$)

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

have *welltyped* $\mathcal{V}$ t ($\mathcal{V}\ x'$)
unfolding t
by (*simp add*: *welltyped.Var*)

moreover have *welltyped* $\mathcal{V}$ t ($\mathcal{V}\ x$)
using *Var*
unfolding t
by (*simp add*: *welltyped.simps*)

ultimately have $\mathcal{V}$-x': $\tau = \mathcal{V}\ x'$
using *Var.hyps*
by *blast*

show *?case*
unfolding t $\mathcal{V}$-x'
by (*simp add*: *welltyped.Var*)
next
case (*Fun f* $\tau s\ \tau\ ts$)

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** $\mathcal{V}$ $\mathcal{V}'$ τ

assume *typed* $\mathcal{V}$ t τ $\forall x \in term.vars\ t.\ \mathcal{V}\ x = \mathcal{V}'\ x$

then show *typed* $\mathcal{V}'$ t τ
by (*cases rule*: *typed.cases*) (*simp-all add*: *typed.simps*)
next
fix t :: ($'f$, $'v$) *term* **and** $\mathcal{V}$ $\mathcal{V}'$ τ

assume *welltyped* $\mathcal{V}$ t τ $\forall x \in term.vars\ t.\ \mathcal{V}\ x = \mathcal{V}'\ x$

```
  then show welltyped 𝒱′ t τ
   by (induction rule: welltyped.induct) (simp-all add: welltyped.simps list.rel-mono-strong)
next
  fix 𝒱 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 𝒱: ∀x∈term.vars t. 𝒱 x = 𝒱′
(term.rename ϱ x)

  show typed 𝒱′ (t ·t ϱ) τ ⟷ typed 𝒱 t τ
  proof(intro iffI)
    assume typed 𝒱′ (t ·t ϱ) τ
    with 𝒱 show typed 𝒱 t τ
    proof(induction t arbitrary: τ)
      case (Var x)

      have 𝒱′ (term.rename ϱ x) = τ
        using Var term.id-subst-rename[OF renaming]
        by (metis eval-term.simps(1) term.typed.right-uniqueD typed.Var)

      then have 𝒱 x = τ
        by (simp add: renaming Var.prems)

      then show ?case
        by(rule typed.Var)
    next
      case (Fun f ts)
      then show ?case
        by (simp add: typed.simps)
    qed
  next
    assume typed 𝒱 t τ
    then show typed 𝒱′ (t ·t ϱ) τ
      using 𝒱
    proof(induction rule: typed.induct)
      case (Var x τ)

      have 𝒱′ (term.rename ϱ x) = τ
        using Var.hyps Var.prems
        by auto

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

      then show ?case
        by (simp add: typed.simps)
    qed
  qed
```

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* ϱ x)

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

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

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

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

      then have $\mathcal{V}$ x = τ
        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* (λs. s $\cdot t$ ϱ) *ts*)) τ
        by *auto*

      then obtain τs **where** τs:
        list-all2 (*welltyped* $\mathcal{V}'$) (*map* (λs. s $\cdot t$ ϱ) *ts*) τs
        $\mathcal{F}$ f (*length* (*map* (λs. s $\cdot t$ ϱ) *ts*)) = (τs, 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* $\mathcal{V}$ *t* τ
then show *welltyped* $\mathcal{V}'$ (*t* $\cdot t$ ϱ) τ
using $\mathcal{V}$
proof(*induction rule*: *welltyped.induct*)
case (*Var x* τ)

then have $\mathcal{V}'$ (*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* $\mathcal{V}'$) (*map* ($\lambda s.$ *s* $\cdot t$ 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

end

locale *nonground-term-inhabited-typing* =
nonground-term-typing **where** $\mathcal{F} = \mathcal{F}$ **for** $\mathcal{F}$ :: ('*f*, '*ty*) *fun-types* +
assumes *types-inhabited*: $\bigwedge\tau.\ \exists f.\ \mathcal{F}\ f\ 0 = ([], \tau)$
begin

sublocale *base-inhabited-typing-properties* **where**
id-subst = *Var* :: '*v* ⇒ ('*f*, '*v*) *term* **and** *comp-subst* = ($\odot$) **and** *subst* = ($\cdot t$) **and**
vars = *term.vars* **and** *welltyped* = *welltyped* **and** *typed* = *typed* **and** *to-ground*
= *term.to-ground* **and**
from-ground = *term.from-ground*
proof *unfold-locales*
fix $\mathcal{V}$ :: ('*v*, '*ty*) *var-types* **and** τ

obtain *f* **where** *f*: $\mathcal{F}\ f\ 0 = ([], \tau)$
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

then show $\exists\, t.$ *term.is-ground t* $\wedge$ *typed* $\mathcal{V}$ *t* τ
using *term.typed-if-welltyped*
by *blast*
qed

end

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

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

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

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

locale *nonground-uniform-typing-lifting* =
functional-substitution-uniform-typing-lifting +
is-typed: *nonground-uniform-typed-lifting* **where** *base-typed* = *base-typed* +
is-welltyped: *nonground-uniform-typed-lifting* **where** *base-typed* = *base-welltyped*
begin

