A Variant of the Superposition Calculus

Nicolas Peltier
CNRS/University of Grenoble (LIG)

March 17, 2025

Abstract

We provide a formalization in Isabelle/Isar of (a variant of) the superposition calculus [1, 4], together with formal proofs of soundness and refutational completeness (w.r.t. the usual redundancy criteria based on clause ordering). This version of the calculus uses all the standard restrictions of the superposition rules, together with the following refinement, inspired by the basic superposition calculus [2, 3]: each clause is associated with a set of terms which are assumed to be in normal form – thus any application of the replacement rule on these terms is blocked. The set is initially empty and terms may be added or removed at each inference step. The set of terms that are assumed to be in normal form includes any term introduced by previous unifiers as well as any term occurring in the parent clauses at a position that is smaller (according to some given ordering on positions) than a previously replaced term. This restriction is slightly weaker than that of the basic superposition calculus (since it is based on terms instead of positions), but it has the advantage that the irreducible terms may be propagated through the inferences (under appropriate conditions), even if they do not occur in the parent clauses. The standard superposition calculus corresponds to the case where the set of irreducible terms is always empty. The term representation and unification algorithm are taken from the theory `Unification.thy` provided in Isabelle.

Contents

1 Preliminaries

theory *multisets-continued*

imports *Main HOL−Library.Multiset*

begin

1.1 Multisets

We use the Multiset theory provided in Isabelle. We prove some additional (mostly trivial) lemmata.

lemma *mset-set-inclusion*:
 assumes *finite E2*
 assumes $E1 \subset E2$
 shows *mset-set E1* ⊂# (*mset-set E2*)
proof (*rule ccontr*)
 let *?s1* = *mset-set E1*
 let *?s2* = *mset-set E2*
 assume ¬ *?s1* ⊂# *?s2*
 from *assms*(*1*) **and** *assms*(*2*) **have** *finite E1* **using** *finite-subset less-imp-le* **by** *auto*
 from ‹¬ *?s1* ⊂# *?s2*› **obtain** *x* **where** (*count ?s1 x* > *count ?s2 x*) **using** *subseteq-mset-def* [*of ?s1 ?s2*]
  by (*metis assms*(*1*) *assms*(*2*) *finite-set-mset-mset-set finite-subset less-imp-le less-le not-le-imp-less subset-mset.le-less*)
 from *this* **have** *count ?s1 x* > *0* **by** *linarith*
 from *this* **and** ‹*finite E1*› **have** *count ?s1 x* = *1* **and** $x \in E1$ **using** *subseteq-mset-def* **by** *auto*
 from *this* **and** *assms*(*2*) **have** $x \in E2$ **by** *auto*
 from *this* **and** ‹*finite E2*› **have** *count ?s2 x* = *1* **by** *auto*
 from *this* **and** ‹*count ?s1 x* = *1*› **and** ‹(*count ?s1 x* > *count ?s2 x*)› **show** *False* **by** *auto*
qed

lemma *mset-ordering-addition*:
 assumes $A = B + C$
 shows *B* ⊆# *A*
 using *assms* **by** *simp*

lemma *equal-image-mset*:
 assumes $\forall x \in E.\ (f\ x) = (g\ x)$
 shows {# (*f x*). *x* ∈# (*mset-set E*) #} = {# (*g x*). *x* ∈# (*mset-set E*) #}
by (*meson assms count-eq-zero-iff count-mset-set*(*3*) *image-mset-cong*)

lemma *multiset-order-inclusion*:
 assumes *E* ⊂# *F*

assumes *trans r*
shows $(E,F) \in (mult\ r)$
proof −
let $?G = F-E$
from *assms*(*1*) **have** $F = E + ?G$
by (*simp add*: *subset-mset.add-diff-inverse subset-mset-def*)
from *this assms*(*1*) **have** $?G \neq \{\#\}$
by *fastforce*
have $E = E + \{\#\}$ **by** *auto*
from *this* ‹$F = E + ?G$› ‹$?G \neq \{\#\}$› *assms*(*2*) **show** *?thesis* **using** *one-step-implies-mult* [*of ?G* $\{\#\}$ *r E*] **by** *auto*
qed

lemma *multiset-order-inclusion-eq*:
assumes $E \subseteq\# F$
assumes *trans r*
shows $E = F \vee (E,F) \in (mult\ r)$
proof (*cases*)
assume $E = F$
then show *?thesis* **by** *auto*
next
assume $E \neq F$
from *this* **and** *assms*(*1*) **have** $E \subset\# F$ **by** *auto*
from *this assms*(*2*) **and** *multiset-order-inclusion* **show** *?thesis* **by** *auto*
qed

lemma *image-mset-ordering*:
assumes $M1 = \{\#\ (f1\ u).\ u \in\# L\ \#\}$
assumes $M2 = \{\#\ (f2\ u).\ u \in\# L\ \#\}$
assumes $\forall u.\ (u \in\# L \longrightarrow (((f1\ u), (f2\ u)) \in r \vee (f1\ u) = (f2\ u)))$
assumes $\exists u.\ (u \in\# L \wedge ((f1\ u), (f2\ u)) \in r)$
assumes *irrefl r*
shows ($(M1,M2) \in (mult\ r)$)
proof −
let $?L' = \{\#\ u \in\# L.\ (f1\ u) = (f2\ u)\ \#\}$
let $?L'' = \{\#\ u \in\# L.\ (f1\ u) \neq (f2\ u)\ \#\}$
have $L = ?L' + ?L''$ **by** (*simp*)
from *assms*(*3*) **have** $\forall u.\ (u \in\# ?L'' \longrightarrow ((f1\ u),(f2\ u)) \in r)$ **by** *auto*
let $?M1' = \{\#\ (f1\ u).\ u \in\# ?L'\ \#\}$
let $?M2' = \{\#\ (f2\ u).\ u \in\# ?L'\ \#\}$
have $?M1' = ?M2'$
by (*metis* (*mono-tags*, *lifting*) *mem-Collect-eq multiset.map-cong0 set-mset-filter*)

let $?M1'' = \{\#\ (f1\ u).\ u \in\# ?L''\ \#\}$
let $?M2'' = \{\#\ (f2\ u).\ u \in\# ?L''\ \#\}$
from ‹$L = ?L' + ?L''$› **have** $M1 = ?M1' + ?M1''$ **by** (*metis assms*(*1*) *image-mset-union*)
from ‹$L = ?L' + ?L''$› **have** $M2 = ?M2' + ?M2''$ **by** (*metis assms*(*2*) *image-mset-union*)

have *dom*: $(\forall k \in \textit{set-mset ?M1''}.\ \exists j \in \textit{set-mset ?M2''}.\ (k, j) \in r)$
proof
fix *k* **assume** $k \in \textit{set-mset ?M1''}$
from *this* **obtain** *u* **where** $k = (f1\ u)$ **and** $u \in\#\ ?L''$ **by** *auto*
from ‹$u \in\#\ ?L''$› **have** $(f2\ u) \in\#\ ?M2''$ **by** *simp*
from ‹$\forall u.\ (u \in\#\ ?L'' \longrightarrow ((f1\ u),(f2\ u)) \in r)$› **and** ‹$u \in\#\ ?L''$›
have $((f1\ u),(f2\ u)) \in r$ **by** *auto*
from *this* **and** ‹$k = (f1\ u)$› **and** ‹$(f2\ u) \in \textit{set-mset ?M2''}$›
show $\exists j \in \textit{set-mset ?M2''}.\ (k, j) \in r$ **by** *auto*
qed
have $?L'' \neq \{\#\}$
proof −
from *assms*(*4*) **obtain** *u* **where** $u \in\#\ L$ **and** $(\ (f1\ u),(f2\ u)\ ) \in r$ **by** *auto*
from *assms*(*5*) ‹$(\ (f1\ u),(f2\ u)\ ) \in r$› **have** $(\ (f1\ u) \neq (f2\ u)\ )$
unfolding *irrefl-def* **by** *fastforce*
from ‹$u \in\#\ L$› ‹$(\ (f1\ u) \neq (f2\ u)\ )$› **have** $u \in\#\ ?L''$ **by** *auto*
from *this* **show** *?thesis* **by** *force*
qed
from *this* **have** $?M2'' \neq \{\#\}$ **by** *auto*
from *this* **and** *dom* **and** ‹$M1 = ?M1' + ?M1''$› ‹$M2 = ?M2' + ?M2''$› ‹$?M1'=?M2'$›

show $(M1,M2) \in (\textit{mult}\ r)$ **by** (*simp add*: *one-step-implies-mult*)
qed

lemma *image-mset-ordering-eq*:
assumes $M1 = \{\#\ (f1\ u).\ u \in\#\ L\ \#\}$
assumes $M2 = \{\#\ (f2\ u).\ u \in\#\ L\ \#\}$
assumes $\forall u.\ (u \in\#\ L \longrightarrow (((f1\ u),\ (f2\ u)) \in r \lor (f1\ u) = (f2\ u)))$
shows $(M1 = M2) \lor (\ (M1,M2) \in (\textit{mult}\ r)\ )$
proof (*cases*)
assume $M1 = M2$ **then show** *?thesis* **by** *auto*
next assume $M1 \neq M2$
let $?L' = \{\#\ u \in\#\ L.\ \ (f1\ u) = (f2\ u)\ \#\}$
let $?L'' = \{\#\ u \in\#\ L.\ \ (f1\ u) \neq (f2\ u)\ \#\}$
have $L = ?L' + ?L''$ **by** (*simp*)
from *assms*(*3*) **have** $\forall u.\ (u \in\#\ ?L'' \longrightarrow ((f1\ u),(f2\ u)) \in r)$ **by** *auto*
let $?M1' = \{\#\ (f1\ u).\ u \in\#\ ?L'\ \#\}$
let $?M2' = \{\#\ (f2\ u).\ u \in\#\ ?L'\ \#\}$
have $?M1' = ?M2'$
by (*metis* (*mono-tags*, *lifting*) *mem-Collect-eq multiset.map-cong0 set-mset-filter*)
let $?M1'' = \{\#\ (f1\ u).\ u \in\#\ ?L''\ \#\}$
let $?M2'' = \{\#\ (f2\ u).\ u \in\#\ ?L''\ \#\}$
from ‹$L = ?L' + ?L''$› **have** $M1 = ?M1' + ?M1''$ **by** (*metis assms*(*1*) *image-mset-union*)
from ‹$L = ?L' + ?L''$› **have** $M2 = ?M2' + ?M2''$ **by** (*metis assms*(*2*) *image-mset-union*)
have *dom*: $(\forall k \in \textit{set-mset ?M1''}.\ \exists j \in \textit{set-mset ?M2''}.\ (k, j) \in r)$
proof
fix *k* **assume** $k \in \textit{set-mset ?M1''}$

from *this* **obtain** *u* **where** $k = (f1\ u)$ **and** $u \in\#\ ?L''$ **by** *auto*
from ‹$u \in\#\ ?L''$› **have** $(f2\ u) \in\#\ ?M2''$ **by** *simp*
from ‹$\forall u.\ (u \in\#\ ?L'' \longrightarrow ((f1\ u),(f2\ u)) \in r)$› **and** ‹$u \in\#\ ?L''$›
have $((f1\ u),(f2\ u)) \in r$ **by** *auto*
from *this* **and** ‹$k = (f1\ u)$› **and** ‹$(f2\ u) \in$ *set-mset* $?M2''$›
show $\exists j \in$ *set-mset* $?M2''.\ (k, j) \in r$ **by** *auto*
qed
from ‹$M1 \neq M2$› **have** $?M2'' \neq \{\#\}$
using ‹$M1 =$ *image-mset* $f1\ \{\#\ u \in\#\ L.\ f1\ u = f2\ u\#\} +$ *image-mset* $f1\ \{\#\ u \in\#\ L.\ f1\ u \neq f2\ u\#\}$› ‹$M2 =$ *image-mset* $f2\ \{\#\ u \in\#\ L.\ f1\ u = f2\ u\#\} +$ *image-mset* $f2\ \{\#\ u \in\#\ L.\ f1\ u \neq f2\ u\#\}$› ‹*image-mset* $f1\ \{\#\ u \in\#\ L.\ f1\ u = f2\ u\#\} =$ *image-mset* $f2\ \{\#\ u \in\#\ L.\ f1\ u = f2\ u\#\}$› **by** *auto*
from *this* **and** *dom* **and** ‹$M1 = ?M1' + ?M1''$› ‹$M2 = ?M2' + ?M2''$› ‹$?M1'=?M2'$›

have $(M1,M2) \in (mult\ r)$ **by** (*simp add*: *one-step-implies-mult*)
from *this* **show** *?thesis* **by** *auto*
qed

lemma *mult1-def-lemma* :
assumes $M = M0 + \{\#a\#\} \wedge N = M0 + K \wedge (\forall b.\ b \in\#\ K \longrightarrow (b, a) \in r)$
shows $(N,M) \in (mult1\ r)$
proof −
from *assms*(*1*) **show** *?thesis* **using** *mult1-def* [*of r*] **by** *auto*
qed

lemma *mset-ordering-add1*:
assumes $(E1,E2) \in (mult\ r)$
shows $(E1,E2 + \{\#\ a\ \#\}) \in (mult\ r)$
proof −
have *i*: $(E2,E2 + \{\#\ a\ \#\}) \in (mult1\ r)$ **using** *mult1-def-lemma* [*of* $E2 + \{\#\ a\ \#\}\ E2\ a\ E2\ \{\#\}\ r$]
by *auto*
have *i*: $(E2,E2 + \{\#\ a\ \#\}) \in (mult1\ r)$ **using** *mult1-def-lemma* [*of* $E2 + \{\#\ a\ \#\}\ E2\ a\ E2\ \{\#\}\ r$]
by *auto*
from *assms*(*1*) **have** $(E1,E2) \in (mult1\ r)^+$ **using** *mult-def* **by** *auto*
from *this* **and** *i* **have** $(E1,E2 + \{\#\ a\ \#\}) \in (mult1\ r)^+$ **by** *auto*
then show *?thesis* **using** *mult-def* **by** *auto*
qed

lemma *mset-ordering-singleton*:
assumes $\forall x.\ (x \in\#\ E1 \longrightarrow (x,a) \in r)$
shows $(E1, \{\#\ a\ \#\}) \in (mult\ r)$
proof −
let $?K = E1$
let $?M0 = \{\#\}$
have *i*: $E1 = ?M0 + ?K$ **by** *auto*
have *ii*: $\{\#\ a\ \#\} = ?M0 + \{\#\ a\ \#\}$ **by** *auto*
from *assms*(*1*) **have** *iii*: $\forall x.\ (x \in\#\ ?K \longrightarrow (x,a) \in r)$ **by** *auto*

from *i* **and** *ii* **and** *iii* **show** *?thesis* **using** *mult1-def-lemma* [*of* {# *a* #} *?M0* *a E1 ?K r*] *mult-def* **by** *auto*
qed

lemma *monotonic-fun-mult1*:
assumes $\bigwedge$ *t s*. ((*t*,*s*) ∈ *r* ⟹ ((*f t*), (*f s*)) ∈ *r*)
assumes (*E1*,*E2*) ∈ (*mult1 r*)
shows ({# (*f x*). *x* ∈# *E1* #},{# (*f x*). *x* ∈# *E2* #}) ∈ (*mult1 r*)
proof −
let *?E1* = {# (*f x*). *x* ∈# *E1* #}
let *?E2* = {# (*f x*). *x* ∈# *E2* #}
from *assms*(*2*) **obtain** *M0 a K* **where** *E2* = *M0* + {#*a*#} **and** *E1* = *M0* + *K* **and** (∀ *b*. *b* ∈# *K* ⟶ (*b*, *a*) ∈ *r*)
unfolding *mult1-def* [*of r*] **by** *auto*
let *?K* = {# (*f x*). *x* ∈# *K* #}
let *?M0* = {# (*f x*). *x* ∈# *M0* #}
from ‹*E2* = *M0* + {#*a*#}› **have** *?E2* = *?M0* + {# (*f a*) #} **by** *simp*
from ‹*E1* = *M0* + *K*› **have** *?E1* = *?M0* + *?K* **by** *simp*
have (∀ *b*. *b* ∈# *?K* ⟶ (*b*, (*f a*)) ∈ *r*)
proof ((*rule allI*),(*rule impI*))
fix *b* **assume** *b* ∈# *?K*
from ‹*b* ∈# *?K*› **obtain** *b′* **where** *b* = (*f b′*) **and** *b′* ∈# *K*
by (*auto simp*: *insert-DiffM2 msed-map-invR union-single-eq-member*)
from ‹*b′* ∈# *K*› **and** ‹(∀ *b*. *b* ∈# *K* ⟶ (*b*, *a*) ∈ *r*)› **have** (*b′*,*a*) ∈ *r* **by** *auto*
from *assms*(*1*) **and** *this* **and** ‹*b* = (*f b′*)› **show** (*b*, (*f a*)) ∈ *r* **by** *auto*
qed
from ‹*?E1* = *?M0* + *?K*› **and** ‹*?E2* = *?M0* + {# (*f a*) #}› **and** ‹(∀ *b*. *b* ∈# *?K* ⟶ (*b*, (*f a*)) ∈ *r*)›
show (*?E1*,*?E2*) ∈ (*mult1 r*) **by** (*metis mult1-def-lemma*)
qed

lemma *monotonic-fun-mult*:
assumes $\bigwedge$ *t s*. ((*t*,*s*) ∈ *r* ⟹ ((*f t*), (*f s*)) ∈ *r*)
assumes (*E1*,*E2*) ∈ (*mult r*)
shows ({# (*f x*). *x* ∈# *E1* #},{# (*f x*). *x* ∈# *E2* #}) ∈ (*mult r*)
proof −
let *?E1* = {# (*f x*). *x* ∈# *E1* #}
let *?E2* = {# (*f x*). *x* ∈# *E2* #}
let *?P* = λ*x*. (*?E1*,{# (*f y*). *y* ∈# *x* #}) ∈ (*mult r*)
show *?thesis*
proof (*rule trancl-induct* [*of E1 E2* (*mult1 r*) *?P*])
from *assms*(*1*) **show** (*E1*, *E2*) ∈ (*mult1 r*)$^+$ **using** *assms*(*2*) *mult-def* **by** *blast*
next
fix *x* **assume** (*E1*, *x*) ∈ *mult1 r*
have (*image-mset f E1*, *image-mset f x*) ∈ *mult1 r*
by (*simp add*: ‹(*E1*, *x*) ∈ *mult1 r*› *assms*(*1*) *monotonic-fun-mult1*)
from *this* **show** (*image-mset f E1*, *image-mset f x*) ∈ *mult r* **by** (*simp add*: *mult-def*)

next
fix *x z* **assume** (*E1*, *x*) ∈ (*mult1 r*)$^+$
(*x*, *z*) ∈ *mult1 r* **and** (*image-mset f E1*, *image-mset f x*) ∈ *mult r*
from ‹(*x*, *z*) ∈ *mult1 r*› **have** (*image-mset f x*, *image-mset f z*) ∈ *mult1 r*
by (*simp add*: *assms*(*1*) *monotonic-fun-mult1*)
from *this* **and** ‹(*image-mset f E1*, *image-mset f x*) ∈ *mult r*›
show (*image-mset f E1*, *image-mset f z*) ∈ *mult r*
using *mult-def trancl.trancl-into-trancl* **by** *fastforce*
qed
qed

lemma *mset-set-insert-eq*:
assumes *finite E*
shows *mset-set* (*E* ∪ { *x* }) ⊆# *mset-set E* + {# *x* #}
proof (*rule ccontr*)
assume ¬ *?thesis*
from *this* **obtain** *y* **where** (*count* (*mset-set* (*E* ∪ { *x* })) *y*)
> (*count* (*mset-set E* + {# *x* #}) *y*)
by (*meson leI subseteq-mset-def*)
from *assms*(*1*) **have** *finite* (*E* ∪ { *x* }) **by** *auto*
have (*count* (*mset-set E* + {# *x* #}) *y*) = (*count* (*mset-set E*) *y*) + (*count* {# *x* #} *y*) **by** *auto*
have *x* ≠ *y*
proof
assume *x* = *y*
then have *y* ∈ *E* ∪ { *x* } **by** *auto*
from ‹*finite* (*E* ∪ { *x* })› *this* **have** (*count* (*mset-set* (*E* ∪ { *x* })) *y*) = *1*
using *count-mset-set*(*1*) **by** *auto*
from *this* **and** ‹(*count* (*mset-set* (*E* ∪ { *x* })) *y*) > (*count* (*mset-set E* + {# *x* #}) *y*)› **have**
(*count* (*mset-set E* + {# *x* #}) *y*) = *0* **by** *auto*
from ‹(*count* (*mset-set E* + {# *x* #}) *y*) = *0*› **have** *count* {# *x* #} *y* = *0* **by** *auto*
from ‹*x* = *y*› **have** *count* {# *x* #} *y* = *1* **using** *count-mset-set* **by** *auto*
from *this* **and** ‹*count* {# *x* #} *y* = *0*› **show** *False* **by** *auto*
qed
have *y* ∉ *E*
proof
assume *y* ∈ *E*
then have *y* ∈ *E* ∪ { *x* } **by** *auto*
from ‹*finite* (*E* ∪ { *x* })› *this* **have** (*count* (*mset-set* (*E* ∪ { *x* })) *y*) = *1*
using *count-mset-set*(*1*) **by** *auto*
from *this* **and** ‹(*count* (*mset-set* (*E* ∪ { *x* })) *y*) > (*count* (*mset-set E* + {# *x* #}) *y*)› **have**
(*count* (*mset-set E* + {# *x* #}) *y*) = *0* **by** *auto*
from ‹(*count* (*mset-set E* + {# *x* #}) *y*) = *0*› **have** *count* (*mset-set E*) *y* = *0* **by** (*simp split*: *if-splits*)
from ‹*y* ∈ *E*› ‹*finite E*› **have** *count* (*mset-set E*) *y* = *1* **using** *count-mset-set*(*1*) **by** *auto*

from *this* **and** ‹*count* (*mset-set E*) *y* = *0*› **show** *False* **by** *auto*
qed
from *this* **and** ‹$x \neq y$› **have** $y \notin E \cup \{ x \}$ **by** *auto*
from *this* **have** (*count* (*mset-set* ($E \cup \{ x \}$)) *y*) = *0* **by** *auto*
from *this* **and** ‹(*count* (*mset-set* ($E \cup \{ x \}$)) *y*)
> (*count* (*mset-set E* + {# *x* #}) *y*)› **show** *False* **by** *auto*
qed

lemma *mset-set-insert*:
assumes $x \notin E$
assumes *finite E*
shows *mset-set* ($E \cup \{ x \}$) = *mset-set E* + {# *x* #}
proof (*rule ccontr*)
assume ¬ *?thesis*
from *this* **obtain** *y* **where** (*count* (*mset-set* ($E \cup \{ x \}$)) *y*)
≠ (*count* (*mset-set E* + {# *x* #}) *y*) **by** (*meson multiset-eqI*)
have (*count* (*mset-set E* + {# *x* #}) *y*) = (*count* (*mset-set E*) *y*) + (*count* {# *x* #} *y*) **by** *auto*
from *assms*(*2*) **have** *finite* ($E \cup \{ x \}$) **by** *auto*
have $x \neq y$
proof
assume $x = y$
then have $y \in E \cup \{ x \}$ **by** *auto*
from ‹*finite* ($E \cup \{ x \}$)› *this* **have** (*count* (*mset-set* ($E \cup \{ x \}$)) *y*) = *1*
using *count-mset-set*(*1*) **by** *auto*
from ‹$x = y$› **have** *count* {# *x* #} *y* = *1* **using** *count-mset-set* **by** *auto*
from ‹$x = y$› ‹$x \notin E$› **have** (*count* (*mset-set E*) *y*) = *0* **using** *count-mset-set* **by** *auto*
from ‹*count* {# *x* #} *y* = *1*› ‹(*count* (*mset-set E*) *y*) = *0*›
‹(*count* (*mset-set E* + {# *x* #}) *y*) = (*count* (*mset-set E*) *y*) + (*count* {# *x* #} *y*)›
have (*count* (*mset-set E* + {# *x* #}) *y*) = *1* **by** *auto*
from *this* **and** ‹(*count* (*mset-set* ($E \cup \{ x \}$)) *y*) = *1*› **and** ‹(*count* (*mset-set* ($E \cup \{ x \}$)) *y*)
≠ (*count* (*mset-set E* + {# *x* #}) *y*)› **show** *False* **by** *auto*
qed
from ‹$x \neq y$› **have** *count* {# *x* #} *y* = *0* **using** *count-mset-set* **by** *auto*
have $y \notin E$
proof
assume $y \in E$
then have $y \in E \cup \{ x \}$ **by** *auto*
from ‹*finite* ($E \cup \{ x \}$)› *this* **have** (*count* (*mset-set* ($E \cup \{ x \}$)) *y*) = *1*
using *count-mset-set*(*1*) **by** *auto*
from *assms*(*2*) ‹$y \in E$› **have** (*count* (*mset-set E*) *y*) = *1* **using** *count-mset-set* **by** *auto*
from ‹*count* {# *x* #} *y* = *0*› ‹(*count* (*mset-set E*) *y*) = *1*›
‹(*count* (*mset-set E* + {# *x* #}) *y*) = (*count* (*mset-set E*) *y*) + (*count* {# *x* #} *y*)›
have (*count* (*mset-set E* + {# *x* #}) *y*) = *1* **by** *auto*

from *this* **and** ‹(*count* (*mset-set* (*E* ∪ { *x* })) *y*) = *1*› **and** ‹(*count* (*mset-set* (*E* ∪ { *x* })) *y*)
≠ (*count* (*mset-set E* + {# *x* #}) *y*)› **show** *False* **by** *auto*
qed
from *this* **and** ‹*x* ≠ *y*› **have** *y* ∉ *E* ∪ { *x* } **by** *auto*
from *this* **have** (*count* (*mset-set* (*E* ∪ { *x* })) *y*) = *0* **by** *auto*
from ‹*y* ∉ *E*› **have** (*count* (*mset-set E*) *y*) = *0* **using** *count-mset-set* **by** *auto*
from ‹*count* {# *x* #} *y* = *0*› ‹(*count* (*mset-set E*) *y*) = *0*›
‹(*count* (*mset-set E* + {# *x* #}) *y*) = (*count* (*mset-set E*) *y*) + (*count* {# *x* #} *y*)›
have (*count* (*mset-set E* + {# *x* #}) *y*) = *0* **by** *auto*
from *this* **and** ‹(*count* (*mset-set* (*E* ∪ { *x* })) *y*) = *0*› **and** ‹(*count* (*mset-set* (*E* ∪ { *x* })) *y*)
≠ (*count* (*mset-set E* + {# *x* #}) *y*)› **show** *False* **by** *auto*
qed

lemma *mset-image-comp*:
shows {# (*f x*). *x* ∈# {# (*g x*). *x* ∈# *E* #} #} = {# (*f* (*g x*)). *x* ∈# *E* #}
by (*simp add*: *image-mset.compositionality comp-def*)

lemma *mset-set-mset-image*:
shows ⋀ *E*. *card E* = *N* ⟹ *finite E* ⟹ *mset-set* (*g* ‘ *E*) ⊆# {# (*g x*). *x* ∈# *mset-set* (*E*) #}
proof (*induction N*)
case *0*
assume *card E* = *0*
assume *finite E*
from *this* **and** ‹*card E* = *0*› **have** *E* = {} **by** *auto*
then show *mset-set* (*g* ‘ *E*) ⊆# {# (*g x*). *x* ∈# *mset-set* (*E*) #} **by** *auto*
next
case (*Suc N*)
assume *card E* = (*Suc N*)
assume *finite E*
from *this* **and** ‹*card E* = (*Suc N*)› **have** *E* ≠ {} **by** *auto*
from *this* **obtain** *x* **where** *x* ∈ *E* **by** *auto*
let *?E* = *E* − { *x* }
from ‹*finite E*› ‹*card E* = (*Suc N*)› **and** ‹*x* ∈ *E*› **have** *card ?E* = *N* **by** *auto*
from ‹*finite E*› **have** *finite ?E* **by** *auto*
from *this* **and** *Suc.IH* [*of ?E*] **and** ‹*card ?E* = *N*›
have *ind*: *mset-set* (*g* ‘ *?E*) ⊆# {# (*g x*). *x* ∈# *mset-set* (*?E*) #} **by** *force*
from ‹*x* ∈ *E*› **have** *E* = *?E* ∪ { *x* } **by** *auto*
have *x* ∉ *?E* **by** *auto*
from ‹*finite ?E*› ‹*E* = *?E* ∪ { *x* }› **and** ‹*x* ∉ *?E*› **have** *mset-set* (*?E* ∪ { *x* }) = *mset-set ?E* + {# *x* #}
using *mset-set-insert* [*of x ?E*] **by** *auto*
from *this* **have**
{# (*g x*). *x* ∈# *mset-set* (*?E* ∪ { *x* }) #} = {# (*g x*). *x* ∈# *mset-set ?E* #} + {# (*g x*) #}
by *auto*

have $(g \text{ ` } (?E \cup \{ x \}) = (g \text{ ` } ?E) \cup \{ g\ x \})$ **by** *auto*
from *this* **have** *i*: *mset-set* $(g \text{ ` } (?E \cup \{ x \}))$ = *mset-set* $( (g \text{ ` } ?E) \cup \{ g\ x \})$ **by** *auto*
from ‹*finite ?E*› **have** *finite* $(g \text{ ` } ?E)$ **by** *auto*
from *this* **have** *mset-set* $((g \text{ ` } ?E) \cup \{ g\ x \} ) \subseteq\#$ *mset-set* $(g \text{ ` } ?E) + \{\#\ (g\ x)\ \#\}$
using *mset-set-insert-eq* [*of* $(g \text{ ` } ?E)$ $(g\ x)$] **by** *meson*
from *this i* **have** *ii*: *mset-set* $(g \text{ ` } (?E \cup \{ x \})) \subseteq\#$ *mset-set* $(g \text{ ` } ?E) + \{\#\ (g\ x)\ \#\}$ **by** *auto*
from *ind* **have** *mset-set* $(g \text{ ` } ?E) + \{\#\ (g\ x)\ \#\} \subseteq\#\ \{\#\ (g\ x).\ x \in\#$ *mset-set* $(?E)\ \#\} + \{\#\ (g\ x)\ \#\}$
using *Multiset.subset-mset.add-right-mono* **by** *metis*
from *this* **and** *ii* **have** *mset-set* $(g \text{ ` } (?E \cup \{ x \})) \subseteq\# \{\#\ (g\ x).\ x \in\#$ *mset-set* $(?E)\ \#\} + \{\#\ (g\ x)\ \#\}$
using *subset-mset.trans* [*of mset-set* $(g \text{ ` } (?E \cup \{ x \}))$] **by** *metis*
from *this* **and** ‹$E = ?E \cup \{ x \}$› ‹$\{\#\ (g\ x).\ x \in\#$ *mset-set* $(?E \cup \{ x \})\ \#\} = \{\#\ (g\ x).\ x \in\#$ *mset-set* $?E\ \#\} + \{\#\ (g\ x)\ \#\}$›
show *mset-set* $(g \text{ ` } E) \subseteq\# \{\#\ (g\ x).\ x \in\#$ *mset-set* $E\ \#\}$
by *metis*
qed

lemma *split-mset-set*:
assumes $C = C1 \cup C2$
assumes $C1 \cap C2 = \{\}$
assumes *finite C1*
assumes *finite C2*
shows (*mset-set C*) = (*mset-set C1*) + (*mset-set C2*)
proof (*rule ccontr*)
assume (*mset-set C*) $\neq$ (*mset-set C1*) + (*mset-set C2*)
then obtain *x* **where** *count* (*mset-set C*) *x* $\neq$ *count* ((*mset-set C1*) + (*mset-set C2*)) *x*
by (*meson multiset-eqI*)
from *assms*(*3*) *assms*(*4*) *assms*(*1*) **have** *finite C* **by** *auto*

have *count* ((*mset-set C1*) + (*mset-set C2*)) *x* = (*count* (*mset-set C1*) *x*) + (*count* (*mset-set C2*) *x*)
by *auto*
from *this* **and** ‹*count* (*mset-set C*) *x* $\neq$ *count* ((*mset-set C1*) + (*mset-set C2*)) *x*› **have**
count (*mset-set C*) *x* $\neq$ (*count* (*mset-set C1*) *x*) + (*count* (*mset-set C2*) *x*) **by** *auto*
have $x \in C1 \vee x \in C2$
proof (*rule ccontr*)
assume $\neg (x \in C1 \vee x \in C2)$
then have $x \notin C1$ **and** $x \notin C2$ **by** *auto*
from *assms*(*1*) ‹$x \notin C1$› **and** ‹$x \notin C2$› **have** $x \notin C$ **by** *auto*
from ‹$x \notin C1$› **have** (*count* (*mset-set C1*) *x*) = *0* **by** *auto*
from ‹$x \notin C2$› **have** (*count* (*mset-set C2*) *x*) = *0* **by** *auto*
from ‹$x \notin C$› **have** (*count* (*mset-set C*) *x*) = *0* **by** *auto*

```
    from ‹(count (mset-set C1) x) = 0› ‹(count (mset-set C2) x) = 0›
      ‹(count (mset-set C) x) = 0›
      ‹count (mset-set C) x ≠ (count (mset-set C1) x) + (count (mset-set C2) x)›
      show False by auto
  qed

  have (x ∉ C1 ∨ x ∈ C2)
  proof (rule ccontr)
    assume ¬ (x ∉ C1 ∨ x ∈ C2)
    then have x ∈ C1 and  x∉ C2 by auto
    from assms(1) ‹x ∈ C1› have x ∈ C by auto
    from assms(3) ‹x ∈ C1› have (count (mset-set C1) x) = 1 by auto
    from ‹x ∉ C2› have (count (mset-set C2) x) = 0 by auto
    from assms(3) assms(4) assms(1) have finite C by auto
    from ‹finite C› ‹x ∈ C› have (count (mset-set C) x) = 1 by auto
    from ‹(count (mset-set C1) x) = 1› ‹(count (mset-set C2) x) = 0›
      ‹(count (mset-set C) x) = 1›
      ‹count (mset-set C) x ≠ (count (mset-set C1) x) + (count (mset-set C2) x)›
      show False by auto
  qed
  have (x ∉ C2 ∨ x ∈ C1)
  proof (rule ccontr)
    assume ¬ (x ∉ C2 ∨ x ∈ C1)
    then have x ∈ C2 and  x∉ C1 by auto
    from assms(1) ‹x ∈ C2› have x ∈ C by auto
    from assms(4) ‹x ∈ C2› have (count (mset-set C2) x) = 1 by auto
    from ‹x ∉ C1› have (count (mset-set C1) x) = 0 by auto
    from ‹finite C› ‹x ∈ C› have (count (mset-set C) x) = 1 by auto
    from ‹(count (mset-set C2) x) = 1› ‹(count (mset-set C1) x) = 0›
      ‹(count (mset-set C) x) = 1›
      ‹count (mset-set C) x ≠ (count (mset-set C1) x) + (count (mset-set C2) x)›
      show False by auto
  qed
  from ‹x ∈ C1 ∨ x ∈ C2› ‹(x ∉ C1 ∨ x ∈ C2)› ‹(x ∉ C2 ∨ x ∈ C1)›
    have x ∈ C1 ∧ x ∈ C2 by blast
  from this and assms(2) show False by auto
qed

lemma image-mset-thm:
  assumes E = {# (f x). x ∈# E' #}
  assumes x ∈# E
  shows ∃ y. ((y ∈# E') ∧ x = (f y))
using assms by auto

lemma split-image-mset:
  assumes M = M1 + M2
  shows {# (f x). x ∈# M #} = {# (f x). x ∈# M1 #} + {# (f x). x ∈# M2
#}
by (simp add: assms)
```

end
theory *well-founded-continued*

imports *Main*

begin

1.2 Well-Founded Sets

Most useful lemmata are already proven in the Well_Founded theory available in Isabelle. We only establish a few convenient results for constructing well-founded sets and relations.

lemma *measure-wf*:
  assumes $wf\ (r :: ('a \times 'a)\ set)$
  assumes $r' = \{\ (x,y).\ ((m\ x),(m\ y)) \in r\ \}$
  shows $wf\ r'$
proof −
  have $(\forall\, Q{::}'b\ set.\ \forall\, x{::}\ 'b.\ x{\in}Q \longrightarrow (\exists\, z{\in}Q.\ \forall\, y.\ (y,z){\in}\ r'\ {-}{-}{>}\ y{\notin}Q))$
  proof $((rule\ allI){+},(rule\ impI))$
    fix $Q{::}\ 'b\ set$ **fix** $x{::}\ 'b$ **assume** $x{\in}Q$
    let $?Q' = (m\ `\ Q)$
    from ‹$x \in Q$› **have** *Q'-not-empty*: $m\ x \in\ ?Q'$ **by** *auto*
    from *assms*(*1*) **and** *Q'-not-empty* **obtain** z' **where** $z' \in\ ?Q'$ **and** $z'min$: $\forall\, y.$
$(y,z'){\in}\ r$
      $\longrightarrow y{\notin}?Q'$ **using** *wf-eq-minimal* [*of r*] **by** *blast*
    from ‹$z' \in\ ?Q'$› **obtain** z **where** $z' = (m\ z)$ **and** $z \in Q$ **by** *auto*
    have $\forall\, y.\ (y,z){\in}\ r' \longrightarrow y{\notin}Q$
    proof $((rule\ allI),(rule\ impI))$
      fix y **assume** $(y,z){\in}\ r'$
      from *assms*(*2*) **and** *this* **and** ‹$z' = (m\ z)$› **have** $((m\ y),z') \in r$ **by** *auto*
      from *this* **and** $z'min$ **have** $(m\ y) \notin\ ?Q'$ **by** *auto*
      then show $y{\notin}Q$ **by** *auto*
    qed
    from *this* **and** ‹$z \in Q$› **show** $(\exists\, z{\in}Q.\ \forall\, y.\ (y,z){\in}\ r'\ {-}{-}{>}\ y{\notin}Q)$ **by** *auto*
  qed
  then show *?thesis* **using** *wf-eq-minimal* **by** *auto*
qed

lemma *finite-proj-wf*:
  assumes *finite E*
  assumes $x \in E$
  assumes *acyclic r*
  shows $(\exists\ y.\ y \in E \wedge (\forall\ z.\ (z,\ y) \in r \longrightarrow z \notin E))$
proof −
  let $?r = \{\ (u,v).\ (u \in E \wedge v \in E \wedge (u,v) \in r)\ \}$
  from *assms*(*1*) **have** *finite* $(E \times E)$ **by** *auto*

have $?r \subseteq (E \times E)$ **by** *auto*
have $?r \subseteq r$ **by** *auto*
from ‹$?r \subseteq (E \times E)$› **and** ‹*finite* $(E \times E)$› **have** *finite* $?r$ **using** *finite-subset* **by** *auto*
from *assms*(3) **and** ‹$?r \subseteq r$› **have** *acyclic* $?r$ **unfolding** *acyclic-def* **using** *trancl-mono* **by** *blast*
from ‹*acyclic* $?r$› ‹*finite* $?r$› **have** *wf* $?r$ **using** *finite-acyclic-wf* **by** *auto*
from *this assms*(2) **obtain** y **where** $y \in E$ **and** i: $\bigwedge z.\ (z,\ y) \in ?r \Longrightarrow z \notin E$
using *wfE-min* [*of* $?r$ x E] **by** *blast*
have $\forall z.\ (z,\ y) \in r \longrightarrow z \notin E$
proof (*rule allI*,*rule impI*)
fix z **assume** $(z,y) \in r$
show $z \notin E$
proof
assume $z \in E$
from *this* **and** ‹$y \in E$› **and** ‹$(z,y) \in r$› **have** $(z,y) \in ?r$ **by** *auto*
from *this* **and** i [*of* z] **and** ‹$z \in E$› **show** *False* **by** *auto*
qed
qed
from *this* **and** ‹$y \in E$› **show** *?thesis* **by** *auto*
qed

end
theory *terms*

imports *HOL−ex.Unification*

begin

2 Terms

2.1 Basic Syntax

We use the same term representation as in the Unification theory provided in Isabelle. Terms are represented by binary trees built on variables and constant symbols.

fun *is-a-variable*
where
(*is-a-variable* (*Var* x)) = *True* |
(*is-a-variable* (*Const* x)) = *False* |
(*is-a-variable* (*Comb* x y)) = *False*

fun *is-a-constant*
where
(*is-a-constant* (*Var* x)) = *False* |
(*is-a-constant* (*Const* x)) = *True* |

(*is-a-constant* (*Comb* x y)) = *False*

fun *is-compound*
where
(*is-compound* (*Var* x)) = *False* |
(*is-compound* (*Const* x)) = *False* |
(*is-compound* (*Comb* x y)) = *True*

definition *ground-term* :: *'a trm* ⇒ *bool*
where
(*ground-term* t) = (*vars-of* t = {})

lemma *constants-are-not-variables* :
assumes *is-a-constant* x
shows ¬ (*is-a-variable* x)
by (*metis assms is-a-constant.elims*(*2*) *is-a-variable.elims*(*2*) *trm.distinct*(*2*))

lemma *constants-are-ground* :
assumes *is-a-constant* x
shows *ground-term* x
proof −
from *assms* **obtain** y **where** x = (*Const* y) **using** *is-a-constant.elims*(*2*) **by** *auto*
then show *?thesis* **by** (*simp add*: *ground-term-def*)
qed

2.2 Positions

We define the notion of a position together with functions to access to subterms and replace them. We establish some basic properties of these functions.

Since terms are binary trees, positions are sequences of binary digits.

datatype *indices* = *Left* | *Right*

type-synonym *position* = *indices list*

fun *left-app*
where *left-app* x = *Left* # x

fun *right-app*
where *right-app* x = *Right* # x

definition *strict-prefix*
where
strict-prefix p q = (∃ r. ($r \neq []$) ∧ (q = (*append* p r)))

fun *subterm* :: *'a trm* ⇒ *position* ⇒ *'a trm* ⇒ *bool*
where

$(subterm\ T\ []\ S) = (T = S)\ |$
$(subterm\ (Var\ v)\ (first\ \#\ next)\ S) = False\ |$
$(subterm\ (Const\ c)\ (first\ \#\ next)\ S) = False\ |$
$(subterm\ (Comb\ x\ y)\ (Left\ \#\ next)\ S) = (subterm\ x\ next\ S)\ |$
$(subterm\ (Comb\ x\ y)\ (Right\ \#\ next)\ S) = (subterm\ y\ next\ S)$

definition *occurs-in* :: $'a\ trm \Rightarrow 'a\ trm \Rightarrow bool$
where
occurs-in $t\ s = (\exists\, p.\ subterm\ s\ p\ t)$

definition *position-in* :: $position \Rightarrow 'a\ trm \Rightarrow bool$
where
position-in $p\ s = (\exists\, t.\ subterm\ s\ p\ t)$

fun *subterms-of*
where
subterms-of $t = \{\ s.\ (occurs\text{-}in\ s\ t)\ \}$

fun *proper-subterms-of*
where
proper-subterms-of $t = \{\ s.\ \exists\, p.\ (p \neq Nil \wedge (subterm\ t\ p\ s))\ \}$

fun *pos-of*
where
pos-of $t = \{\ p.\ (position\text{-}in\ p\ t)\ \}$

fun *replace-subterm* ::
$'a\ trm \Rightarrow position \Rightarrow 'a\ trm \Rightarrow 'a\ trm \Rightarrow bool$
where
$(replace\text{-}subterm\ T\ []\ u\ S) = (S = u)\ |$
$(replace\text{-}subterm\ (Var\ x)\ (first\ \#\ next)\ u\ S) = False\ |$
$(replace\text{-}subterm\ (Const\ c)\ (first\ \#\ next)\ u\ S) = False\ |$
$(replace\text{-}subterm\ (Comb\ x\ y)\ (Left\ \#\ next)\ u\ S) =$
$(\exists\, S1.\ (replace\text{-}subterm\ x\ next\ u\ S1) \wedge (S = Comb\ S1\ y))\ |$
$(replace\text{-}subterm\ (Comb\ x\ y)\ (Right\ \#\ next)\ u\ S) =$
$(\exists\, S2.\ (replace\text{-}subterm\ y\ next\ u\ S2) \wedge (S = Comb\ x\ S2))$

lemma *replace-subterm-is-a-function*:
shows $\bigwedge t\ u\ v.\ subterm\ t\ p\ u \Longrightarrow \exists\, s.\ replace\text{-}subterm\ t\ p\ v\ s$
proof (*induction p,auto*)
next case $(Cons\ i\ q)$
from ‹$subterm\ t\ (Cons\ i\ q)\ u$› **obtain** $t1\ t2$ **where** $t = (Comb\ t1\ t2)$
using *subterm.elims*(2) **by** *blast*
have $i = Right \vee i = Left$ **using** *indices.exhaust* **by** *auto*
then show *?case*
proof
assume $i = Right$
from *this* **and** ‹$t = (Comb\ t1\ t2)$› **and** ‹$subterm\ t\ (Cons\ i\ q)\ u$› **have** $subterm\ t2\ q\ u$ **by** *auto*

from *this* **obtain** *s* **where** *replace-subterm t2 q v s* **using** *Cons.IH* [*of t2 u*] **by** *auto*
from *this* **and** ‹$t = (Comb\ t1\ t2)$› **and** ‹$i = Right$› **have** *replace-subterm t* (*Cons i q*) *v* (*Comb t1 s*)
by *auto*
from *this* **show** *?case* **by** *auto*
next assume $i = Left$
from *this* **and** ‹$t = (Comb\ t1\ t2)$› **and** ‹*subterm t* (*Cons i q*) *u*› **have** *subterm t1 q u* **by** *auto*
from *this* **obtain** *s* **where** *replace-subterm t1 q v s* **using** *Cons.IH* [*of t1 u*] **by** *auto*
from *this* **and** ‹$t = (Comb\ t1\ t2)$› **and** ‹$i = Left$› **have** *replace-subterm t* (*Cons i q*) *v* (*Comb s t2*)
by *auto*
from *this* **show** *?case* **by** *auto*
qed
qed

We prove some useful lemmata concerning the set of variables or subterms occurring in a term.

lemma *root-subterm*:
shows $t \in$ (*subterms-of t*)
by (*metis mem-Collect-eq occurs-in-def subterm.simps*(*1*) *subterms-of.simps*)

lemma *root-position*:
shows *Nil* $\in$ (*pos-of t*)
by (*metis mem-Collect-eq subterm.simps*(*1*) *position-in-def pos-of.simps*)

lemma *subterms-of-an-atomic-term*:
assumes *is-a-variable t* $\vee$ *is-a-constant t*
shows *subterms-of t* $= \{ t \}$
proof
show *subterms-of t* $\subseteq \{ t \}$
proof
fix *x* **assume** $x \in$ *subterms-of t*
then have *occurs-in x t* **by** *auto*
then have $\exists p.$ (*subterm t p x*) **unfolding** *occurs-in-def* **by** *auto*
from *this* **and** *assms* **have** $x = t$
by (*metis is-a-constant.simps*(*3*) *is-a-variable.simps*(*3*) *subterm.elims*(*2*))
thus $x \in \{ t \}$ **by** *auto*
qed
next
show $\{ t \} \subseteq$ *subterms-of t*
proof
fix *x* **assume** $x \in \{ t \}$
then show $x \in$ *subterms-of t* **using** *root-subterm* **by** *auto*
qed
qed

```
lemma positions-of-an-atomic-term:
  assumes is-a-variable t ∨ is-a-constant t
  shows pos-of t = { Nil }
proof
  show pos-of t ⊆ { Nil }
  proof
    fix x assume x ∈ pos-of t
    then have position-in x t by auto
    then have ∃ s. (subterm t x s) unfolding position-in-def by auto
    from this and assms have x = Nil
     by (metis is-a-constant.simps(3) is-a-variable.simps(3) subterm.elims(2))
    thus x ∈ { Nil } by auto
  qed
next
  show { Nil } ⊆ pos-of t
  proof
    fix x :: indices list assume x ∈ { Nil }
    then show x ∈ pos-of t using root-position by auto
  qed
qed

lemma subterm-of-a-subterm-is-a-subterm :
  assumes subterm u q v
  shows ⋀ t. subterm t p u ⟹ subterm t (append p q) v
proof (induction p)
  case Nil
    show ?case using Nil.prems assms by auto
  next case (Cons i p′)
    from ‹subterm t (Cons i p′) u› obtain t1 t2 where t = (Comb t1 t2)
      using subterm.elims(2) by blast
    have i = Right ∨ i = Left using indices.exhaust by auto
    then show ?case
    proof
      assume i = Right
      from this and ‹subterm t (Cons i p′) u› and ‹t = (Comb t1 t2)›
        have subterm t2 p′ u by auto
      from this have subterm t2 (append p′ q) v by (simp add: Cons.IH)
     from this and ‹t = (Comb t1 t2)› and ‹i = Right› show subterm t (append
(Cons i p′) q) v
        by simp
    next assume i = Left
      from this and ‹subterm t (Cons i p′) u› and ‹t = (Comb t1 t2)›
        have subterm t1 p′ u by auto
      from this have subterm t1 (append p′ q) v by (simp add: Cons.IH)
      from this and ‹t = (Comb t1 t2)› and ‹i = Left› show subterm t (append
(Cons i p′) q) v
        by simp
    qed
qed
```

lemma *occur-in-subterm*:
  assumes *occurs-in u t*
  assumes *occurs-in t s*
  shows *occurs-in u s*
by (*meson assms*(*1*) *assms*(*2*) *occurs-in-def subterm-of-a-subterm-is-a-subterm*)

lemma *vars-of-subterm* :
  assumes $x \in$ *vars-of s*
  shows $\bigwedge$ *t. subterm t p s* $\Longrightarrow x \in$ *vars-of t*
proof (*induction p*)
  case *Nil*
    show *?case* **using** *Nil.prems assms* **by** *auto*
  next case (*Cons i p'*)
    from ‹*subterm t* (*Cons i p'*) *s*› **obtain** *t1 t2* **where** *t* = (*Comb t1 t2*)
      using *subterm.elims*(*2*) **by** *blast*
    have *i* = *Right* ∨ *i* = *Left* **using** *indices.exhaust* **by** *auto*
    then show *?case*
    proof
      assume *i* = *Right*
      from *this* **and** ‹*subterm t* (*Cons i p'*) *s*› **and** ‹*t* = (*Comb t1 t2*)›
        have *subterm t2 p' s* **by** *auto*
      from *this* **have** $x \in$ *vars-of t2* **by** (*simp add*: *Cons.IH*)
      from *this* **and** ‹*t* = (*Comb t1 t2*)› **and** ‹*i* = *Right*› **show** *?case*
        by *simp*
    next assume *i* = *Left*
      from *this* **and** ‹*subterm t* (*Cons i p'*) *s*› **and** ‹*t* = (*Comb t1 t2*)›
        have *subterm t1 p' s* **by** *auto*
      from *this* **have** $x \in$ *vars-of t1* **by** (*simp add*: *Cons.IH*)
      from *this* **and** ‹*t* = (*Comb t1 t2*)› **and** ‹*i* = *Left*› **show** *?case*
        by *simp*
    qed
qed

lemma *vars-subterm* :
  assumes *subterm t p s*
  shows *vars-of s* ⊆ *vars-of t*
by (*meson assms subsetI vars-of-subterm*)

lemma *vars-subterms-of* :
  assumes $s \in$ *subterms-of t*
  shows *vars-of s* ⊆ *vars-of t*
using *assms occurs-in-def vars-subterm* **by** *fastforce*

lemma *subterms-of-a-non-atomic-term*:
  shows *subterms-of* (*Comb t1 t2*) = (*subterms-of t1*) ∪ (*subterms-of t2*) ∪ { (*Comb t1 t2*) }
proof
  show *subterms-of* (*Comb t1 t2*) ⊆ (*subterms-of t1*) ∪ (*subterms-of t2*) ∪ { (*Comb

t1 t2) }
proof
fix *x* **assume** $x \in$ (*subterms-of* (*Comb t1 t2*))
then have *occurs-in x* (*Comb t1 t2*) **by** *auto*
then obtain *p* **where** *subterm* (*Comb t1 t2*) *p x* **unfolding** *occurs-in-def* **by** *auto*
have $p = [] \vee (\exists\, i\ q.\ p = i \# q)$ **using** *neq-Nil-conv* **by** *blast*
then show $x \in$ (*subterms-of t1*) $\cup$ (*subterms-of t2*) $\cup$ { (*Comb t1 t2*) }
proof
assume $p = []$
from *this* **and** ‹*subterm* (*Comb t1 t2*) *p x*› **show** *?thesis* **by** *auto*
next
assume $\exists\, i\ q.\ p = i \# q$
then obtain *i q* **where** $p = i \# q$ **by** *auto*
have $i = Left \vee i = Right$ **using** *indices.exhaust* **by** *blast*
then show $x \in$ (*subterms-of t1*) $\cup$ (*subterms-of t2*) $\cup$ { (*Comb t1 t2*) }
proof
assume $i = Left$
from *this* **and** ‹$p = i \# q$› **and** ‹*subterm* (*Comb t1 t2*) *p x*›
have *subterm t1 q x* **by** *auto*
then have *occurs-in x t1* **unfolding** *occurs-in-def* **by** *auto*
then show $x \in$ (*subterms-of t1*) $\cup$ (*subterms-of t2*) $\cup$ { (*Comb t1 t2*) } **by** *auto*
next
assume $i = Right$
from *this* **and** ‹$p = i \# q$› **and** ‹*subterm* (*Comb t1 t2*) *p x*›
have *subterm t2 q x* **by** *auto*
then have *occurs-in x t2* **unfolding** *occurs-in-def* **by** *auto*
then show $x \in$ (*subterms-of t1*) $\cup$ (*subterms-of t2*) $\cup$ { (*Comb t1 t2*) } **by** *auto*
qed
qed
qed
next
show (*subterms-of t1*) $\cup$ (*subterms-of t2*) $\cup$ { (*Comb t1 t2*) } $\subseteq$ *subterms-of* (*Comb t1 t2*)
proof
fix *x* **assume** $x \in$ (*subterms-of t1*) $\cup$ (*subterms-of t2*) $\cup$ { (*Comb t1 t2*) }
then have $x \in$ (*subterms-of t1*) $\vee$ ($x \in$ (*subterms-of t2*) $\vee$ $x =$ (*Comb t1 t2*)) **by** *auto*
thus $x \in$ *subterms-of* (*Comb t1 t2*)
proof
assume $x \in$ (*subterms-of t1*)
then have *occurs-in x t1* **by** *auto*
then obtain *p* **where** *subterm t1 p x* **unfolding** *occurs-in-def* **by** *auto*
then have *subterm* (*Comb t1 t2*) (*Left* # *p*) *x* **by** *auto*
then have *occurs-in x* (*Comb t1 t2*) **using** *occurs-in-def* **by** *blast*
then show $x \in$ *subterms-of* (*Comb t1 t2*) **by** *auto*
next

assume $(x \in (\textit{subterms-of } t2) \lor x = (\textit{Comb } t1\ t2))$
then show $x \in \textit{subterms-of } (\textit{Comb } t1\ t2)$
proof
assume $x \in (\textit{subterms-of } t2)$
then have *occurs-in* x $t2$ **by** *auto*
then obtain p **where** *subterm* $t2$ p x **unfolding** *occurs-in-def* **by** *auto*
then have *subterm* $(\textit{Comb } t1\ t2)$ $(\textit{Right} \# p)$ x **by** *auto*
then have *occurs-in* x $(\textit{Comb } t1\ t2)$ **using** *occurs-in-def* **by** *blast*
then show $x \in \textit{subterms-of } (\textit{Comb } t1\ t2)$ **by** *auto*
next
assume $x = (\textit{Comb } t1\ t2)$
show $x \in \textit{subterms-of } (\textit{Comb } t1\ t2)$ **using** $\langle x = t1 \cdot t2\rangle$ *root-subterm* **by** *blast*
qed
qed
qed
qed

lemma *positions-of-a-non-atomic-term*:
shows $\textit{pos-of } (\textit{Comb } t1\ t2) = (\textit{left-app } ` (\textit{pos-of } t1)) \cup (\textit{right-app } ` (\textit{pos-of } t2)) \cup \{ \textit{Nil} \}$
proof
show $\textit{pos-of } (\textit{Comb } t1\ t2) \subseteq (\textit{left-app } ` (\textit{pos-of } t1)) \cup (\textit{right-app } ` (\textit{pos-of } t2)) \cup \{ \textit{Nil} \}$
proof
fix x **assume** $x \in \textit{pos-of } (\textit{Comb } t1\ t2)$
then have *position-in* x $(\textit{Comb } t1\ t2)$ **by** *auto*
then obtain s **where** *subterm* $(\textit{Comb } t1\ t2)$ x s **unfolding** *position-in-def* **by** *auto*
have $x = [] \lor (\exists\, i\ q.\ x = i \# q)$ **using** *neq-Nil-conv* **by** *blast*
then show $x \in (\textit{left-app } ` (\textit{pos-of } t1)) \cup (\textit{right-app } ` (\textit{pos-of } t2)) \cup \{ \textit{Nil} \}$
proof
assume $x = []$
from *this* **and** $\langle \textit{subterm } (\textit{Comb } t1\ t2)\ x\ s\rangle$ **show** *?thesis* **by** *auto*
next
assume $\exists\, i\ q.\ x = i \# q$
then obtain i q **where** $x = i \# q$ **by** *auto*
have $i = \textit{Left} \lor i = \textit{Right}$ **using** *indices.exhaust* **by** *blast*
then show $x \in (\textit{left-app } ` (\textit{pos-of } t1)) \cup (\textit{right-app } ` (\textit{pos-of } t2)) \cup \{ \textit{Nil} \}$
proof
assume $i = \textit{Left}$
from *this* **and** $\langle x = i \# q\rangle$ **and** $\langle \textit{subterm } (\textit{Comb } t1\ t2)\ x\ s\rangle$
have *subterm* $t1$ q s **by** *auto*
then have *position-in* q $t1$ **unfolding** *position-in-def* **by** *auto*
from *this* **and** $\langle x = i \# q\rangle$ $\langle i = \textit{Left}\rangle$
show $x \in (\textit{left-app } ` (\textit{pos-of } t1)) \cup (\textit{right-app } ` (\textit{pos-of } t2)) \cup \{ \textit{Nil} \}$ **by** *auto*
next
assume $i = \textit{Right}$

from *this* **and** ‹$x = i \# q$› **and** ‹*subterm* (*Comb t1 t2*) *x s*›
have *subterm t2 q s* **by** *auto*
then have *position-in q t2* **unfolding** *position-in-def* **by** *auto*
from *this* **and** ‹$x = i \# q$› ‹$i = Right$›
show $x \in$ (*left-app* ‘ (*pos-of t1*)) $\cup$ (*right-app* ‘ (*pos-of t2*)) $\cup$ { *Nil* } **by** *auto*
qed
qed
qed
next
show (*left-app* ‘ (*pos-of t1*)) $\cup$ (*right-app* ‘ (*pos-of t2*)) $\cup$ { *Nil* } $\subseteq$ *pos-of* (*Comb t1 t2*)
proof
fix *x* **assume** $x \in$ (*left-app* ‘ (*pos-of t1*)) $\cup$ (*right-app* ‘ (*pos-of t2*)) $\cup$ { *Nil* }
then have ($x \in$ *left-app* ‘ (*pos-of t1*)) $\vee$ (($x \in$ (*right-app* ‘ (*pos-of t2*))) $\vee$ (x = *Nil*)) **by** *auto*
thus $x \in$ *pos-of* (*Comb t1 t2*)
proof
assume $x \in$ *left-app* ‘ (*pos-of t1*)
then obtain *q* **where** $x = Left \# q$ **and** *position-in q t1* **by** *auto*
then obtain *s* **where** *subterm t1 q s* **unfolding** *position-in-def* **by** *auto*
then have *subterm* (*Comb t1 t2*) (*Left* # *q*) *s* **by** *auto*
from *this* **and** ‹$x = Left \# q$› **have** *position-in x* (*Comb t1 t2*) **using** *position-in-def* **by** *blast*
then show $x \in$ *pos-of* (*Comb t1 t2*) **by** *auto*
next
assume ($x \in$ (*right-app* ‘ (*pos-of t2*))) $\vee$ ($x = Nil$)
then show $x \in$ *pos-of* (*Comb t1 t2*)
proof
assume $x \in$ *right-app* ‘ (*pos-of t2*)
then obtain *q* **where** $x = Right \# q$ **and** *position-in q t2* **by** *auto*
then obtain *s* **where** *subterm t2 q s* **unfolding** *position-in-def* **by** *auto*
then have *subterm* (*Comb t1 t2*) (*Right* # *q*) *s* **by** *auto*
from *this* **and** ‹$x = Right \# q$› **have** *position-in x* (*Comb t1 t2*) **using** *position-in-def* **by** *blast*
then show $x \in$ *pos-of* (*Comb t1 t2*) **by** *auto*
next
assume $x = Nil$
show $x \in$ *pos-of* (*Comb t1 t2*) **using** ‹$x = Nil$› *root-position* **by** *blast*
qed
qed
qed
qed

lemma *set-of-subterms-is-finite* :
shows (*finite* (*subterms-of* (t :: $'a$ *trm*)))
proof (*induction t*)
case (*Var x*)
then show *?case* **using** *subterms-of-an-atomic-term* [*of Var x*] **by** *simp*

next
case (*Const x*)
then show *?case* **using** *subterms-of-an-atomic-term* [*of Const x*] **by** *simp*
next
case (*Comb t1 t2*) **assume** *finite* (*subterms-of t1*) **and** *finite* (*subterms-of t2*)
have *subterms-of* (*Comb t1 t2*) = *subterms-of t1* ∪ *subterms-of t2* ∪ { *Comb t1 t2* }
using *subterms-of-a-non-atomic-term* **by** *auto*
from *this* **and** ‹*finite* (*subterms-of t1*)› **and** ‹*finite* (*subterms-of t2*)›
show *finite* (*subterms-of* (*Comb t1 t2*)) **by** *simp*
qed

lemma *set-of-positions-is-finite* :
shows (*finite* (*pos-of* (*t* :: *'a trm*)))
proof (*induction t*)
case (*Var x*)
then show *?case* **using** *positions-of-an-atomic-term* [*of Var x*] **by** *simp*
next
case (*Const x*)
then show *?case* **using** *positions-of-an-atomic-term* [*of Const x*] **by** *simp*
next
case (*Comb t1 t2*) **assume** *finite* (*pos-of t1*) **and** *finite* (*pos-of t2*)
from ‹*finite* (*pos-of t1*)› **have** *i*: *finite* (*left-app* ‘ (*pos-of t1*)) **by** *auto*
from ‹*finite* (*pos-of t2*)› **have** *ii*: *finite* (*right-app* ‘ (*pos-of t2*)) **by** *auto*
have *pos-of* (*Comb t1 t2*) = (*left-app* ‘ (*pos-of t1*)) ∪ (*right-app* ‘ (*pos-of t2*)) ∪ { *Nil* }
using *positions-of-a-non-atomic-term* **by** *metis*
from *this* **and** *i ii* **show** *finite* (*pos-of* (*Comb t1 t2*)) **by** *simp*
qed

lemma *vars-of-instances*:
shows *vars-of* (*subst t σ*)
$= \bigcup$ { *V*. ∃ *x*. (*x* ∈ (*vars-of t*)) ∧ (*V* = *vars-of* (*subst* (*Var x*) *σ*)) }
proof (*induction t*)
case (*Const a*)
have *vars-of* (*Const a*) = {} **by** *auto*
then have *rhs-empty*: $\bigcup$ { *V*. ∃ *x*. (*x* ∈ (*vars-of* (*Const a*))) ∧ (*V* = *vars-of* (*subst* (*Var x*) *σ*)) } = {} **by** *auto*
have *lhs-empty*: (*subst* (*Const a*) *σ*) = (*Const a*) **by** *auto*
from *rhs-empty* **and** *lhs-empty* **show** *?case* **by** *auto*
next
case (*Var a*)
have *vars-of* (*Var a*) = { *a* } **by** *auto*
then have *rhs*: $\bigcup$ { *V*. ∃ *x*. (*x* ∈ (*vars-of* (*Var a*))) ∧ (*V* = *vars-of* (*subst* (*Var x*) *σ*)) } =
vars-of (*subst* (*Var a*) *σ*) **by** *auto*
have *lhs*: (*subst* (*Var a*) *σ*) = (*subst* (*Var a*) *σ*) **by** *auto*
from *rhs* **and** *lhs* **show** *?case* **by** *auto*
next

case (*Comb t1 t2*)
have *vars-of* (*Comb t1 t2*) = (*vars-of t1*) ∪ (*vars-of t2*) **by** *auto*
then have $\bigcup$ { *V*. ∃ *x*. (*x* ∈ (*vars-of* (*Comb t1 t2*))) ∧ (*V* = *vars-of* (*subst* (*Var x*) σ)) }
= $\bigcup$ { *V*. ∃ *x*. (*x* ∈ (*vars-of t1*)) ∧ (*V* = *vars-of* (*subst*(*Var x*) σ)) }
∪ $\bigcup$ { *V*. ∃ *x*. (*x* ∈ (*vars-of t2*)) ∧ (*V* = *vars-of* (*subst* (*Var x*) σ)) }
by *auto*
then have *rhs*: $\bigcup$ { *V*. ∃ *x*. (*x* ∈ (*vars-of* (*Comb t1 t2*))) ∧ (*V* = *vars-of* (*subst* (*Var x*) σ)) }
= (*vars-of* (*subst t1* σ)) ∪ (*vars-of* (*subst t2* σ))
using ‹*vars-of* (*subst t1* σ)
= $\bigcup$ { *V*. ∃ *x*. (*x* ∈ (*vars-of t1*)) ∧ (*V* = *vars-of* (*subst* (*Var x*) σ)) }›
and
‹*vars-of* (*subst t2* σ)
= $\bigcup$ { *V*. ∃ *x*. (*x* ∈ (*vars-of t2*)) ∧ (*V* = *vars-of* (*subst* (*Var x*) σ)) }›
by *auto*
have (*subst* (*Comb t1 t2*) σ) = (*Comb* (*subst t1* σ) (*subst t2* σ))
by *auto*
then have *lhs*: (*vars-of* (*subst* (*Comb t1 t2*) σ)) =
(*vars-of* (*subst t1* σ)) ∪ (*vars-of* (*subst t2* σ)) **by** *auto*
from *lhs* **and** *rhs* **show** *?case* **by** *auto*
qed

lemma *subterms-of-instances* :
∀ *u v u′ s*. (*u* = (*subst v s*) ⟶ (*subterm u p u′*)
⟶ (∃ *x q1 q2*. (*is-a-variable x*) ∧ (*subterm* (*subst x s*) *q1 u′*) ∧
(*subterm v q2 x*) ∧ (*p* = (*append q2 q1*))) ∨
((∃ *v′*. ((¬ *is-a-variable v′*) ∧ (*subterm v p v′*) ∧ (*u′* = (*subst v′ s*)))))) (**is** *?prop p*)
proof (*induction p*)
case *Nil*
show *?case*
proof ((*rule allI*)+,(*rule impI*)+)
fix *u* ::′*a trm* **fix** *v u′ s* **assume** *u* = (*subst v s*) **and** *subterm u* [] *u′*
then have *u* = *u′* **by** *auto*
show (∃ *x q1 q2*. (*is-a-variable x*) ∧ (*subterm* (*subst x s*) *q1 u′*) ∧
(*subterm v q2 x*) ∧ ([] = (*append q2 q1*))) ∨
((∃ *v′*. ((¬ *is-a-variable v′*) ∧ (*subterm v* [] *v′*) ∧ (*u′* = (*subst v′ s*)))))
proof (*cases*)
assume *is-a-variable v*
from ‹*u* = *u′*›**and** ‹*u* = (*subst v s*)›
have (*subterm* (*subst v s*) [] *u′*) **by** *auto*
have *subterm v* [] *v* **by** *auto*
from *this* **and** ‹(*subterm* (*subst v s*) [] *u′*)› **and** ‹*is-a-variable v*›
show *?thesis* **by** *auto*
next assume ¬ *is-a-variable v*
from ‹*u* = *u′*› **and** ‹*u* = (*subst v s*)›
have ((*subterm v* [] *v*) ∧ (*u′* = (*subst v s*))) **by** *auto*
then show *?thesis* **by** *auto*

qed
qed
next
case (*Cons i q*)
show *?case*
proof ((*rule allI*)+,(*rule impI*)+)
fix u ::'*a trm* **fix** $v\ u'\ s$ **assume** $u = (subst\ v\ s)$
and *subterm* u (*Cons i q*) u'
show ($\exists x\ q1\ q2.$ (*is-a-variable* x) $\wedge$ (*subterm* (*subst* $x\ s$) $q1\ u'$) $\wedge$
(*subterm* $v\ q2\ x$) $\wedge$ ((*Cons i q*) = (*append q2 q1*))) $\vee$
(($\exists$ v'. (($\neg$ *is-a-variable* v') $\wedge$ (*subterm* v (*Cons i q*) v') $\wedge$ (u' = (*subst* v' s)))))
proof (*cases v*)
fix x **assume** v = (*Var x*)
then have *subterm* v [] v **by** *auto*
from ‹v = (*Var x*)› **have** *is-a-variable* v **by** *auto*
have *Cons i q* = (*append* [] (*Cons i q*)) **by** *auto*
from ‹*subterm* u (*Cons i q*) u'› **and** ‹u = (*subst v s*)›
and ‹v = (*Var x*)› **have** *subterm* (*subst v s*) (*Cons i q*) u' **by** *auto*
from ‹*is-a-variable v*› **and** ‹*subterm v* [] *v*› **and** ‹*Cons i q* = (*append* [] (*Cons i q*))› **and** *this*
show *?thesis* **by** *blast*
next
fix x **assume** v = (*Const x*)
from *this* **and** ‹u = (*subst v s*)› **have** $u = v$ **by** *auto*
from *this* **and** ‹v = (*Const x*)› **and** ‹*subterm* u (*Cons i q*) u'› **show** *?thesis* **by** *auto*
next
fix *t1 t2* **assume** v = (*Comb t1 t2*)
from *this* **and** ‹u = (*subst v s*)›
have u = (*Comb* (*subst t1 s*) (*subst t2 s*)) **by** *auto*
have i = *Left* $\vee$ i = *Right* **using** *indices.exhaust* **by** *auto*
from ‹i = *Left* $\vee$ i = *Right*› **and** ‹u = (*Comb* (*subst t1 s*) (*subst t2 s*))›
and ‹*subterm* u (*Cons i q*) u'› **obtain** *ti* **where**
subterm (*subst ti s*) $q\ u'$ **and** *ti* = *t1* $\vee$ *ti* = *t2* **and** *subterm* v [i] *ti*
using ‹v = *t1* · *t2*› **by** *auto*
from ‹*?prop q*› **and** ‹*subterm* (*subst ti s*) $q\ u'$› **have**
($\exists x\ q1\ q2.$ (*is-a-variable* x) $\wedge$ (*subterm* (*subst* $x\ s$) $q1\ u'$) $\wedge$
(*subterm ti q2 x*) $\wedge$ (q = (*append q2 q1*))) $\vee$
(($\exists$ v'. (($\neg$ *is-a-variable* v') $\wedge$ (*subterm ti q* v') $\wedge$ (u' = (*subst* v' s)))))
by *auto*
then show *?thesis*
proof
assume ($\exists x\ q1\ q2.$ (*is-a-variable* x) $\wedge$ (*subterm* (*subst* $x\ s$) $q1\ u'$) $\wedge$
(*subterm ti q2 x*) $\wedge$ (q = (*append q2 q1*)))
then obtain *x q1 q2* **where** *is-a-variable x* **and** *subterm* (*subst x s*) *q1* u'
and *subterm ti q2 x* **and** q = (*append q2 q1*)
by *auto*

from ‹*subterm ti q2 x*› **and** ‹*subterm v* [*i*] *ti*› **have** *subterm v* (*i* # *q2*) *x*
using ‹*i* = *indices.Left* ∨ *i* = *indices.Right*› ‹*v* = *t1* · *t2*› **by** *auto*
from ‹*q* = *append q2 q1*› **have** *i* # *q* = (*append* (*i* # *q2*) *q1*) **by** *auto*
from ‹*i* # *q* = (*append* (*i* # *q2*) *q1*)› **and** ‹*is-a-variable x*›
and ‹*subterm* (*subst x s*) *q1 u′*› **and** ‹*subterm v* (*i* # *q2*) *x*›
show *?thesis* **by** *blast*
next
assume ((∃ *v′*. ((¬ *is-a-variable v′*) ∧ (*subterm ti q v′*) ∧ (*u′* = (*subst v′ s*)))))
then obtain *v′* **where** (¬ *is-a-variable v′*) (*subterm ti q v′*) **and** *u′* = (*subst v′ s*) **by** *auto*
from ‹*subterm ti q v′*› **and** ‹*subterm v* [*i*] *ti*› **have** *subterm v* (*i* # *q*) *v′*
using ‹*i* = *indices.Left* ∨ *i* = *indices.Right*› ‹*v* = *t1* · *t2*› **by** *auto*
from *this* **and** ‹(¬ *is-a-variable v′*)› ‹*subterm ti q v′*› **and** ‹*u′* = (*subst v′ s*)›
show *?thesis* **by** *auto*
qed
qed
qed
qed

lemma *vars-of-replacement*:
shows ⋀ *t s*. *x* ∈ *vars-of s* ⟹ *replace-subterm t p v s* ⟹ *x* ∈ (*vars-of t*) ∪ (*vars-of v*)
proof (*induction p*)
case *Nil*
from ‹*replace-subterm t Nil v s*› **have** *s* = *v* **by** *auto*
from *this* **and** ‹*x* ∈ *vars-of s*› **show** *?case* **by** *auto*
next case (*Cons i q*)
from ‹*replace-subterm t* (*Cons i q*) *v s*› **obtain** *t1 t2* **where**
t = (*Comb t1 t2*)
by (*metis is-a-variable.cases replace-subterm.simps*(*2*) *replace-subterm.simps*(*3*))

have *i* = *Left* ∨ *i* = *Right* **using** *indices.exhaust* **by** *blast*
then show *?case*
proof
assume *i* = *Left*
from *this* **and** ‹*t* = *Comb t1 t2*› **and** ‹*replace-subterm t* (*Cons i q*) *v s*›
obtain *s1* **where** *s* = *Comb s1 t2* **and** *replace-subterm t1 q v s1*
using *replace-subterm.simps*(*4*) **by** *auto*
from ‹*s* = *Comb s1 t2*› **and** ‹*x* ∈ *vars-of s*› **have** *x* ∈ *vars-of s1* ∨ *x* ∈ *vars-of t2*
by *simp*
then show *?case*
proof
assume *x* ∈ *vars-of s1*
from *this* **and** *Cons.IH* [*of s1 t1*] **and** ‹*replace-subterm t1 q v s1*› **have** *x* ∈ (*vars-of t1*) ∪ (*vars-of v*)
by *auto*

from *this* **and** ‹$t = (Comb\ t1\ t2)$› **show** *?case* **by** *auto*
next
assume $x \in$ *vars-of t2*
from *this* **and** ‹$t = (Comb\ t1\ t2)$› **show** *?case* **by** *auto*
qed
next
assume $i = Right$
from *this* **and** ‹$t = Comb\ t1\ t2$› **and** ‹*replace-subterm t* (*Cons i q*) *v s*›
obtain *s2* **where** $s = Comb\ t1\ s2$ **and** *replace-subterm t2 q v s2*
using *replace-subterm.simps* **by** *auto*
from ‹$s = Comb\ t1\ s2$› **and** ‹$x \in$ *vars-of s*› **have** $x \in$ *vars-of t1* $\vee$ $x \in$ *vars-of s2*
by *simp*
then show *?case*
proof
assume $x \in$ *vars-of s2*
from *this* **and** *Cons.IH* [*of s2 t2*] **and** ‹*replace-subterm t2 q v s2*› **have** $x \in$ (*vars-of t2*) $\cup$ (*vars-of v*)
by *auto*
from *this* **and** ‹$t = (Comb\ t1\ t2)$› **show** *?case* **by** *auto*
next
assume $x \in$ *vars-of t1*
from *this* **and** ‹$t = (Comb\ t1\ t2)$› **show** *?case* **by** *auto*
qed
qed
qed

lemma *vars-of-replacement-set*:
assumes *replace-subterm t p v s*
shows *vars-of s* $\subseteq$ (*vars-of t*) $\cup$ (*vars-of v*)
by (*meson assms subsetI vars-of-replacement*)

2.3 Substitutions and Most General Unifiers

Substitutions are defined in the Unification theory. We provide some additional definitions and lemmata.