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

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

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

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

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

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

end

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

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

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

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

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

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

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

end

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

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

sub-is-typed = *sub-is-welltyped* **and** *base-typed* = *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 *term-based-nonground-uniform-typing-lifting* =
term: *nonground-term* +
nonground-uniform-typing-lifting **where**
id-subst = *Var* **and** *comp-subst* = (⊙) **and** *map* = *map-uprod* **and** *to-set* = *set-uprod* **and**
base-vars = *term.vars* **and** *base-subst* = (·*t*) **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*

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

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

sublocale *clause-typing typed* ($\mathcal{V}$ :: (′*v*, ′*ty*) *var-types*) *welltyped* $\mathcal{V}$
by *unfold-locales*

sublocale *atom*: *term-based-nonground-uniform-typing-lifting* **where**
base-typed = *typed* :: (′*v* ⇒ ′*ty*) ⇒ (′*f*, ′*v*) *Term.term* ⇒ ′*ty* ⇒ *bool* **and**
base-welltyped = *welltyped*

by *unfold-locales*

sublocale *literal*: *term-based-nonground-typing-lifting* **where**
base-typed = *typed* :: ('*v* ⇒ '*ty*) ⇒ ('*f*, '*v*) *Term.term* ⇒ '*ty* ⇒ *bool* **and**
base-welltyped = *welltyped* **and** *sub-vars* = *atom.vars* **and** *sub-subst* = (·*a*) **and**
map = *map-literal* **and** *to-set* = *set-literal* **and** *sub-is-typed* = *atom.is-typed* **and**
sub-is-welltyped = *atom.is-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-typed = *typed* **and** *base-welltyped* = *welltyped* **and**
sub-vars = *literal.vars* **and** *sub-subst* = (·*l*) **and** *map* = *image-mset* **and** *to-set*
= *set-mset* **and**
sub-is-typed = *literal.is-typed* **and** *sub-is-welltyped* = *literal.is-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-uniform-inhabited-typing-lifting* **where**
base-typed = *typed* :: ('*v* ⇒ '*ty*) ⇒ ('*f*, '*v*) *Term.term* ⇒ '*ty* ⇒ *bool* **and**
base-welltyped = *welltyped*
by *unfold-locales*

sublocale *literal*: *term-based-nonground-inhabited-typing-lifting* **where**
base-typed = *typed* :: ('*v* ⇒ '*ty*) ⇒ ('*f*, '*v*) *Term.term* ⇒ '*ty* ⇒ *bool* **and**
base-welltyped = *welltyped* **and** *sub-vars* = *atom.vars* **and** *sub-subst* = (·*a*) **and**
map = *map-literal* **and** *to-set* = *set-literal* **and** *sub-is-typed* = *atom.is-typed* **and**
sub-is-welltyped = *atom.is-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-typed = *typed* **and** *base-welltyped* = *welltyped* **and**
sub-vars = *literal.vars* **and** *sub-subst* = (·*l*) **and** *map* = *image-mset* **and** *to-set*

= *set-mset* **and**
sub-is-typed = *literal.is-typed* **and** *sub-is-welltyped* = *literal.is-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*: $(\exists_{\leq 1}x.\ P\ x) \Longrightarrow \{x.\ P\ x\} = \{\} \lor (\exists x.\ \{x.\ P\ x\} = \{x\})$
using *Uniq-D* **by** *fastforce*

lemma *Collect-eq-if-Uniq-prod*:
$(\exists_{\leq 1}(x, y).\ P\ x\ y) \Longrightarrow \{(x, y).\ P\ x\ y\} = \{\} \lor (\exists x\ y.\ \{(x, y).\ P\ x\ y\} = \{(x, y)\})$
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 $select_G\ C_G\ C\ \mathcal{V}\ \gamma \equiv$
$C \cdot \gamma = clause.from\text{-}ground\ C_G \land$
$select_G\ C_G = clause.to\text{-}ground\ ((select\ C) \cdot \gamma) \land$
clause.is-welltyped-ground-instance $C\ \mathcal{V}\ \gamma$

abbreviation *select-subst-stability-on* **where**
select-subst-stability-on select $select_G\ N \equiv$
$\forall C_G \in \bigcup$ (*clause.welltyped-ground-instances* ‘ N). $\exists (C, \mathcal{V}) \in N.\ \exists \gamma.$
select-subst-stability-on-clause select $select_G\ C_G\ C\ \mathcal{V}\ \gamma$

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 $?N_G = \bigcup$(*clause.welltyped-ground-instances* ‘ N)

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

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

then have

$\forall C_G \in ?N_G. \exists\gamma. \exists(C, \mathcal{V}) \in N. \exists select_G.$
select-subst-stability-on-clause select select$_G$ C_G C $\mathcal{V}$ γ
unfolding *clause.welltyped-ground-instances-def*
by *auto*

then have *select*$_G$*-exists-for-premises*:
$\forall C_G \in ?N_G. \exists select_G\ \gamma. \exists(C, \mathcal{V}) \in N.$
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**
$\bigwedge C_G.$ *select*$_G$ C_G = (
if $C_G \in ?N_G$
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 $\mathcal{F}$
for
select :: ('*f*, '*v* :: *infinite*) *atom clause* ⇒ ('*f*, '*v*) *atom clause* **and**
$\mathcal{F}$ :: ('*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*$_G$ $C \subseteq\# C$
  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 $l_G \in\#$ *select*$_G$ C_G
  shows *is-neg* l_G
proof −
  obtain $C\ \gamma$ **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 $I \longleftrightarrow (\forall\, t\ t'\ ctxt.\ (t, t') \in I \longrightarrow (ctxt\langle t\rangle_G, ctxt\langle t'\rangle_G) \in I)$

lemma *compatible-with-gctxtD*:
  compatible-with-gctxt $I \Longrightarrow (t, t') \in I \Longrightarrow (ctxt\langle t\rangle_G, ctxt\langle t'\rangle_G) \in I$
  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↔)
  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 $(t,\ t'') \in I1\ O\ I2$
  then obtain t' **where** $(t,\ t') \in I1$ **and** $(t',\ t'') \in I2$
    by *auto*

  have $(ctxt\langle t\rangle_G,\ ctxt\langle t'\rangle_G) \in I1$
    using ‹$(t,\ t') \in I1$› *assms*(*1*) *compatible-with-gctxtD* **by** *blast*
  moreover have $(ctxt\langle t'\rangle_G,\ ctxt\langle t''\rangle_G) \in I2$
    using ‹$(t',\ t'') \in I2$› *assms*(*2*) *compatible-with-gctxtD* **by** *blast*
  ultimately show $(ctxt\langle t\rangle_G,\ ctxt\langle t''\rangle_G) \in I1\ O\ I2$
    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**
  $rewrite\text{-}inside\text{-}gctxt\ R = \{(ctxt\langle t1\rangle_G,\ ctxt\langle t2\rangle_G) \mid ctxt\ t1\ t2.\ (t1,\ t2) \in R\}$

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

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

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

lemma *rewrite-inside-gctxt-insert*:
  $rewrite\text{-}inside\text{-}gctxt\ (insert\ r\ R) = rewrite\text{-}inside\text{-}gctxt\ \{r\} \cup rewrite\text{-}inside\text{-}gctxt\ R$

using *rewrite-inside-gctxt-union*[*of* {*r*} *R*, *simplified*] **.**

lemma *converse-rewrite-steps*: $(\mathit{rewrite\text{-}inside\text{-}gctxt}\ R)^{-1} = \mathit{rewrite\text{-}inside\text{-}gctxt}\ (R^{-1})$
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*) ∈ *rewrite-inside-gctxt R* **and**
ball-R-rhs-lt-lhs: $\bigwedge$*t1 t2*. (*t1*, *t2*) ∈ *R* ⟹ *t2* $\prec_t$ *t1* **and**
compatible-with-gctxt: $\bigwedge$*t1 t2 ctxt*. *t2* $\prec_t$ *t1* ⟹ $ctxt\langle t2\rangle_G \prec_t ctxt\langle t1\rangle_G$
shows *t2* $\prec_t$ *t1*
proof −
from *rule-in* **obtain** *t1' t2' ctxt* **where**
(*t1'*, *t2'*) ∈ *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'*) ∈ *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'*) ∈ *rewrite-inside-gctxt* (*R1* ∪ *R2*)
t ∈ *NF* (*rewrite-inside-gctxt R2*)
shows (*t*, *t'*) ∈ *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*) ∈ *rewrite-inside-gctxt R* **and**
ball-P: $\bigwedge$*t1 t2*. (*t1*, *t2*) ∈ *R* ⟹ *P t1 t2* **and**
preservation: $\bigwedge$*t1 t2 ctxt σ*. (*t1*, *t2*) ∈ *R* ⟹ *P t1 t2* ⟹ *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'*) ∈ *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') ∈ R›]
    using preservation[OF ‹(t1', t2') ∈ 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 ⊆ rewrite-inside-gctxt E
proof (rule Set.subsetI)
  fix e assume e-in: e ∈ E
  moreover obtain s t where e-def: e = (s, t)
    by fastforce
  show e ∈ rewrite-inside-gctxt E
    unfolding rewrite-inside-gctxt-def
    unfolding mem-Collect-eq
  proof (intro exI conjI)
    show e = (□⟨s⟩G, □⟨t⟩G)
      unfolding e-def
      by simp
  next
    show (s, t) ∈ 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: ⋀ctxt t t'. R t t' ⟹ R ctxt⟨t⟩G ctxt⟨t'⟩G and
    equations-subset-R: ⋀x y. (x, y) ∈ E ⟹ R y x
  shows wf ((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 (rewrite-inside-gctxt E)⁻¹ ⊆ {(x, y). R x y}
  proof (rule Set.subsetI)
    fix e assume e ∈ (rewrite-inside-gctxt E)⁻¹
    then obtain ctxt s t where e-def: e = (ctxt⟨s⟩G, ctxt⟨t⟩G) and (t, s) ∈ 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⟨s⟩G ctxt⟨t⟩G
```