fun *subst-set* :: *'a trm set => 'a subst => 'a trm set*
where
(*subst-set S* σ) = { $u.\ \exists t.\ u = (subst\ t\ \sigma) \wedge t \in S$ }

definition *subst-codomain*
where
subst-codomain σ V = { $x.\ \exists y.\ (subst\ (Var\ y)\ \sigma) = (Var\ x) \wedge (y \in V)$ }

lemma *subst-codomain-is-finite*:
assumes *finite A*
shows *finite* (*subst-codomain* η *A*)
proof –
have *i*: $((\lambda x.\ (Var\ x))$ ‘ (*subst-codomain* η *A*)) $\subseteq$ $((\lambda x.\ (subst\ (Var\ x)\ \eta))$ ‘ *A*)

proof
fix *x* **assume** $x \in ((\lambda x.\ (Var\ x))$ ‘ $(subst\text{-}codomain\ \eta\ A))$
from *this* **obtain** *y* **where** $y \in (subst\text{-}codomain\ \eta\ A)$ **and** $x = (Var\ y)$ **by** *auto*
from ‹$y \in (subst\text{-}codomain\ \eta\ A)$› **obtain** *z* **where** $(subst\ (Var\ z)\ \eta) = (Var\ y)\ (z \in A)$
unfolding *subst-codomain-def* **by** *auto*
from ‹$(z \in A)$› ‹$x = (Var\ y)$› ‹$(subst\ (Var\ z)\ \eta) = (Var\ y)$› *this* **show**
$x \in ((\lambda x.\ (subst\ (Var\ x)\ \eta))$ ‘ $A)$**using** *image-iff* **by** *fastforce*
qed
have *inj-on* $(\lambda x.\ (Var\ x))\ (subst\text{-}codomain\ \eta\ A)$ **by** (*meson inj-onI trm.inject*(*1*))
from *assms*(*1*) **have** *finite* $((\lambda x.\ (subst\ (Var\ x)\ \eta))$ ‘ $A)$ **by** *auto*
from *this* **and** *i* **have** *finite* $((\lambda x.\ (Var\ x))$ ‘ $(subst\text{-}codomain\ \eta\ A))$
using *rev-finite-subset* **by** *auto*
from *this* **and** ‹*inj-on* $(\lambda x.\ (Var\ x))\ (subst\text{-}codomain\ \eta\ A)$› **show** *?thesis* **using** *finite-imageD* [*of* $(\lambda x.\ (Var\ x))$ *subst-codomain* η *A*]
by *auto*
qed

The notions of unifiers, most general unifiers, the unification algorithm and a proof of correctness are provided in the Unification theory. Below, we prove that the algorithm is complete.

lemma *subt-decompose*:
shows $\forall\, t1\ t2.\ Comb\ t1\ t2 \prec s \longrightarrow (t1 \prec s \wedge t2 \prec s)$
proof ((*induction s*),(*simp*),(*simp*))
case (*Comb s1 s2*)
show *?case*
proof ((*rule allI*)+,(*rule impI*))
fix *t1 t2* **assume** $Comb\ t1\ t2 \prec Comb\ s1\ s2$
show $t1 \prec (Comb\ s1\ s2) \wedge t2 \prec (Comb\ s1\ s2)$
proof (*rule ccontr*)
assume *neg*: $\neg(t1 \prec (Comb\ s1\ s2) \wedge t2 \prec (Comb\ s1\ s2))$
from ‹$Comb\ t1\ t2 \prec Comb\ s1\ s2$› **have**
d: $Comb\ t1\ t2 = s1 \vee Comb\ t1\ t2 = s2 \vee Comb\ t1\ t2 \prec s1 \vee Comb\ t1\ t2 \prec s2$
by *auto*
have *i*: $\neg\ (Comb\ t1\ t2 = s1)$
proof
assume $(Comb\ t1\ t2 = s1)$
then have $t1 \prec s1$ **and** $t2 \prec s1$ **by** *auto*
from *this* **and** *neg* **show** *False* **by** *auto*
qed
have *ii*: $\neg\ (Comb\ t1\ t2 = s2)$
proof
assume $(Comb\ t1\ t2 = s2)$
then have $t1 \prec s2$ **and** $t2 \prec s2$ **by** *auto*
from *this* **and** *neg* **show** *False* **by** *auto*
qed
have *iii*: $\neg\ (Comb\ t1\ t2 \prec s1)$
proof

```
        assume (Comb t1 t2 ≺ s1)
        then have t1 ≺ s1 ∧ t2 ≺ s1 using Comb.IH by metis
        from this and neg  show False by auto
      qed
      have iv: ¬ (Comb t1 t2 ≺ s2)
      proof
        assume (Comb t1 t2 ≺ s2)
        then have t1 ≺ s2  ∧ t2 ≺ s2 using Comb.IH by metis
        from this and neg show False by auto
      qed
      from d and i ii iii iv show False by auto
    qed
  qed
qed

lemma subt-irrefl:
  shows ¬ (s ≺ s)
proof ((induction s),(simp),(simp))
    case (Comb t1 t2)
     show ?case
    proof
      assume Comb t1 t2 ≺ Comb t1 t2
      then have i: Comb t1 t2 ≠ t1 using Comb.IH(1) by fastforce
     from ‹Comb t1 t2 ≺ Comb t1 t2› have ii: Comb t1 t2 ≠ t2 using Comb.IH(2)
by fastforce
      from i ii and ‹Comb t1 t2 ≺ Comb t1 t2› have Comb t1 t2 ≺ t1 ∨ Comb
t1 t2 ≺ t2 by auto
      then show False
      proof
        assume Comb t1 t2  ≺ t1
        then have t1 ≺ t1  using subt-decompose [of t1] by metis
        from this show False using Comb.IH by auto
      next
        assume Comb t1 t2  ≺ t2
        then have t2 ≺ t2  using subt-decompose [of t2] by metis
        from this show False using Comb.IH by auto
      qed
    qed
qed

lemma MGU-exists:
  shows ∀σ. ((subst t σ) = (subst s σ) ⟶ unify t s ≠ None)
proof (rule unify.induct)
     fix x s1 s2 show ∀σ :: 'a subst .((subst (Const x) σ) = (subst (Comb s1 s2)
σ)
      ⟶ unify (Const x) (Comb s1 s2) ≠ None) by simp
    next
    fix t1 t2 y show ∀σ :: 'a subst.(subst (Comb t1 t2) σ) = (subst (Const y) σ)
      ⟶ unify (Comb t1 t2) (Const y) ≠ None by simp
```

next
fix $x\ y$ **show** $\forall\sigma :: {'a}\ subst.(subst\ (Const\ x)\ \sigma) = (subst\ (Var\ y)\ \sigma)$
$\longrightarrow unify\ (Const\ x)\ (Var\ y) \neq None$ **using** *unify.simps(3)* **by** *fastforce*
next
fix $t1\ t2\ y$ **show** $\forall\sigma :: {'a}\ subst.(subst\ (Comb\ t1\ t2)\ \sigma) = (subst\ (Var\ y)\ \sigma)$
$\longrightarrow unify\ (Comb\ t1\ t2)\ (Var\ y) \neq None$
by (*metis option.distinct(1) subst-mono subt-irrefl unify.simps(4)*)
next
fix $x\ s$ **show** $\forall\sigma :: {'a}\ subst.(subst\ (Var\ x)\ \sigma) = (subst\ s\ \sigma)$
$\longrightarrow unify\ (Var\ x)\ s \neq None$
by (*metis option.distinct(1) subst-mono subt-irrefl unify.simps(5)*)
next
fix $x\ y$ **show** $\forall\sigma :: {'a}\ subst.(subst\ (Const\ x)\ \sigma) = (subst\ (Const\ y)\ \sigma)$
$\longrightarrow unify\ (Const\ x)\ (Const\ y) \neq None$ **by** *simp*
next
fix $t1\ t2\ s1\ s2$
show $\forall\sigma :: {'a}\ subst.\ (subst\ t1\ \sigma = subst\ s1\ \sigma \longrightarrow unify\ t1\ s1 \neq None) \Longrightarrow$
$(\bigwedge x2.\ unify\ t1\ s1 = Some\ x2 \Longrightarrow$
$\forall\sigma.\ subst\ (t2 \lhd x2)\ \sigma = subst\ (s2 \lhd x2)\ \sigma \longrightarrow$
$unify\ (t2 \lhd x2)\ (s2 \lhd x2) \neq None) \Longrightarrow$
$\forall\sigma.\ (subst\ (t1 \cdot t2)\ \sigma = subst\ (s1 \cdot s2)\ \sigma \longrightarrow$
$unify\ (t1 \cdot t2)\ (s1 \cdot s2) \neq None)$
proof −
assume $h1$: $\forall\sigma.\ (subst\ t1\ \sigma = subst\ s1\ \sigma \longrightarrow unify\ t1\ s1 \neq None)$
assume $h2$: $(\bigwedge x2.\ unify\ t1\ s1 = Some\ x2 \Longrightarrow$
$\forall\sigma.\ subst\ (t2 \lhd x2)\ \sigma = subst\ (s2 \lhd x2)\ \sigma \longrightarrow$
$unify\ (t2 \lhd x2)\ (s2 \lhd x2) \neq None)$
show $\forall\sigma.\ (subst\ (t1 \cdot t2)\ \sigma = subst\ (s1 \cdot s2)\ \sigma \longrightarrow$
$unify\ (t1 \cdot t2)\ (s1 \cdot s2) \neq None)$
proof ((*rule allI*),(*rule impI*))
fix σ **assume** $h3$: $subst\ (t1 \cdot t2)\ \sigma = subst\ (s1 \cdot s2)\ \sigma$
from $h3$ **have** $subst\ t1\ \sigma = subst\ s1\ \sigma$ **by** *auto*
from *this* **and** $h1$ **have** $unify\ t1\ s1 \neq None$ **by** *auto*
from *this* **obtain** ϑ **where** $unify\ t1\ s1 = Some\ \vartheta$ **and** $MGU\ \vartheta\ t1\ s1$
by (*meson option.exhaust unify-computes-MGU*)
from $\langle subst\ t1\ \sigma = subst\ s1\ \sigma\rangle$ **have** $Unifier\ \sigma\ t1\ s1$
unfolding *Unifier-def* **by** *auto*
from *this* **and** $\langle MGU\ \vartheta\ t1\ s1\rangle$ **obtain** η **where** $\sigma \doteq \vartheta \lozenge \eta$ **using** *MGU-def*
by *metis*
from $h3$ **have** $subst\ t2\ \sigma = subst\ s2\ \sigma$ **by** *auto*
from *this* **and** $\langle\sigma \doteq \vartheta \lozenge \eta\rangle$
have $subst\ (subst\ t2\ \vartheta)\ \eta$
$= subst\ (subst\ s2\ \vartheta)\ \eta$
by (*simp add*: *subst-eq-def*)
from *this* **and** $\langle unify\ t1\ s1 = Some\ \vartheta\rangle$ **and** $h2$ [*of* ϑ] **have** $unify\ (t2 \lhd \vartheta)$
$(s2 \lhd \vartheta) \neq None$
by *auto*
from *this* **show** $unify\ (t1 \cdot t2)\ (s1 \cdot s2) \neq None$
by (*simp add*: $\langle unify\ t1\ s1 = Some\ \vartheta\rangle$ *option.case-eq-if*)

qed
qed
qed

We establish some useful properties of substitutions and instances.

definition *ground-on* :: $'a\ set \Rightarrow\ 'a\ subst \Rightarrow bool$
where *ground-on* $V\ \sigma = (\forall\, x \in V.\ (\mathit{ground\text{-}term}\ (\mathit{subst}\ (\mathit{Var}\ x)\ \ \sigma)))$

lemma *comp-subst-terms*:
assumes $\sigma \doteq \vartheta \lozenge \eta$
shows $(\mathit{subst}\ t\ \sigma) = (\mathit{subst}\ (\mathit{subst}\ t\ \vartheta)\ \eta)$
proof −
from ‹$\sigma \doteq \vartheta \lozenge \eta$› **have** $((\mathit{subst}\ t\ \sigma) = (\mathit{subst}\ t\ (\vartheta \lozenge \eta)))$ **by** *auto*
have $(\mathit{subst}\ t\ (\vartheta \lozenge \eta) = (\mathit{subst}\ (\mathit{subst}\ t\ \vartheta)\ \eta))$ **by** *auto*
from *this* **and** ‹$((\mathit{subst}\ t\ \sigma) = (\mathit{subst}\ t\ (\vartheta \lozenge \eta)))$› **show** *?thesis* **by** *auto*
qed

lemma *ground-instance*:
assumes *ground-on* $(\mathit{vars\text{-}of}\ t)\ \sigma$
shows *ground-term* $(\mathit{subst}\ t\ \sigma)$
proof $(\mathit{rule\ ccontr})$
assume $\neg$ *ground-term* $(\mathit{subst}\ t\ \sigma)$
then have *vars-of* $(\mathit{subst}\ t\ \sigma) \neq \{\}$ **unfolding** *ground-term-def* **by** *auto*
then obtain x **where** $x \in \mathit{vars\text{-}of}\ (\mathit{subst}\ t\ \sigma)$ **by** *auto*
then have $x \in \bigcup\ \{\ V.\ \exists\, x.\ (x \in (\mathit{vars\text{-}of}\ t)) \wedge (V = \mathit{vars\text{-}of}\ (\mathit{subst}\ (\mathit{Var}\ x)\ \sigma))$
$\}$
using *vars-of-instances* **by** *force*
then obtain y **where** $x \in (\mathit{vars\text{-}of}\ (\mathit{subst}\ (\mathit{Var}\ y)\ \sigma))$ **and** $y \in (\mathit{vars\text{-}of}\ t)$ **by** *blast*
from *assms(1)* **and** ‹$y \in (\mathit{vars\text{-}of}\ t)$› **have** *ground-term* $(\mathit{subst}\ (\mathit{Var}\ y)\ \sigma)$ **unfolding** *ground-on-def*
by *auto*
from *this* **and** ‹$x \in (\mathit{vars\text{-}of}\ (\mathit{subst}\ (\mathit{Var}\ y)\ \sigma))$› **show** *False* **unfolding** *ground-term-def* **by** *auto*
qed

lemma *substs-preserve-groundness*:
assumes *ground-term t*
shows *ground-term* $(\mathit{subst}\ t\ \sigma)$
by (*metis assms equals0D ground-instance ground-on-def ground-term-def*)

lemma *ground-subst-exists* :
finite $V \Longrightarrow \exists\,\sigma.\ (\mathit{ground\text{-}on}\ V\ \sigma)$
proof (*induction rule*: *finite.induct*)
case *emptyI*
show *?case* **unfolding** *ground-on-def* **by** *simp*
next
fix A :: $'a\ set$ **and** $a{::}'a$
assume *finite A*

assume *hyp-ind*: $\exists\sigma$. *ground-on A* σ
then obtain σ **where** *ground-on A* σ **by** *auto*
then show $\exists\sigma$. *ground-on* (*insert a A*) σ
proof *cases*
assume $a \in A$
from *this* **and** *hyp-ind* **show** $\exists\sigma$. *ground-on* (*insert a A*) σ
unfolding *ground-on-def* **by** *auto*
next
assume $a \notin A$
obtain *c* **where** *c* = (*Const a*) **and** *is-a-constant c* **by** *auto*
obtain ϑ **where** ϑ = (*a*,*c*) # σ **by** *auto*
have $\forall x$. ($x \in$ *insert a A* $\longrightarrow$ (*ground-term* (*subst* (*Var x*) ϑ)))
proof ((*rule allI*)+,(*rule impI*)+)
fix *x* **assume** $x \in$ *insert a A*
show *ground-term* (*subst* (*Var x*) ϑ)
proof *cases*
assume $x = a$
from *this* **and** ‹ϑ = (*a*,*c*) # σ› **have** (*subst* (*Var x*) ϑ) = *c* **by** *auto*
from ‹*is-a-constant c*› **have** *ground-term c* **using** *constants-are-ground* **by** *auto*
from *this* **and** ‹(*subst* (*Var x*) ϑ) = *c*› **show** *ground-term* (*subst* (*Var x*) ϑ) **by** *auto*
next
assume $x \neq a$
from ‹$x \neq a$› **and** ‹$x \in$ *insert a A*› **have** $x \in A$ **by** *auto*
from ‹$x \neq a$› **and** ‹ϑ = (*a*,*c*) # σ› **have** (*subst* (*Var x*) ϑ) = (*subst* (*Var x*) σ) **by** *auto*
from *this* **and** ‹$x \in A$› **and** ‹*ground-on A* σ›
show *ground-term* (*subst* (*Var x*) ϑ) **unfolding** *ground-on-def* **by** *auto*
qed
qed
from *this* **show** *?thesis* **unfolding** *ground-on-def* **by** *auto*
qed
qed

lemma *substs-preserve-ground-terms* :
assumes *ground-term t*
shows *subst t* σ = *t*
by (*metis agreement assms equals0D ground-term-def subst-Nil*)

lemma *substs-preserve-subterms* :
shows $\bigwedge$ *t s*. *subterm t p s* $\Longrightarrow$ *subterm* (*subst t* σ) *p* (*subst s* σ)
proof (*induction p*)
case *Nil*
then have *t* = *s* **using** *subterm.elims*(*2*) **by** *auto*
from ‹*t* = *s*› **have** (*subst t* σ) = (*subst s* σ) **by** *auto*
from *this* **show** *?case* **using** *Nil.prems* **by** *auto*
next case (*Cons i q*)
from ‹*subterm t* (*i* # *q*) *s*› **obtain** *t1 t2* **where**

$t = (Comb\ t1\ t2)$ **using** *subterm.elims*(2) **by** *blast*
have $i = Left \lor i = Right$ **using** *indices.exhaust* **by** *blast*
then show *subterm* (*subst t σ*) (*i # q*) (*subst s σ*)
proof
assume $i = Left$
from *this* **and** ‹$t = Comb\ t1\ t2$› **and** ‹*subterm t* (*i # q*) *s*›
have *subterm t1 q s* **by** *auto*
from *this* **have** *subterm* (*subst t1 σ*) *q* (*subst s σ*) **using** *Cons.IH* **by** *auto*
from *this* **and** ‹$t = Comb\ t1\ t2$›
show *subterm* (*subst t σ*) (*i # q*) (*subst s σ*)
by (*simp add*: ‹$i = indices.Left$›)
next assume $i = Right$
from *this* **and** ‹$t = Comb\ t1\ t2$› **and** ‹*subterm t* (*i # q*) *s*›
have *subterm t2 q s* **by** *auto*
from *this* **have** *subterm* (*subst t2 σ*) *q* (*subst s σ*) **using** *Cons.IH* **by** *auto*
from *this* **and** ‹$t = Comb\ t1\ t2$›
show *subterm* (*subst t σ*) (*i # q*) (*subst s σ*)
by (*simp add*: ‹$i = indices.Right$›)
qed
qed

lemma *substs-preserve-occurs-in*:
assumes *occurs-in s t*
shows *occurs-in* (*subst s σ*) (*subst t σ*)
using *substs-preserve-subterms*
using *assms occurs-in-def* **by** *blast*

definition *coincide-on*
where *coincide-on* $\sigma\ \eta\ V = (\forall x \in V.\ (subst\ (Var\ x)\ \sigma) = (\ (subst\ (Var\ x)\ \eta)))$

lemma *coincide-sym*:
assumes *coincide-on* $\sigma\ \eta\ V$
shows *coincide-on* $\eta\ \sigma\ V$
by (*metis assms coincide-on-def*)

lemma *coincide-on-term*:
shows $\bigwedge \sigma\ \eta.\ coincide\text{-}on\ \sigma\ \eta\ (vars\text{-}of\ t) \Longrightarrow subst\ t\ \sigma = subst\ t\ \eta$
proof (*induction t*)
case (*Var x*)
from *this* **show** $subst\ (Var\ x)\ \sigma = subst\ \ (Var\ x)\ \eta$
unfolding *coincide-on-def* **by** *auto*
next case (*Const x*)
show $subst\ (Const\ x)\ \sigma = subst\ \ (Const\ x)\ \eta$ **by** *auto*
next case (*Comb t1 t2*)
from *this* **show** *?case* **unfolding** *coincide-on-def* **by** *auto*
qed

lemma *ground-replacement*:
assumes *replace-subterm t p v s*

assumes *ground-term* (*subst t σ*)
assumes *ground-term* (*subst v σ*)
shows *ground-term* (*subst s σ*)
proof −
from *assms*(*1*) **have** *vars-of s* ⊆ (*vars-of t*) ∪ (*vars-of v*) **using** *vars-of-replacement-set* [*of t p v s*]
by *auto*
from *assms*(*2*) **have** *ground-on* (*vars-of t*) *σ* **unfolding** *ground-on-def*
by (*metis contra-subsetD ex-in-conv ground-term-def occs-vars-subset subst-mono vars-iff-occseq*)
from *assms*(*3*) **have** *ground-on* (*vars-of v*) *σ* **unfolding** *ground-on-def*
by (*metis contra-subsetD ex-in-conv ground-term-def occs-vars-subset subst-mono vars-iff-occseq*)
from ‹*vars-of s* ⊆ (*vars-of t*) ∪ (*vars-of v*)› ‹*ground-on* (*vars-of t*) *σ*›
and ‹*ground-on* (*vars-of v*) *σ*› **have** *ground-on* (*vars-of s*) *σ*
by (*meson UnE ground-on-def rev-subsetD*)
from *this* **show** *?thesis* **using** *ground-instance* **by** *blast*
qed

We now show that two disjoint substitutions can always be fused.

lemma *combine-substs*:
assumes *finite V1*
assumes *V1* ∩ *V2* = {}
assumes *ground-on V1 η1*
shows ∃*σ*. (*coincide-on σ η1 V1*) ∧ (*coincide-on σ η2 V2*)
proof −
have *finite V1* ⟹ *ground-on V1 η1* ⟹ *V1* ∩ *V2* = {} ⟹ ∃*σ*. (*coincide-on σ η1 V1*) ∧ (*coincide-on σ η2 V2*)
proof (*induction rule*: *finite.induct*)
case *emptyI*
show *?case* **unfolding** *coincide-on-def* **by** *auto*
next fix *V1* :: *'a set* **and** *a*::*'a*
assume *finite V1*
assume *hyp-ind*: *ground-on V1 η1* ⟹ *V1* ∩ *V2* = {}
⟹ ∃*σ*. (*coincide-on σ η1 V1*) ∧ (*coincide-on σ η2 V2*)
assume *ground-on* (*insert a V1*) *η1*
assume (*insert a V1*) ∩ *V2* = {}
from *this* **have** *V1* ∩ *V2* = {} **by** *auto*
from ‹*ground-on* (*insert a V1*) *η1*› **have** *ground-on V1 η1*
unfolding *ground-on-def* **by** *auto*
from *this* **and** *hyp-ind* **and** ‹*V1* ∩ *V2* = {}› **obtain** *σ'*
where *c*:(*coincide-on σ' η1 V1*) ∧ (*coincide-on σ' η2 V2*) **by** *auto*
let *?t* = *subst* (*Var a*) *η1*
from *assms*(*2*) **have** *ground-term ?t*
by (*meson* ‹*ground-on* (*insert a V1*) *η1*› *ground-on-def insertI1*)
let *?σ* = *comp* [(*a*,*?t*)] *σ'*
have *coincide-on ?σ η1* (*insert a V1*)
proof (*rule ccontr*)
assume ¬*coincide-on ?σ η1* (*insert a V1*)

then obtain x **where** $x \in (insert\ a\ V1)$ **and**
$(subst\ (Var\ x)\ ?\sigma) \neq (\ (subst\ (Var\ x)\ \eta 1))$
unfolding *coincide-on-def* **by** *blast*
have $subst\ (Var\ a)\ ?\sigma\ = subst\ ?t\ \sigma'$ **by** *simp*
from ‹*ground-term ?t*› **have** $subst\ (Var\ a)\ ?\sigma\ = ?t$
using *substs-preserve-ground-terms* **by** *auto*
from *this* **and** ‹$(subst\ (Var\ x)\ ?\sigma) \neq (\ (subst\ (Var\ x)\ \eta 1))$›
have $x \neq a$ **by** *blast*
from *this* **and** ‹$x \in (insert\ a\ V1)$› **have** $x \in V1$ **by** *auto*
from ‹$x \neq a$› **have** $(subst\ (Var\ x)\ ?\sigma) = (subst\ (Var\ x)\ \sigma')$ **by** *auto*
from *c* **and** ‹$x \in V1$› **have** $(subst\ (Var\ x)\ \sigma') = (subst\ (Var\ x)\ \eta 1)$
unfolding *coincide-on-def* **by** *blast*
from *this* **and** ‹$(subst\ (Var\ x)\ ?\sigma) = (subst\ (Var\ x)\ \sigma')$›
and ‹$(subst\ (Var\ x)\ ?\sigma) \neq (\ (subst\ (Var\ x)\ \eta 1))$› **show** *False* **by** *auto*
qed
have *coincide-on* $?\sigma\ \eta 2\ V2$
proof (*rule ccontr*)
assume $\neg$*coincide-on* $?\sigma\ \eta 2\ V2$
then obtain x **where** $x \in V2$ **and**
$(subst\ (Var\ x)\ ?\sigma) \neq (\ (subst\ (Var\ x)\ \eta 2))$
unfolding *coincide-on-def* **by** *blast*
from ‹$(insert\ a\ V1) \cap V2 = \{\}$› **and** ‹$x \in V2$› **have** $x \neq a$ **by** *auto*
from *this* **have** $(subst\ (Var\ x)\ ?\sigma) = (subst\ (Var\ x)\ \sigma')$ **by** *auto*
from *c* **and** ‹$x \in V2$› **have** $(subst\ (Var\ x)\ \sigma') = (subst\ (Var\ x)\ \eta 2)$
unfolding *coincide-on-def* **by** *blast*
from *this* **and** ‹$(subst\ (Var\ x)\ ?\sigma) = (subst\ (Var\ x)\ \sigma')$›
and ‹$(subst\ (Var\ x)\ ?\sigma) \neq (\ (subst\ (Var\ x)\ \eta 2))$› **show** *False* **by** *auto*
qed
from ‹*coincide-on* $?\sigma\ \eta 1\ (insert\ a\ V1)$› ‹*coincide-on* $?\sigma\ \eta 2\ V2$›
show $\exists \sigma.\ (coincide\text{-}on\ \sigma\ \eta 1\ (insert\ a\ V1)) \wedge (coincide\text{-}on\ \sigma\ \eta 2\ V2)$ **by**
auto
qed
from *this* **and** *assms* **show** *?thesis* **by** *auto*
qed

We define a map function for substitutions and prove its correctness.

fun *map-subst*
where *map-subst f Nil = Nil*
| *map-subst f* $((x,y)\ \#\ l) = (x,(f\ y))\ \#\ (map\text{-}subst\ f\ l)$

lemma *map-subst-lemma*:
shows $((subst\ (Var\ x)\ \sigma) \neq (Var\ x) \vee (subst\ (Var\ x)\ \sigma) \neq (subst\ (Var\ x)$
$(map\text{-}subst\ f\ \sigma)))$
$\longrightarrow ((subst\ (Var\ x)\ (map\text{-}subst\ f\ \sigma)) = (f\ (subst\ (Var\ x)\ \sigma)))$
proof (*induction* σ,*simp*)
next case (*Cons p* σ)
let $?u = (fst\ p)$
let $?v = (snd\ p)$
show *?case*

proof
assume ((*subst* (*Var x*) (*Cons p σ*)) ≠ (*Var x*)
∨ (*subst* (*Var x*) (*Cons p σ*))
≠ (*subst* (*Var x*) (*map-subst f* (*Cons p σ*))))
have *map-subst f* (*Cons p σ*) = ((*?u*, (*f ?v*)) # (*map-subst f σ*))
by (*metis map-subst.simps*(*2*) *prod.collapse*)
show (*subst* (*Var x*) (*map-subst f* (*Cons p σ*))) = (*f* (*subst* (*Var x*) (*Cons p σ*)))
proof *cases*
assume *x* = *?u*
from *this* **have** *subst* (*Var x*) (*Cons p σ*) = *?v*
by (*metis assoc.simps*(*2*) *prod.collapse subst.simps*(*1*))
from ‹*map-subst f* (*Cons p σ*) = ((*?u*, (*f ?v*)) # (*map-subst f σ*))›
and ‹*x* = *?u*›
have *subst* (*Var x*) (*map-subst f* (*Cons p σ*)) = (*f ?v*) **by** *simp*
from ‹*subst* (*Var x*) (*Cons p σ*) = *?v*› ‹*subst* (*Var x*) (*map-subst f* (*Cons p σ*)) = (*f ?v*)› **show** *?thesis* **by** *auto*
next
assume *x* ≠ *?u*
from *this* **have** *subst* (*Var x*) (*Cons p σ*) = (*subst* (*Var x*) *σ*)
by (*metis assoc.simps*(*2*) *prod.collapse subst.simps*(*1*))
from ‹*map-subst f* (*Cons p σ*) = ((*?u*, (*f ?v*)) # (*map-subst f σ*))›
and ‹*x* ≠ *?u*›
have *subst* (*Var x*) (*map-subst f* (*Cons p σ*)) =
subst (*Var x*) (*map-subst f σ*) **by** *simp*
from *this* **and** *Cons.IH* **have**
subst (*Var x*) (*map-subst f* (*Cons p σ*)) = (*f* (*subst* (*Var x*) *σ*))
using ‹*subst* (*Var x*) (*p* # *σ*) = *subst* (*Var x*) *σ*› ‹*subst* (*Var x*) (*p* # *σ*) ≠ *Var x* ∨ *subst* (*Var x*) (*p* # *σ*) ≠ *subst* (*Var x*) (*map-subst f* (*p* # *σ*))› **by** *auto*

from *this* **and** ‹*subst* (*Var x*) (*Cons p σ*) = (*subst* (*Var x*) *σ*)› **show** *?thesis* **by** *auto*
qed
qed
qed

2.3.1 Minimum Idempotent Most General Unifier

definition *min-IMGU* :: *'a subst* ⇒ *'a trm* ⇒ *'a trm* ⇒ *bool* **where**
min-IMGU μ t u ⟷
IMGU μ t u ∧ *fst* ‘ *set μ* ⊆ *vars-of t* ∪ *vars-of u* ∧ *range-vars μ* ⊆ *vars-of t* ∪ *vars-of u*

lemma *unify-computes-min-IMGU*:
unify M N = *Some σ* ⟹ *min-IMGU σ M N*
by (*simp add*: *min-IMGU-def IMGU-iff-Idem-and-MGU unify-computes-MGU unify-gives-Idem*
unify-gives-minimal-domain unify-gives-minimal-range)

2.4 Congruences

We now define the notion of a congruence on ground terms, i.e., an equivalence relation that is closed under contextual embedding.

type-synonym $'a$ *binary-relation-on-trms* $= {'a}$ *trm* $\Rightarrow {'a}$ *trm* $\Rightarrow$ *bool*

definition *reflexive* :: $'a$ *binary-relation-on-trms* $\Rightarrow$ *bool*
where
$(\mathit{reflexive}\ x) = (\forall\, y.\ (x\ y\ y))$

definition *symmetric* :: $'a$ *binary-relation-on-trms* $\Rightarrow$ *bool*
where
$(\mathit{symmetric}\ x) = (\forall\, y.\ \forall\, z.\ ((x\ y\ z) = (x\ z\ y)))$

definition *transitive* :: $'a$ *binary-relation-on-trms* $\Rightarrow$ *bool*
where
$(\mathit{transitive}\ x) = (\forall\, y.\ \forall\, z.\ \forall\, u.\ (x\ y\ z) \longrightarrow (x\ z\ u) \longrightarrow (x\ y\ u))$

definition *equivalence-relation* :: $'a$ *binary-relation-on-trms* $\Rightarrow$ *bool*
where
$(\mathit{equivalence\text{-}relation}\ x) = ((\mathit{reflexive}\ x) \land (\mathit{symmetric}\ x) \land (\mathit{transitive}\ x))$

definition *compatible-with-structure* :: $('a$ *binary-relation-on-trms*$) \Rightarrow$ *bool*
where
$(\mathit{compatible\text{-}with\text{-}structure}\ x) = (\forall\, t1\ t2\ s1\ s2.$
$(x\ t1\ s1) \longrightarrow (x\ t2\ s2) \longrightarrow (x\ (\mathit{Comb}\ t1\ t2)\ (\mathit{Comb}\ s1\ s2)))$

definition *congruence* :: $'a$ *binary-relation-on-trms* $\Rightarrow$ *bool*
where
$(\mathit{congruence}\ x) = ((\mathit{equivalence\text{-}relation}\ x) \land (\mathit{compatible\text{-}with\text{-}structure}\ x))$

lemma *replacement-preserves-congruences* :
shows $\bigwedge t\ \ s.\ (\mathit{congruence}\ I) \Longrightarrow (I\ (\mathit{subst}\ u\ \sigma)\ \ (\mathit{subst}\ v\ \sigma))$
$\Longrightarrow \mathit{subterm}\ t\ p\ u \Longrightarrow \mathit{replace\text{-}subterm}\ t\ p\ v\ s$
$\Longrightarrow (I\ (\mathit{subst}\ t\ \sigma)\ \ (\mathit{subst}\ s\ \sigma))$
proof (*induction p*)
case *Nil*
from ‹*subterm t Nil u*› **have** $t = u$ **by** *auto*
from ‹*replace-subterm t Nil v s*› **have** $s = v$ **by** *auto*
from ‹$t = u$› **and** ‹$s = v$› **and** ‹$(I\ (\mathit{subst}\ u\ \sigma)\ \ (\mathit{subst}\ v\ \sigma))$›
show *?case* **by** *auto*
next case (*Cons i q*)
from ‹*subterm t* $(i\ \#\ q)$ *u*› **obtain** *t1 t2* **where**
$t = (\mathit{Comb}\ t1\ t2)$ **using** *subterm.elims*(*2*) **by** *blast*
have $i = \mathit{Left} \lor i = \mathit{Right}$ **using** *indices.exhaust* **by** *blast*
then show $I\ (\mathit{subst}\ t\ \sigma)\ \ (\mathit{subst}\ s\ \sigma)$
proof
assume $i = \mathit{Left}$
from *this* **and** ‹$t = \mathit{Comb}\ t1\ t2$› **and** ‹*subterm t* $(i\ \#\ q)$ *u*›

```
      have subterm t1 q u by auto
    from ‹i = Left› and ‹t = Comb t1 t2› and ‹replace-subterm t (i # q) v s›
      obtain t1' where replace-subterm t1 q v t1' and s = Comb t1' t2 by auto
    from ‹congruence I› and ‹(I (subst u σ)  (subst v σ))›
      and ‹subterm t1 q u› and ‹replace-subterm t1 q v t1'› have
      I (subst t1 σ) (subst t1' σ) using Cons.IH Cons.prems(1) by blast
    from ‹congruence I› have I (subst t2 σ)  (subst t2 σ)
      unfolding congruence-def equivalence-relation-def reflexive-def by auto
    from ‹I (subst t1 σ) (subst t1' σ)›
      and ‹I (subst t2 σ)  (subst t2 σ)›
      and ‹congruence I› and ‹t = (Comb t1 t2)› and ‹s = (Comb t1' t2)›
      show I (subst t σ)  (subst s σ)
        unfolding congruence-def compatible-with-structure-def by auto
  next
    assume i = Right
    from this and ‹t = Comb t1 t2› and ‹subterm t (i # q) u›
      have subterm t2 q u by auto
    from ‹i = Right› and ‹t = Comb t1 t2› and ‹replace-subterm t (i # q) v s›

      obtain t2' where replace-subterm t2 q v t2' and s = Comb t1 t2' by auto
    from ‹congruence I› and ‹(I (subst u σ)  (subst v σ))›
      and ‹subterm t2 q u› and ‹replace-subterm t2 q v t2'› have
      I (subst t2 σ) (subst t2' σ) using Cons.IH Cons.prems(1) by blast
    from ‹congruence I› have I (subst t1 σ)  (subst t1 σ)
      unfolding congruence-def equivalence-relation-def reflexive-def by auto
    from ‹I (subst t2 σ) (subst t2' σ)›
      and ‹I (subst t1 σ)  (subst t1 σ)›
      and ‹congruence I› and ‹t = (Comb t1 t2)› and ‹s = (Comb t1 t2')›
      show I (subst t σ)  (subst s σ)
        unfolding congruence-def compatible-with-structure-def by auto
  qed
qed

definition equivalent-on
where equivalent-on σ η V I = (∀ x ∈ V.
  (I (subst (Var x) σ) ( (subst (Var x) η))))

lemma equivalent-on-term:
  assumes congruence I
  shows ⋀ σ η. equivalent-on σ η (vars-of t) I ⟹ (I (subst t σ) (subst t η))
proof (induction t)
  case (Var x)
    from this show (I (subst (Var x) σ) (subst  (Var x) η))
      unfolding equivalent-on-def by auto
  next case (Const x)
    from assms(1) show (I (subst (Const x) σ) (subst  (Const x) η))
      unfolding congruence-def equivalence-relation-def reflexive-def by auto
  next case (Comb t1 t2)
    from this assms(1) show ?case unfolding equivalent-on-def
```

unfolding *congruence-def compatible-with-structure-def* **by** *auto*
qed

2.5 Renamings

We define the usual notion of a renaming. We show that fresh renamings always exist (provided the set of variables is infinite) and that renamings admit inverses.