```
      using R-compatible-with-gctxt by simp
    then show e ∈ {(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 ⇒ 'var set and
  base-subst :: 'base ⇒ ('var ⇒ 'base) ⇒ 'base and
  base-to-ground :: 'base ⇒ 'base_G and
  base-from-ground +
fixes entails-def :: 'expr ⇒ bool and I :: ('base_G × 'base_G) set
assumes
  congruence: ⋀expr γ var update.
        based.base.is-ground update ⟹
        based.base.is-ground (γ var) ⟹
        (base-to-ground (γ var), base-to-ground update) ∈ I ⟹
        based.is-ground (subst expr γ) ⟹
        entails-def (subst expr (γ(var := update))) ⟹
        entails-def (subst expr γ)
begin

abbreviation entails ≡ 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 (γ var) and
    var-update: (base-to-ground (γ var), base-to-ground update) ∈ I and
    expr-grounding: based.is-ground (subst expr γ)
  shows
    entails (subst expr (γ(var := update))) ⟷ entails (subst expr γ)
  using congruence[OF var-grounding, of γ(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** · *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 ⟹
    is-ground (*γ var*) ⟹
    (*to-ground* (*γ var*), *to-ground update*) ∈ *I* ⟹
    is-ground (*expr* · *γ*) ⟹
    (*to-ground* (*expr* · (*γ*(*var* := *update*))), *expr′*) ∈ *I* ⟹
    (*to-ground* (*expr* · *γ*), *expr′*) ∈ *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*) ∈ *I*
  shows (*to-ground* (*expr* · (*γ*(*var* := *update*))), *expr′*) ∈ *I* ⟷ (*to-ground* (*expr* · *γ*), *expr′*) ∈ *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*) ∈ *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*)))) ∈ *I* ⟷
      (*to-ground* (*expr* · *γ*), *to-ground* (*expr′* · *γ*)) ∈ *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 :: $'d \Rightarrow$ *bool* **and**
empty :: *bool* **and**
connective :: *bool* ⇒ *bool* ⇒ *bool* **and**
entails-def
assumes
is-negated-subst: $\bigwedge$*expr* σ. *is-negated* (*subst expr* σ) ⟷ *is-negated expr* **and**
entails-def: $\bigwedge$*expr*. *entails-def expr* ⟷
(*if is-negated expr then Not else* (λ*x*. *x*))
(*Finite-Set.fold connective empty* (*sub-entails* ‘ *to-set expr*))
begin

notation *sub-entails* ((⊨$_s$ -) [*50*] *50*)
notation *entails-def* ((⊨ -) [*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* · γ)
(*base-to-ground* (γ *var*), *base-to-ground update*) ∈ *I*
⊨ *expr* · γ(*var* := *update*)

moreover then have ∀ *sub* ∈ *to-set expr*. (⊨$_s$ *sub* $\cdot_s$ γ(*var* := *update*)) ⟷ ⊨$_s$ *sub* $\cdot_s$ γ
using *sub.symmetric-congruence*[*of update* γ] *to-set-is-ground-subst*
by *blast*

ultimately show ⊨ *expr* · 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* = (∧) **and** *empty* = *True*

locale *entailment-lifting-disj* = *entailment-lifting*
where *connective* = (∨) **and** *empty* = *False*

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

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

lemma *comp-fun-idem-disj*: *comp-fun-idem-on X* (∨)
  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 $x \in vars\ t$
  obtains *c* **where** $t = c\langle Var\ x\rangle$
  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** $t' \in set\ args$ $x \in vars\ t'$
    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*⟨*Var x*⟩. *x′* = *x*#} = *Suc* (*size* {#*x′* ∈# *vars-term-ms c*⟨*t*⟩. *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*]: $t = c\langle Var\ var\rangle$
by (*meson term.var-in-term*)

have [*simp*]:
(*?context-to-ground* ($c \cdot t_c\ \gamma$))$\langle$*term.to-ground* (γ *var*)$\rangle_G$ = *term.to-ground* ($c\langle Var\ var\rangle \cdot t\ \gamma$)
using *Suc*
by(*induction c*) *simp-all*

have *context-update* [*simp*]:
(*?context-to-ground* ($c \cdot t_c\ \gamma$))$\langle$*term.to-ground update*$\rangle_G$ = *term.to-ground* ($c\langle update\rangle \cdot t\ \gamma$)
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* ($c\langle update\rangle \cdot t\ \gamma$)
using *Suc.prems update-is-ground*
by *auto*

moreover have (*term.to-ground* ($c\langle update\rangle \cdot t\ \gamma(var := update)$), 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* ($c\langle update\rangle \cdot t\ \gamma$), t') ∈ *I*
using *Suc calculation*
by *blast*

ultimately have ((*?context-to-ground* ($c \cdot t_c\ \gamma$))$\langle$*term.to-ground* (γ *var*)$\rangle_G$, 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* = λ*l*. *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* (λ*x*. *x*))
(*Finite-Set.fold* (∧) *True* ((λ*a*. *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* = λ*C*. *upair* ‘ *I* ⊫ *clause.to-ground C*
proof *unfold-locales*
  fix *C* :: (′*f*, ′*v*) *atom clause*

  show *upair* ‘ *I* ⊫ *clause.to-ground C* ⟷
  (*if False then Not else* (λ*x*. *x*)) (*Finite-Set.fold* (∨) *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* ≈ *t*′) ⟹ *literal.entails* (*c*⟨*t*⟩ ≈ *c*⟨*t*′⟩)
  by (*simp add*: *upair-compatible-with-gctxtI*)

lemma *symmetric-literal-context-congruence*:
  assumes *Upair t t*′ ∈ *upair* ‘ *I*
  shows
    upair ‘ *I* ⊫*l* $c\langle t\rangle_G$ ≈ *t*″ ⟷ *upair* ‘ *I* ⊫*l* $c\langle t'\rangle_G$ ≈ *t*″
    upair ‘ *I* ⊫*l* $c\langle t\rangle_G$!≈ *t*″ ⟷ *upair* ‘ *I* ⊫*l* $c\langle t'\rangle_G$!≈ *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* ($\lambda P.\ P \cdot \sigma$) *Ps*) ($C \cdot \sigma$)
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** ≺ *50*)
  assumes
    transp [*intro*]: *transp* (≺) **and**
    asymp [*intro*]: *asymp* (≺)
begin

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

notation *less-eq* (**infix** ⪯ *50*)

sublocale *order* (⪯) (≺)
  by(*rule order-reflclp-if-transp-and-asymp*[*OF transp asymp*])

end

locale *strict-order-restriction* =
  strict-order +
  relation-restriction **where** *R* = (≺)
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-eq*$_r$ ≡ *restriction.less-eq*
notation *less-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* (≺)

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* = (≺)

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 $c \Longrightarrow$
$t_1 \prec t_2 \Longrightarrow$
$\mathit{Fun}\langle c;t_1\rangle \prec \mathit{Fun}\langle c;t_2\rangle$
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-context c
restricted-context c'
$\bigwedge t.$ *restricted* $t \Longrightarrow \mathit{Fun}\langle c;t\rangle \prec \mathit{Fun}\langle c';t\rangle$
$t \preceq t'$
shows $\mathit{Fun}\langle c;t\rangle \prec \mathit{Fun}\langle c';t'\rangle$
using *assms context-compatible dual-order.strict-trans*
by *blast*

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

end

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

sublocale *restricted*: *context-compatible-restricted-order* **where**
restriction = *UNIV* **and** *context-restriction* = *UNIV* **and** *restricted* = λ-. *True*
and

restricted-context = λ-. *True*
using *less-context-compatible*
by *unfold-locales simp-all*

sublocale *context-compatibility* (≺)
by *unfold-locales*

sublocale *less-eq*: *context-compatibility* (⪯)
by *unfold-locales*

lemma *context-less-term-lesseq*:
assumes
⋀*t*. *Fun*⟨*c*;*t*⟩ ≺ *Fun*⟨*c*′;*t*⟩
t ⪯ *t*′
shows *Fun*⟨*c*;*t*⟩ ≺ *Fun*⟨*c*′;*t*′⟩
using *assms restricted.context-less-term-lesseq*
by *blast*

lemma *context-lesseq-term-less*:
assumes
⋀*t*. *Fun*⟨*c*;*t*⟩ ⪯ *Fun*⟨*c*′;*t*⟩
t ≺ *t*′
shows *Fun*⟨*c*;*t*⟩ ≺ *Fun*⟨*c*′;*t*′⟩
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* ⇒ ′*t* ⇒ *bool*
begin

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

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

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 $\bigwedge z.\ (x, z) \in A^* \Longrightarrow z \notin \mathit{Domain}\ B$
  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
    $\bigwedge z.\ (x, z) \in B^* \Longrightarrow z \notin \mathit{Domain}\ A$
  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** $\mathit{less} = \mathit{less}_t$
  for $\mathit{less}_t :: 'f\ \mathit{gterm} \Rightarrow 'f\ \mathit{gterm} \Rightarrow \mathit{bool}$ +
  assumes
    subterm-property [*simp*]: $\bigwedge t\ c.\ c \neq \Box \Longrightarrow \mathit{less}_t\ t\ c\langle t\rangle_G$
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*$_G$ ≡ *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'* $\longleftrightarrow$ *e* $\prec$ *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'* $\longleftrightarrow$ *e* $\preceq$ *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:
$\bigwedge expr_1\ expr_2\ \gamma.$
is-ground $(expr_1 \cdot \gamma) \Longrightarrow$
is-ground $(expr_2 \cdot \gamma) \Longrightarrow$
$R\ expr_1\ expr_2 \Longrightarrow$
$R\ (expr_1 \cdot \gamma)\ (expr_2 \cdot \gamma)$

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:
$\bigwedge update\ x\ \gamma\ expr.$
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* ⇒ *'a multiset*
begin

definition *multiset-extension* :: *'b* ⇒ *'b* ⇒ *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*) $\subseteq$ *restriction*$\}$
begin

sublocale *restricted-total-strict-order* $(\prec_m)$ $\{b.$ *set-mset* (*to-mset b*) $\subseteq$ *restriction*$\}$
proof *unfold-locales*
  have *totalp-on* $\{b.$ *set-mset b* $\subseteq$ *restriction*$\}$ (*multp* $(\prec)$)
    using *totalp-on-multp*[*OF base.totalp base.transp*]
    by *fastforce*

  then show *totalp-on* $\{b.$ *set-mset* (*to-mset b*) $\subseteq$ *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* = (≺) :: *'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: ⋀*expr*. *set-mset* (*to-mset expr*) = *to-set expr* **and**
to-mset-map: ⋀*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** ⪯ *50*)
notation *sub.less-eq* (**infix** ⪯ *50*)

lemma *lesseq-if-all-lesseq*:
assumes ∀ *sub* ∈# *to-mset expr*. *sub* $\cdot_s$ σ' ⪯ *sub* $\cdot_s$ σ
shows *expr* · σ' $\preceq_m$ *expr* · 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
∀ *sub*∈#*to-mset expr*. *sub* $\cdot_s$ σ' ⪯ *sub* $\cdot_s$ σ
∃ *sub*∈#*to-mset expr*. *sub* $\cdot_s$ σ' ≺ *sub* $\cdot_s$ σ
shows
expr · σ' $\prec_m$ *expr* · 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* ($\prec_m$) *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* ($\prec_m$) *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*}
($\prec_m$)
    using *inj-to-mset*
    unfolding *inj-def multiset-extension-def totalp-on-def*
    by *blast*