definition *renaming*
where
renaming σ V = ($\forall$ $x \in V$. (*is-a-variable* (*subst* (*Var* x) σ))
$\wedge$ ($\forall$ x y. (($x \in V$) $\longrightarrow$ ($y \in V$) $\longrightarrow$ $x \neq y$ $\longrightarrow$ (*subst* (*Var* x) σ) $\neq$ (*subst* (*Var* y) σ))))

lemma *renamings-admit-inverse*:
shows *finite* V $\Longrightarrow$ *renaming* σ V $\Longrightarrow$ $\exists$ ϑ. ($\forall$ $x \in V$. (*subst* (*subst* (*Var* x) σ) ϑ) = (*Var* x))
$\wedge$ ($\forall$ x. ($x \notin$ (*subst-codomain* σ V) $\longrightarrow$ (*subst* (*Var* x) ϑ) = (*Var* x)))
$\wedge$ ($\forall$ x. *is-a-variable* (*subst* (*Var* x) ϑ))
proof (*induction rule*: *finite.induct*)
case *emptyI*
let *?*ϑ = []
have *i*: ($\forall$ $x \in \{\}$. (*subst* (*subst* (*Var* x) σ) *?*ϑ) = (*Var* x)) **by** *auto*
have *ii*: ($\forall$ x. ($x \notin$ (*subst-codomain* σ $\{\}$) $\longrightarrow$ (*subst* (*Var* x) *?*ϑ) = (*Var* x))) **by** *auto*
have *iii*: $\forall$ x. *is-a-variable* (*subst* (*Var* x) *?*ϑ) **by** *simp*
from *i ii iii* **show** *?case* **by** *metis*
next
fix A :: *'a set* **and** a::*'a*
assume *finite* A
assume *hyp-ind*: *renaming* σ A $\Longrightarrow$ $\exists$ ϑ. ($\forall$ $x \in A$. (*subst* (*subst* (*Var* x) σ) ϑ) = (*Var* x))
$\wedge$ ($\forall$ x. ($x \notin$ (*subst-codomain* σ A) $\longrightarrow$ (*subst* (*Var* x) ϑ) = (*Var* x)))
$\wedge$ ($\forall$ x. *is-a-variable* (*subst* (*Var* x) ϑ))
show *renaming* σ (*insert* a A) $\Longrightarrow$ $\exists$ ϑ. ($\forall$ $x \in$ (*insert* a A). (*subst* (*subst* (*Var* x) σ) ϑ) = (*Var* x))
$\wedge$ ($\forall$ x. ($x \notin$ (*subst-codomain* σ (*insert* a A)) $\longrightarrow$ (*subst* (*Var* x) ϑ) = (*Var* x)))
$\wedge$ ($\forall$ x. *is-a-variable* (*subst* (*Var* x) ϑ))
proof −
assume *renaming* σ (*insert* a A)
show $\exists$ ϑ. ($\forall$ $x \in$ (*insert* a A). (*subst* (*subst* (*Var* x) σ) ϑ) = (*Var* x))
$\wedge$ ($\forall$ x. ($x \notin$ (*subst-codomain* σ (*insert* a A)) $\longrightarrow$ (*subst* (*Var* x) ϑ) = (*Var* x)))
$\wedge$ ($\forall$ x. *is-a-variable* (*subst* (*Var* x) ϑ))
proof (*cases*)
assume $a \in A$
from *this* **have** *insert* a A = A **by** *auto*
from *this* **and** ‹*renaming* σ (*insert* a A)› *hyp-ind* **show** *?thesis* **by** *metis*
next assume $a \notin A$

from ‹*renaming* σ (*insert a A*)› **have** *renaming* σ *A* **unfolding** *renaming-def* **by** *blast*
from *this* **and** *hyp-ind* **obtain** ϑ **where** *i*: ($\forall$ $x \in A$. (*subst* (*subst* (*Var x*) σ) ϑ) = (*Var x*)) **and**
ii: ($\forall$ *x*. ($x \notin$ (*subst-codomain* σ *A*) $\longrightarrow$ (*subst* (*Var x*) ϑ) = (*Var x*))) **and**
iii: $\forall$ *x*. *is-a-variable* (*Var x* $\lhd$ ϑ) **by** *metis*
from ‹*renaming* σ (*insert a A*)› **have** *is-a-variable* (*subst* (*Var a*) σ) **unfolding** *renaming-def* **by** *blast*
from *this* **obtain** *b* **where** (*subst* (*Var a*) σ) = (*Var b*) **using** *is-a-variable.elims(2)* **by** *auto*
let *?*η = (*b*,(*Var a*)) # ϑ
have *i'*: ($\forall$ $x \in$ (*insert a A*). (*subst* (*subst* (*Var x*) σ) *?*η) = (*Var x*))
proof
fix *x* **assume** $x \in$ (*insert a A*)
show (*subst* (*subst* (*Var x*) σ) *?*η) = (*Var x*)
proof (*cases*)
assume $x = a$
from *this*
have (*subst* (*Var b*) ((*b*,(*Var a*)) # *Nil*)) = (*Var a*)
by *simp*
have $b \notin$ (*subst-codomain* σ *A*)
proof
assume $b \in$ (*subst-codomain* σ *A*)
from *this* **have** $\exists$ *y*. (*subst* (*Var y*) σ) = (*Var b*) $\wedge$ ($y \in A$) **unfolding** *subst-codomain-def*
by *force*
then obtain *a'* **where** $a' \in A$ **and** *subst* (*Var a'*) σ = (*Var b*)
by *metis*
from ‹$a' \in A$› **and** ‹$a \notin A$› **have** $a \neq a'$ **by** *auto*
have $a \in$ (*insert a A*) **by** *auto*
from ‹$a \neq a'$› **and** ‹$a' \in A$› **and** ‹$a \in$ (*insert a A*)› **and** ‹*renaming* σ (*insert a A*)›
have (*subst* (*Var a*) $\sigma \neq$ (*subst* (*Var a'*) σ))
unfolding *renaming-def* **by** *blast*
from *this* **and** ‹*subst* (*Var a'*) σ = (*Var b*)› ‹(*subst* (*Var a*) σ) = (*Var b*)›
show *False* **by** *auto*
qed
from *this* **and** *ii* **have** (*subst* (*Var b*) ϑ) = (*Var b*) **by** *auto*
from *this* **and** ‹$x = a$› ‹(*subst* (*Var a*) σ) = (*Var b*)›
‹(*subst* (*Var b*) ((*b*,(*Var a*)) # *Nil*)) = (*Var a*)›
show (*subst* (*subst* (*Var x*) σ) *?*η) = (*Var x*)
by *simp*
next assume $x \neq a$
from *this* **and** ‹$x \in$ *insert a A*› **obtain** $x \in A$ **by** *auto*
from *this i* **have** (*subst* (*subst* (*Var x*) σ) ϑ) = (*Var x*)
by *auto*
then show (*subst* (*subst* (*Var x*) σ) *?*η) = (*Var x*)
by (*metis* ‹*subst* (*Var a*) σ = *Var b*› ‹*renaming* σ (*insert a A*)›

$\langle x \in$ *insert a A*$\rangle$ $\langle x \neq a\rangle$ *insertI1 is-a-variable.elims*(*2*)
occs.simps(*1*) *renaming-def repl-invariance vars-iff-occseq*)
qed
qed
have *ii'*: ($\forall$ *x*. (*x* $\notin$ (*subst-codomain* σ (*insert a A*)) $\longrightarrow$ (*subst* (*Var x*) *?*η)
= (*Var x*)))
proof ((*rule allI*),(*rule impI*))
fix *x* **assume** *x* $\notin$ *subst-codomain* σ (*insert a A*)
from *this* $\langle$(*subst* (*Var a*) σ) = (*Var b*)$\rangle$ **have** *x*$\neq$ *b* **unfolding** *subst-codomain-def*

by *auto*
from *this* **have** (*subst* (*Var x*) *?*η) = (*subst* (*Var x*) ϑ) **by** *auto*
from $\langle$*x* $\notin$ *subst-codomain* σ (*insert a A*)$\rangle$ **have** *x* $\notin$ (*subst-codomain* σ *A*)
unfolding *subst-codomain-def*
by *auto*
from *this* **and** *ii* **have** (*subst* (*Var x*) ϑ) = (*Var x*) **by** *auto*
from $\langle$(*subst* (*Var x*) *?*η) = (*subst* (*Var x*) ϑ)$\rangle$
and $\langle$(*subst* (*Var x*) ϑ) = (*Var x*)$\rangle$ **show** (*subst* (*Var x*) *?*η) = (*Var x*)
by *auto*
qed
have *iii'*: $\forall$ *x*. *is-a-variable* (*subst* (*Var x*) *?*η)
using *iii* **by** *auto*
from *i' ii' iii'* **show** *?thesis* **by** *auto*
qed
qed
qed

lemma *renaming-exists*:
assumes $\neg$ *finite* (*Vars* :: (*'a set*))
shows *finite V* $\Longrightarrow$ ($\forall$ *V'*::*'a set*. *finite V'* $\longrightarrow$ ($\exists$ η. ((*renaming* η *V*) $\wedge$ ((*subst-codomain*
η *V*) $\cap$ *V'*) = {})))
proof (*induction rule*: *finite.induct*)
case *emptyI*
let *?*η = []
show *?case* **unfolding** *renaming-def subst-codomain-def* **by** *auto*
next
fix *A* :: *'a set* **and** *a*::*'a*
assume *finite A*
assume *hyp-ind*: $\forall$ *V'* :: *'a set*. *finite V'* $\longrightarrow$ ($\exists$ η. *renaming* η *A* $\wedge$ *subst-codomain*
η *A* $\cap$ *V'* = {})
show $\forall$ *V'*:: *'a set*. *finite V'* $\longrightarrow$ ($\exists$ η. *renaming* η (*insert a A*) $\wedge$ *subst-codomain*
η (*insert a A*) $\cap$ *V'* = {})
proof ((*rule allI*),(*rule impI*))
fix *V'*:: *'a set* **assume** *finite V'*
from *this* **have** *finite* (*insert a V'*) **by** *auto*
from *this* **and** *hyp-ind* **obtain** η **where** *renaming* η *A* **and** (*subst-codomain*
η *A*) $\cap$ (*insert a V'*) = {} **by** *metis*
from $\langle$*finite A*$\rangle$ **have** *finite* (*subst-codomain* η *A*)
using *subst-codomain-is-finite* **by** *auto*

from *this* ‹*finite* V'› **have** *finite* ($V' \cup$ (*subst-codomain* η A)) **by** *auto*
from *this* **have** *finite* ((*insert* a V') $\cup$ (*subst-codomain* η A)) **by** *auto*
from *this* ‹$\neg$ *finite Vars*› **have** $\neg$ (*Vars* $\subseteq$ ((*insert* a V') $\cup$ (*subst-codomain* η A))) **using** *rev-finite-subset*
by *metis*
from *this* **obtain** *nv* **where** $nv \in$ *Vars* **and** $nv \notin$ (*insert* a V') **and** $nv \notin$ (*subst-codomain* η A) **by** *auto*
let $?\eta$ = (a,(*Var nv*)) # η
have *i*: ($\forall x \in$ (*insert* a A). (*is-a-variable* (*subst* (*Var* x) $?\eta$)))
proof (*rule ccontr*)
assume $\neg$ ($\forall x \in$ (*insert* a A). (*is-a-variable* (*subst* (*Var* x) $?\eta$)))
then obtain x **where** $x \in$ (*insert* a A) **and** $\neg$*is-a-variable* (*subst* (*Var* x) $?\eta$)
by *auto*
from ‹$\neg$*is-a-variable* (*subst* (*Var* x) $?\eta$)› **have** $x \neq a$ **by** *auto*
from *this* **and** ‹$x \in$ (*insert* a A)› **have** $x \in A$ **by** *auto*
from ‹$x \neq a$› **have** (*subst* (*Var* x) $?\eta$) = (*subst* (*Var* x) η) **by** *auto*
from ‹*renaming* η A› **and** ‹$x \in A$› **have** *is-a-variable* (*subst* (*Var* x) η)
unfolding *renaming-def* **by** *metis*
from *this* **and** ‹$\neg$*is-a-variable* (*subst* (*Var* x) $?\eta$)›
‹(*subst* (*Var* x) $?\eta$) = (*subst* (*Var* x) η)› **show** *False* **by** *auto*
qed
have *ii*: ($\forall$ x y. (($x \in$ (*insert* a A)) $\longrightarrow$ ($y \in$ (*insert* a A)) $\longrightarrow$ $x \neq y$
$\longrightarrow$ (*subst* (*Var* x) $?\eta$) $\neq$ (*subst* (*Var* y) $?\eta$)))
proof (*rule ccontr*)
assume $\neg$($\forall$ x y. (($x \in$ (*insert* a A)) $\longrightarrow$ ($y \in$ (*insert* a A)) $\longrightarrow$ $x \neq y$
$\longrightarrow$ (*subst* (*Var* x) $?\eta$) $\neq$ (*subst* (*Var* y) $?\eta$)))
from *this* **obtain** x y **where** $x \in$ *insert* a A $y \in$ *insert* a A $x \neq y$
(*subst* (*Var* x) $?\eta$) = (*subst* (*Var* y) $?\eta$) **by** *blast*
from *i* **obtain** y' **where** (*subst* (*Var* y) $?\eta$) = (*Var* y')
using *is-a-variable.simps* **using** ‹$y \in$ *insert* a A› *is-a-variable.elims*(*2*) **by** *auto*
from *i* **obtain** x' **where** (*subst* (*Var* x) $?\eta$) = (*Var* x')
using *is-a-variable.simps* **using** ‹$x \in$ *insert* a A› *is-a-variable.elims*(*2*) **by** *auto*
from ‹(*subst* (*Var* x) $?\eta$) = (*Var* x')› ‹(*subst* (*Var* y) $?\eta$) = (*Var* y')›
‹(*subst* (*Var* x) $?\eta$) = (*subst* (*Var* y) $?\eta$)› **have** $x' = y'$ **by** *auto*
have $x \neq a$
proof
assume $x = a$
from *this* **and** ‹$x \neq y$› **and** ‹$y \in$ *insert* a A› **have** $y \in A$ **by** *auto*
from *this* **and** ‹$x \neq y$› **and** ‹$x = a$› **and** ‹(*subst* (*Var* y) $?\eta$) = (*Var* y')›
have $y' \in$ (*subst-codomain* η A) **unfolding** *subst-codomain-def* **by** *auto*
from ‹$x = a$› **and** ‹(*subst* (*Var* x) $?\eta$) = (*Var* x')› **have** $x' = nv$ **by** *auto*
from *this* **and** ‹$y' \in$ (*subst-codomain* η A)› **and** ‹$x' = y'$› **and** ‹$nv \notin$ (*subst-codomain* η A)›
show *False* **by** *auto*
qed
from *this* **and** ‹$x \in$ *insert* a A› **have** $x \in A$ **and**

$(subst\ (Var\ x)\ ?\eta) = (subst\ (Var\ x)\ \eta)$ **by** *auto*
have $y \neq a$
proof
assume $y = a$
from *this* **and** ‹$x \neq y$› **and** ‹$x \in insert\ a\ A$› **have** $x \in A$ **by** *auto*
from *this* **and** ‹$x \neq y$› **and** ‹$y = a$› **and** ‹$(subst\ (Var\ x)\ ?\eta) = (Var\ x')$›
have $x' \in (subst\text{-}codomain\ \eta\ A)$ **unfolding** *subst-codomain-def* **by** *auto*
from ‹$y = a$› **and** ‹$(subst\ (Var\ y)\ ?\eta) = (Var\ y')$› **have** $y' = nv$ **by** *auto*
from *this* **and** ‹$x' \in (subst\text{-}codomain\ \eta\ A)$› **and** ‹$x' = y'$› **and** ‹$nv \notin (subst\text{-}codomain\ \eta\ A)$›
show *False* **by** *auto*
qed
from *this* **and** ‹$y \in insert\ a\ A$› **have** $y \in A$ **and**
$(subst\ (Var\ y)\ ?\eta) = (subst\ (Var\ y)\ \eta)$ **by** *auto*
from ‹$(subst\ (Var\ x)\ ?\eta) = (subst\ (Var\ x)\ \eta)$›
‹$(subst\ (Var\ y)\ ?\eta) = (subst\ (Var\ y)\ \eta)$›
‹$(subst\ (Var\ x)\ ?\eta) = (subst\ (Var\ y)\ ?\eta)$›
have $(subst\ (Var\ x)\ \eta) = (subst\ (Var\ y)\ \eta)$ **by** *auto*
from *this* **and** ‹$x \in A$› **and** ‹$y \in A$›**and** ‹$renaming\ \eta\ A$› **and** ‹$x \neq y$› **show** *False*
unfolding *renaming-def* **by** *metis*
qed
from *i ii* **have** $renaming\ ?\eta\ (insert\ a\ A)$ **unfolding** *renaming-def* **by** *auto*
have $((subst\text{-}codomain\ ?\eta\ (insert\ a\ A)) \cap V') = \{\}$
proof (*rule ccontr*)
assume $(subst\text{-}codomain\ ?\eta\ (insert\ a\ A)) \cap V' \neq \{\}$
then obtain x **where** $x \in (subst\text{-}codomain\ ?\eta\ (insert\ a\ A))$ **and** $x \in V'$ **by** *auto*
from ‹$x \in (subst\text{-}codomain\ ?\eta\ (insert\ a\ A))$› **obtain** x' **where** $x' \in (insert\ a\ A)$
and $subst\ (Var\ x')\ ?\eta = (Var\ x)$ **unfolding** *subst-codomain-def* **by** *blast*
have $x' \neq a$
proof
assume $x' = a$
from *this* **and** ‹$subst\ (Var\ x')\ ?\eta = (Var\ x)$› **have** $x = nv$ **by** *auto*
from *this* **and** ‹$x \in V'$› **and** ‹$nv \notin (insert\ a\ V')$› **show** *False* **by** *auto*
qed
from *this* **and** ‹$x' \in (insert\ a\ A)$› **have** $x' \in A$ **by** *auto*
from ‹$x' \neq a$› **and** ‹$subst\ (Var\ x')\ ?\eta = (Var\ x)$› **have**
$(Var\ x) = (subst\ (Var\ x')\ \eta)$ **by** *auto*
from *this* **and** ‹$x' \in A$› **have** $x \in subst\text{-}codomain\ \eta\ A$ **unfolding** *subst-codomain-def* **by** *auto*
from ‹$x \in subst\text{-}codomain\ \eta\ A$› **and** ‹$(subst\text{-}codomain\ \eta\ A) \cap (insert\ a\ V') = \{\}$› **and** ‹$x \in V'$›
show *False* **by** *auto*
qed
from *this* **and** ‹$renaming\ ?\eta\ (insert\ a\ A)$›
show $\exists\eta.\ renaming\ \eta\ (insert\ a\ A) \wedge subst\text{-}codomain\ \eta\ (insert\ a\ A) \cap V' = \{\}$ **by** *auto*

qed
qed

end
theory *equational-clausal-logic*

imports *Main terms HOL−Library.Multiset*

begin

3 Equational Clausal Logic

The syntax and semantics of clausal equational logic are defined as usual. Interpretations are congruences on binary trees.

3.1 Syntax

We first define the syntax of equational clauses.

datatype $'a$ *equation* $=$ *Eq* $'a$ *trm* $'a$ *trm*

fun *lhs*
where *lhs* (*Comb t1 t2*) = *t1* |
lhs (*Var x*) = (*Var x*) |
lhs (*Const x*) = (*Const x*)

fun *rhs*
where *rhs* (*Comb t1 t2*) = *t2* |
rhs (*Var x*) = (*Var x*) |
rhs (*Const x*) = (*Const x*)

datatype $'a$ *literal* = *Pos* $'a$ *equation* | *Neg* $'a$ *equation*

fun *atom* :: $'a$ *literal* ⇒ $'a$ *equation*
where
(*atom* (*Pos x*)) = *x* |
(*atom* (*Neg x*)) = *x*

datatype *sign* = *pos* | *neg*

fun *get-sign* :: $'a$ *literal* ⇒ *sign*
where
(*get-sign* (*Pos x*)) = *pos* |
(*get-sign* (*Neg x*)) = *neg*

fun *positive-literal* :: $'a$ *literal* ⇒ *bool*

where
  $(\textit{positive-literal}\ (\textit{Pos}\ x)) = \textit{True}\ |$
  $(\textit{positive-literal}\ (\textit{Neg}\ x)) = \textit{False}$

fun $\textit{negative-literal} :: {'a}\ \textit{literal} \Rightarrow \textit{bool}$
  where
  $(\textit{negative-literal}\ (\textit{Pos}\ x)) = \textit{False}\ |$
  $(\textit{negative-literal}\ (\textit{Neg}\ x)) = \textit{True}$

fun $\textit{mk-lit} :: \textit{sign} \Rightarrow {'a}\ \textit{equation} \Rightarrow {'a}\ \textit{literal}$
  where
  $(\textit{mk-lit}\ \textit{pos}\ x) = (\textit{Pos}\ x)\ |$
  $(\textit{mk-lit}\ \textit{neg}\ x) = (\textit{Neg}\ x)$

definition *decompose-equation*
  where $\textit{decompose-equation}\ e\ t\ s = (e = (\textit{Eq}\ t\ s) \lor (e = (\textit{Eq}\ s\ t)))$

definition *decompose-literal*
  where $\textit{decompose-literal}\ L\ t\ s\ p =$
    $(\exists\, e.\ ((p = \textit{pos} \land (L = (\textit{Pos}\ e)) \land \textit{decompose-equation}\ e\ t\ s)$
    $\lor (p = \textit{neg} \land (L = (\textit{Neg}\ e)) \land \textit{decompose-equation}\ e\ t\ s)))$

fun *subterms-of-eq*
  where $\textit{subterms-of-eq}\ (\textit{Eq}\ t\ s) = (\textit{subterms-of}\ t \cup \textit{subterms-of}\ s)$

fun *vars-of-eq*
  where $\textit{vars-of-eq}\ (\textit{Eq}\ t\ s) = (\textit{vars-of}\ t \cup \textit{vars-of}\ s)$

lemma *decompose-equation-vars*:
  assumes $\textit{decompose-equation}\ e\ t\ s$
  shows $\textit{vars-of}\ t \subseteq \textit{vars-of-eq}\ e$
by (*metis assms decompose-equation-def sup.cobounded1 sup-commute vars-of-eq.simps*)

fun *subterms-of-lit*
  where
  $\textit{subterms-of-lit}\ (\textit{Pos}\ e) = (\textit{subterms-of-eq}\ e)\ |$
  $\textit{subterms-of-lit}\ (\textit{Neg}\ e) = (\textit{subterms-of-eq}\ e)$

fun *vars-of-lit*
  where
  $\textit{vars-of-lit}\ (\textit{Pos}\ e) = (\textit{vars-of-eq}\ e)\ |$
  $\textit{vars-of-lit}\ (\textit{Neg}\ e) = (\textit{vars-of-eq}\ e)$

fun *vars-of-cl*
  where $\textit{vars-of-cl}\ C = \{\ x.\ \exists\, L.\ x \in (\textit{vars-of-lit}\ L) \land L \in C\ \}$

fun *subterms-of-cl*
  where $\textit{subterms-of-cl}\ C = \{\ x.\ \exists\, L.\ x \in (\textit{subterms-of-lit}\ L) \land L \in C\ \}$

Note that clauses are defined as sets and not as multisets (identical literals are always merged).

type-synonym *'a clause* = *'a literal set*

fun *ground-clause* :: *'a clause* ⇒ *bool*
where
  (*ground-clause C*) = ((*vars-of-cl C*) = {})

fun *subst-equation* :: *'a equation* ⇒ *'a subst* ⇒ *'a equation*
where
  (*subst-equation* (*Eq u v*) *s*)
    = (*Eq* (*subst u s*) (*subst v s*))

fun *subst-lit* :: *'a literal* ⇒ *'a subst* ⇒ *'a literal*
where
  (*subst-lit* (*Pos e*) *s*)
    = (*Pos* (*subst-equation e s*)) |
  (*subst-lit* (*Neg e*) *s*)
    = (*Neg* (*subst-equation e s*))

fun *subst-cl* :: *'a clause* ⇒ *'a subst* ⇒ *'a clause*
where
  (*subst-cl C s*) = { *L*. (∃ *L'*. (*L'* ∈ *C*) ∧ (*L* = (*subst-lit L' s*))) }

We establish some properties of the functions returning the set of variables occurring in an object.

lemma *decompose-literal-vars*:
  assumes *decompose-literal L t s p*
  shows *vars-of t* ⊆ *vars-of-lit L*
by (*metis assms decompose-equation-vars decompose-literal-def vars-of-lit.simps*(*1*) *vars-of-lit.simps*(*2*))

lemma *vars-of-cl-lem*:
  assumes *L* ∈ *C*
  shows *vars-of-lit L* ⊆ *vars-of-cl C*
using *assms* **by** *auto*

lemma *set-of-variables-is-finite-eq*:
  shows *finite* (*vars-of-eq e*)
proof −
  obtain *t* **and** *s* **where** *e* = *Eq t s* **using** *equation.exhaust* **by** *auto*
  then have *vars-of-eq e* = (*vars-of t*) ∪ (*vars-of s*) **by** *auto*
  from *this* **show** *?thesis* **by** *auto*
qed

lemma *set-of-variables-is-finite-lit*:
  shows *finite* (*vars-of-lit l*)
proof −
  obtain *e* **where** *l* = *Pos e* ∨ *l* = *Neg e* **using** *literal.exhaust* **by** *auto*
  then have *vars-of-lit l* = (*vars-of-eq e*) **by** *auto*

from *this* **show** *?thesis* **using** *set-of-variables-is-finite-eq* **by** *auto*
qed

lemma *set-of-variables-is-finite-cl*:
assumes *finite C*
shows *finite* (*vars-of-cl C*)
proof −
let *?S* = { *x*. ∃ *l*. *x* = *vars-of-lit l* ∧ *l* ∈ *C* }
have *vars-of-cl C* = ⋃ *?S* **by** *auto*
from *assms* **have** *finite ?S* **by** *auto*
{ **fix** *x* **have** *x* ∈ *?S* ⟹ *finite x* **using** *set-of-variables-is-finite-lit* **by** *auto* }
from *this* **and** ‹*finite ?S*› **have** *finite* (⋃ *?S*) **using** *finite-Union* **by** *auto*
from *this* **and** ‹*vars-of-cl C* = ⋃ *?S*› **show** *?thesis* **by** *auto*
qed

lemma *subterm-lit-vars* :
assumes *u* ∈ *subterms-of-lit L*
shows *vars-of u* ⊆ *vars-of-lit L*
proof −
obtain *e* **where** *def-e*: *L* = (*Pos e*) ∨ *L* = (*Neg e*) **and** *vars-of-lit L* = *vars-of-eq e*
by (*metis negative-literal.elims*(*2*) *negative-literal.elims*(*3*) *vars-of-lit.simps*(*1*) *vars-of-lit.simps*(*2*))
obtain *t* **and** *s* **where** *def-ts*: *e* = (*Eq t s*) ∨ *e* = (*Eq s t*) **and** *vars-of-eq e* = *vars-of t* ∪ *vars-of s*
by (*metis equation.exhaust vars-of-eq.simps*)
from *this* **and** ‹*vars-of-lit L* = *vars-of-eq e*› **have** *vars-of-lit L* = *vars-of t* ∪ *vars-of s* **by** *auto*
from *assms*(*1*) **and** *def-e def-ts* **have** *u* ∈ *subterms-of t* ∪ *subterms-of s* **by** *auto*
from *this* **have** *vars-of u* ⊆ *vars-of t* ∪ *vars-of s*
by (*meson UnE sup.coboundedI1 sup.coboundedI2 vars-subterms-of*)
from *this* **and** ‹*vars-of-lit L* = *vars-of t* ∪ *vars-of s*› **show** *?thesis* **by** *auto*
qed

lemma *subterm-vars* :
assumes *u* ∈ *subterms-of-cl C*
shows *vars-of u* ⊆ *vars-of-cl C*
proof −
from *assms*(*1*) **obtain** *L* **where** *u* ∈ *subterms-of-lit L* **and** *L* ∈ *C* **by** *auto*
from ‹*u* ∈ *subterms-of-lit L*› **have** *vars-of u* ⊆ *vars-of-lit L* **using** *subterm-lit-vars* **by** *auto*
from ‹*L* ∈ *C*› **have** *vars-of-lit L* ⊆ *vars-of-cl C* **using** *vars-of-cl.simps* **by** *auto*
from *this* **and** ‹*vars-of u* ⊆ *vars-of-lit L*› **show** *?thesis* **by** *auto*
qed

We establish some basic properties of substitutions.

lemma *subterm-cl-subst*:
assumes *x* ∈ (*subterms-of-cl C*)
shows (*subst x σ*) ∈ (*subterms-of-cl* (*subst-cl C σ*))

proof −
from *assms*(*1*) **obtain** L **where** $L \in C$ **and** $x \in$ *subterms-of-lit* L **by** *auto*
from ‹$L \in C$› **have** (*subst-lit* $L\ \sigma$) $\in$ (*subst-cl* $C\ \sigma$) **by** *auto*
obtain e **where** $L = (Pos\ e) \vee L = (Neg\ e)$ **using** *literal.exhaust* **by** *auto*
then show *?thesis*
proof
assume $L = (Pos\ e)$
from *this* **and** ‹$x \in$ *subterms-of-lit* L› **have** $x \in$ *subterms-of-eq* e **by** *auto*
from ‹$L = (Pos\ e)$› **have** (*subst-lit* $L\ \sigma$) = (*Pos* (*subst-equation* $e\ \sigma$))
by *auto*
obtain $t\ s$ **where** $e = (Eq\ t\ s)$ **using** *equation.exhaust* **by** *auto*
from *this* **have** (*subst-equation* $e\ \sigma$) = (*Eq* (*subst* $t\ \sigma$) (*subst* $s\ \sigma$))
by *auto*
from ‹$x \in$ *subterms-of-eq* e› **and** ‹$e = (Eq\ t\ s)$› **have** $x \in$ *subterms-of* $t \vee x \in$ *subterms-of* s **by** *auto*
then show *?thesis*
proof
assume $x \in$ *subterms-of* t
then have *occurs-in* $x\ t$ **by** *auto*
then obtain p **where** *subterm* $t\ p\ x$ **unfolding** *occurs-in-def* **by** *blast*
from *this* **have** *subterm* (*subst* $t\ \sigma$) p (*subst* $x\ \sigma$)
using *substs-preserve-subterms* **by** *auto*
from *this* **have** *occurs-in* (*subst* $x\ \sigma$) (*subst* $t\ \sigma$) **unfolding** *occurs-in-def* **by** *auto*
then have (*subst* $x\ \sigma$) $\in$ *subterms-of* (*subst* $t\ \sigma$) **by** *auto*
then have (*subst* $x\ \sigma$) $\in$ *subterms-of-eq* (*Eq* (*subst* $t\ \sigma$) (*subst* $s\ \sigma$)) **by** *auto*
from *this* **and** ‹$L = (Pos\ e)$› **and** ‹$e = Eq\ t\ s$›
have (*subst* $x\ \sigma$) $\in$ (*subterms-of-lit* (*subst-lit* $L\ \sigma$)) **by** *auto*
from *this* **and** ‹(*subst-lit* $L\ \sigma$) $\in$ (*subst-cl* $C\ \sigma$)›
show (*subst* $x\ \sigma$) $\in$ *subterms-of-cl* (*subst-cl* $C\ \sigma$) **by** *auto*
next
assume $x \in$ *subterms-of* s
then have *occurs-in* $x\ s$ **by** *auto*
then obtain p **where** *subterm* $s\ p\ x$ **unfolding** *occurs-in-def* **by** *blast*
from *this* **have** *subterm* (*subst* $s\ \sigma$) p (*subst* $x\ \sigma$)
using *substs-preserve-subterms* **by** *auto*
from *this* **have** *occurs-in* (*subst* $x\ \sigma$) (*subst* $s\ \sigma$) **unfolding** *occurs-in-def* **by** *auto*
then have (*subst* $x\ \sigma$) $\in$ *subterms-of* (*subst* $s\ \sigma$) **by** *auto*
then have (*subst* $x\ \sigma$) $\in$ *subterms-of-eq* (*Eq* (*subst* $t\ \sigma$) (*subst* $s\ \sigma$)) **by** *auto*
from *this* **and** ‹$L = (Pos\ e)$› **and** ‹$e = Eq\ t\ s$›
have (*subst* $x\ \sigma$) $\in$ (*subterms-of-lit* (*subst-lit* $L\ \sigma$)) **by** *auto*
from *this* **and** ‹(*subst-lit* $L\ \sigma$) $\in$ (*subst-cl* $C\ \sigma$)›
show (*subst* $x\ \sigma$) $\in$ *subterms-of-cl* (*subst-cl* $C\ \sigma$) **by** *auto*
qed
next
assume $L = (Neg\ e)$
from *this* **and** ‹$x \in$ *subterms-of-lit* L› **have** $x \in$ *subterms-of-eq* e **by** *auto*
from ‹$L = (Neg\ e)$› **have** (*subst-lit* $L\ \sigma$) = (*Neg* (*subst-equation* $e\ \sigma$))

by *auto*
obtain *t s* **where** $e = (Eq\ t\ s)$ **using** *equation.exhaust* **by** *auto*
from *this* **have** $(\textit{subst-equation}\ e\ \sigma) = (Eq\ (\textit{subst}\ t\ \sigma)\ \ (\textit{subst}\ s\ \sigma))$
by *auto*
from ‹$x \in \textit{subterms-of-eq}\ e$› **and** ‹$e = (Eq\ t\ s)$› **have** $x \in \textit{subterms-of}\ t \vee x \in \textit{subterms-of}\ s$ **by** *auto*
then show *?thesis*
proof
assume $x \in \textit{subterms-of}\ t$
then have *occurs-in x t* **by** *auto*
then obtain *p* **where** *subterm t p x* **unfolding** *occurs-in-def* **by** *blast*
from *this* **have** *subterm* $(\textit{subst}\ t\ \sigma)\ p\ (\textit{subst}\ x\ \sigma)$
using *substs-preserve-subterms* **by** *auto*
from *this* **have** *occurs-in* $(\textit{subst}\ x\ \sigma)\ (\textit{subst}\ t\ \sigma)$ **unfolding** *occurs-in-def* **by** *auto*
then have $(\textit{subst}\ x\ \sigma) \in \textit{subterms-of}\ \ (\textit{subst}\ t\ \sigma)$ **by** *auto*
then have $(\textit{subst}\ x\ \sigma) \in \textit{subterms-of-eq}\ (Eq\ (\textit{subst}\ t\ \sigma)\ (\textit{subst}\ s\ \sigma))$ **by** *auto*
from *this* **and** ‹$L = (Neg\ e)$› **and** ‹$e = Eq\ t\ s$›
have $(\textit{subst}\ x\ \sigma) \in (\textit{subterms-of-lit}\ (\textit{subst-lit}\ L\ \sigma))$ **by** *auto*
from *this* **and** ‹$(\textit{subst-lit}\ L\ \sigma) \in (\textit{subst-cl}\ C\ \sigma)$›
show $(\textit{subst}\ x\ \sigma) \in \textit{subterms-of-cl}\ (\textit{subst-cl}\ C\ \sigma)$ **by** *auto*
next
assume $x \in \textit{subterms-of}\ s$
then have *occurs-in x s* **by** *auto*
then obtain *p* **where** *subterm s p x* **unfolding** *occurs-in-def* **by** *blast*
from *this* **have** *subterm* $(\textit{subst}\ s\ \sigma)\ p\ (\textit{subst}\ x\ \sigma)$
using *substs-preserve-subterms* **by** *auto*
from *this* **have** *occurs-in* $(\textit{subst}\ x\ \sigma)\ (\textit{subst}\ s\ \sigma)$ **unfolding** *occurs-in-def* **by** *auto*
then have $(\textit{subst}\ x\ \sigma) \in \textit{subterms-of}\ \ (\textit{subst}\ s\ \sigma)$ **by** *auto*
then have $(\textit{subst}\ x\ \sigma) \in \textit{subterms-of-eq}\ (Eq\ (\textit{subst}\ t\ \sigma)\ (\textit{subst}\ s\ \sigma))$ **by** *auto*
from *this* **and** ‹$L = (Neg\ e)$› **and** ‹$e = Eq\ t\ s$›
have $(\textit{subst}\ x\ \sigma) \in (\textit{subterms-of-lit}\ (\textit{subst-lit}\ L\ \sigma))$ **by** *auto*
from *this* **and** ‹$(\textit{subst-lit}\ L\ \sigma) \in (\textit{subst-cl}\ C\ \sigma)$›
show $(\textit{subst}\ x\ \sigma) \in \textit{subterms-of-cl}\ (\textit{subst-cl}\ C\ \sigma)$ **by** *auto*
qed
qed
qed

lemma *ground-substs-yield-ground-clause*:
assumes *ground-on* $(\textit{vars-of-cl}\ C)\ \sigma$
shows *ground-clause* $(\textit{subst-cl}\ C\ \sigma)$
proof (*rule ccontr*)
let $?D = (\textit{subst-cl}\ C\ \sigma)$
let $?V = (\textit{vars-of-cl}\ C)$
assume $\neg(\textit{ground-clause}\ ?D)$
then obtain *x* **where** $x \in (\textit{vars-of-cl}\ ?D)$ **by** *auto*
then obtain *l* **where** $l \in C$ **and** $x \in (\textit{vars-of-lit}\ (\textit{subst-lit}\ l\ \ \sigma))$ **by** *auto*
from ‹$l \in C$› **have** $\textit{vars-of-lit}\ l \subseteq \textit{vars-of-cl}\ C$ **by** *auto*

obtain *e* **where** $l = Pos\ e \vee l = Neg\ e$ **using** *literal.exhaust* **by** *auto*
then have *vars-of-lit* l = *vars-of-eq* e **by** *auto*
let $?l' = ($*subst-lit* $l\ \sigma)$
let $?e' = ($*subst-equation* $e\ \sigma)$
obtain t **and** s **where** $e = Eq\ t\ s$ **using** *equation.exhaust* **by** *auto*
then have *vars-of-eq* e = *vars-of* $t \cup$ *vars-of* s **by** *auto*
let $?t' = ($*subst* $t\ \sigma)$
let $?s' = ($*subst* $s\ \sigma)$
from ‹$e = Eq\ t\ s$› **have** $?e' = (Eq\ ?t'\ ?s')$ **by** *auto*
from ‹$l = Pos\ e \vee l = Neg\ e$› **have** $?l' = Pos\ ?e' \vee ?l' = Neg\ ?e'$ **by** *auto*
from ‹$l \in C$› **have** $?l' \in ?D$ **by** *auto*
from ‹$?l' = Pos\ ?e' \vee ?l' = Neg\ ?e'$› **and** ‹$x \in ($*vars-of-lit* $?l')$›
have $x \in ($*vars-of-eq* $?e')$ **by** *auto*
from *this* **and** ‹$?e' = (Eq\ ?t'\ ?s')$› **have** $x \in ($*vars-of* $?t' \cup$ *vars-of* $?s')$ **by** *auto*

then have i:$\neg($*ground-term* $?t') \vee \neg($*ground-term* $?s')$ **unfolding** *ground-term-def*
by *auto*
from ‹*vars-of-eq* e = *vars-of* $t \cup$ *vars-of* s› **and** ‹*vars-of-lit* l = *vars-of-eq* e›
and
‹*vars-of-lit* $l \subseteq ?V$› **have** *vars-of* $t \subseteq ?V$ **and** *vars-of* $s \subseteq ?V$ **by** *auto*
from ‹*vars-of* $t \subseteq ?V$› **and** ‹*ground-on* $?V\ \sigma$› **have** *ground-on* $($*vars-of* $t)\ \sigma$
unfolding *ground-on-def* **by** *auto*
then have ii:*ground-term* $?t'$ **using** *ground-instance* **by** *auto*
from ‹*vars-of* $s \subseteq ?V$› **and** ‹*ground-on* $?V\ \sigma$› **have** *ground-on* $($*vars-of* $s)\ \sigma$
unfolding *ground-on-def* **by** *auto*
then have iii:*ground-term* $?s'$ **using** *ground-instance* **by** *auto*
from i **and** ii **and** iii **show** *False* **by** *auto*
qed

lemma *ground-clauses-and-ground-substs*:
assumes *ground-clause* $($*subst-cl* $C\ \sigma)$
shows *ground-on* $($*vars-of-cl* $C)\ \sigma$
proof $($*rule ccontr*$)$
assume $\neg$*ground-on* $($*vars-of-cl* $C)\ \sigma$
from *this* **obtain** x **where** $x \in$ *vars-of-cl* C **and** $\neg$ *ground-term* $($*subst* $(Var\ x)$
$\sigma)$
unfolding *ground-on-def* **by** *auto*
from ‹$\neg$ *ground-term* $($*subst* $(Var\ x)\ \sigma)$› **obtain** y **where**
$y \in$ *vars-of* $($*subst* $(Var\ x)\ \sigma)$ **unfolding** *ground-term-def* **by** *auto*
from ‹$x \in$ *vars-of-cl* C› **obtain** L **where** $L \in C$ **and** $x \in$ *vars-of-lit* L **by** *auto*
from ‹$x \in$ *vars-of-lit* L› **obtain** e **where** $L = Pos\ e \vee L = Neg\ e$ **and** $x \in$
vars-of-eq e
by $($*metis vars-of-lit.elims*$)$
from ‹$x \in$ *vars-of-eq* e› **obtain** $t\ s$ **where** $e = (Eq\ t\ s)$ **and** $x \in$ *vars-of* $t \cup$
vars-of s
by $($*metis vars-of-eq.elims*$)$
from *this* **have** $x \in$ *vars-of* $t \vee x \in$ *vars-of* s **by** *auto*
then have $y \in$ *vars-of-eq* $($*subst-equation* $e\ \sigma)$
proof

assume $x \in$ *vars-of t*
have *i*: *vars-of* (*subst t σ*) = $\bigcup$ { *V*. ∃ *x*. *x* ∈ *vars-of t* ∧ *V* = *vars-of* (*subst* (*Var x*) *σ*) }
using *vars-of-instances* [*of t σ*] **by** *meson*
from ‹*x* ∈ *vars-of t*› *i*
have *vars-of* (*subst* (*Var x*) *σ*) ⊆ *vars-of* (*subst t σ*)
by *auto*
from *this* **and** ‹*y* ∈ *vars-of* (*subst* (*Var x*) *σ*)› ‹*e* = (*Eq t s*)› **show** *?thesis* **by** *auto*
next
assume *x* ∈ *vars-of s*
have *i*: *vars-of* (*subst s σ*) = $\bigcup$ { *V*. ∃ *x*. *x* ∈ *vars-of s* ∧ *V* = *vars-of* (*subst* (*Var x*) *σ*) }
using *vars-of-instances* [*of s σ*] **by** *meson*
from ‹*x* ∈ *vars-of s*› *i*
have *vars-of* (*subst* (*Var x*) *σ*) ⊆ *vars-of* (*subst s σ*)
by *auto*
from *this* **and** ‹*y* ∈ *vars-of* (*subst* (*Var x*) *σ*)› ‹*e* = (*Eq t s*)› **show** *?thesis* **by** *auto*
qed
from *this* **and** ‹*L* = *Pos e* ∨ *L* = *Neg e*› **have** *y* ∈ *vars-of-lit* (*subst-lit L σ*)
by *auto*
from *this* **and** ‹*L* ∈ *C*› **have** *y* ∈ *vars-of-cl* (*subst-cl C σ*) **by** *auto*
from *this* **and** *assms*(*1*) **show** *False* **by** *auto*
qed

lemma *ground-instance-exists*:
assumes *finite C*
shows ∃ *σ*. (*ground-clause* (*subst-cl C σ*))
proof −
let *?V* = (*vars-of-cl C*)
from *assms* **have** *finite ?V* **using** *set-of-variables-is-finite-cl* **by** *auto*
from *this* **obtain** *σ* **where** *ground-on ?V σ*
using *ground-subst-exists* **by** *blast*
let *?D* = (*subst-cl C σ*)
from ‹*ground-on ?V σ*› **have** (*ground-clause ?D*) **using** *ground-substs-yield-ground-clause* [*of C σ*] **by** *auto*
then show *?thesis* **by** *auto*
qed

lemma *composition-of-substs* :
shows (*subst* (*subst t σ*) *η*)
= (*subst t* (*comp σ η*))
by *simp*

lemma *composition-of-substs-eq* :
shows (*subst-equation* (*subst-equation e σ*) *η*)
= (*subst-equation e* (*comp σ η*))
by (*metis subst-equation.simps composition-of-substs vars-of-eq.elims*)

```
lemma composition-of-substs-lit :
  shows (subst-lit (subst-lit l σ) η)
    = (subst-lit l (comp σ η))
by (metis subst-lit.simps(1) subst-lit.simps(2)
    composition-of-substs-eq positive-literal.cases)

lemma composition-of-substs-cl :
  shows (subst-cl (subst-cl C σ) η)
    = (subst-cl C (comp σ η))
proof −
  let ?f = (λx. (subst-lit (subst-lit x  σ) η))
  let ?f′ = (λx. (subst-lit x (comp σ η)))
  have ∀ l. (?f l) = (?f′ l) using composition-of-substs-lit by auto
  then show ?thesis by auto
qed

lemma substs-preserve-ground-lit :
  assumes ground-clause C
  assumes y ∈ C
  shows subst-lit y σ = y
proof −
    obtain t and s where y = Pos (Eq t s) ∨ y = Neg (Eq t s)
      by (metis subst-equation.elims get-sign.elims)
    from this have vars-of t ⊆ vars-of-lit y by auto
    from this and ‹y ∈ C› have vars-of t ⊆ vars-of-cl C by auto
     from this and assms(1) have ground-term t unfolding ground-term-def by
auto
    then have subst t σ = t using substs-preserve-ground-terms by auto
    from ‹y = Pos (Eq t s) ∨ y = Neg (Eq t s)› have vars-of s ⊆ vars-of-lit y by
auto
    from this and ‹y ∈ C› have vars-of s ⊆ vars-of-cl C by auto
     from this and assms(1) have ground-term s unfolding ground-term-def by
auto
    then have subst s σ = s using substs-preserve-ground-terms by auto
     from ‹subst s σ = s› and ‹subst t σ = t› and ‹y = Pos (Eq t s) ∨ y = Neg
(Eq t s)›
    show subst-lit y σ = y by auto
qed

lemma substs-preserve-ground-clause :
  assumes ground-clause C
  shows subst-cl C σ = C
proof
  show subst-cl C σ ⊆ C
  proof
    fix x assume x ∈ subst-cl C σ
    then obtain y where y ∈ C and x = subst-lit y σ by auto
    from assms(1) and ‹y ∈ C› and ‹x = subst-lit y σ›
```

have $x = y$ **using** *substs-preserve-ground-lit* **by** *auto*
from *this* **and** ‹$y \in C$› **show** $x \in C$ **by** *auto*
qed
next
show $C \subseteq$ *subst-cl* $C\ \sigma$
proof
fix x **assume** $x \in C$
then have *subst-lit* $x\ \sigma \in$ *subst-cl* $C\ \sigma$ **by** *auto*
from *assms*(*1*) **and** ‹$x \in C$› **have** $x =$ *subst-lit* $x\ \sigma$
using *substs-preserve-ground-lit* [*of* $C\ x$] **by** *auto*
from *this* **and** ‹$x \in C$› **show** $x \in$ *subst-cl* $C\ \sigma$ **by** *auto*
qed
qed

lemma *substs-preserve-finiteness* :
assumes *finite* C
shows *finite* (*subst-cl* $C\ \sigma$)
proof −
from *assms*(*1*) **show** *?thesis* **using** *Finite-Set.finite-imageI* **by** *auto*
qed

We prove that two equal substitutions yield the same objects.

lemma *subst-eq-eq* :
assumes *subst-eq* $\sigma\ \eta$
shows *subst-equation* $e\ \sigma =$ *subst-equation* $e\ \eta$
proof −
obtain t **and** s **where** $e =$ *Eq* $t\ s$ **using** *equation.exhaust* **by** *auto*
from *assms*(*1*) **have** *subst* $s\ \sigma =$ *subst* $s\ \eta$ **by** *auto*
from *assms*(*1*) **have** *subst* $t\ \sigma =$ *subst* $t\ \eta$ **by** *auto*
from ‹*subst* $s\ \sigma =$ *subst* $s\ \eta$› ‹*subst* $t\ \sigma =$ *subst* $t\ \eta$›
and ‹$e =$ *Eq* $t\ s$› **show** *?thesis* **by** *auto*
qed

lemma *subst-eq-lit* :
assumes *subst-eq* $\sigma\ \eta$
shows *subst-lit* $l\ \sigma =$ *subst-lit* $l\ \eta$
proof −
obtain e **where** $l =$ *Pos* $e \lor l =$ *Neg* e **using** *literal.exhaust* **by** *auto*
from *assms*(*1*) **have** *subst-equation* $e\ \sigma =$ *subst-equation* $e\ \eta$ **using** *subst-eq-eq*
by *auto*
from *this* **and** ‹$l =$ *Pos* $e \lor l =$ *Neg* e› **show** *?thesis* **by** *auto*
qed

lemma *subst-eq-cl*:
assumes *subst-eq* $\sigma\ \eta$
shows *subst-cl* $C\ \sigma =$ *subst-cl* $C\ \eta$
proof (*rule ccontr*)
assume *subst-cl* $C\ \sigma \neq$ *subst-cl* $C\ \eta$
then obtain L **where** $L \in C$ **and** *subst-lit* $L\ \sigma \neq$ *subst-lit* $L\ \eta$

by *force*
from *assms*(*1*) **and** ‹*subst-lit L σ* ≠ *subst-lit L η*›
show *False* **using** *subst-eq-lit* **by** *auto*
qed

lemma *coincide-on-eq* :
assumes *coincide-on σ η* (*vars-of-eq e*)
shows *subst-equation e σ* = *subst-equation e η*
proof −
obtain *t* **and** *s* **where** *e* = *Eq t s* **using** *equation.exhaust* **by** *auto*
then have *vars-of t* ⊆ *vars-of-eq e* **by** *simp*
from *this* **and** ‹*coincide-on σ η* (*vars-of-eq e*)› **have** *coincide-on σ η* (*vars-of t*)
unfolding *coincide-on-def* **by** *auto*
from *this* **have** *subst t σ* = *subst t η* **using** *coincide-on-term* **by** *auto*
from ‹*e* = *Eq t s*› **have** *vars-of s* ⊆ *vars-of-eq e* **by** *simp*
from *this* **and** ‹*coincide-on σ η* (*vars-of-eq e*)› **have** *coincide-on σ η* (*vars-of s*)
unfolding *coincide-on-def* **by** *auto*
from *this* **have** *subst s σ* = *subst s η* **using** *coincide-on-term* **by** *auto*
from ‹*subst t σ* = *subst t η*›
and ‹*subst s σ* = *subst s η*›
and ‹*e* = *Eq t s*› **show** *?thesis* **by** *auto*
qed

lemma *coincide-on-lit* :
assumes *coincide-on σ η* (*vars-of-lit l*)
shows *subst-lit l σ* = *subst-lit l η*
proof −
obtain *e* **where** *l* = *Pos e* ∨ *l* = *Neg e* **using** *literal.exhaust* **by** *auto*
then have *vars-of-eq e* ⊆ *vars-of-lit l* **by** *auto*
from *this* **and** ‹*coincide-on σ η* (*vars-of-lit l*)› **have** *coincide-on σ η* (*vars-of-eq e*)
unfolding *coincide-on-def* **by** *auto*
from *this* **have** *subst-equation e σ* = *subst-equation e η*
using *coincide-on-eq* **by** *auto*
from *this* **and** ‹*l* = *Pos e* ∨ *l* = *Neg e*› **show** *?thesis* **by** *auto*
qed

lemma *coincide-on-cl* :
assumes *coincide-on σ η* (*vars-of-cl C*)
shows *subst-cl C σ* = *subst-cl C η*
proof (*rule ccontr*)
assume *subst-cl C σ* ≠ *subst-cl C η*
then obtain *L* **where** *L* ∈ *C* **and** *subst-lit L σ* ≠ *subst-lit L η*
by *force*
from ‹*L* ∈ *C*› **have** *vars-of-lit L* ⊆ *vars-of-cl C* **by** *auto*
from *this* **and** *assms* **have** *coincide-on σ η* (*vars-of-lit L*) **unfolding** *coincide-on-def* **by** *auto*
from *this* **and** ‹*subst-lit L σ* ≠ *subst-lit L η*›
show *False* **using** *coincide-on-lit* **by** *auto*

qed

3.2 Semantics

Interpretations are congruences on the set of terms.

definition *fo-interpretation* :: *'a binary-relation-on-trms* $\Rightarrow$ *bool*
where
(*fo-interpretation* x) = (*congruence* x)

fun *validate-ground-eq* :: *'a binary-relation-on-trms* $\Rightarrow$ *'a equation* $\Rightarrow$ *bool*
where
(*validate-ground-eq* I (*Eq* t s) = (I t s))

fun *validate-ground-lit* :: *'a binary-relation-on-trms* $\Rightarrow$ *'a literal* $\Rightarrow$ *bool*
where
validate-ground-lit I (*Pos* E) = (*validate-ground-eq* I E) |
validate-ground-lit I (*Neg* E) = ($\neg$(*validate-ground-eq* I E))

fun *validate-ground-clause* :: *'a binary-relation-on-trms* $\Rightarrow$ *'a clause* $\Rightarrow$ *bool*
where
validate-ground-clause I C = ($\exists$ L.($L \in C$) $\wedge$ (*validate-ground-lit* I L))

fun *validate-clause* :: *'a binary-relation-on-trms* $\Rightarrow$ *'a clause* $\Rightarrow$ *bool*
where
validate-clause I C = ($\forall$ s. (*ground-clause* (*subst-cl* C s))
$\longrightarrow$ (*validate-ground-clause* I (*subst-cl* C s)))

fun *validate-clause-set* :: *'a binary-relation-on-trms* $\Rightarrow$ *'a clause set* $\Rightarrow$ *bool*
where
validate-clause-set I S = ($\forall$ C. ($C \in S$ $\longrightarrow$ (*validate-clause* I C)))

definition *clause-entails-clause* :: *'a clause* $\Rightarrow$ *'a clause* $\Rightarrow$ *bool*
where
clause-entails-clause C D = ($\forall$ I. (*fo-interpretation* I $\longrightarrow$ *validate-clause* I C $\longrightarrow$ *validate-clause* I D))

definition *set-entails-clause* :: *'a clause set* $\Rightarrow$ *'a clause* $\Rightarrow$ *bool*
where
set-entails-clause S C = ($\forall$ I. (*fo-interpretation* I $\longrightarrow$ *validate-clause-set* I S $\longrightarrow$ *validate-clause* I C))

definition *satisfiable-clause-set* :: *'a clause set* $\Rightarrow$ *bool*
where
(*satisfiable-clause-set* S) = ($\exists$ I. (*fo-interpretation* I) $\wedge$ (*validate-clause-set* I S))

We state basic properties of the entailment relation.

lemma *set-entails-clause-member*:
assumes $C \in S$
shows *set-entails-clause* S C

proof (*rule ccontr*)
  assume ¬ *?thesis*
   from *this* **obtain** *I* **where** *fo-interpretation I validate-clause-set I S* ¬ *validate-clause I C*
    unfolding *set-entails-clause-def* **by** *auto*
  from ‹*validate-clause-set I S*› **and** *assms*(*1*) ‹¬ *validate-clause I C*› **show** *False* **by** *auto*
qed

lemma *instances-are-entailed* :
  assumes *validate-clause I C*
  shows *validate-clause I* (*subst-cl C σ*)
proof (*rule ccontr*)
  assume ¬*validate-clause I* (*subst-cl C σ*)
  then obtain *η*
    where ¬*validate-ground-clause I* (*subst-cl* (*subst-cl C σ*) *η*)
      and *ground-clause* (*subst-cl* (*subst-cl C σ*) *η*)
    by *auto*
  then have *i*: ¬*validate-ground-clause I* (*subst-cl C* (*comp σ η*))
   using *composition-of-substs-cl* **by** *metis*
  from ‹*ground-clause* (*subst-cl* (*subst-cl C σ*) *η*)›
    have *ii*: *ground-clause* (*subst-cl C* (*comp σ η*))
    using *composition-of-substs-cl* **by** *metis*
  from *i* **and** *ii* **have** ¬*validate-clause I C* **by** *auto*
  from ‹¬*validate-clause I C*› **and** ‹*validate-clause I C*› **show** *False* **by** *blast*
qed

We prove that two equivalent substitutions yield equivalent objects.

lemma *equivalent-on-eq* :
  assumes *equivalent-on σ η* (*vars-of-eq e*) *I*
  assumes *fo-interpretation I*
 shows (*validate-ground-eq I* (*subst-equation e σ*)) = (*validate-ground-eq I* (*subst-equation e η*))
proof −
  obtain *t* **and** *s* **where** *e* = *Eq t s* **using** *equation.exhaust* **by** *auto*
  then have *vars-of t* ⊆ *vars-of-eq e* **by** *simp*
  from *this* **and** *assms*(*1*) **have** *equivalent-on σ η* (*vars-of t*) *I*
    unfolding *equivalent-on-def* **by** *auto*
  from *this assms*(*2*)
    have *I* (*subst t σ*) (*subst t η*) **using** *equivalent-on-term*
    unfolding *fo-interpretation-def* **by** *auto*
  from ‹*e* = *Eq t s*› **have** *vars-of s* ⊆ *vars-of-eq e* **by** *simp*
 from *this* **and** ‹*equivalent-on σ η* (*vars-of-eq e*) *I*› **have** *equivalent-on σ η* (*vars-of s*) *I*
    unfolding *equivalent-on-def* **by** *auto*
  from *this assms*(*2*) **have** *I* (*subst s σ*) (*subst s η*)
    using *equivalent-on-term* **unfolding** *fo-interpretation-def* **by** *auto*
  from *assms*(*2*) ‹*I* (*subst t σ*) (*subst t η*)›
    and ‹*I* (*subst s σ*) (*subst s η*)›

and ‹*e* = *Eq t s*› **show** *?thesis* **unfolding** *fo-interpretation-def congruence-def equivalence-relation-def*
transitive-def symmetric-def reflexive-def
by (*metis* (*full-types*) *subst-equation.simps validate-ground-eq.simps*)
qed

lemma *equivalent-on-lit* :
assumes *equivalent-on* σ η (*vars-of-lit l*) *I*
assumes *fo-interpretation I*
shows (*validate-ground-lit I* (*subst-lit l* σ))
= (*validate-ground-lit I* (*subst-lit l* η))
proof −
obtain *e* **where** $l = Pos\ e \vee l = Neg\ e$ **using** *literal.exhaust* **by** *auto*
then have *vars-of-eq e* $\subseteq$ *vars-of-lit l* **by** *auto*
from *this* **and** ‹*equivalent-on* σ η (*vars-of-lit l*) *I*› **have** *equivalent-on* σ η (*vars-of-eq e*) *I*
unfolding *equivalent-on-def* **by** *auto*
from *this assms*(*2*) **have** (*validate-ground-eq I* (*subst-equation e* σ)) = (*validate-ground-eq I* (*subst-equation e* η))
using *equivalent-on-eq* **by** *auto*
from *this* **and** ‹$l = Pos\ e \vee l = Neg\ e$› **show** *?thesis* **by** *auto*
qed

lemma *equivalent-on-cl* :
assumes *equivalent-on* σ η (*vars-of-cl C*) *I*
assumes *fo-interpretation I*
shows (*validate-ground-clause I* (*subst-cl C* σ))
= (*validate-ground-clause I* (*subst-cl C* η))
proof (*rule ccontr*)
assume (*validate-ground-clause I* (*subst-cl C* σ))
$\neq$ (*validate-ground-clause I* (*subst-cl C* η))
then obtain *L* **where** $L \in C$ **and** (*validate-ground-lit I* (*subst-lit L* σ))
$\neq$ (*validate-ground-lit I* (*subst-lit L* η))
by *force*
from ‹$L \in C$› **have** *vars-of-lit L* $\subseteq$ *vars-of-cl C* **by** *auto*
from *this* **and** *assms* **have** *equivalent-on* σ η (*vars-of-lit L*) *I* **unfolding** *equivalent-on-def* **by** *auto*
from *this assms*(*2*) **and** ‹(*validate-ground-lit I* (*subst-lit L* σ))
$\neq$ (*validate-ground-lit I* (*subst-lit L* η))›
show *False* **using** *equivalent-on-lit* **by** *metis*
qed

end
theory *superposition*

imports *Main terms equational-clausal-logic well-founded-continued HOL−Library.Multiset multisets-continued*

begin

4 Definition of the Superposition Calculus

4.1 Extended Clauses

An extended clause is a clause associated with a set of terms. The intended meaning is that the terms occurring in the attached set are assumed to be in normal form: any application of the superposition rule on these terms is therefore useless and can be blocked. Initially the set of irreducible terms attached to each clause is empty. At each inference step, new terms can be added or deleted from this set.

datatype $'a$ *eclause* = *Ecl* $'a$ *clause* $'a$ *trm set*

fun *subst-ecl*
where
$(\mathit{subst\text{-}ecl}\ (\mathit{Ecl}\ C\ S)\ \sigma) =$
$(\mathit{Ecl}\ (\mathit{subst\text{-}cl}\ C\ \sigma)\ \{\ s.\ (\exists\, t.\ (s = (\mathit{subst}\ t\ \sigma) \wedge t \in S))\ \})$

fun *cl-ecl*
where
$(\mathit{cl\text{-}ecl}\ (\mathit{Ecl}\ C\ X)) = C$

fun *trms-ecl*
where
$(\mathit{trms\text{-}ecl}\ (\mathit{Ecl}\ C\ X)) = X$

definition *renaming-cl*
where $\mathit{renaming\text{-}cl}\ C\ D = (\exists\ \eta.\ (\mathit{renaming}\ \eta\ (\mathit{vars\text{-}of\text{-}cl}\ (\mathit{cl\text{-}ecl}\ C))) \wedge D = (\mathit{subst\text{-}ecl}\ C\ \eta))$

definition *closed-under-renaming*
where $\mathit{closed\text{-}under\text{-}renaming}\ S = (\forall\ C\ D.$
$(C \in S) \longrightarrow (\mathit{renaming\text{-}cl}\ C\ D) \longrightarrow (D \in S))$

definition *variable-disjoint*
where $(\mathit{variable\text{-}disjoint}\ C\ D) = ((\mathit{vars\text{-}of\text{-}cl}\ (\mathit{cl\text{-}ecl}\ C)) \cap (\mathit{vars\text{-}of\text{-}cl}\ (\mathit{cl\text{-}ecl}\ D)) = \{\})$

4.2 Orders and Selection Functions

We assume that the set of variables is infinite (so that shared variables can be renamed away) and that the following objects are given:

(i) A term ordering that is total on ground terms, well-founded and closed under contextual embedding and substitutions. This ordering is used as usual to orient equations and to restrict the application of the replacement rule.

(ii) A selection function, mapping each clause to a (possibly empty) set of negative literals. We assume that this selection function is closed under renamings.

(iii) A function mapping every extended clause to an order on positions, which contains the (reversed) prefix ordering. This order allows one to control the order in which the subterms are rewritten (terms occurring at minimal positions are considered with the highest priority).

(iv) A function *filter-trms* that allows one to filter away some of the terms attached to a given extended clause (it can be used for instance to remove terms if the set becomes too big). The standard superposition calculus corresponds to the case where this function always returns the empty set.

locale *basic-superposition* =

fixes *trm-ord* :: (*'a trm* × *'a trm*) *set*
fixes *sel* :: *'a clause* ⇒ *'a clause*
fixes *pos-ord* :: *'a eclause* ⇒ *'a trm* ⇒ (*position* × *position*) *set*
fixes *vars* :: *'a set*
fixes *filter-trms* :: *'a clause* ⇒ *'a trm set* ⇒ *'a trm set*
assumes
trm-ord-wf : (*wf trm-ord*)
and *trm-ord-ground-total* :
(∀ *x y*. ((*ground-term x*) ⟶ (*ground-term y*) ⟶ *x* ≠ *y*
⟶ ((*x*,*y*) ∈ *trm-ord* ∨ (*y*,*x*) ∈ *trm-ord*)))
and *trm-ord-trans* : *trans trm-ord*
and *trm-ord-irrefl* : *irrefl trm-ord*
and *trm-ord-reduction-left* : ∀ *x1 x2 y*. (*x1*,*x2*) ∈ *trm-ord*
⟶ ((*Comb x1 y*),(*Comb x2 y*)) ∈ *trm-ord*
and *trm-ord-reduction-right* : ∀ *x1 x2 y*. (*x1*,*x2*) ∈ *trm-ord*
⟶ ((*Comb y x1*),(*Comb y x2*)) ∈ *trm-ord*
and *trm-ord-subterm* : ∀ *x1 x2*. (*x1*,(*Comb x1 x2*)) ∈ *trm-ord*
∧ (*x2*,(*Comb x1 x2*)) ∈ *trm-ord*
and *trm-ord-subst* :
∀ *s x y*. (*x*,*y*) ∈ *trm-ord* ⟶ ((*subst x s*),(*subst y s*)) ∈ *trm-ord*
and *pos-ord-irrefl* : (∀ *x y*. (*irrefl* (*pos-ord x y*)))
and *pos-ord-trans* : (∀ *x*. (*trans* (*pos-ord x y*)))
and *pos-ord-prefix* : ∀ *x y p q r*. ((*q*,*p*) ∈ (*pos-ord x y*) ⟶ ((*append q r*),*p*) ∈ (*pos-ord x y*))
and *pos-ord-nil* : ∀ *x y p* . (*p* ≠ *Nil*) ⟶ (*p*,*Nil*) ∈ (*pos-ord x y*)
and *sel-neg*: (∀ *x*. ((*sel* (*cl-ecl x*)) ⊆ (*cl-ecl x*))
∧ (∀ *y* ∈ *sel* (*cl-ecl x*). (*negative-literal y*)))
and *sel-renaming*: ∀ η *C*. ((*renaming* η (*vars-of-cl C*)) ⟶ *sel C* = {} ⟶ *sel* (*subst-cl C* η) = {})
and *infinite-vars*: ¬ (*finite vars*)
and *filter-trms-inclusion*: *filter-trms C E* ⊆ *E*
begin

We provide some functions to decompose a literal in a way that is compatible with the ordering and establish some basic properties.

definition *orient-lit* :: *'a literal* ⇒ *'a trm* ⇒ *'a trm* ⇒ *sign* ⇒ *bool*
where
  (*orient-lit L u v s*) =
    ((((*L* = (*Pos* (*Eq u v*))) ∨ (*L* = (*Pos* (*Eq v u*)))) ∧ ((*u*,*v*) ∉ *trm-ord*) ∧ (*s* = *pos*))
    ∨
    (((*L* = (*Neg* (*Eq u v*))) ∨ (*L* = (*Neg* (*Eq v u*)))) ∧ ((*u*,*v*) ∉ *trm-ord*) ∧ (*s* = *neg*)))

definition *orient-lit-inst* :: *'a literal* ⇒ *'a trm* ⇒ *'a trm* ⇒ *sign* ⇒ *'a subst* ⇒ *bool*
where
  (*orient-lit-inst L u v s σ*) =
    ((((*L* = (*Pos* (*Eq u v*))) ∨ (*L* = (*Pos* (*Eq v u*))))
      ∧ (((*subst u σ*),(*subst v σ*)) ∉ *trm-ord*) ∧ (*s* = *pos*))
    ∨
    (((*L* = (*Neg* (*Eq u v*))) ∨ (*L* = (*Neg* (*Eq v u*)))) ∧ (((*subst u σ*),(*subst v σ*))
    ∉ *trm-ord*) ∧ (*s* = *neg*)))

lemma *lift-orient-lit*:
  assumes *orient-lit-inst L t s p σ*
  shows *orient-lit* (*subst-lit L σ*) (*subst t σ*) (*subst s σ*) *p*
unfolding *orient-lit-inst-def orient-lit-def* **using** *assms orient-lit-inst-def* **by** *auto*

lemma *orient-lit-vars*:
  assumes *orient-lit L t s p*
  shows *vars-of t* ⊆ *vars-of-lit L* ∧ *vars-of s* ⊆ *vars-of-lit L*
proof −
  have *p* = *neg* ∨ *p* = *pos* **using** *sign.exhaust* **by** *auto*
  then show *?thesis*
  proof
    assume *p* = *neg*
    from *this* **and** *assms*(*1*) **have** (*L* = *Neg* (*Eq t s*)) ∨ (*L* = (*Neg* (*Eq s t*)))
      unfolding *orient-lit-def* **by** *auto*
    then show *?thesis*
    proof
      assume *L* = *Neg* (*Eq t s*)
      then have *vars-of-lit L* = *vars-of t* ∪ *vars-of s* **by** *simp*
      from *this* **show** *?thesis* **by** *simp*
    next
      assume *L* = *Neg* (*Eq s t*)
      then have *vars-of-lit L* = *vars-of s* ∪ *vars-of t* **by** *simp*
      from *this* **show** *?thesis* **by** *simp*
    qed
    next assume *p* = *pos*
    from *this* **and** *assms*(*1*) **have** (*L* = *Pos* (*Eq t s*)) ∨ (*L* = (*Pos* (*Eq s t*)))

```
      unfolding orient-lit-def by auto
    then show ?thesis
    proof
      assume L = Pos (Eq t s)
      then have vars-of-lit L = vars-of t ∪ vars-of s by simp
      from this show ?thesis by simp
    next
      assume L = Pos (Eq s t)
      then have vars-of-lit L = vars-of s ∪ vars-of t by simp
      from this show ?thesis by simp
    qed
  qed
qed

lemma orient-lit-inst-vars:
  assumes orient-lit-inst L t s p σ
  shows vars-of t ⊆ vars-of-lit L ∧ vars-of s ⊆ vars-of-lit L
proof –
  have p = neg ∨ p = pos using sign.exhaust by auto
  then show ?thesis
  proof
    assume p = neg
    from this and assms(1) have (L = Neg (Eq t s)) ∨ (L = (Neg (Eq s t)))
      unfolding orient-lit-inst-def by auto
    then show ?thesis
    proof
      assume L = Neg (Eq t s)
      then have vars-of-lit L = vars-of t ∪ vars-of s by simp
      from this show ?thesis by simp
    next
      assume L = Neg (Eq s t)
      then have vars-of-lit L = vars-of s ∪ vars-of t by simp
      from this show ?thesis by simp
    qed
    next assume p = pos
    from this and assms(1) have (L = Pos (Eq t s)) ∨ (L = (Pos (Eq s t)))
      unfolding orient-lit-inst-def by auto
    then show ?thesis
    proof
      assume L = Pos (Eq t s)
      then have vars-of-lit L = vars-of t ∪ vars-of s by simp
      from this show ?thesis by simp
    next
      assume L = Pos (Eq s t)
      then have vars-of-lit L = vars-of s ∪ vars-of t by simp
      from this show ?thesis by simp
    qed
  qed
qed
```

lemma *orient-lit-inst-coincide*:
  assumes *orient-lit-inst L1 t s polarity* σ
  assumes *coincide-on* σ η (*vars-of-lit L1*)
  shows *orient-lit-inst L1 t s polarity* η
proof −
  have *polarity* = *pos* ∨ *polarity* = *neg* **using** *sign.exhaust* **by** *auto*
  then show *?thesis*
  proof
    assume *polarity* = *pos*
    from *this* **and** *assms*(*1*) **have** *L1* = *Pos* (*Eq t s*) ∨ *L1* = *Pos* (*Eq s t*)
      and ((*subst t* σ), (*subst s* σ)) ∉ *trm-ord*
      unfolding *orient-lit-inst-def* **by** *auto*
    from ‹*L1* = *Pos* (*Eq t s*) ∨ *L1* = *Pos* (*Eq s t*)›
      have *vars-of t* ⊆ *vars-of-lit L1* **and** *vars-of s* ⊆ *vars-of-lit L1* **by** *auto*
    from ‹*vars-of t* ⊆ *vars-of-lit L1*› **and** *assms*(*2*) **have** *coincide-on* σ η (*vars-of*
t)
      unfolding *coincide-on-def* **by** *auto*
    from ‹*vars-of s* ⊆ *vars-of-lit L1*› **and** *assms*(*2*) **have** *coincide-on* σ η (*vars-of*
s)
      unfolding *coincide-on-def* **by** *auto*
    from ‹((*subst t* σ), (*subst s* σ)) ∉ *trm-ord*›
     and ‹*coincide-on* σ η (*vars-of t*)› **and** ‹*coincide-on* σ η (*vars-of s*)› *assms*(*2*)

    have ((*subst t* η), (*subst s* η)) ∉ *trm-ord*
      using *coincide-on-term* **by** *metis*
    from *this* **and** ‹*polarity* = *pos*› **and** ‹*L1* = *Pos* (*Eq t s*) ∨ *L1* = *Pos* (*Eq s t*)›
show *?thesis*
      unfolding *orient-lit-inst-def* **by** *auto*
    next assume *polarity* = *neg*
    from *this* **and** *assms*(*1*) **have** *L1* = *Neg* (*Eq t s*) ∨ *L1* = *Neg* (*Eq s t*)
      and ((*subst t* σ), (*subst s* σ)) ∉ *trm-ord*
      unfolding *orient-lit-inst-def* **by** *auto*
    from ‹*L1* = *Neg* (*Eq t s*) ∨ *L1* = *Neg* (*Eq s t*)›
      have *vars-of t* ⊆ *vars-of-lit L1* **and** *vars-of s* ⊆ *vars-of-lit L1* **by** *auto*
    from ‹*vars-of t* ⊆ *vars-of-lit L1*› **and** *assms*(*2*) **have** *coincide-on* σ η (*vars-of*
t)
      unfolding *coincide-on-def* **by** *auto*
    from ‹*vars-of s* ⊆ *vars-of-lit L1*› **and** *assms*(*2*) **have** *coincide-on* σ η (*vars-of*
s)
      unfolding *coincide-on-def* **by** *auto*
    from ‹((*subst t* σ), (*subst s* σ)) ∉ *trm-ord*›
     and ‹*coincide-on* σ η (*vars-of t*)› **and** ‹*coincide-on* σ η (*vars-of s*)› *assms*(*2*)

    have ((*subst t* η), (*subst s* η)) ∉ *trm-ord*
      using *coincide-on-term* **by** *metis*
    from *this* **and** ‹*polarity* = *neg*› **and** ‹*L1* = *Neg* (*Eq t s*) ∨ *L1* = *Neg* (*Eq s t*)›
show *?thesis*
      unfolding *orient-lit-inst-def* **by** *auto*

```
  qed
qed

lemma orient-lit-inst-subterms:
  assumes orient-lit-inst L t s polarity σ
  assumes u ∈ subterms-of t
  shows u ∈ subterms-of-lit L
proof −
  have polarity = pos ∨ polarity = neg using sign.exhaust by auto
  then show ?thesis
  proof
    assume polarity = pos
    from this and assms(1) have L = (Pos (Eq t s)) ∨ L = (Pos (Eq s t))
      unfolding orient-lit-inst-def by auto
    then show ?thesis
    proof
      assume L = (Pos (Eq t s))
      from this and assms(2) show ?thesis  by simp
      next assume L = (Pos (Eq s t))
      from this and assms(2) show ?thesis  by simp
    qed
  next
    assume polarity = neg
    from this and assms(1) have L = (Neg (Eq t s)) ∨ L = (Neg (Eq s t))
      unfolding orient-lit-inst-def by auto
    then show ?thesis
    proof
      assume L = (Neg (Eq t s))
      from this and assms(2) show ?thesis  by simp
      next assume L = (Neg (Eq s t))
      from this and assms(2) show ?thesis  by simp
    qed
  qed
qed
```

4.3 Clause Ordering

Clauses and extended clauses are ordered by transforming them into multisets of multisets of terms. To avoid any problem with the merging of identical literals, the multiset is assigned to a pair clause-substitution rather than to an instantiated clause.