  then show *totalp-on* (*range from-ground*) ($\prec_m$)
    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* $expr_2$)) $(\prec)$ $(\prec)$ $(\lambda sub.\ sub \cdot_s \gamma)$
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* · *γ*) *x* ∈ *vars expr*

moreover then have ∀ *sub* ∈# *to-mset expr*. *sub* $\cdot_s$ *γ*(*x* := *update*) ⪯ *sub* $\cdot_s$ *γ*
using
sub.subst-update-stability
sub.subst-reduntant-upd
to-mset-to-set
to-set-is-ground-subst
by *blast*

moreover have ∃ *sub* ∈# *to-mset expr*. *sub* $\cdot_s$ *γ*(*x* := *update*) ≺ (*sub* $\cdot_s$ *γ*)
using *sub.subst-update-stability assms*
unfolding *vars-def subst-def to-mset-to-set*
by *fastforce*

ultimately show *expr* · *γ*(*x* := *update*) $\prec_m$ *expr* · *γ*
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 $l \in\# C$
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* = (⊙) **and** *subst* = (·*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* ⇒ ('*f*, '*v*) *Term.term* ⇒ *bool*
begin

interpretation *term-order-notation*.

sublocale *base-grounded-order* **where**
comp-subst = (⊙) **and** *subst* = (·*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* ·*t* γ) **and**
var: *x* ∈ *term.vars t*

from *term-grounding var*
show *t* ·*t* γ(*x* := *update*) $\prec_t$ *t* ·*t* γ
proof(*induction t*)
case *Var*
then show *?case*
using *update-is-ground update-less*
by *simp*
next
case (*Fun f subs*)

then have ∀ *sub* ∈ *set subs*. *sub* ·*t* γ(*x* := *update*) $\preceq_t$ *sub* ·*t* γ
by (*metis eval-with-fresh-var is-ground-iff reflclp-iff term.set-intros*(*4*))

moreover then have ∃ *sub* ∈ *set subs*. *sub* ·*t* γ(*x* := *update*) $\prec_t$ *sub* ·*t* γ

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* $\cdot t$ γ(x := *update*) $\prec_t$ *sub* $\cdot t$ γ) *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 $\cdot t$ γ(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* $\cdot t$ γ(x := *update*) = *s* $\cdot t$ γ
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* $\cdot t$ γ(x := *update*) = *s* $\cdot t$ γ
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* □ *ss2* $\cdot t_c$ γ

have *context.is-ground ?c*
using *subs first*(*5*)
by *auto*

moreover have *s* $\cdot t$ γ(x := *update*) $\prec_t$ *s* $\cdot t$ γ

using *first.hyps*(*2*)
by (*meson Cons-eq-filterD*)

ultimately have *?c*$\langle$*s* $\cdot t$ γ(*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* $\cdot t$ γ(*x* := *update*) = *?c*$\langle$*s* $\cdot t$ γ(*x* := *update*)$\rangle$
unfolding *subs*
using *ss1 ss2*
by *simp*

moreover have *Fun f subs* $\cdot t$ γ = *?c*$\langle$*s* $\cdot t$ $\gamma\rangle$
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: $\forall$ *s*$\in$*set ss1*. $\neg$ *s* $\cdot t$ γ(*x* := *update*) $\prec_t$ *s* $\cdot t$ γ **and**
less: *s* $\cdot t$ γ(*x* := *update*) $\prec_t$ *s* $\cdot t$ γ **and**
ss: *ss* = *filter* (λ*term*. *term* $\cdot t$ γ(*x* := *update*)$\prec_t$ *term* $\cdot t$ γ) *ss2*
using *Cons-eq-filter-iff*[*of s ss* (λ*s*. *s* $\cdot t$ γ(*x* := *update*) $\prec_t$ *s* $\cdot t$ γ)]
by *blast*

let *?subs'* = *ss1* @ (*s* $\cdot t$ γ(*x* := *update*)) # *ss2*

have [*simp*]: *s* $\cdot t$ γ(*x* := *update*) $\cdot t$ γ = *s* $\cdot t$ γ(*x* := *update*)
using *first.prems*(*3*) *update-is-ground*
unfolding *subs*
by (*simp add*: *is-ground-iff*)

have [*simp*]: *s* $\cdot t$ γ(*x* := *update*) $\cdot t$ γ(*x* := *update*) = *s* $\cdot t$ γ(*x* := *update*)
using *first.prems*(*3*) *update-is-ground*
unfolding *subs*
by (*simp add*: *is-ground-iff*)

have *ss*: *ss* = *filter* (λ*sub*. *sub* $\cdot t$ γ(*x* := *update*) $\prec_t$ *sub* $\cdot t$ γ) *?subs'*
using *ss1 ss*
by *auto*

moreover have $\forall$ *sub* $\in$ *set ?subs'*. *sub* $\cdot t$ γ(*x* := *update*) $\preceq_t$ *sub* $\cdot t$ γ
using *first.prems*(*1*)
unfolding *subs*