We first map every literal to a multiset of terms, using the usual conventions and then define the multisets associated with clauses and extended clauses.

```
fun mset-lit :: 'a literal ⇒ 'a trm multiset
  where mset-lit (Pos (Eq t s)) = {# t,s #} |
        mset-lit (Neg (Eq t s)) = {# t,t,s,s #}

fun mset-cl
```

where *mset-cl* $(C,\sigma) = \{\#$ (*mset-lit* (*subst-lit* x σ)). $x \in\#$ (*mset-set* C) $\#\}$

fun *mset-ecl*
where *mset-ecl* $(C,\sigma) = \{\#$ (*mset-lit* (*subst-lit* x σ)). $x \in\#$ (*mset-set* (*cl-ecl* C)) $\#\}$

lemma *mset-ecl-conv*: *mset-ecl* (C, σ) = *mset-cl* (*cl-ecl* C, σ)
by *simp*

lemma *mset-ecl-and-mset-lit*:
assumes $L \in$ (*cl-ecl* C)
assumes *finite* (*cl-ecl* C)
shows (*mset-lit* (*subst-lit* L σ)) $\in\#$ (*mset-ecl* (C,σ))
proof −
from *assms*(*1*) *assms*(*2*) **have** $L \in\#$ (*mset-set* (*cl-ecl* C)) **by** (*simp*)
then show *?thesis*
proof −
have *f1*: *mset-set* (*cl-ecl* C) − $\{\#L\#\}$ + $\{\#L\#\}$ = *mset-set* (*cl-ecl* C)
by (*meson* ‹$L \in\#$ *mset-set* (*cl-ecl* C)› *insert-DiffM2*)
have *count* $\{\#$*mset-lit* (*subst-lit* L σ)$\#\}$ (*mset-lit* (*subst-lit* L σ)) = *1*
by *simp*
then show *?thesis*
by (*metis* (*no-types*, *lifting*) *f1 image-mset-add-mset insert-iff mset-ecl.simps set-mset-add-mset-insert union-mset-add-mset-right*)
qed
qed

lemma *ecl-ord-coincide*:
assumes *coincide-on* σ σ' (*vars-of-cl* (*cl-ecl* C))
shows *mset-ecl* (C,σ) = *mset-ecl* (C,σ')
proof −
have $\forall x$. ($x \in$ (*cl-ecl* C) $\longrightarrow$ ((*subst-lit* x σ) = (*subst-lit* x σ')))
proof ((*rule allI*),(*rule impI*))
fix x **assume** $x \in$ (*cl-ecl* C)
from *this* **have** *vars-of-lit* $x \subseteq$ (*vars-of-cl* (*cl-ecl* C)) **by** *auto*
from *this* **and** *assms*(*1*) **have** *coincide-on* σ σ' (*vars-of-lit* x) **unfolding** *coincide-on-def* **by** *auto*
from *this* **show** ((*subst-lit* x σ) = (*subst-lit* x σ'))
by (*simp add*: *coincide-on-lit*)
qed
then show *?thesis* **using** *equal-image-mset*
[*of* *cl-ecl* C λx. (*mset-lit* (*subst-lit* x σ)) λx. (*mset-lit* (*subst-lit* x σ'))]
by (*metis* *mset-ecl.simps*)
qed

Literal and clause orderings are obtained as usual as the multiset extensions of the term ordering.

definition *lit-ord* :: ($'a$ *literal* $\times$ $'a$ *literal*) *set*
where

lit-ord =
{ (x,y). (((*mset-lit* x),(*mset-lit* y)) ∈ (*mult trm-ord*)) }

lemma *mult-trm-ord-trans*:
shows *trans* (*mult trm-ord*)
by (*metis* (*no-types*, *lifting*) *mult-def transI transitive-closure-trans*(*2*))

lemma *lit-ord-trans*:
shows *trans lit-ord*
by (*metis* (*no-types*, *lifting*) *basic-superposition.lit-ord-def basic-superposition-axioms*

case-prodD case-prodI mem-Collect-eq mult-def transI transitive-closure-trans(*2*))

lemma *lit-ord-wf*:
shows *wf lit-ord*
proof −
from *trm-ord-wf* **have** *wf* (*mult trm-ord*) **using** *wf-mult* **by** *auto*
then show *?thesis*
using *lit-ord-def*
and *measure-wf* [*of* (*mult trm-ord*) *lit-ord mset-lit*]
by *blast*
qed

definition *ecl-ord* :: (('*a eclause* × '*a subst*) × ('*a eclause* × '*a subst*)) *set*
where
ecl-ord =
{ (x,y). (((*mset-ecl* x),(*mset-ecl* y)) ∈ (*mult* (*mult trm-ord*))) }

definition *ecl-ord-eq* :: (('*a eclause* × '*a subst*) × ('*a eclause* × '*a subst*)) *set*
where
ecl-ord-eq =
ecl-ord ∪ { (x,y). ((*mset-ecl* x) = (*mset-ecl* y)) }

definition *cl-ord* :: (('*a clause* × '*a subst*) × ('*a clause* × '*a subst*)) *set*
where
cl-ord =
{ (x,y). (((*mset-cl* x),(*mset-cl* y)) ∈ (*mult* (*mult trm-ord*))) }

definition *cl-ord-eq* :: (('*a clause* × '*a subst*) × ('*a clause* × '*a subst*)) *set*
where
cl-ord-eq = *cl-ord* ∪
{ (x,y). (*mset-cl* x) = (*mset-cl* y) }

lemma *member-ecl-ord-iff*:
$((C, \sigma_C), (D, \sigma_D))$ ∈ *ecl-ord* ⟷ ((*cl-ecl* C, σ_C), (*cl-ecl* D, σ_D)) ∈ *cl-ord*
by (*simp add*: *ecl-ord-def cl-ord-def*)

lemma *mult-mult-trm-ord-trans*:
shows *trans* (*mult* (*mult trm-ord*))

by (*metis* (*no-types*, *lifting*) *mult-def transI transitive-closure-trans*(*2*))

lemma *ecl-ord-trans*:
shows *trans ecl-ord*
by (*metis* (*no-types*, *lifting*) *basic-superposition.ecl-ord-def basic-superposition-axioms case-prodD*
case-prodI mem-Collect-eq mult-def transI transitive-closure-trans(*2*))

lemma *cl-ord-trans*:
shows *trans cl-ord*
by (*metis* (*no-types*, *lifting*) *basic-superposition.cl-ord-def basic-superposition-axioms case-prodD*
case-prodI mem-Collect-eq mult-def transI transitive-closure-trans(*2*))

lemma *cl-ord-eq-trans*:
shows *trans cl-ord-eq*
proof −
have $\forall$ *r*. *trans r* = ($\forall$ *p pa pb*. ((*p*::*'a literal set* × (*'a* × *'a trm*) *list*, *pa*) ∉ *r* ∨ (*pa*, *pb*) ∉ *r*)
∨ (*p*, *pb*) ∈ *r*)
by (*simp add*: *trans-def*)
then obtain *pp* :: ((*'a literal set* × (*'a* × *'a trm*) *list*) × *'a literal set* × (*'a* × *'a trm*) *list*) *set* ⇒ *'a literal set* × (*'a* × *'a trm*) *list* **and** *ppa* :: ((*'a literal set* × (*'a* × *'a trm*) *list*) × *'a literal set* × (*'a* × *'a trm*) *list*) *set* ⇒ *'a literal set* × (*'a* × *'a trm*) *list* **and** *ppb* :: ((*'a literal set* × (*'a* × *'a trm*) *list*) × *'a literal set* × (*'a* × *'a trm*) *list*) *set* ⇒ *'a literal set* × (*'a* × *'a trm*) *list* **where**
f1: ∀ *r*. (¬ *trans r* ∨ (∀ *p pa pb*. (*p*, *pa*) ∉ *r* ∨ (*pa*, *pb*) ∉ *r* ∨ (*p*, *pb*) ∈ *r*)) ∧ (*trans r* ∨ (*pp r*, *ppa r*) ∈ *r* ∧ (*ppa r*, *ppb r*) ∈ *r* ∧ (*pp r*, *ppb r*) ∉ *r*)
by (*metis* (*no-types*))
have *f2*: *trans* {(*p*, *pa*). (*mset-cl p*, *mset-cl pa*) ∈ *mult* (*mult trm-ord*)}
using *cl-ord-def cl-ord-trans* **by** *presburger*
{ assume ¬ (*case* (*ppa* (*cl-ord* ∪ {(*p*, *pa*). *mset-cl p* = *mset-cl pa*}), *ppb* (*cl-ord* ∪ {(*p*, *pa*). *mset-cl p* = *mset-cl pa*})) *of* (*p*, *pa*) ⇒ *mset-cl p* = *mset-cl pa*)
{ assume (*ppa* (*cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*}), *ppb* (*cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*})) ∈ {(*pa*, *p*). (*mset-cl pa*, *mset-cl p*) ∈ *mult* (*mult trm-ord*)}
moreover
{ assume (*ppa* (*cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*}), *ppb* (*cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*})) ∈ {(*pa*, *p*). (*mset-cl pa*, *mset-cl p*) ∈ *mult* (*mult trm-ord*)} ∧ (*mset-cl* (*pp* (*cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*})), *mset-cl* (*ppb* (*cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*}))) ∉ *mult* (*mult trm-ord*)
then have (*ppa* (*cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*}), *ppb* (*cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*})) ∈ {(*pa*, *p*). (*mset-cl pa*, *mset-cl p*) ∈ *mult* (*mult trm-ord*)} ∧ *mset-cl* (*pp* (*cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*})) ≠ *mset-cl* (*ppa* (*cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*}))
by *force*
then have ((*pp* (*cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*}), *ppb* (*cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*})) ∈ {(*pa*, *p*). (*mset-cl pa*, *mset-cl p*) ∈ *mult* (*mult trm-ord*)} ∨ (*pp* (*cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*}), *ppb* (*cl-ord* ∪ {(*pa*,

p). *mset-cl pa* = *mset-cl p*})) ∈ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*}) ∨ (*pp* (*cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*}), *ppa* (*cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*})) ∉ {(*pa*, *p*). (*mset-cl pa*, *mset-cl p*) ∈ *mult* (*mult trm-ord*)} ∧ (*pp* (*cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*}), *ppa* (*cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*})) ∉ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*}
 using *f2 f1* **by** *blast* **}**
 ultimately have (*mset-cl* (*pp* (*cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*})), *mset-cl* (*ppb* (*cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*}))) ∈ *mult* (*mult trm-ord*) ∨ ((*pp* (*cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*}), *ppb* (*cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*})) ∈ {(*pa*, *p*). (*mset-cl pa*, *mset-cl p*) ∈ *mult* (*mult trm-ord*)} ∨ (*pp* (*cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*}), *ppb* (*cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*})) ∈ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*}) ∨ (*pp* (*cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*}), *ppa* (*cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*})) ∉ {(*pa*, *p*). (*mset-cl pa*, *mset-cl p*) ∈ *mult* (*mult trm-ord*)} ∧ (*pp* (*cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*}), *ppa* (*cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*})) ∉ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*}
 by *fastforce* **}**
 then have (*mset-cl* (*pp* (*cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*})), *mset-cl* (*ppb* (*cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*}))) ∈ *mult* (*mult trm-ord*) ∨ ((*pp* (*cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*}), *ppb* (*cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*})) ∈ {(*pa*, *p*). (*mset-cl pa*, *mset-cl p*) ∈ *mult* (*mult trm-ord*)} ∨ (*pp* (*cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*}), *ppb* (*cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*})) ∈ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*}) ∨ (*pp* (*cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*}), *ppa* (*cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*})) ∉ {(*pa*, *p*). (*mset-cl pa*, *mset-cl p*) ∈ *mult* (*mult trm-ord*)} ∧ (*pp* (*cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*}), *ppa* (*cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*})) ∉ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*} ∨ (*pp* (*cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*}), *ppa* (*cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*})) ∉ *cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*} ∨ (*ppa* (*cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*}), *ppb* (*cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*})) ∉ *cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*} ∨ (*pp* (*cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*}), *ppb* (*cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*})) ∈ *cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*}
 using *cl-ord-def* **by** *auto* **}**
 then have (*pp* (*cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*}), *ppa* (*cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*})) ∉ *cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*} ∨ (*ppa* (*cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*}), *ppb* (*cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*})) ∉ *cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*} ∨ (*pp* (*cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*}), *ppb* (*cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*})) ∈ *cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*}
 using *cl-ord-def* **by** *force*
 then have *trans* (*cl-ord* ∪ {(*pa*, *p*). *mset-cl pa* = *mset-cl p*})
 using *f1* **by** *meson*
 from *this* **show** *?thesis* **unfolding** *cl-ord-eq-def* **by** *auto*
qed

lemma *wf-ecl-ord*:
 shows *wf ecl-ord*
proof −
 have *wf* (*mult trm-ord*) **using** *trm-ord-wf* **and** *wf-mult* **by** *auto*

then have *wf* (*mult* (*mult trm-ord*)) **using** *wf-mult* **by** *auto*
thus *?thesis*
using *ecl-ord-def*
and *measure-wf* [*of* (*mult* (*mult trm-ord*)) *ecl-ord mset-ecl*]
by *blast*
qed

definition *maximal-literal* :: $'a$ *literal* $\Rightarrow$ $'a$ *clause* $\Rightarrow$ *bool*
where
(*maximal-literal L C*) = ($\forall$ *x*. (*x* $\in$ *C* $\longrightarrow$ (*L*,*x*) $\notin$ *lit-ord*))

definition *eligible-literal*
where
(*eligible-literal L C* σ) = (*L* $\in$ *sel* (*cl-ecl C*) $\vee$
(*sel*(*cl-ecl C*) = {}
$\wedge$ (*maximal-literal* (*subst-lit L* σ) (*subst-cl* (*cl-ecl C*) σ))))

definition *strictly-maximal-literal*
where *strictly-maximal-literal C L* σ =
($\forall$ *x* $\in$ (*cl-ecl C*) $-$ { *L* }. ((*subst-lit x* σ),(*subst-lit L* σ))
$\in$ *lit-ord*)

definition *lower-or-eq*
where *lower-or-eq t s* = ((*t* = *s*) $\vee$ ((*t*,*s*) $\in$ *trm-ord*))

lemma *eligible-literal-coincide*:
assumes *coincide-on* σ σ' (*vars-of-cl* (*cl-ecl C*))
assumes *eligible-literal L C* σ
assumes *L* $\in$ (*cl-ecl C*)
shows *eligible-literal L C* σ'
proof $-$
from *assms*(*2*) **have**
L $\in$ *sel* (*cl-ecl C*) $\vee$ (*sel* (*cl-ecl C*) = {} $\wedge$ *maximal-literal* (*subst-lit L* σ)
(*subst-cl* (*cl-ecl C*) σ))
unfolding *eligible-literal-def* **by** *auto*
then show *?thesis*
proof
assume *L* $\in$ *sel* (*cl-ecl C*)
then show *?thesis* **unfolding** *eligible-literal-def* **by** *auto*
next
assume *sel* (*cl-ecl C*) = {} $\wedge$ *maximal-literal* (*subst-lit L* σ) (*subst-cl* (*cl-ecl C*) σ)
then have *sel* (*cl-ecl C*) = {} **and** *maximal-literal* (*subst-lit L* σ) (*subst-cl* (*cl-ecl C*) σ)
by *auto*
from *assms*(*1*) **have** (*subst-cl* (*cl-ecl C*) σ) = (*subst-cl* (*cl-ecl C*) σ')
using *coincide-on-cl* **by** *blast*
from *assms*(*3*) **and** *assms*(*1*) **have** *coincide-on* σ σ' (*vars-of-lit L*) **unfolding** *coincide-on-def*

by *auto*
from *this* **have** $(subst\text{-}lit\ L\ \sigma) = (subst\text{-}lit\ L\ \sigma')$
using *coincide-on-lit* **by** *auto*
from *this* **and** ‹$(subst\text{-}cl\ (cl\text{-}ecl\ C)\ \sigma) = (subst\text{-}cl\ (cl\text{-}ecl\ C)\ \sigma')$›
and ‹$maximal\text{-}literal\ (subst\text{-}lit\ L\ \sigma)\ \ (subst\text{-}cl\ (cl\text{-}ecl\ C)\ \sigma)$›
have $maximal\text{-}literal\ (subst\text{-}lit\ L\ \sigma')\ \ (subst\text{-}cl\ (cl\text{-}ecl\ C)\ \sigma')$
by *auto*
from *this* **and** ‹$sel\ (cl\text{-}ecl\ C) = \{\}$› **show** *?thesis* **unfolding** *eligible-literal-def*
by *auto*
qed
qed

The next definition extends the ordering to substitutions.

definition *lower-on*
where $lower\text{-}on\ \sigma\ \eta\ V = (\forall x \in V.$
$(lower\text{-}or\text{-}eq\ (subst\ (Var\ x)\ \sigma)\ (\ (subst\ (Var\ x)\ \eta))))$

We now establish some properties of the ordering relations.

lemma *lower-or-eq-monotonic*:
assumes *lower-or-eq t1 s1*
assumes *lower-or-eq t2 s2*
shows $lower\text{-}or\text{-}eq\ (Comb\ t1\ t2)\ (Comb\ s1\ s2)$
unfolding *lower-or-eq-def* **using** *trm-ord-reduction-left trm-ord-reduction-right*
by (*metis assms*(*1*) *assms*(*2*) *lower-or-eq-def trm-ord-trans transD*)

lemma *lower-on-term*:
shows $\bigwedge \sigma\ \eta.\ lower\text{-}on\ \sigma\ \eta\ (vars\text{-}of\ t) \Longrightarrow$
$(lower\text{-}or\text{-}eq\ (subst\ t\ \sigma)\ (subst\ t\ \eta))$
proof (*induction t*)
case (*Var x*)
from *this* **show** *?case*
unfolding *lower-on-def* **by** *auto*
next case (*Const x*)
show *?case*
unfolding *lower-or-eq-def* **by** *auto*
next case (*Comb t1 t2*)
show $\bigwedge \sigma\ \eta.\ lower\text{-}on\ \sigma\ \eta\ (vars\text{-}of\ (Comb\ t1\ t2)) \Longrightarrow$
$(lower\text{-}or\text{-}eq\ (subst\ \ (Comb\ t1\ t2)\ \sigma)\ (subst\ \ (Comb\ t1\ t2)\ \eta))$
proof −
fix $\sigma\ \eta$ **assume** $lower\text{-}on\ \sigma\ \eta\ (vars\text{-}of\ (Comb\ t1\ t2))$
from *this* **have** $lower\text{-}on\ \sigma\ \eta\ (vars\text{-}of\ t1)$ **and** $lower\text{-}on\ \sigma\ \eta\ (vars\text{-}of\ t2)$
unfolding *lower-on-def* **by** *auto*
from ‹$lower\text{-}on\ \sigma\ \eta\ (vars\text{-}of\ t1)$› **have** $lower\text{-}or\text{-}eq\ (subst\ t1\ \sigma)\ (subst\ t1\ \eta)$
using *Comb.IH* **by** *auto*
from ‹$lower\text{-}on\ \sigma\ \eta\ (vars\text{-}of\ t2)$› **have** $lower\text{-}or\text{-}eq\ (subst\ t2\ \sigma)\ (subst\ t2\ \eta)$
using *Comb.IH* **by** *auto*
from ‹$lower\text{-}or\text{-}eq\ (subst\ t1\ \sigma)\ (subst\ t1\ \eta)$› ‹$lower\text{-}or\text{-}eq\ (subst\ t2\ \sigma)\ (subst\ t2\ \eta)$›
show $lower\text{-}or\text{-}eq\ (subst\ \ (Comb\ t1\ t2)\ \sigma)\ (subst\ (Comb\ t1\ t2)\ \eta)$

        using *lower-or-eq-monotonic* **by** *auto*
    qed
qed

lemma *diff-substs-yield-diff-trms*:
  assumes (*subst* (*Var x*) σ) $\neq$ (*subst* (*Var x*) η)
  shows ($x \in$ *vars-of t*)
   $\Longrightarrow$ (*subst t* σ) $\neq$ (*subst t* η)
proof (*induction t*)
  case (*Var y*)
    show ($x \in$ *vars-of* (*Var y*)) $\Longrightarrow$ (*subst* (*Var y*) σ) $\neq$ (*subst* (*Var y*) η)
    proof −
      assume ($x \in$ *vars-of* (*Var y*))
      from ‹($x \in$ *vars-of* (*Var y*))› **have** $x = y$ **by** *auto*
      from *this* **and** *assms*(*1*)
      show (*subst* (*Var y*) σ) $\neq$ (*subst* (*Var y*) η)
      by *auto*
    qed
  next case (*Const y*)
    show ($x \in$ *vars-of* (*Const y*))
   $\Longrightarrow$ (*subst* (*Const y*) σ) $\neq$ (*subst* (*Const y*) η)
    proof (*rule ccontr*)
      from ‹($x \in$ *vars-of* (*Const y*))› **show** *False* **by** *auto*
    qed
  next case (*Comb t1 t2*)
    show ($x \in$ *vars-of* (*Comb t1 t2*))
   $\Longrightarrow$ (*subst* (*Comb t1 t2*) σ) $\neq$ (*subst* (*Comb t1 t2*) η)
    proof −
      assume ($x \in$ *vars-of* (*Comb t1 t2*))
      from ‹$x \in$ *vars-of* (*Comb t1 t2*)› **have** $x \in$ *vars-of t1* $\vee$ $x \in$ *vars-of t2* **by**
auto
      then show (*subst* (*Comb t1 t2*) σ) $\neq$ (*subst* (*Comb t1 t2*) η)
      proof
        assume $x \in$ *vars-of t1*
        from *this* **have** (*subst t1* σ) $\neq$ (*subst t1* η)
          using *Comb.IH* **by** *auto*
        then show *?thesis* **by** *auto*
      next
        assume $x \in$ *vars-of t2*
        from *this* **have** (*subst t2* σ) $\neq$ (*subst t2* η)
          using *Comb.IH* **by** *auto*
        then show *?thesis* **by** *auto*
      qed
    qed
qed

lemma *lower-subst-yields-lower-trms*:
  assumes *lower-on* σ η (*vars-of t*)
  assumes ((*subst* (*Var x*) σ),(*subst* (*Var x*) η)) $\in$ *trm-ord*

assumes ($x \in$ *vars-of* t)
shows ((*subst* t σ),(*subst* t η)) $\in$ *trm-ord*
proof –
from *assms*(1) **have** *lower-or-eq* (*subst* t σ) (*subst* t η)
using *lower-on-term* **by** *auto*
from *assms*(2) **have** (*subst* (*Var* x) σ) $\neq$ (*subst* (*Var* x) η)
using *trm-ord-irrefl irrefl-def* **by** *fastforce*
from *this* **and** *assms*(3) **have** (*subst* t σ) $\neq$ (*subst* t η)
using *diff-substs-yield-diff-trms* **by** *fastforce*
from *this* **and** ‹*lower-or-eq* (*subst* t σ) (*subst* t η)›
show *?thesis* **unfolding** *lower-or-eq-def* **by** *auto*
qed

lemma *lower-on-lit*:
assumes *lower-on* σ η (*vars-of-lit* L)
assumes ((*subst* (*Var* x) σ),(*subst* (*Var* x) η)) $\in$ *trm-ord*
assumes $x \in$ *vars-of-lit* L
shows ((*subst-lit* L σ), (*subst-lit* L η)) $\in$ *lit-ord*
proof –
obtain t s **where** *def-l*: L = *Pos* (*Eq* t s) | L = (*Neg* (*Eq* t s))
by (*metis mset-lit.cases*)
from *this* **have** *vars-of* $t \subseteq$ *vars-of-lit* L **and** *vars-of* $s \subseteq$ *vars-of-lit* L **by** *auto*
from ‹*vars-of* $s \subseteq$ *vars-of-lit* L› **and** *assms*(1) **have** *lower-on* σ η (*vars-of* s)
unfolding *lower-on-def* **by** *auto*
from *def-l* **have** *def-ms-l*: *mset-lit* L = {# t,s #} $\vee$ *mset-lit* L = {# t,t,s,s #}
by *auto*
from *this* **have** $t \in$# (*mset-lit* L) **and** $s \in$# (*mset-lit* L) **by** *auto*
from *def-l* **have** *mset-lit* (*subst-lit* L σ) = {# (*subst* u σ). $u \in$# (*mset-lit* L) #} **by** *auto*
from *def-l* **have** *mset-lit* (*subst-lit* L η) = {# (*subst* u η). $u \in$# (*mset-lit* L) #} **by** *auto*
from ‹*lower-on* σ η (*vars-of* s)› **have** *lower-or-eq* (*subst* s σ) (*subst* s η)
using *lower-on-term* **by** *auto*
let *?L* = *mset-lit* L
let *?M1* = *mset-lit* (*subst-lit* L σ)
let *?M2* = *mset-lit* (*subst-lit* L η)
from ‹*vars-of* $t \subseteq$ *vars-of-lit* L› **and** *assms*(1) **have** *lower-on* σ η (*vars-of* t)
unfolding *lower-on-def* **by** *auto*
from ‹*vars-of* $s \subseteq$ *vars-of-lit* L› **and** *assms*(1) **have** *lower-on* σ η (*vars-of* s)
unfolding *lower-on-def* **by** *auto*
have *all-lower*: $\forall$ u. ($u \in$# (*mset-lit* L) $\longrightarrow$ (((*subst* u σ), (*subst* u η)) $\in$ *trm-ord*

$\vee$ (*subst* u σ) = (*subst* u η)))
proof (*rule allI*,*rule impI*)
fix u **assume** $u \in$# (*mset-lit* L)
have $u = t \vee u = s$
proof (*cases*)
assume *mset-lit* L = {# t,s #}
from *this* **and** ‹$u \in$# (*mset-lit* L)› **show** *?thesis*

by (*simp add*: *count-single insert-DiffM2 insert-noteq-member not-gr0*)
next
assume ¬*mset-lit L* = {# *t*,*s* #}
from *this* and *def-ms-l* have *mset-lit L* = {# *t*,*t*,*s*,*s* #}
by *auto*
from *this* and ‹*u* ∈# (*mset-lit L*)› show *?thesis*
using *not-gr0* by *fastforce*
qed
then show (((*subst u σ*), (*subst u η*)) ∈ *trm-ord*
∨ (*subst u σ*) = (*subst u η*))
proof
assume *u* = *t*
from ‹*lower-on σ η* (*vars-of t*)› have *lower-or-eq* (*subst t σ*) (*subst t η*)
using *lower-on-term* by *auto*
from *this* show *?thesis* unfolding *lower-or-eq-def* using ‹*u* = *t*› by *auto*
next
assume *u* = *s*
from ‹*lower-on σ η* (*vars-of s*)› have *lower-or-eq* (*subst s σ*) (*subst s η*)
using *lower-on-term* by *auto*
from *this* show *?thesis* unfolding *lower-or-eq-def* using ‹*u* = *s*› by *auto*
qed
qed
have *sl-exists*: ∃ *u*. (*u* ∈# (*mset-lit L*) ∧ ((*subst u σ*), (*subst u η*)) ∈ *trm-ord*)
proof −
from ‹*x* ∈ *vars-of-lit L*› and *def-l* have
x ∈ *vars-of t* ∨ *x* ∈ *vars-of s* by *auto*
then show *?thesis*
proof
assume *x* ∈ *vars-of t*
from *this* and ‹*lower-on σ η* (*vars-of t*)› *assms*(*1*) *assms*(*2*)
have ((*subst t σ*),(*subst t η*)) ∈ *trm-ord*
using *lower-subst-yields-lower-trms* by *auto*
from *this* and ‹*t* ∈# (*mset-lit L*)› show *?thesis* by *auto*
next
assume *x* ∈ *vars-of s*
from *this* and ‹*lower-on σ η* (*vars-of s*)› *assms*(*1*) *assms*(*2*)
have ((*subst s σ*),(*subst s η*)) ∈ *trm-ord*
using *lower-subst-yields-lower-trms* by *auto*
from *this* and ‹*s* ∈# (*mset-lit L*)› show *?thesis* by *auto*
qed
qed
from *all-lower sl-exists* and
‹*mset-lit* (*subst-lit L σ*) = {# (*subst u σ*). *u* ∈# (*mset-lit L*) #}›
‹*mset-lit* (*subst-lit L η*) = {# (*subst u η*). *u* ∈# (*mset-lit L*) #}›
have (*?M1*,*?M2*) ∈ (*mult trm-ord*)
using *trm-ord-irrefl image-mset-ordering*
[*of ?M1 λx*. (*subst x σ*) *?L ?M2 λx*. (*subst x η*) *trm-ord*]
by *blast*
from *this* show *?thesis* unfolding *lit-ord-def* by *auto*

qed

lemma *lower-on-lit-eq*:
assumes *lower-on* σ η (*vars-of-lit L*)
shows ((*subst-lit L* σ) = (*subst-lit L* η)) $\vee$ ((*subst-lit L* σ), (*subst-lit L* η)) $\in$ *lit-ord*
proof (*cases*)
assume *coincide-on* σ η (*vars-of-lit L*)
then show *?thesis* **using** *coincide-on-lit* **by** *auto*
next
assume $\neg$*coincide-on* σ η (*vars-of-lit L*)
then obtain *x* **where** $x \in$ *vars-of-lit L*
and (*subst* (*Var x*) σ) $\neq$ (*subst* (*Var x*) η)
unfolding *coincide-on-def* **by** *auto*
from ‹$x \in$ *vars-of-lit L*› *assms*(*1*)
‹(*subst* (*Var x*) σ) $\neq$ (*subst* (*Var x*) η)›**and** *assms*(*1*)
have ((*subst* (*Var x*) σ),(*subst* (*Var x*) η)) $\in$ *trm-ord*
unfolding *lower-on-def lower-or-eq-def* **by** *auto*
from *this assms*(*1*) ‹$x \in$ *vars-of-lit L*› **show** *?thesis* **using** *lower-on-lit* **by** *auto*
qed

lemma *lower-on-cl*:
assumes *lower-on* σ η (*vars-of-cl* (*cl-ecl C*))
assumes ((*subst* (*Var x*) σ),(*subst* (*Var x*) η)) $\in$ *trm-ord*
assumes $x \in$ *vars-of-cl* (*cl-ecl C*)
assumes *finite* (*cl-ecl C*)
shows ((*C*,σ), (*C*, η)) $\in$ *ecl-ord*
proof −
let *?M1* = *mset-ecl* (*C*,σ)
let *?M2* = *mset-ecl* (*C*,η)
let *?M* = (*mset-set* (*cl-ecl C*))
let *?f1* = λx. (*mset-lit* (*subst-lit x* σ))
let *?f2* = λx. (*mset-lit* (*subst-lit x* η))
have *?M1* = {# (*?f1 u*). *u* ∈# *?M* #} **using** *mset-ecl.simps* **by** *blast*
have *?M2* = {# (*?f2 u*). *u* ∈# *?M* #} **using** *mset-ecl.simps* **by** *blast*
have *i*: $\forall$ *u*. (*u* ∈# *?M* $\longrightarrow$ (((*?f1 u*), (*?f2 u*)) $\in$ (*mult trm-ord*) $\vee$ (*?f1 u*) = (*?f2 u*)))
proof ((*rule allI*),(*rule impI*))
fix *u* **assume** *u* ∈# *?M*
from *this* **have** $u \in$ (*cl-ecl C*) **using** *count-mset-set*(*3*) **by** (*simp add*: *assms*(*4*))
from *this* **and** *assms*(*1*) **have** *lower-on* σ η (*vars-of-lit u*) **unfolding** *lower-on-def* **by** *auto*
then have ((*subst-lit u* σ) = (*subst-lit u* η))
$\vee$ ((*subst-lit u* σ), (*subst-lit u* η)) $\in$ *lit-ord*
using *lower-on-lit-eq* **by** *blast*
from *this* **show** (((*?f1 u*), (*?f2 u*)) $\in$ (*mult trm-ord*) $\vee$ (*?f1 u*) = (*?f2 u*))
unfolding *lit-ord-def* **by** *auto*
qed
have *irrefl* (*mult trm-ord*) **by** (*simp add*: *irreflI trm-ord-wf wf-mult*)

have *ii*: $\exists\, u.\ (u \in\# \ ?M \wedge ((?f1\ u), (?f2\ u)) \in (mult\ trm\text{-}ord))$
proof –
from ‹$x \in vars\text{-}of\text{-}cl\ (cl\text{-}ecl\ C)$› **obtain** *u* **where** $u \in (cl\text{-}ecl\ C)$ **and** $x \in vars\text{-}of\text{-}lit\ u$
by *auto*
from *assms*(*4*) ‹$u \in (cl\text{-}ecl\ C)$› **have** $u \in\# \ ?M$ **by** *auto*
from ‹$u \in (cl\text{-}ecl\ C)$› **and** *assms*(*1*) **have** *lower-on* $\sigma\ \eta$ (*vars-of-lit u*)
unfolding *lower-on-def* **by** *auto*
from ‹$x \in vars\text{-}of\text{-}lit\ u$› ‹*lower-on* $\sigma\ \eta$ (*vars-of-lit u*)› *assms*(*2*)
have $((subst\text{-}lit\ u\ \sigma), (subst\text{-}lit\ u\ \eta)) \in lit\text{-}ord$
using *lower-on-lit* **by** *blast*
from *this* ‹$u \in\# \ ?M$› **have** $(u \in\# \ ?M \wedge ((?f1\ u), (?f2\ u)) \in (mult\ trm\text{-}ord))$
unfolding *lit-ord-def* **by** *auto*
then show *?thesis* **by** *auto*
qed
from *i ii* ‹$?M1 = \{\#\ (?f1\ u).\ u \in\# \ ?M\ \#\}$› ‹$?M2 = \{\#\ (?f2\ u).\ u \in\# \ ?M\ \#\}$› ‹*irrefl* (*mult trm-ord*)›
have $(?M1,?M2) \in (mult\ (mult\ trm\text{-}ord))$
using *image-mset-ordering* [*of ?M1 ?f1 ?M ?M2 ?f2* (*mult trm-ord*)] **by** *auto*
then show *?thesis* **unfolding** *ecl-ord-def* **by** *auto*
qed

lemma *subterm-trm-ord* :
shows $\bigwedge t\ \ s.$
$subterm\ t\ p\ s \Longrightarrow p \neq []$
$\Longrightarrow (s,t) \in trm\text{-}ord$
proof (*induction p*)
case *Nil*
from ‹$Nil \neq []$› **show** *?case* **by** *auto*
next case (*Cons i q*)
from ‹*subterm t* (i # q) *s*› **obtain** *t1 t2* **where**
$t = (Comb\ t1\ t2)$ **using** *subterm.elims*(*2*) **by** *blast*
have $i = Left \vee i = Right$ **using** *indices.exhaust* **by** *blast*
then show $(s,t) \in trm\text{-}ord$
proof
assume $i = Left$
from *this* **and** ‹$t = Comb\ t1\ t2$› **and** ‹*subterm t* (i # q) *s*›
have *subterm t1 q s* **by** *auto*
show *?thesis*
proof (*cases*)
assume $q = Nil$
from *this* **and** ‹*subterm t1 q s*› **have** $t1 = s$ **by** *auto*
from *this* **and** ‹$t = Comb\ t1\ t2$› **show** *?case* **using** *trm-ord-subterm* **by** *auto*
next
assume $q \neq Nil$
from *this* **and** ‹*subterm t1 q s*› **have** $(s,t1) \in trm\text{-}ord$ **using** *Cons.IH* **by** *auto*
from ‹$t = Comb\ t1\ t2$› **have** $(t1,t) \in trm\text{-}ord$ **using** *trm-ord-subterm* **by**

auto
    from *this* **and** ‹(s,$t1$) ∈ *trm-ord*› **show** *?case*
      using *trm-ord-trans* **unfolding** *trans-def* **by** *metis*
   qed
  next
   assume i = *Right*
   from *this* **and** ‹t = *Comb t1 t2*› **and** ‹*subterm t* (i # q) s›
    have *subterm t2 q s* **by** *auto*
   show *?thesis*
   proof (*cases*)
    assume q = *Nil*
    from *this* **and** ‹*subterm t2 q s*› **have** $t2 = s$ **by** *auto*
    from *this* **and** ‹t = *Comb t1 t2*› **show** *?case* **using** *trm-ord-subterm* **by**
auto
   next
    assume $q \neq$ *Nil*
    from *this* **and** ‹*subterm t2 q s*› **have** (s,$t2$) ∈ *trm-ord* **using** *Cons.IH* **by**
auto
    from ‹t = *Comb t1 t2*› **have** ($t2$,t) ∈ *trm-ord* **using** *trm-ord-subterm* **by**
auto
    from *this* **and** ‹(s,$t2$) ∈ *trm-ord*› **show** *?case*
      using *trm-ord-trans* **unfolding** *trans-def* **by** *metis*
   qed
 qed
qed

lemma *subterm-trm-ord-eq* :
 assumes *subterm t p s*
 shows $s = t \vee (s,t) \in$ *trm-ord*
proof (*cases*)
 assume p = *Nil*
  from *this* **and** *assms*(*1*) **show** *?thesis* **by** *auto*
 next assume $p \neq$ *Nil*
  from *this* **and** *assms*(*1*) **show** *?thesis* **using** *subterm-trm-ord* **by** *auto*
qed

lemma *subterms-of-trm-ord-eq* :
 assumes $s \in$ *subterms-of t*
 shows $s = t \vee (s,t) \in$ *trm-ord*
proof −
 from *assms*(*1*) **obtain** p **where** *subterm t p s* **using** *occurs-in-def* **by** *auto*
 from *this* **show** *?thesis* **using** *subterm-trm-ord-eq* **by** *auto*
qed

lemma *subt-trm-ord*:
 shows $t \prec s \longrightarrow (t,s) \in$ *trm-ord*
proof (*induction s*)
  case (*Var x*)
  show *?case*

proof
assume $t \prec Var\ x$
then show $(t, Var\ x) \in trm\text{-}ord$ **by** *auto*
qed
case $(Const\ x)$
show *?case*
proof
assume $t \prec Const\ x$
then show $(t,Const\ x) \in trm\text{-}ord$ **by** *auto*
qed
case $(Comb\ t1\ t2)$
show *?case*
proof
assume $t \prec Comb\ t1\ t2$
show $(t,\ Comb\ t1\ t2) \in trm\text{-}ord$
proof (*rule ccontr*)
assume $(t,\ Comb\ t1\ t2) \notin trm\text{-}ord$
then have *i*: $t \neq t1$ **using** *trm-ord-subterm* **by** *auto*
from ‹$(t,\ Comb\ t1\ t2) \notin trm\text{-}ord$› **have** *ii*: $t \neq t2$ **using** *trm-ord-subterm* **by** *auto*
from *i ii* **and** ‹$t \prec Comb\ t1\ t2$› **have** $t \prec t1 \lor t \prec t2$ **by** *auto*
from *this* **and** ‹$(t,\ Comb\ t1\ t2) \notin trm\text{-}ord$›
show *False* **using** *Comb.IH trm-ord-subterm trm-ord-trans trans-def* **by** *metis*
qed
qed
qed

lemma *trm-ord-vars*:
assumes $(t,s) \in trm\text{-}ord$
shows $vars\text{-}of\ t \subseteq vars\text{-}of\ s$
proof (*rule ccontr*)
assume $\neg vars\text{-}of\ t \subseteq vars\text{-}of\ s$
then obtain x **where** $x \in vars\text{-}of\ t$ **and** $x \notin vars\text{-}of\ s$ **by** *auto*
let $?\sigma = [(x,s)]$
from *assms* **have** $((subst\ t\ ?\sigma),(subst\ s\ ?\sigma)) \in trm\text{-}ord$
using *trm-ord-subst* **by** *auto*
let $?\vartheta = []$
let $?V = vars\text{-}of\ s$
have $subst\ s\ ?\vartheta = s$ **by** *simp*
have $subst\ (Var\ x)\ ?\sigma = s$ **by** *simp*
have $coincide\text{-}on\ ?\sigma\ ?\vartheta\ ?V$
proof (*rule ccontr*)
assume $\neg\ coincide\text{-}on\ ?\sigma\ ?\vartheta\ ?V$
then obtain y **where** $y \in ?V\ subst\ (Var\ y)\ ?\sigma \neq subst\ (Var\ y)\ ?\vartheta$
unfolding *coincide-on-def* **by** *auto*
from ‹$subst\ (Var\ y)\ ?\sigma \neq subst\ (Var\ y)\ ?\vartheta$› **have** $y = x$
by (*metis assoc.simps*(*2*) *subst.simps*(*1*))
from *this* **and** ‹$x \notin vars\text{-}of\ s$› ‹$y \in ?V$› **show** *False* **by** *auto*

  qed
  from *this* **and** ‹*subst s ?ϑ = s*› **have** *subst s ?σ = s*
    using *coincide-on-term* **by** *metis*
  from ‹*x ∈ vars-of t*› **have** (*Var x*) ≺ *t*
    using ‹(*subst t* [(*x*, *s*)], *subst s* [(*x*, *s*)]) ∈ *trm-ord*›
      ‹*subst s* [(*x*, *s*)] = *s*› *trm-ord-wf vars-iff-occseq* **by** *fastforce*
  from *this* **have** ((*Var x*), *t*) ∈ *trm-ord* **using** *subt-trm-ord* **by** *auto*
  from *this* **and** *assms*(*1*) **have** (*Var x*,*s*) ∈*trm-ord* **using** *trm-ord-trans trans-def*
by *metis*
  from *this* **have** ((*subst* (*Var x*) *?σ*),(*subst s ?σ*)) ∈ *trm-ord*
    using *trm-ord-subst* **by** *metis*
  from *this* **and** ‹*subst s ?σ = s*› ‹*subst* (*Var x*) *?σ = s*›
    have (*s*,*s*) ∈ *trm-ord* **by** *auto*
  from *this* **show** *False* **using** *trm-ord-irrefl irrefl-def* **by** *metis*
qed

lemma *lower-on-ground*:
  assumes *lower-on σ η V*
  assumes *ground-on V η*
  shows *ground-on V σ*
proof (*rule ccontr*)
  assume ¬ *ground-on V σ*
  from *this* **obtain** *x* **where** *x ∈ V* **and** *vars-of* (*subst* (*Var x*) *σ*) ≠ {}
    unfolding *ground-on-def ground-term-def* **by** *metis*
  from *assms*(*1*) ‹*x ∈ V*› **have** (*subst* (*Var x*) *σ*) = (*subst* (*Var x*) *η*)
    ∨ ((*subst* (*Var x*) *σ*),(*subst* (*Var x*) *η*)) ∈ *trm-ord*
    unfolding *lower-on-def lower-or-eq-def* **by** *metis*
  from *this* **have** *vars-of* (*subst* (*Var x*) *σ*) ⊆ *vars-of* (*subst* (*Var x*) *η*)
    using *trm-ord-vars* **by** *auto*
  from *this* **and** ‹*vars-of* (*subst* (*Var x*) *σ*) ≠ {}›
    have *vars-of* (*subst* (*Var x*) *η*) ≠ {} **by** *auto*
  from *this* **and** ‹*x ∈ V*› **and** *assms*(*2*) **show** *False*
    unfolding *ground-on-def ground-term-def* **by** *metis*
qed

lemma *replacement-monotonic* :
  shows ⋀ *t s*. ((*subst v σ*), (*subst u σ*)) ∈ *trm-ord*
        ⟹ *subterm t p u* ⟹ *replace-subterm t p v s*
        ⟹ ((*subst s σ*), (*subst t σ*)) ∈ *trm-ord*
proof (*induction p*)
  case *Nil*
    from ‹*subterm t Nil u*› **have** *t = u* **by** *auto*
    from ‹*replace-subterm t Nil v s*› **have** *s = v* **by** *auto*
    from ‹*t = u*› **and** ‹*s = v*› **and** ‹((*subst v σ*), (*subst u σ*)) ∈ *trm-ord*›
      show *?case* **by** *auto*
  next case (*Cons i q*)
    from ‹*subterm t* (*i # q*) *u*› **obtain** *t1 t2* **where**
        t = (*Comb t1 t2*) **using** *subterm.elims*(*2*) **by** *blast*
    have *i = Left* ∨ *i = Right* **using** *indices.exhaust* **by** *blast*

```
    then show ((subst s σ), (subst t σ)) ∈ trm-ord
    proof
      assume i = Left
      from this and ‹t = Comb t1 t2› and ‹subterm t (i # q) u›
        have subterm t1 q u by auto
      from ‹i = Left› and ‹t = Comb t1 t2› and ‹replace-subterm t (i # q) v s›
        obtain t1' where replace-subterm t1 q v t1' and s = Comb t1' t2 by auto
      from ‹((subst v σ), (subst u σ)) ∈ trm-ord›
        and ‹subterm t1 q u› and ‹replace-subterm t1 q v t1'› have
        ((subst t1' σ), (subst t1 σ)) ∈ trm-ord
        using Cons.IH Cons.prems(1) by blast
      from this and ‹t = (Comb t1 t2)› and ‹s = (Comb t1' t2)›
        show ((subst s σ), (subst t σ)) ∈ trm-ord
        by (simp add: trm-ord-reduction-left)
    next
      assume i = Right
      from this and ‹t = Comb t1 t2› and ‹subterm t (i # q) u›
        have subterm t2 q u by auto
      from ‹i = Right› and ‹t = Comb t1 t2› and ‹replace-subterm t (i # q) v s›

        obtain t2' where replace-subterm t2 q v t2' and s = Comb t1 t2' by auto
      from ‹((subst v σ), (subst u σ)) ∈ trm-ord›
        and ‹subterm t2 q u› and ‹replace-subterm t2 q v t2'› have
        ((subst t2' σ), (subst t2 σ)) ∈ trm-ord
        using Cons.IH Cons.prems(2) by blast
      from this and ‹t = (Comb t1 t2)› and ‹s = (Comb t1 t2')›
        show ((subst s σ), (subst t σ)) ∈ trm-ord
        by (simp add: trm-ord-reduction-right)
    qed
qed

lemma mset-lit-subst:
  shows (mset-lit (subst-lit L σ)) =
    {# (subst x σ). x ∈# (mset-lit L) #}
proof −
  have positive-literal L ∨ negative-literal L
    using negative-literal.simps(2) positive-literal.elims(3) by blast
  then show ?thesis
  proof
    assume positive-literal L
    then obtain t s where L = Pos (Eq t s)
      by (metis equation.exhaust positive-literal.elims(2))
    from this show ?thesis by auto
  next
    assume negative-literal L
    then obtain t s where L = Neg (Eq t s)
      by (metis equation.exhaust negative-literal.elims(2))
    from this show ?thesis by auto
  qed
```

qed

lemma *lit-ord-irrefl*:
  shows $(L,L) \notin$ *lit-ord*
by (*simp add*: *lit-ord-wf*)

lemma *lit-ord-subst*:
  assumes $(L,M) \in$ *lit-ord*
  shows ((*subst-lit* L σ), (*subst-lit* M σ)) $\in$ *lit-ord*
proof –
  let *?f* = λx. (*subst* x σ)
  have *i*: $\bigwedge$ *t s*. $((t,s) \in$ *trm-ord* $\Longrightarrow$ ((*?f t*), (*?f s*)) $\in$ *trm-ord*)
    using *trm-ord-subst* **by** *auto*
  from *assms*(*1*) **have** *ii*: ((*mset-lit L*),(*mset-lit M*)) $\in$ (*mult trm-ord*)
    unfolding *lit-ord-def* **by** *auto*
  let *?L* = {# (*?f x*). x $\in$# (*mset-lit L*) #}
  let *?M* = {# (*?f x*). x $\in$# (*mset-lit M*) #}
  from *i* **and** *ii* **have** *iii*: (*?L*,*?M*) $\in$ (*mult trm-ord*) **using** *monotonic-fun-mult*
by *metis*
  have *l*: *?L* = (*mset-lit* (*subst-lit L* σ))
    using *mset-lit-subst* **by** *auto*
  have *m*: *?M* = (*mset-lit* (*subst-lit M* σ))
    using *mset-lit-subst* **by** *auto*
  from *l m iii* **show** *?thesis* **unfolding** *lit-ord-def* **by** *auto*
qed

lemma *args-are-strictly-lower*:
   assumes *is-compound t*
   shows (*lhs t*,*t*) $\in$ *trm-ord* $\wedge$ (*rhs t*,*t*) $\in$ *trm-ord*
by (*metis assms is-compound.elims*(*2*) *lhs.simps*(*1*) *rhs.simps*(*1*) *trm-ord-subterm*)

lemma *mset-subst*:
  assumes $C' =$ *subst-cl* D ϑ
  assumes $\sigma \doteq \vartheta \lozenge \eta$
  assumes *finite D*
  shows *mset-cl* (C',η) = *mset-cl* (D,σ) $\vee$ (*mset-cl* (C',η),*mset-cl* (D,σ)) $\in$ (*mult*
(*mult trm-ord*))
proof –
  let *?f* = λx. (*subst-lit* x ϑ)
  let *?g* = λx. (*mset-lit* (*subst-lit* x η))
  let *?h* = λx. (*mset-lit* (*subst-lit* x σ))
  have *i*: $\forall x \in D$. ((*?g* (*?f x*)) = (*?h x*))
  proof
    fix *x*
    have (*subst-lit* (*subst-lit* x ϑ) η) = (*subst-lit* x ($\vartheta \lozenge \eta$))
      using *composition-of-substs-lit* **by** *auto*
    from *assms*(*2*) **have** (*subst-lit* x σ) = (*subst-lit* x ($\vartheta \lozenge \eta$))
      using *subst-eq-lit* **by** *auto*
    from *this* ‹(*subst-lit* (*subst-lit* x ϑ) η) = (*subst-lit* x ($\vartheta \lozenge \eta$))›

    show (*?g* (*?f x*)) = (*?h x*) **by** *auto*
  qed
  from *assms*(*3*) **have** *mset-set* (*?f* ‘ *D*) ⊆# {# (*?f x*). *x* ∈# *mset-set* (*D*) #}
    using *mset-set-mset-image* **by** *auto*
  from *this* **have** *ii*: {# (*?g x*). *x* ∈# *mset-set* (*?f* ‘ *D*) #} ⊆# {# (*?g x*). *x* ∈# {# (*?f x*). *x* ∈# *mset-set* (*D*) #} #}
    using *image-mset-subseteq-mono* **by** *auto*
  have {# (*?g x*). *x* ∈# {# (*?f x*). *x* ∈# *mset-set* (*D*) #} #} = {# (*?g* (*?f x*)). *x* ∈# *mset-set D* #}
    using *mset-image-comp* [*of ?g ?f*] **by** *auto*
  from *this* **and** *ii* **have**
    iii: {# (*?g x*). *x* ∈# *mset-set* (*?f* ‘ *D*) #} ⊆# {# (*?g* (*?f x*)). *x* ∈# *mset-set D* #} **by** *auto*
  from *i* **have** {# (*?g* (*?f x*)). *x* ∈# (*mset-set D*) #} = {# (*?h x*). *x* ∈# (*mset-set D*) #}
    using *equal-image-mset* [*of D* λ*x*. (*?g* (*?f x*))] **by** *auto*
  from *this* **and** *iii*
    have {# (*?g x*). *x* ∈# *mset-set* (*?f* ‘ *D*) #} ⊆# {# (*?h x*). *x* ∈# *mset-set D* #} **by** *auto*
  from *this*
    have *iv*: {# (*?g x*). *x* ∈# *mset-set* (*?f* ‘ *D*) #} ⊆# *mset-cl* (*D*,σ) **by** *auto*
  from *assms*(*1*) **have** ((λ*x*. *subst-lit x* ϑ) ‘ *D*) = *C′* **by** *auto*
  from *this* **and** *iv* **have** {#*mset-lit* (*subst-lit x* η). *x* ∈# *mset-set C′* #} ⊆# *mset-cl* (*D*, σ)
    by *auto*
  from *this* **have** *mset-cl* (*C′*, η) ⊆# *mset-cl* (*D*, σ) **by** *auto*
  from *this* **show** *?thesis* **using** *multiset-order-inclusion-eq mult-trm-ord-trans* **by** *auto*
qed

lemma *max-lit-exists*:
  shows *finite C* ⟹ *C* ≠ {} ⟶ *ground-clause C* ⟶ (∃ *L*. (*L* ∈ *C* ∧ (*maximal-literal L C*)))
proof (*induction rule*: *finite.induct*)
  case *emptyI*
  show *?case* **by** *simp*
next
  fix *A* :: *′a clause* **and** *a*:: *′a literal*
  assume *finite A*
  assume *hyp-ind*: *A* ≠ {} ⟶ *ground-clause A* ⟶ (∃ *L*. (*L* ∈ *A* ∧ (*maximal-literal L A*)))
  show (*insert a A*) ≠ {} ⟶ *ground-clause* (*insert a A*)
        ⟶ (∃ *L*. (*L* ∈ (*insert a A*) ∧ (*maximal-literal L* (*insert a A*))))
  proof ((*rule impI*)+)
    assume *insert a A* ≠ {}
    assume *ground-clause* (*insert a A*)
    show (∃ *L*. (*L* ∈ (*insert a A*) ∧ (*maximal-literal L* (*insert a A*))))
    proof (*cases*)
      assume *A* = {}

then have $a \in (insert\ a\ A) \wedge (maximal\text{-}literal\ a\ (insert\ a\ A))$
unfolding *maximal-literal-def* **using** *lit-ord-irrefl* **by** *auto*
then show *?thesis* **by** *auto*
next assume $A \neq \{\}$
have $insert\ a\ A = \{a\} \cup A$ **by** *auto*
then have $vars\text{-}of\text{-}cl\ (insert\ a\ A) = vars\text{-}of\text{-}cl\ A \cup vars\text{-}of\text{-}lit\ a$ **by** *auto*
from *this* **and** ‹$ground\text{-}clause\ (insert\ a\ A)$› **have**
$vars\text{-}of\text{-}lit\ a = \{\}$ **and** *ground-clause A* **by** *auto*
from ‹*ground-clause A*› **and** ‹$A \neq \{\}$› **and** *hyp-ind* **obtain** b **where**
$b \in A$ **and** *maximal-literal b A* **by** *auto*
show *?thesis*
proof (*cases*)
assume *maximal-literal a A*
then have *maximal-literal a* (*insert a A*)
using *lit-ord-wf maximal-literal-def* **by** *auto*
then show *?thesis* **by** *auto*
next
assume $\neg$*maximal-literal a A*
then obtain a' **where** $a' \in A$ **and** $(a,a') \in lit\text{-}ord$
unfolding *maximal-literal-def* **by** *auto*
from ‹$a' \in A$› **and** ‹*maximal-literal b A*› **have** $(b,a') \notin lit\text{-}ord$
unfolding *maximal-literal-def* **by** *auto*
from *this* **and** ‹$(a,a') \in lit\text{-}ord$›
have $(b,a) \notin lit\text{-}ord$ **unfolding** *lit-ord-def*
using *mult-def trancl-trans* **by** *fastforce*

from *this* **and** ‹*maximal-literal b A*› **have** *maximal-literal b* (*insert a A*)
unfolding *maximal-literal-def* **by** *simp*
from *this* **and** ‹$b \in A$› **show** *?thesis* **by** *auto*
qed
qed
qed
qed

We deduce that a clause contains at least one eligible literal.

lemma *eligible-lit-exists*:
assumes *finite* (*cl-ecl C*)
assumes $(cl\text{-}ecl\ C) \neq \{\}$
assumes (*ground-clause* (*subst-cl* (*cl-ecl C*) σ))
shows $\exists L.\ ((eligible\text{-}literal\ L\ C\ \sigma) \wedge (L \in (cl\text{-}ecl\ C)))$
proof (*cases*)
assume $sel\ (cl\text{-}ecl\ C) = \{\}$
let $?C = (subst\text{-}cl\ (cl\text{-}ecl\ C)\ \sigma)$
have *finite ?C* **by** (*simp add*: *assms*(*1*))
have $?C \neq \{\}$
proof $-$
from *assms*(*2*) **obtain** L **where** $L \in (cl\text{-}ecl\ C)$ **by** *auto*
from *this* **have** $(subst\text{-}lit\ L\ \sigma) \in ?C$ **by** *auto*
from *this* **show** $?C \neq \{\}$ **by** *auto*

qed
from ‹*finite ?C*› ‹*?C* ≠ {}› *assms*(*3*) **obtain** *L* **where** *L* ∈ *?C maximal-literal L ?C* **using** *max-lit-exists* **by** *metis*
from ‹*L* ∈ *?C*› **obtain** *L′* **where** *L′* ∈ (*cl-ecl C*) **and** *L* = (*subst-lit L′ σ*) **by** *auto*
from ‹*L′* ∈ (*cl-ecl C*)› ‹*L* = (*subst-lit L′ σ*)› ‹*maximal-literal L ?C*› ‹*sel* (*cl-ecl C*) = {}›
show *?thesis* **unfolding** *eligible-literal-def* **by** *metis*
next
assume *sel* (*cl-ecl C*) ≠ {}
then obtain *L* **where** *L* ∈ *sel* (*cl-ecl C*) **by** *auto*
from *this* **show** *?thesis* **unfolding** *eligible-literal-def* **using** *sel-neg* **by** *blast*
qed

The following lemmata provide various ways of proving that literals are ordered, depending on the relations between the terms they contain.

lemma *lit-ord-dominating-term*:
assumes (*s1*,*s2*) ∈ *trm-ord* ∨ (*s1*,*t2*) ∈ *trm-ord*
assumes *orient-lit x1 s1 t1 p1*
assumes *orient-lit x2 s2 t2 p2*
assumes *vars-of-lit x1* = {}
assumes *vars-of-lit x2* = {}
shows (*x1*,*x2*) ∈ *lit-ord*
proof −
from ‹*vars-of-lit x1* = {}› **and** ‹*orient-lit x1 s1 t1 p1*› **have** *vars-of t1* = {} **and** *vars-of s1* = {}
and ¬(*s1*,*t1*) ∈ *trm-ord* **unfolding** *orient-lit-def* **by** *auto*
from *assms*(*5*) **and** ‹*orient-lit x2 s2 t2 p2*› **have** *vars-of t2* = {} **and** *vars-of s2* = {}
and ¬(*s2*,*t2*) ∈ *trm-ord* **unfolding** *orient-lit-def* **by** *auto*
from ‹*vars-of t1* = {}› **and** ‹*vars-of s1* = {}› **and** ‹¬(*s1*,*t1*) ∈ *trm-ord*›
have *o1*: *t1* = *s1* ∨ (*t1*,*s1*) ∈ *trm-ord* **using** *trm-ord-ground-total*
unfolding *ground-term-def* **by** *auto*
from ‹*vars-of t2* = {}› **and** ‹*vars-of s2* = {}› **and** ‹¬(*s2*,*t2*) ∈ *trm-ord*›
have *o2*: *t2* = *s2* ∨ (*t2*,*s2*) ∈ *trm-ord* **using** *trm-ord-ground-total*
unfolding *ground-term-def* **by** *auto*
from ‹¬(*s2*,*t2*) ∈ *trm-ord*› **and** *assms*(*1*) **have** (*s1*,*s2*) ∈ *trm-ord*
by (*metis assms*(*1*) *o2 trm-ord-trans transE*)
let *?m1* = *mset-lit x1*
let *?m2* = *mset-lit x2*
from *assms*(*1*) **and** *o1* **and** *o2* **have** (*t1*,*s2*) ∈ *trm-ord* **using** *trm-ord-trans trans-def* **by** *metis*
from *this* **and** ‹(*s1*,*s2*) ∈ *trm-ord*› **have**
s2max: ∀ *x*. (*x* ∈# {# *t1*,*t1*,*s1*,*s1* #} ⟶ (*x*,*s2*) ∈ *trm-ord*)
by *auto*
have {# *s2* #} ⊂# {# *t2*,*t2*,*s2*,*s2* #} **by** *simp*
from ‹{# *s2* #} ⊂# {# *t2*,*t2*,*s2*,*s2* #}›
have ({# *s2* #}, {# *t2*,*t2*,*s2*,*s2* #}) ∈ *mult trm-ord*
using *trm-ord-trans multiset-order-inclusion* [*of* {# *s2* #} {# *t2*,*t2*,*s2*,*s2* #}

trm-ord] **by** *auto*
  have *p1* = *neg* ∨ *p1* = *pos* **using** *sign.exhaust* **by** *auto*
  then show *?thesis*
  proof
    assume *p1* = *neg*
    from *this* **and** ‹*orient-lit x1 s1 t1 p1*› **have** *x1* = (*Neg* (*Eq t1 s1*)) ∨ *x1* = (*Neg* (*Eq s1 t1*))
      using *orient-lit-def* **by** *blast*
    from *this* **have** *m1*: *?m1* = {# *t1*,*t1*,*s1*,*s1* #} **using** *mset-lit.simps* **by** *auto*

    have *p2* = *neg* ∨ *p2* = *pos* **using** *sign.exhaust* **by** *auto*
    then show *?thesis*
    proof
      assume *p2* = *neg*
      from *this* **and** ‹*orient-lit x2 s2 t2 p2*› **have** *x2* = (*Neg* (*Eq t2 s2*)) ∨ *x2* = (*Neg* (*Eq s2 t2*))
        using *orient-lit-def* **by** *blast*
      from *this* **have** *m2*: *?m2* = {# *t2*,*t2*,*s2*,*s2* #} **using** *mset-lit.simps* **by** *auto*
      from *s2max* **have** ({# *t1*,*t1*,*s1*,*s1* #}, {# *s2* #}) ∈ *mult trm-ord*
          using *mult1-def-lemma* [*of* {# *s2* #} {#} *s2* {# *t1*,*t1*,*s1*,*s1* #} {# *t1*,*t1*,*s1*,*s1* #} *trm-ord*]
        mult-def
        by *auto*

      from ‹({# *s2* #}, {# *t2*,*t2*,*s2*,*s2* #}) ∈ *mult trm-ord*› **and** ‹({# *t1*,*t1*,*s1*,*s1* #}, {# *s2* #}) ∈ *mult trm-ord*›
        have ({# *t1*,*t1*,*s1*,*s1* #}, {# *t2*,*t2*,*s2*,*s2* #}) ∈ *mult trm-ord*
        using *mult-trm-ord-trans* **unfolding** *trans-def* **by** *blast*

      from *this* **and** *m1* **and** *m2* **show** *?thesis*
        using *lit-ord-def* **by** *auto*
      next assume *p2* = *pos*
      from *this* **and** ‹*orient-lit x2 s2 t2 p2*› **have** *x2* = (*Pos* (*Eq t2 s2*)) ∨ *x2* = (*Pos* (*Eq s2 t2*))
        using *orient-lit-def* **by** *blast*
      from *this* **have** *m2*: *?m2* = {# *t2*,*s2* #} **using** *mset-lit.simps* **by** *auto*
      from *s2max* **have** ({# *t1*,*t1*,*s1*,*s1* #}, {# *s2* #}) ∈ *mult trm-ord*
          using *mult1-def-lemma* [*of* {# *s2* #} {#} *s2* {# *t1*,*t1*,*s1*,*s1* #} {# *t1*,*t1*,*s1*,*s1* #} *trm-ord*]
        mult-def
        by *auto*
      from *this* **and** ‹({# *s2* #}, {# *t2*,*t2*,*s2*,*s2* #}) ∈ *mult trm-ord*›
        have ({# *t1*,*t1*,*s1*,*s1* #}, {# *t2*,*s2* #}) ∈ *mult trm-ord*
       using *mset-ordering-add1* [*of* {# *t1*,*t1*,*s1*,*s1* #} {# *s2* #} *trm-ord t2*] **by** (*auto*)
      from *this* **and** *m1* **and** *m2* **show** *?thesis*
        using *lit-ord-def* **by** *auto*
   qed
   next

assume *p1* = *pos*
from *this* **and** ‹*orient-lit x1 s1 t1 p1*› **have** *x1* = (*Pos* (*Eq t1 s1*)) ∨ *x1* = (*Pos* (*Eq s1 t1*))
using *orient-lit-def* **by** *blast*
from *this* **have** *m1*: *?m1* = {# *t1*,*s1* #} **using** *mset-lit.simps* **by** *auto*
have *p2* = *neg* ∨ *p2* = *pos* **using** *sign.exhaust* **by** *auto*
then show *?thesis*
proof
assume *p2* = *neg*
from *this* **and** ‹*orient-lit x2 s2 t2 p2*› **have** *x2* = (*Neg* (*Eq t2 s2*)) ∨ *x2* = (*Neg* (*Eq s2 t2*))
using *orient-lit-def* **by** *blast*
from *this* **have** *m2*: *?m2* = {# *t2*,*t2*,*s2*,*s2* #} **using** *mset-lit.simps* **by** *auto*
from *s2max* **have** ({# *t1*,*s1* #}, {# *s2* #}) ∈ *mult trm-ord*
using *mult1-def-lemma* [*of* {# *s2* #} {#} *s2* {# *t1*,*s1* #} {# *t1*,*s1* #} *trm-ord*]
mult-def
by *auto*
from ‹({# *s2* #}, {# *t2*,*t2*,*s2*,*s2* #}) ∈ *mult trm-ord*› **and** ‹({# *t1*,*s1* #}, {# *s2* #}) ∈ *mult trm-ord*›
have ({# *t1*,*s1* #}, {# *t2*,*t2*,*s2*,*s2* #}) ∈ *mult trm-ord*
using *mult-trm-ord-trans* **unfolding** *trans-def* **by** *blast*

from *this* **and** *m1* **and** *m2* **show** *?thesis*
using *lit-ord-def* **by** *auto*
next assume *p2* = *pos*
from *this* **and** ‹*orient-lit x2 s2 t2 p2*› **have** *x2* = (*Pos* (*Eq t2 s2*)) ∨ *x2* = (*Pos* (*Eq s2 t2*))
using *orient-lit-def* **by** *blast*
from *this* **have** *m2*: *?m2* = {# *t2*,*s2* #} **using** *mset-lit.simps* **by** *auto*
from *s2max* **have** ({# *t1*,*s1* #}, {# *s2* #}) ∈ *mult trm-ord*
using *mult1-def-lemma* [*of* {# *s2* #} {#} *s2* {# *t1*,*s1* #} {# *t1*,*s1* #} *trm-ord*]
mult-def
by *auto*
from *this* **have** ({# *t1*,*s1* #}, {# *t2*,*s2* #}) ∈ *mult trm-ord*
using *mset-ordering-add1* [*of* {# *t1*,*s1* #} {# *s2* #} *trm-ord t2*] **by** *auto*
from *this* **and** *m1* **and** *m2* **show** *?thesis*
using *lit-ord-def* **by** *auto*
qed
qed
qed

lemma *lit-ord-neg-lit-lhs*:
assumes *orient-lit x1 s t1 pos*
assumes *orient-lit x2 s t2 neg*
assumes *vars-of-lit x1* = {}
assumes *vars-of-lit x2* = {}
shows (*x1*,*x2*) ∈ *lit-ord*

proof −
  from *assms*(*3*) **and** *assms*(*1*) **have** *vars-of t1* = {} **and** *vars-of s* = {}
    and ¬(*s*,*t1*) ∈ *trm-ord* **unfolding** *orient-lit-def* **by** *auto*
  from *assms*(*4*) **and** *assms*(*2*) **have** *vars-of t2* = {}
    and ¬(*s*,*t2*) ∈ *trm-ord* **unfolding** *orient-lit-def* **by** *auto*
  from ‹*vars-of t1* = {}› **and** ‹*vars-of s* = {}› **and** ‹¬(*s*,*t1*) ∈ *trm-ord*›
    have *o1*: *t1* = *s* ∨ (*t1*,*s*) ∈ *trm-ord* **using** *trm-ord-ground-total*
    unfolding *ground-term-def* **by** *auto*
  from ‹*vars-of t2* = {}› **and** ‹*vars-of s* = {}› **and** ‹¬(*s*,*t2*) ∈ *trm-ord*›
    have *o2*: *t2* = *s* ∨ (*t2*,*s*) ∈ *trm-ord* **using** *trm-ord-ground-total*
    unfolding *ground-term-def* **by** *auto*
  let *?m1* = *mset-lit x1*
  let *?m2* = *mset-lit x2*
  from ‹*orient-lit x1 s t1 pos*› **have** *x1* = (*Pos* (*Eq t1 s*)) ∨ *x1* = (*Pos* (*Eq s t1*))
      using *orient-lit-def* **by** *blast*
  from *this* **have** *m1*: *?m1* = {# *t1*,*s* #} **using** *mset-lit.simps* **by** *auto*
  from ‹*orient-lit x2 s t2 neg*› **have** *x2* = (*Neg* (*Eq t2 s*)) ∨ *x2* = (*Neg* (*Eq s t2*))
       using *orient-lit-def* **by** *blast*
  from *this* **have** *m2*: *?m2* = {# *t2*,*t2*,*s*,*s* #} **using** *mset-lit.simps* **by** *auto*
  show *?thesis*
  proof (*cases*)
    assume *t1* = *s*
    have ({# *s*,*s* #}, {# *t2*,*s*,*s* #}) ∈ *mult trm-ord*
         using *mult1-def-lemma* [*of* {# *t2*,*s*,*s* #} {# *s*,*s* #} *t2* {# *s*,*s* #} {#}
trm-ord]
       mult-def **by** *auto*
    then have ({# *s*,*s* #}, {# *t2*,*t2*,*s*,*s* #}) ∈ *mult trm-ord*
       using *mset-ordering-add1* [*of* {# *s*,*s* #} {# *t2*,*s*,*s* #} *trm-ord t2*] **by** *auto*
    from *this* **and** ‹*t1* = *s*› **and** *m1* **and** *m2* **show** *?thesis* **using** *lit-ord-def* **by**
auto
  next
    assume *t1* ≠ *s*
    from *this* **and** *o1* **have** (*t1*,*s*) ∈ *trm-ord* **by** *auto*
    from *this* **have** *smax*: ∀ *x*. (*x* ∈# {# *t1* #} ⟶ (*x*,*s*) ∈ *trm-ord*)
     by *auto*
    from *smax* **have** ({# *t1*,*s* #}, {# *s*,*s* #}) ∈ *mult trm-ord*
        using *mult1-def-lemma* [*of* {# *s*,*s* #} {# *s* #} *s* {# *t1*,*s* #} {# *t1* #}
trm-ord]
       mult-def **by** *auto*
    from *this* **have** ({# *t1*,*s* #}, {# *t2*,*s*,*s* #}) ∈ *mult trm-ord*
       using *mset-ordering-add1* [*of* {# *t1*,*s* #} {# *s*,*s* #} *trm-ord t2*] **by** *auto*
    from *this* **have** ({# *t1*,*s* #}, {# *t2*,*t2*,*s*,*s* #}) ∈ *mult trm-ord*
        using *mset-ordering-add1* [*of* {# *t1*,*s* #} {# *t2*,*s*,*s* #} *trm-ord t2*] **by**
auto
    from *this* **and** *m1* **and** *m2* **show** *?thesis* **using** *lit-ord-def* **by** *auto*
   qed
qed

lemma *lit-ord-neg-lit-rhs*:

assumes *orient-lit x1 s t1 pos*
assumes *orient-lit x2 t2 s neg*
assumes *vars-of-lit x1* = {}
assumes *vars-of-lit x2* = {}
shows (*x1*,*x2*) ∈ *lit-ord*
proof −
from *assms*(*3*) **and** *assms*(*1*) **have** *vars-of t1* = {} **and** *vars-of s* = {}
and ¬(*s*,*t1*) ∈ *trm-ord* **unfolding** *orient-lit-def* **by** *auto*
from *assms*(*4*) **and** *assms*(*2*) **have** *vars-of t2* = {}
and ¬(*t2*,*s*) ∈ *trm-ord* **unfolding** *orient-lit-def* **by** *auto*
from ‹*vars-of t1* = {}› **and** ‹*vars-of s* = {}› **and** ‹¬(*s*,*t1*) ∈ *trm-ord*›
have *o1*: *t1* = *s* ∨ (*t1*,*s*) ∈ *trm-ord* **using** *trm-ord-ground-total*
unfolding *ground-term-def* **by** *auto*
from ‹*vars-of t2* = {}› **and** ‹*vars-of s* = {}› **and** ‹¬(*t2*,*s*) ∈ *trm-ord*›
have *o2*: *t2* = *s* ∨ (*s*,*t2*) ∈ *trm-ord* **using** *trm-ord-ground-total*
unfolding *ground-term-def* **by** *auto*
let *?m1* = *mset-lit x1*
let *?m2* = *mset-lit x2*
from ‹*orient-lit x1 s t1 pos*› **have** *x1* = (*Pos* (*Eq t1 s*)) ∨ *x1* = (*Pos* (*Eq s t1*))
using *orient-lit-def* **by** *blast*
from *this* **have** *m1*: *?m1* = {# *t1*,*s* #} **using** *mset-lit.simps* **by** *auto*
from ‹*orient-lit x2 t2 s neg*› **have** *x2* = (*Neg* (*Eq t2 s*)) ∨ *x2* = (*Neg* (*Eq s t2*))
using *orient-lit-def* **by** *blast*
from *this* **have** *m2*: *?m2* = {# *t2*,*t2*,*s*,*s* #} **using** *mset-lit.simps* **by** *auto*
show *?thesis*
proof (*cases*)
assume *t1* = *s*
have ({# *s*,*s* #}, {# *t2*,*s*,*s* #}) ∈ *mult trm-ord*
using *mult1-def-lemma* [*of* {# *t2*,*s*,*s* #} {# *s*,*s* #} *t2* {# *s*,*s* #} {#}
trm-ord]
mult-def **by** *auto*
then have ({# *s*,*s* #}, {# *t2*,*t2*,*s*,*s* #}) ∈ *mult trm-ord*
using *mset-ordering-add1* [*of* {# *s*,*s* #} {# *t2*,*s*,*s* #} *trm-ord t2*] **by** *auto*
from *this* **and** ‹*t1* = *s*› **and** *m1* **and** *m2* **show** *?thesis* **using** *lit-ord-def* **by**
auto
next
assume *t1* ≠ *s*
from *this* **and** *o1* **have** (*t1*,*s*) ∈ *trm-ord* **by** *auto*
from *this* **have** *smax*: ∀ *x*. (*x* ∈# {# *t1* #} ⟶ (*x*,*s*) ∈ *trm-ord*)
by *auto*
from *smax* **have** ({# *t1*,*s* #}, {# *s*,*s* #}) ∈ *mult trm-ord*
using *mult1-def-lemma* [*of* {# *s*,*s* #} {# *s* #} *s* {# *t1*,*s* #} {# *t1* #}
trm-ord]
mult-def **by** *auto*
from *this* **have** ({# *t1*,*s* #}, {# *t2*,*s*,*s* #}) ∈ *mult trm-ord*
using *mset-ordering-add1* [*of* {# *t1*,*s* #} {# *s*,*s* #} *trm-ord t2*] **by** *auto*
from *this* **have** ({# *t1*,*s* #}, {# *t2*,*t2*,*s*,*s* #}) ∈ *mult trm-ord*
using *mset-ordering-add1* [*of* {# *t1*,*s* #} {# *t2*,*s*,*s* #} *trm-ord t2*] **by**
auto

from *this* **and** *m1* **and** *m2* **show** *?thesis* **using** *lit-ord-def* **by** *auto*
qed
qed

lemma *lit-ord-rhs*:
assumes $(t1,t2) \in$ *trm-ord*
assumes *orient-lit x1 s t1 p*
assumes *orient-lit x2 s t2 p*
assumes *vars-of-lit x1* = {}
assumes *vars-of-lit x2* = {}
shows $(x1,x2) \in$ *lit-ord*
proof −
from *assms(2)* **and** *assms(4)* **have** *vars-of t1* = {} **and** *vars-of s* = {}
and ¬(*s*,*t1*) ∈ *trm-ord* **unfolding** *orient-lit-def* **by** *auto*
from *assms(3)* **and** *assms(5)* **have** *vars-of t2* = {}
and ¬(*s*,*t2*) ∈ *trm-ord* **unfolding** *orient-lit-def* **by** *auto*
from ‹*vars-of t1* = {}› **and** ‹*vars-of s* = {}› **and** ‹¬(*s*,*t1*) ∈ *trm-ord*›
have *o1*: *t1* = *s* ∨ (*t1*,*s*) ∈ *trm-ord* **using** *trm-ord-ground-total*
unfolding *ground-term-def* **by** *auto*
from ‹*vars-of t2* = {}› **and** ‹*vars-of s* = {}› **and** ‹¬(*s*,*t2*) ∈ *trm-ord*›
have *o2*: *t2* = *s* ∨ (*t2*,*s*) ∈ *trm-ord* **using** *trm-ord-ground-total*
unfolding *ground-term-def* **by** *auto*
let *?m1* = *mset-lit x1*
let *?m2* = *mset-lit x2*
have *p* = *pos* ∨ *p* = *neg* **using** *sign.exhaust* **by** *auto*
then show *?thesis*
proof
assume *p* = *pos*
from *this* **and** ‹*orient-lit x1 s t1 p*› **have** *x1* = (*Pos* (*Eq t1 s*)) ∨ *x1* = (*Pos* (*Eq s t1*))
using *orient-lit-def* **by** *blast*
from *this* **have** *m1*: *?m1* = {# *t1*,*s* #} **using** *mset-lit.simps* **by** *auto*
from ‹*p* = *pos*› **and** ‹*orient-lit x2 s t2 p*› **have** *x2* = (*Pos* (*Eq t2 s*)) ∨ *x2* = (*Pos* (*Eq s t2*))
using *orient-lit-def* **by** *blast*
from *this* **have** *m2*: *?m2* = {# *t2*,*s* #} **using** *mset-lit.simps* **by** *auto*
from *assms(1)* **have** (∀ *b*. *b* ∈# {#*t1*#} ⟶ (*b*, *t2*) ∈ *trm-ord*) **by** *auto*
then have ({# *t1*,*s* #}, {# *t2*,*s* #}) ∈ *mult trm-ord*
using *mult1-def-lemma* [*of* {# *t2*,*s* #} {# *s* #} *t2* {# *t1*,*s* #} {# *t1* #} *trm-ord*]
mult-def **by** *auto*
from *this* **and** *m1* **and** *m2* **show** *?thesis* **using** *lit-ord-def* **by** *auto*
next assume *p* = *neg*
from *this* **and** ‹*orient-lit x1 s t1 p*› **have** *x1* = (*Neg* (*Eq t1 s*)) ∨ *x1* = (*Neg* (*Eq s t1*))
using *orient-lit-def* **by** *blast*
from *this* **have** *m1*: *?m1* = {# *t1*,*t1*,*s*,*s* #} **using** *mset-lit.simps* **by** *auto*
from ‹*p* = *neg*› **and** ‹*orient-lit x2 s t2 p*› **have** *x2* = (*Neg* (*Eq t2 s*)) ∨ *x2* = (*Neg* (*Eq s t2*))

using *orient-lit-def* **by** *blast*
from *this* **have** *m2*: *?m2* = {# *t2*,*t2*,*s*,*s* #} **using** *mset-lit.simps* **by** *auto*
from *assms*(*1*) **have** *max*: (∀ *b*. *b* ∈# {#*t1*,*t1*#} ⟶ (*b*, *t2*) ∈ *trm-ord*) **by** *auto*

have *i*: {# *t2*,*s*,*s* #} = {# *s*,*s*,*t2* #} **by** (*simp add*: *add.commute add.left-commute*)

have *ii*: {# *t1*,*t1*,*s*,*s* #} = {# *s*,*s*,*t1*,*t1* #} **by** (*simp add*: *add.commute add.left-commute*)
from *i* **and** *ii* **and** *max* **have** ({# *t1*,*t1*,*s*,*s* #}, {# *t2*,*s*,*s* #}) ∈ *mult trm-ord*
using *mult1-def-lemma* [*of* {# *t2*,*s*,*s* #} {# *s*,*s* #} *t2* {# *t1*,*t1*,*s*,*s* #} {# *t1*,*t1* #} *trm-ord*]
mult-def **by** *auto*
then have ({# *t1*,*t1*,*s*,*s* #}, {# *t2*,*t2*,*s*,*s* #}) ∈ *mult trm-ord*
using *mset-ordering-add1* [*of* {# *t1*,*t1*,*s*,*s* #} {# *t2*,*s*,*s* #} *trm-ord t2*] **by** *auto*
from *this* **and** *m1* **and** *m2* **show** *?thesis* **using** *lit-ord-def* **by** *auto*
qed
qed