by *simp*

moreover have *ex-less*: $\exists\, sub \in set\ ?subs'.\ sub \cdot t\ \gamma(x := update) \prec_t sub \cdot t$
γ
using *ss Cons neq-Nil-conv*
by *force*

moreover have *subs'-grounding*: *term.is-ground* $(Fun\ f\ ?subs' \cdot t\ \gamma)$
using *first.prems*(*3*)
unfolding *subs*
by *simp*

moreover have $x \in term.vars\ (Fun\ f\ ?subs')$
by (*metis ex-less eval-with-fresh-var term.set-intros*(*4*) *order.less-irrefl*)

ultimately have *less-subs'*: $Fun\ f\ ?subs' \cdot t\ \gamma(x := update) \prec_t Fun\ f\ ?subs'$
$\cdot t\ \gamma$
using *first.hyps*(*1*) *first.prems*(*3*)
by *blast*

have *context-grounding*: *context.is-ground* $(More\ f\ ss1\ \Box\ ss2\ \cdot t_c\ \gamma)$
using *subs'-grounding*
by *auto*

have $Fun\ f\ (ss1\ @\ s \cdot t\ \gamma(x := update)\ \#\ ss2) \cdot t\ \gamma \prec_t Fun\ f\ subs \cdot t\ \gamma$
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** $\prec_{tG}$ *50*)
notation $order.less\text{-}eq_G$ (**infix** $\preceq_{tG}$ *50*)

sublocale *restriction*: *ground-term-order* $(\prec_{tG})$
proof *unfold-locales*
fix $c\ t\ t'$
assume $t \prec_{tG} t'$
then show $c\langle t\rangle_G \prec_{tG} c\langle t'\rangle_G$
using *order.ground-context-compatibility*[*OF*
term.ground-is-ground term.ground-is-ground context.ground-is-ground]
unfolding $order.less_G\text{-}def$

by *simp*
next
fix t :: *'f gterm* **and** c :: *'f ground-context*
assume $c \neq \square$
then show $t \prec_{tG} c\langle t\rangle_G$
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* $l\ \sigma$ = *literal.subst* $l\ \sigma$ **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 ·l γ) (C · γ)
proof−

  have grounding-not-empty: C · γ ≠ {#}
    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* $(C \cdot \gamma)$ **and**
ground-strictly-maximal: *is-strictly-maximal* l_G $(C \cdot \gamma)$
obtains l **where**
is-strictly-maximal l C $l_G = l \cdot l\ \gamma$
proof−

have *grounding-not-empty*: $C \cdot \gamma \neq \{\#\}$
using *is-strictly-maximal-not-empty*[*OF ground-strictly-maximal*]**.**

have l_G*-in-grounding*: $l_G \in\# C \cdot \gamma$
using *strictly-maximal-in-clause*[*OF ground-strictly-maximal*]**.**

obtain l **where**
l-in-C: $l \in\# C$ **and**
l_G [*simp*]: $l_G = l \cdot l\ \gamma$
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*: $l' \in\# C - \{\# l \#\}$ **and**
l-less-eq-l': $l \preceq_l l'$
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* $(l' \cdot l\ \gamma)$
using l'*-in-C grounding*
by (*meson clause.to-set-is-ground-subst in-diffD*)

have $l \cdot l\ \gamma \preceq_l l' \cdot l\ \gamma$
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* $(C \cdot \gamma)$ **and**
grounding-strictly-maximal: *is-strictly-maximal* $(l \cdot l\ \gamma)$ $(C \cdot \gamma)$
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*
    (λ*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*: $t_1 \,!\!\approx t_3 \prec_l t_2 \,!\!\approx t_3$
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
Functional-Substitution-Typing
Typed-Functional-Substitution
Abstract-Substitution.Functional-Substitution-Example
begin

type-synonym ('*f*, '*ty*) *fun-types* = '*f* ⇒ '*ty list* × '*ty*

Inductive predicates defining well-typed terms.

inductive *typed* :: ('*f*, '*ty*) *fun-types* ⇒ ('*v*, '*ty*) *var-types* ⇒ ('*f*,'*v*) *term* ⇒ '*ty* ⇒

bool
for $\mathcal{F}$ $\mathcal{V}$ **where**
Var: $\mathcal{V}$ $x = \tau \Longrightarrow$ *typed* $\mathcal{F}$ $\mathcal{V}$ (*Var* x) τ
| *Fun*: $\mathcal{F}$ $f = (\tau s, \tau) \Longrightarrow$ *typed* $\mathcal{F}$ $\mathcal{V}$ (*Fun f ts*) τ

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*: *explicit-typing typed* $\mathcal{F}$ $\mathcal{V}$ *welltyped* $\mathcal{F}$ $\mathcal{V}$
proof *unfold-locales*
show *right-unique* (*typed* $\mathcal{F}$ $\mathcal{V}$)
proof (*rule right-uniqueI*)
fix t τ_1 τ_2
assume *typed* $\mathcal{F}$ $\mathcal{V}$ t τ_1 **and** *typed* $\mathcal{F}$ $\mathcal{V}$ t τ_2
thus $\tau_1 = \tau_2$
by (*auto elim*!: *typed.cases*)
qed
next
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
next
fix t τ
assume *welltyped* $\mathcal{F}$ $\mathcal{V}$ t τ
then show *typed* $\mathcal{F}$ $\mathcal{V}$ t τ
by (*metis* (*full-types*) *typed.simps welltyped.cases*)
qed

global-interpretation *term*: *base-functional-substitution-typing* **where**
typed = *typed* ($\mathcal{F}$ :: ('*f*, '*ty*) *fun-types*) **and** *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*
by (*unfold-locales*; *intro typed.Var welltyped.Var refl*)

A selection of substitution properties for typed terms.

locale *typed-term-subst-properties* =
typed: *explicitly-typed-subst-stability* **where** *typed* = *typed* $\mathcal{F}$ +
welltyped: *explicitly-typed-subst-stability* **where** *typed* = *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 τ **and** $\mathcal{V}$ **and** *t* :: ('f, 'v) *term* **and** σ
  assume *is-typed-on*: ∀ *x* ∈ *vars-term t*. *typed* $\mathcal{F}$ $\mathcal{V}$ (σ *x*) ($\mathcal{V}$ *x*)

  show *typed* $\mathcal{F}$ $\mathcal{V}$ (*t* · σ) τ ⟷ *typed* $\mathcal{F}$ $\mathcal{V}$ *t* τ
  proof(*rule iffI*)
    assume *typed* $\mathcal{F}$ $\mathcal{V}$ *t* τ
    then show *typed* $\mathcal{F}$ $\mathcal{V}$ (*t* · σ) τ
      using *is-typed-on*
      by(*induction rule*: *typed.induct*)(*auto simp*: *typed.Fun*)
  next
    assume *typed* $\mathcal{F}$ $\mathcal{V}$ (*t* · σ) τ
    then show *typed* $\mathcal{F}$ $\mathcal{V}$ *t* τ
      using *is-typed-on*
      by(*induction t*)(*auto simp*: *typed.simps*)
  qed
next
  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 𝒱-x': τ = 𝒱 x'
      using Var.hyps
      by (simp add: t welltyped.simps)

    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
qed
```

Examples of generated lemmas and definitions

```
thm
  term.welltyped.right-unique
  term.welltyped.explicit-subst-stability
  term.welltyped.subst-stability
  term.welltyped.subst-update

  term.typed.right-unique
  term.typed.explicit-subst-stability
  term.typed.subst-stability
  term.typed.subst-update

  term.is-welltyped-on-subset
  term.is-typed-on-subset
  term.is-welltyped-id-subst
  term.is-typed-id-subst

term term.is-welltyped
term term.subst.is-welltyped-on
term term.subst.is-welltyped
term term.is-typed
term term.subst.is-typed-on
term term.subst.is-typed

end
theory Typed-Functional-Substitution-Lifting-Example
  imports
    Functional-Substitution-Typing-Lifting
    Typed-Functional-Substitution-Lifting
    Typed-Functional-Substitution-Example
    Abstract-Substitution.Functional-Substitution-Lifting-Example
```

begin

All property locales have corresponding lifting locales

locale *nonground-uniform-typing-lifting* =
functional-substitution-uniform-typing-lifting **where**
base-typed = *typed* $\mathcal{F}$ **and** *base-welltyped* = *welltyped* $\mathcal{F}$ +

is-typed: *uniform-typed-subst-stability-lifting* **where**
base-typed = *typed* $\mathcal{F}$ +

is-welltyped: *uniform-typed-subst-stability-lifting* **where**
base-typed = *welltyped* $\mathcal{F}$
for $\mathcal{F}$ :: ('*f*, '*ty*) *fun-types*

locale *nonground-typing-lifting* =
functional-substitution-typing-lifting **where**
base-typed = *typed* $\mathcal{F}$ **and** *base-welltyped* = *welltyped* $\mathcal{F}$ +

is-typed: *typed-subst-stability-lifting* **where** *base-typed* = *typed* $\mathcal{F}$ +

is-welltyped: *typed-subst-stability-lifting* **where**
sub-is-typed = *sub-is-welltyped* **and** *base-typed* = *welltyped* $\mathcal{F}$
for $\mathcal{F}$ :: ('*f*, '*ty*) *fun-types*

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

sublocale *equation*:
uniform-typing-lifting **where**
sub-typed = *typed* $\mathcal{F}$ $\mathcal{V}$ **and** *sub-welltyped* = *welltyped* $\mathcal{F}$ $\mathcal{V}$ **and**
to-set = *set-prod*
by *unfold-locales*

sublocale *equation*:
nonground-uniform-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*
by *unfold-locales*

Lifted lemmas and definitions

thm
equation.is-welltyped-def
equation.is-typed-def

equation.is-welltyped.subst-stability
equation.is-typed.subst-stability

equation.is-typed-if-is-welltyped

We can lift multiple levels

sublocale *equation-set*:
typing-lifting **where**
sub-is-typed = *equation.is-typed* $\mathcal{V}$ **and** *sub-is-welltyped* = *equation.is-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-is-welltyped = *equation.is-welltyped* **and** *sub-is-typed* = *equation.is-typed*
by *unfold-locales*

Lifted lemmas and definitions

thm
equation-set.is-welltyped-def
equation-set.is-typed-def

equation-set.is-welltyped.subst-stability
equation-set.is-typed.subst-stability
equation-set.is-typed-if-is-welltyped

end

Interpretation with Unit-Typing

global-interpretation *example-typing-lifting* λ-. ([], ())**.**

end