We show that the replacement of a term by an equivalent term preserves the semantics.

lemma *trm-rep-preserves-eq-semantics*:
assumes *fo-interpretation I*
assumes (*I* (*subst t1 σ*) (*subst t2 σ*))
assumes (*validate-ground-eq I* (*subst-equation* (*Eq t1 s*) *σ*))
shows (*validate-ground-eq I* (*subst-equation* (*Eq t2 s*) *σ*))
proof −
from *assms*(*1*) **have** *transitive I* **and** *symmetric I* **unfolding**
fo-interpretation-def congruence-def equivalence-relation-def **by** *auto*
have (*subst-equation* (*Eq t1 s*) *σ*) = (*Eq* (*subst t1 σ*) (*subst s σ*)) **by** *simp*
from *this* **and** *assms*(*3*) **have** *I* (*subst t1 σ*) (*subst s σ*) **by** *simp*
from *this* **and** *assms*(*2*) **and** ‹*transitive I*› **and** ‹*symmetric I*›
have *I* (*subst t2 σ*) (*subst s σ*)
unfolding *transitive-def symmetric-def* **by** *metis*
have (*subst-equation* (*Eq t2 s*) *σ*) = (*Eq* (*subst t2 σ*) (*subst s σ*)) **by** *simp*
from *this* **and** ‹*I* (*subst t2 σ*) (*subst s σ*)› **show** *?thesis* **by** *simp*
qed

lemma *trm-rep-preserves-lit-semantics*:
assumes *fo-interpretation I*
assumes (*I* (*subst t1 σ*) (*subst t2 σ*))
assumes *orient-lit-inst L t1 s polarity σ'*
assumes ¬(*validate-ground-lit I* (*subst-lit L σ*))
shows ¬*validate-ground-lit I* (*subst-lit* (*mk-lit polarity* (*Eq t2 s*)) *σ*)

proof −
from *assms*(*1*) **have** *transitive I* **and** *symmetric I* **unfolding**
fo-interpretation-def congruence-def equivalence-relation-def **by** *auto*

have $polarity = pos \lor polarity = neg$ **using** *sign.exhaust* **by** *auto*
then show *?thesis*
proof
assume $polarity = pos$
from *this* **have** *mk*: $(mk\text{-}lit\ polarity\ (Eq\ t2\ s)) = (Pos\ (Eq\ t2\ s))$ **by** *auto*
from ‹$polarity = pos$› **and** *assms*(*3*) **have** $L = (Pos\ (Eq\ t1\ s)) \lor L = (Pos\ (Eq\ s\ t1))$
unfolding *orient-lit-inst-def* **by** *auto*
then show *?thesis*
proof
assume $L = (Pos\ (Eq\ t1\ s))$
from *this* **and** *assms*(*4*) **have** $\neg I\ (subst\ t1\ \sigma)\ (subst\ s\ \sigma)$ **by** *simp*
from *this* **and** *assms*(*2*) **and** ‹*transitive I*› **and** ‹*symmetric I*›
have $\neg I\ (subst\ t2\ \sigma)\ (subst\ s\ \sigma)$
unfolding *transitive-def symmetric-def* **by** *metis*
from *this* **and** *mk* **show** *?thesis* **by** *simp*
next
assume $L = (Pos\ (Eq\ s\ t1))$
from *this* **and** *assms*(*4*) **have** $\neg I\ (subst\ s\ \sigma)\ (subst\ t1\ \sigma)$ **by** *simp*
from *this* **and** *assms*(*2*) **and** ‹*transitive I*› **and** ‹*symmetric I*›
have $\neg I\ (subst\ t2\ \sigma)\ (subst\ s\ \sigma)$
unfolding *transitive-def symmetric-def* **by** *metis*
from *this* **and** *mk* **show** *?thesis* **by** *simp*
qed
next
assume $polarity = neg$
from *this* **have** *mk*: $(mk\text{-}lit\ polarity\ (Eq\ t2\ s)) = (Neg\ (Eq\ t2\ s))$ **by** *auto*
from ‹$polarity = neg$› **and** *assms*(*3*) **have** $L = (Neg\ (Eq\ t1\ s)) \lor L = (Neg\ (Eq\ s\ t1))$
unfolding *orient-lit-inst-def* **by** *auto*
then show *?thesis*
proof
assume $L = (Neg\ (Eq\ t1\ s))$
from *this* **and** *assms*(*4*) **have** $I\ (subst\ t1\ \sigma)\ (subst\ s\ \sigma)$ **by** *simp*
from *this* **and** *assms*(*2*) **and** ‹*transitive I*› **and** ‹*symmetric I*›
have $I\ (subst\ t2\ \sigma)\ (subst\ s\ \sigma)$
unfolding *transitive-def symmetric-def* **by** *metis*
from *this* **and** *mk* **show** *?thesis* **by** *simp*
next
assume $L = (Neg\ (Eq\ s\ t1))$
from *this* **and** *assms*(*4*) **have** $I\ (subst\ s\ \sigma)\ (subst\ t1\ \sigma)$ **by** *simp*
from *this* **and** *assms*(*2*) **and** ‹*transitive I*› **and** ‹*symmetric I*›
have $I\ (subst\ t2\ \sigma)\ (subst\ s\ \sigma)$
unfolding *transitive-def symmetric-def* **by** *metis*
from *this* **and** *mk* **show** *?thesis* **by** *simp*
qed
qed
qed

lemma *subterms-dominated* :
assumes *maximal-literal L C*
assumes *orient-lit L t s p*
assumes $u \in$ *subterms-of-cl C*
assumes *vars-of-lit L* = {}
assumes *vars-of-cl C* = {}
shows $u = t \vee (u,t) \in$ *trm-ord*
proof (*rule ccontr*)
assume *neg-h*: $\neg(u = t \vee (u,t) \in$ *trm-ord*)
from *assms*(*5*) **and** *assms*(*3*) **have** *vars-of u* = {} **using** *subterm-vars* **by** *blast*
from ‹*vars-of-lit L* = {}› **and** ‹*orient-lit L t s p*› **have** *vars-of s* = {} **and** *vars-of t* = {}
and $\neg(t,s) \in$ *trm-ord* **unfolding** *orient-lit-def* **by** *auto*
from *assms*(*3*) **obtain** L' **where** $u \in$ *subterms-of-lit* L' **and** $L' \in C$ **by** *auto*
from *assms*(*5*) **and** ‹$L' \in C$› **have** *vars-of-lit* L' = {} **using** *vars-of-cl.simps* **by** *auto*
from ‹$u \in$ *subterms-of-lit* L'› **obtain** t' s' p' **where** *orient-lit* L' t' s' p'
and $u \in$ *subterms-of* $t' \cup$ *subterms-of* s' **unfolding** *orient-lit-def*
by (*metis Un-commute mset-lit.cases subterms-of-eq.simps subterms-of-lit.simps*(*1*)

subterms-of-lit.simps(*2*) *trm-ord-wf wf-asym*)
from ‹$u \in$ *subterms-of* $t' \cup$ *subterms-of* s'› **have** $u \in$ *subterms-of* $t' \vee u \in$ *subterms-of* s' **by** *auto*
then show *False*
proof
assume $u \in$ *subterms-of* t'
from *this* **have** $u = t' \vee (u,t') \in$ *trm-ord*
using *subterms-of-trm-ord-eq* [*of u t'*] **by** *auto*
from *neg-h* **and** ‹*vars-of u* = {}› **and** ‹*vars-of t* = {}› **have** $(t,u) \in$ *trm-ord*
using *trm-ord-ground-total* **unfolding** *ground-term-def* **by** *auto*
from *this* **and** ‹$u = t' \vee (u,t') \in$ *trm-ord*› **have** $(t,t') \in$ *trm-ord*
using *trm-ord-trans* **unfolding** *trans-def* **by** *metis*
from *this* **and** ‹*vars-of-lit* L' = {}› **and** *assms*(*4*) **and**
‹*orient-lit L t s p*› **and** ‹*orient-lit* L' t' s' p'›
have $(L,L') \in$ *lit-ord* **using** *lit-ord-dominating-term* **by** *blast*
from *this* **and** *assms*(*1*) **and** ‹$L' \in C$› **show** *False* **unfolding** *maximal-literal-def* **by** *auto*
next
assume $u \in$ *subterms-of* s'
from *this* **have** $u = s' \vee (u,s') \in$ *trm-ord*
using *subterms-of-trm-ord-eq* [*of u s'*] **by** *auto*
from *neg-h* **and** ‹*vars-of u* = {}› **and** ‹*vars-of t* = {}› **have** $(t,u) \in$ *trm-ord*
using *trm-ord-ground-total* **unfolding** *ground-term-def* **by** *auto*
from *this* **and** ‹$u = s' \vee (u,s') \in$ *trm-ord*› **have** $(t,s') \in$ *trm-ord*
using *trm-ord-trans* **unfolding** *trans-def* **by** *metis*
from *this* **and** ‹*vars-of-lit* L' = {}› **and** *assms*(*4*) **and**
‹*orient-lit L t s p*› **and** ‹*orient-lit* L' t' s' p'›
have $(L,L') \in$ *lit-ord* **using** *lit-ord-dominating-term* **by** *blast*
from *this* **and** *assms*(*1*) **and** ‹$L' \in C$› **show** *False* **unfolding** *maximal-literal-def*

by *auto*
 qed
qed

A term dominates an expression if the expression contains no strictly greater subterm:

fun *dominate-eq*:: $'a\ trm \Rightarrow 'a\ equation \Rightarrow bool$
  where $(dominate\text{-}eq\ t\ (Eq\ u\ v)) = ((t,u) \notin trm\text{-}ord \wedge (t,v) \notin trm\text{-}ord)$

fun *dominate-lit*:: $'a\ trm \Rightarrow 'a\ literal \Rightarrow bool$
  where $(dominate\text{-}lit\ t\ (Pos\ e)) = (dominate\text{-}eq\ t\ e)\ |$
  $(dominate\text{-}lit\ t\ (Neg\ e)) = (dominate\text{-}eq\ t\ e)$

definition *dominate-cl*:: $'a\ trm \Rightarrow 'a\ clause \Rightarrow bool$
  where $(dominate\text{-}cl\ t\ C) = (\forall\, x \in C.\ (dominate\text{-}lit\ t\ x))$

definition *no-disequation-in-cl*:: $'a\ trm \Rightarrow 'a\ clause \Rightarrow bool$
  where $(no\text{-}disequation\text{-}in\text{-}cl\ t\ C) = (\forall\, u\ v.$
  $(Neg\ (Eq\ u\ v) \in C \longrightarrow (u \neq t \wedge v \neq t)))$

definition *no-taut-eq-in-cl*:: $'a\ trm \Rightarrow 'a\ clause \Rightarrow bool$
  where $(no\text{-}taut\text{-}eq\text{-}in\text{-}cl\ t\ C) = (Pos\ (Eq\ t\ t) \notin C)$

definition *eq-occurs-in-cl*
  where
  $(eq\text{-}occurs\text{-}in\text{-}cl\ t\ s\ C\ \sigma) = (\exists\, L\ t'\ s'.\ (L \in C) \wedge (orient\text{-}lit\text{-}inst\ L\ t'\ s'\ pos\ \sigma)$
  $\wedge\ (t = subst\ t'\ \sigma) \wedge (s = subst\ s'\ \sigma))$

4.4 Inference Rules

We now define the rules of the superposition calculus. Standard superposition is a refinement of the paramodulation rule based on the following ideas:

(i) the replacement of a term by a bigger term is forbidden;

(ii) the replacement can be performed only in the maximal term of a maximal (or selected) literal;

(iii) replacement of variables is forbidden.

Our definition imposes additional conditions on the positions on which the replacements are allowed: any superposition inference inside a term occurring in the set attached to the extended clause is blocked.

We consider two different kinds of inferences: ground or first-order. Ground inferences are those needed for completeness, first-order inferences are those actually used by theorem provers. For conciseness, these two notions of inferences are defined simultaneously, and a parameter is added to the corresponding functions to determine whether the inference is ground or first-order.

datatype *inferences* = *Ground* | *FirstOrder*

The following function checks whether a given substitution is a unifier of two terms. If the inference is first-order then the unifier must be maximal.

definition *ck-unifier* **where**
ck-unifier t s σ type ⟷ (*if type* = *FirstOrder then min-IMGU σ t s else Unifier σ t s*)

lemma *ck-unifier-thm*:
assumes *ck-unifier t s σ k*
shows (*subst t σ*) = (*subst s σ*)
by (*metis assms min-IMGU-def IMGU-iff-Idem-and-MGU MGU-is-Unifier ck-unifier-def Unifier-def*)

lemma *subst-preserve-ck-unifier*:
assumes *ck-unifier t s σ k*
shows *ck-unifier t s* (*comp σ η*) *Ground*
proof −
let *?σ′* = (*comp σ η*)
from *assms* **have** (*subst t σ*) = (*subst s σ*)
using *ck-unifier-thm* **by** *auto*
then have (*subst t ?σ′*) = (*subst s ?σ′*) **by** *simp*
then show *?thesis* **unfolding** *ck-unifier-def Unifier-def* **by** *auto*
qed

The following function checks whether a given term is allowed to be reduced according to the strategy described above, i.e., that it does not occur in the set of terms associated with the clause (we do not assume that the set of irreducible terms is closed under subterm thus we use the function *occurs-in* instead of a mere membership test.

definition *allowed-redex*
where *allowed-redex t C σ* = (¬ (∃ *s* ∈ (*trms-ecl C*).
(*occurs-in* (*subst t σ*) (*subst s σ*))))

The following function allows one to compute the set of irreducible terms attached to the conclusion of an inference. The computation depends on the type of the considered inference: for ground inferences the entire set of irreducible terms is kept. For first-order inferences, the function *filter-trms* is called to remove some of the terms (see also the function *dom-trms* below).

definition *get-trms*
where
get-trms C E t = (*if* (*t* = *FirstOrder*) *then* (*filter-trms C E*) *else E*)

The following definition provides the conditions that allow one to propagate irreducible terms from the parent clauses to the conclusion. A term can be propagated if it is strictly lower than a term occurring in the derived clause, or if it occurs in a negative literal of the derived clause. Note that this condition is slightly more restrictive than that of the basic superposition calculus, because maximal terms occurring in maximal positive literals cannot be kept in the set of irreducible terms. However in our definition, terms can be propagated even if they do not occur in the parent clause or in the conclusion. Extended clauses whose set of irreducible terms fulfills this property are called well-constrained.

definition *dom-trm*
  where *dom-trm t C* =
    (∃ *L u v p*. (*L* ∈ *C* ∧ (*decompose-literal L u v p*)
      ∧ (((*p* = *neg* ∧ *t* = *u*) ∨ (*t*,*u*) ∈ *trm-ord*))))

lemma *dom-trm-lemma*:
  assumes *dom-trm t C*
  shows ∃ *u*. (*u* ∈ (*subterms-of-cl C*) ∧ (*u* = *t* ∨ (*t*,*u*) ∈ *trm-ord*))
proof −
  from *assms*(*1*) **obtain** *L u v p* **where**
    L ∈ *C decompose-literal L u v p* (*u* = *t* ∨ (*t*,*u*) ∈ *trm-ord*)
    unfolding *dom-trm-def* **by** *blast*
  from ‹*decompose-literal L u v p*› **have** *u* ∈ *subterms-of-lit L*
   unfolding *decompose-literal-def decompose-equation-def* **using** *root-subterm* **by** *force*
  from *this* **and** ‹*L* ∈ *C*› **have** *u* ∈ (*subterms-of-cl C*) **by** *auto*
  from *this* **and** ‹(*u* = *t* ∨ (*t*,*u*) ∈ *trm-ord*)› **show** *?thesis* **by** *auto*
qed

definition *dom-trms*
where
  dom-trms C E = { *x*. (*x* ∈ *E*) ∧ (*dom-trm x C*) }

lemma *dom-trms-subset*:
  shows (*dom-trms C E*) ⊆ *E*
unfolding *dom-trms-def* **by** *auto*

lemma *dom-trm-vars*:
  assumes *dom-trm t C*
  shows *vars-of t* ⊆ *vars-of-cl C*
proof −
  from *assms* **obtain** *L u v p* **where** *L* ∈ *C decompose-literal L u v p t* = *u* ∨ (*t*,*u*) ∈ *trm-ord*
    unfolding *dom-trm-def* **by** *auto*
  from ‹*t* = *u* ∨ (*t*,*u*) ∈ *trm-ord*› **have** *vars-of t* ⊆ *vars-of u* **using** *trm-ord-vars* **by** *blast*
  from *this* **and** ‹*decompose-literal L u v p*› **have** *vars-of t* ⊆ *vars-of-lit L* **using** *decompose-literal-vars* **by** *blast*
  from *this* **show** *?thesis* **using** ‹*L* ∈ *C*› **by** *auto*
qed

definition *well-constrained*

where *well-constrained* $C = (\forall\, y.\ (y \in \mathit{trms\text{-}ecl}\ C \longrightarrow \mathit{dom\text{-}trm}\ y\ (\mathit{cl\text{-}ecl}\ C)))$

The next function allows one to check that a set of terms is in normal form. The argument f denotes the function mapping a term to its normal form (we do not assume that f is compatible with the term structure at this point).

definition *all-trms-irreducible*
where $(\mathit{all\text{-}trms\text{-}irreducible}\ E\ f) = (\forall\, x\ y.\ (x \in E \longrightarrow \mathit{occurs\text{-}in}\ y\ x \longrightarrow (f\ y) = y))$

Superposition We now define the superposition rule. Note that we assume that the parent clauses are variable-disjoint, but we do not explicitly rename them at this point, thus for completeness we will have to assume that the clause sets are closed under renaming. During the application of the rule, all the terms occurring at a position that is lower than that of the reduced term can be added in the set of irreducible terms attached to the conclusion (the intuition is that we assume that the terms occurring at minimal positions are reduced first). In particular, every proper subterm of the reduced term u' is added in the set of irreducible terms, thus every application of the superposition rule in a term introduced by unification will be blocked.

Clause *P1* is the "into" clause and clause *P2* is the "from" clause.

definition *superposition* ::
$'a\ \mathit{eclause} \Rightarrow\ 'a\ \mathit{eclause} \Rightarrow\ 'a\ \mathit{eclause} \Rightarrow\ 'a\ \mathit{subst} \Rightarrow \mathit{inferences} \Rightarrow\ 'a\ \mathit{clause} \Rightarrow \mathit{bool}$
where
$(\mathit{superposition}\ P1\ P2\ C\ \sigma\ k\ C') =$
$(\exists\, L\ t\ s\ u\ v\ M\ p\ \mathit{Cl\text{-}P1}\ \mathit{Cl\text{-}P2}\ \mathit{Cl\text{-}C}\ \mathit{polarity}\ t'\ u'\ L'\ \mathit{trms\text{-}C}.$
$(L \in \mathit{Cl\text{-}P1}) \wedge (M \in \mathit{Cl\text{-}P2}) \wedge (\mathit{eligible\text{-}literal}\ L\ P1\ \sigma) \wedge\ (\mathit{eligible\text{-}literal}\ M\ P2\ \sigma)$
$\wedge\ (\mathit{variable\text{-}disjoint}\ P1\ P2)$
$\wedge\ (\mathit{Cl\text{-}P1} = (\mathit{cl\text{-}ecl}\ P1)) \wedge (\mathit{Cl\text{-}P2} = (\mathit{cl\text{-}ecl}\ P2))$
$\wedge\ (\neg\ \mathit{is\text{-}a\text{-}variable}\ u')$
$\wedge\ (\mathit{allowed\text{-}redex}\ u'\ P1\ \sigma)$
$\wedge\ \mathit{trms\text{-}C} = (\mathit{get\text{-}trms}\ \mathit{Cl\text{-}C}\ (\mathit{dom\text{-}trms}\ \mathit{Cl\text{-}C}\ (\mathit{subst\text{-}set}$
$((\mathit{trms\text{-}ecl}\ P1) \cup (\mathit{trms\text{-}ecl}\ P2) \cup$
$\{\ r.\ \exists\ q.\ (q,p) \in (\mathit{pos\text{-}ord}\ P1\ t) \wedge (\mathit{subterm}\ t\ q\ r)\ \})\ \sigma))\ k)$
$\wedge\ (C = (\mathit{Ecl}\ \mathit{Cl\text{-}C}\ \mathit{trms\text{-}C}))$
$\wedge\ (\mathit{orient\text{-}lit\text{-}inst}\ M\ u\ v\ \mathit{pos}\ \sigma)$
$\wedge\ (\mathit{orient\text{-}lit\text{-}inst}\ L\ t\ s\ \mathit{polarity}\ \sigma)$
$\wedge\ ((\mathit{subst}\ u\ \sigma) \neq (\mathit{subst}\ v\ \sigma))$
$\wedge\ (\mathit{subterm}\ t\ p\ u')$
$\wedge\ (\mathit{ck\text{-}unifier}\ u'\ u\ \sigma\ k)$
$\wedge\ (\mathit{replace\text{-}subterm}\ t\ p\ v\ t')$
$\wedge\ ((k = \mathit{FirstOrder}) \vee (\ (\mathit{subst\text{-}lit}\ M\ \sigma),(\mathit{subst\text{-}lit}\ L\ \sigma)) \in \mathit{lit\text{-}ord})$
$\wedge\ ((k = \mathit{FirstOrder}) \vee (\mathit{strictly\text{-}maximal\text{-}literal}\ P2\ M\ \sigma))$
$\wedge\ (L' = \mathit{mk\text{-}lit}\ \mathit{polarity}\ (\mathit{Eq}\ t'\ s))$
$\wedge\ (\mathit{Cl\text{-}C} = (\mathit{subst\text{-}cl}\ C'\ \sigma))$
$\wedge\ (C' = (\mathit{Cl\text{-}P1} - \{\ L\ \}) \cup ((\mathit{Cl\text{-}P2} - \{\ M\ \}) \cup \{\ L'\ \}\ )))$

Reflexion We now define the Reflexion rule, which deletes contradictory literals (after unification). All the terms occurring in these literals can be added into the set of irreducible terms (intuitively, we can assume that these terms have been normalized before applying the rule). It is sufficient to add the term t, since every term occurring in the considered literal is a subterm of t (after unification).

definition *reflexion* ::
$'a$ *eclause* $\Rightarrow$ $'a$ *eclause* $\Rightarrow$ $'a$ *subst* $\Rightarrow$ *inferences* $\Rightarrow$ $'a$ *clause* $\Rightarrow$ *bool*
where
(*reflexion* P C σ k C') =
($\exists$ *L1* t s *Cl-P* *Cl-C* *trms-C*.
(*eligible-literal* *L1* P σ)
$\wedge$ (*L1* $\in$ (*cl-ecl* P)) $\wedge$ (*Cl-C* = (*cl-ecl* C)) $\wedge$ (*Cl-P* = (*cl-ecl* P))
$\wedge$ (*orient-lit-inst* *L1* t s *neg* σ)
$\wedge$ (*ck-unifier* t s σ k)
$\wedge$ (C = (*Ecl* *Cl-C* *trms-C*))
$\wedge$ *trms-C* = (*get-trms* *Cl-C*
(*dom-trms* *Cl-C* (*subst-set* ((*trms-ecl* P) $\cup$ { t }) σ)) k)
$\wedge$ (*Cl-C* = (*subst-cl* C' σ))
$\wedge$ (C' = ((*Cl-P* $-$ { *L1* }))))

Factorization We now define the equational factorization rule, which merges two equations sharing the same left-hand side (after unification), if the right-hand sides are equivalent. Here, contrarily to the previous rule, the term t cannot be added into the set of irreducible terms, because we cannot assume that this term is in normal form (e.g., the application of the equational factorization rule may yield a new rewrite rule of left-hand side t). However, all proper subterms of t can be added.

definition *factorization* ::
$'a$ *eclause* $\Rightarrow$ $'a$ *eclause* $\Rightarrow$ $'a$ *subst* $\Rightarrow$ *inferences* $\Rightarrow$ $'a$ *clause* $\Rightarrow$ *bool*
where
(*factorization* P C σ k C') =
($\exists$ *L1* *L2* L' t s u v *Cl-P* *Cl-C* *trms-C*.
(*eligible-literal* *L1* P σ)
$\wedge$ (*L1* $\in$ (*cl-ecl* P)) $\wedge$ (*L2* $\in$ (*cl-ecl* P) $-$ { *L1* }) $\wedge$ (*Cl-C* = (*cl-ecl* C)) $\wedge$
(*Cl-P* = (*cl-ecl* P))
$\wedge$ (*orient-lit-inst* *L1* t s *pos* σ)
$\wedge$ (*orient-lit-inst* *L2* u v *pos* σ)
$\wedge$ ((*subst* t σ) $\neq$ (*subst* s σ))
$\wedge$ ((*subst* t σ) $\neq$ (*subst* v σ))
$\wedge$ (*ck-unifier* t u σ k)
$\wedge$ (L' = *Neg* (*Eq* s v))
$\wedge$ (C = (*Ecl* *Cl-C* *trms-C*)
$\wedge$ *trms-C* = (*get-trms* *Cl-C*
(*dom-trms* *Cl-C* (*subst-set* ((*trms-ecl* P) $\cup$ (*proper-subterms-of* t)) σ)))
k)
$\wedge$ (*Cl-C* = (*subst-cl* C' σ))

$\wedge$ ($C' = ($ ($Cl\text{-}P - \{ L2 \}) \cup \{ L' \})))

4.5 Derivations

We now define the set of derivable clauses by induction. Note that redundancy criteria are not taken into account at this point. Our definition of derivations also covers renaming.

definition *derivable* :: $'a$ *eclause* $\Rightarrow$ $'a$ *eclause set*
$\Rightarrow$ $'a$ *eclause set* $\Rightarrow$ $'a$ *subst* $\Rightarrow$ *inferences* $\Rightarrow$ $'a$ *clause* $\Rightarrow$ *bool*
where
(*derivable* $C\ P\ S\ \sigma\ k\ C'$) =
(($\exists$ *P1 P2*. (*P1* $\in S \wedge$ *P2* $\in S \wedge P = \{$ *P1*, *P2* $\} \wedge$ *superposition P1 P2* C $\sigma\ k\ C'$))
$\vee$ ($\exists$ *P1*. (*P1* $\in S \wedge P = \{$ *P1* $\} \wedge$ *factorization P1* $C\ \sigma\ k\ C'$))
$\vee$ ($\exists$ *P1*. (*P1* $\in S \wedge P = \{$ *P1* $\} \wedge$ *reflexion P1* $C\ \sigma\ k\ C'$)))

lemma *derivable-premisses*:
assumes *derivable* $C\ P\ S\ \sigma\ k\ C'$
shows $P \subseteq S$
using *assms derivable-def* **by** *auto*

inductive *derivable-ecl* :: $'a$ *eclause* $\Rightarrow$ $'a$ *eclause set* $\Rightarrow$ *bool*
where
init [*simp*, *intro*!]: $C \in S \Longrightarrow$ (*derivable-ecl* $C\ S$) |
rn [*simp*, *intro*!]: (*derivable-ecl* $C\ S$) $\Longrightarrow$ (*renaming-cl* $C\ D$) $\Longrightarrow$ (*derivable-ecl* $D\ S$) |
deriv [*simp*, *intro*!]: ($\forall x$. ($x \in P \longrightarrow$ (*derivable-ecl* $x\ S$)))
$\Longrightarrow$ (*derivable* $C\ P\ S'\ \sigma$ *FirstOrder* C') $\Longrightarrow$ (*derivable-ecl* $C\ S$)

We define a notion of instance by associating clauses with ground substitutions.

definition *instances*:: $'a$ *eclause set* $\Rightarrow$ ($'a$ *eclause* $\times$ $'a$ *subst*) *set*
where *instances* $S = \{ x.\ \exists C\ \sigma.\ (C \in S\ \wedge$ (*ground-clause* (*subst-cl* (*cl-ecl* C) σ))
$\wedge\ x = (\ C,\sigma))\}$

definition *clset-instances*:: ($'a$ *eclause* $\times$ $'a$ *subst*) *set* $\Rightarrow$ $'a$ *clause set*
where
clset-instances $S = \{\ C.\ \exists x.\ (x \in S \wedge C =$ (*subst-cl* (*cl-ecl* (*fst* x)) (*snd* x))) $\}$

definition *grounding-set*
where *grounding-set* $S\ \sigma = (\forall x.\ (x \in S \longrightarrow$ (*ground-clause* (*subst-cl* (*cl-ecl* x) σ))))

5 Soundness

In this section, we prove that the conclusion of every inference rule is a logical consequence of the premises. Thus a clause set is unsatisfiable if the empty clause is derivable. For each rule, we first prove that all ground instances of the conclusion are entailed by the corresponding instances of the parent clauses, then we lift the result to first-order clauses. The proof is standard and straightforward, but note that we also prove that the derived clauses are finite and well-constrained.

lemma *cannot-validate-contradictary-literals* :
  assumes *l* = *Neg* (*Eq t t*)
  assumes *fo-interpretation I*
  shows ¬ (*validate-ground-lit I l*)
proof −
  from *assms*(*2*) **have** *congruence I* **unfolding** *fo-interpretation-def* **by** *auto*
  then have *I t t* **unfolding** *congruence-def reflexive-def equivalence-relation-def* **by** *auto*
  from *this* **and** *assms*(*1*) **show** *?thesis* **by** *auto*
qed

lemma *ground-reflexion-is-sound* :
  assumes *finite* (*cl-ecl C*)
  assumes *reflexion C D* σ *k* C'
  assumes (*ground-clause* (*subst-cl* (*cl-ecl D*) ϑ))
  shows *clause-entails-clause* (*subst-cl* (*subst-cl* (*cl-ecl C*) σ) ϑ)
      (*subst-cl* (*cl-ecl D*) ϑ)
proof (*rule ccontr*)
  let *?C* = (*cl-ecl C*)
  let *?D* = (*cl-ecl D*)
  let *?C′* = (*subst-cl* (*subst-cl* (*cl-ecl C*) σ) ϑ)
  let *?D′* = (*subst-cl* (*cl-ecl D*) ϑ)
  assume ¬ (*clause-entails-clause ?C′ ?D′*)
  then obtain *I* **where** *validate-clause I ?C′* **and** ¬ (*validate-clause I ?D′*) *fo-interpretation I*
    unfolding *clause-entails-clause-def* **by** *auto*
  from *assms*(*2*) **obtain** *L1 t s* **where**
    ?D = (*subst-cl* (*?C* − { *L1* }) σ)
    and *orient-lit-inst L1 t s neg* σ **and** *ck-unifier t s* σ *k*
      using *reflexion-def* [*of C D* σ *k*] **by** *auto*
  from *assms*(*1*) **have** *finite* (*subst-cl* (*subst-cl ?C* σ) ϑ) **by** *auto*
  then obtain η **where** *i*: *ground-clause* (*subst-cl*
      (*subst-cl* (*subst-cl ?C* σ) ϑ) η)
    using *ground-instance-exists* [*of* (*subst-cl* (*subst-cl ?C* σ) ϑ)]
    by *auto*
  let *?CC* = (*subst-cl* (*subst-cl* (*subst-cl ?C* σ) ϑ) η)
  let *?*σ'' = *comp* σ ϑ
  let *?*σ' = *comp ?*σ'' η
  have *?CC* = (*subst-cl* (*subst-cl ?C ?*σ'') η)

using *composition-of-substs-cl* [*of ?C*] **by** *auto*
then have *?CC* = (*subst-cl ?C ?σ′*)
using *composition-of-substs-cl* [*of ?C*] **by** *auto*
from ‹*validate-clause I* (*subst-cl* (*subst-cl* (*cl-ecl C*) *σ*) *ϑ*)›
have *validate-ground-clause I ?CC* **using** *i validate-clause.simps* **by** *blast*
then obtain *l′* **where** *l′* ∈ *?CC* **and** *validate-ground-lit I l′* **by** *auto*
from ‹*l′* ∈ *?CC*› **and** ‹*?CC* = (*subst-cl ?C ?σ′*)› **obtain** *l* **where**
l ∈ *?C* **and** *l′* = (*subst-lit l ?σ′*) **using** *subst-cl.simps* **by** *blast*
have *subst-lit l σ* ∈ *?D*
proof (*rule ccontr*)
assume *subst-lit l σ* ∉ *?D*
from *this* **and** ‹*?D* = (*subst-cl* (*?C* − { *L1* }) *σ*)› **and** ‹*l* ∈ *?C*›
have *l* = *L1* **by** *auto*
from *this* **and** ‹*orient-lit-inst L1 t s neg σ*› **have** *l* = (*Neg* (*Eq t s*)) ∨ *l* = (*Neg* (*Eq s t*))
unfolding *orient-lit-inst-def* **by** *auto*
from ‹*ck-unifier t s σ k*› **have** *subst t σ* = *subst s σ*
using *ck-unifier-thm* **by** *auto*
then have *subst* (*subst* (*subst t σ*) *ϑ*) *η* =
subst (*subst* (*subst s σ*) *ϑ*) *η* **by** *auto*
then have (*subst t ?σ′*) = *subst s ?σ′* **by** *auto*
from *this* **and** ‹*l* = (*Neg* (*Eq t s*)) ∨ *l* = (*Neg* (*Eq s t*))›
have (*subst-lit l ?σ′*) = (*Neg* (*Eq* (*subst t ?σ′*) (*subst t ?σ′*)))
by *auto*
from *this* **and** ‹*fo-interpretation I*› **have** ¬ (*validate-ground-lit I* (*subst-lit l ?σ′*))
using *cannot-validate-contradictary-literals* [*of* (*subst-lit l ?σ′*) (*subst t ?σ′*) *I*]
by *auto*
from *this* **and** ‹*l′* = *subst-lit l ?σ′*› **and** ‹*validate-ground-lit I l′*› **show** *False* **by** *auto*
qed
from ‹*subst-lit l σ* ∈ *?D*› **and** ‹*l′* = *subst-lit l ?σ′*›
have *l′* ∈ (*subst-cl* (*subst-cl ?D ϑ*) *η*)
using *subst-cl.simps composition-of-substs-lit mem-Collect-eq*
by (*metis* (*mono-tags*, *lifting*))
from *this* **and** ‹*validate-ground-lit I l′*› **have**
validate-ground-clause I (*subst-cl* (*subst-cl ?D ϑ*) *η*) **by** *auto*
from ‹*ground-clause* (*subst-cl ?D ϑ*)› **have**
(*subst-cl ?D ϑ*) = (*subst-cl* (*subst-cl ?D ϑ*) *η*)
using *substs-preserve-ground-clause* [*of* (*subst-cl ?D ϑ*) *η*] **by** *blast*
from *this* **and** ‹*validate-ground-clause I* (*subst-cl* (*subst-cl ?D ϑ*) *η*)›
have *validate-ground-clause I* (*subst-cl ?D ϑ*) **by** *force*
from *this* **and** *assms*(*3*) **and** ‹¬ *validate-clause I* (*subst-cl* (*cl-ecl D*) *ϑ*)› **show** *False*
using *substs-preserve-ground-clause validate-clause.elims*(*3*) **by** *metis*
qed

lemma *reflexion-is-sound* :
assumes *finite* (*cl-ecl C*)

assumes *reflexion C D σ k C′*
shows *clause-entails-clause* (*cl-ecl C*) (*cl-ecl D*)
proof (*rule ccontr*)
let *?C* = (*cl-ecl C*)
let *?D* = (*cl-ecl D*)
assume ¬ (*clause-entails-clause ?C ?D*)
then obtain *I* **where** *validate-clause I ?C* **and** ¬ (*validate-clause I ?D*) *fo-interpretation I*
unfolding *clause-entails-clause-def* **by** *auto*
from ‹¬ (*validate-clause I ?D*)› **obtain** *ϑ*
where *D-false*: ¬ (*validate-ground-clause I* (*subst-cl ?D ϑ*))
and (*ground-clause* (*subst-cl ?D ϑ*)) **by** *auto*
have *validate-clause I* (*subst-cl* (*subst-cl ?C σ*) *ϑ*)
using ‹*validate-clause I* (*cl-ecl C*)› *instances-are-entailed* **by** *blast*
from *this* **and** *assms*(*1*) **and** *assms*(*2*) **have** *validate-clause I* (*subst-cl ?D ϑ*)
using *ground-reflexion-is-sound* **unfolding** *clause-entails-clause-def*
using ‹*fo-interpretation I*› ‹*ground-clause* (*subst-cl* (*cl-ecl D*) *ϑ*)› **by** *blast*
from *this* **and** *D-false* **show** *False*
by (*metis* ‹*ground-clause* (*subst-cl* (*cl-ecl D*) *ϑ*)›
substs-preserve-ground-clause validate-clause.elims(*1*))
qed

lemma *orient-lit-semantics-pos* :
assumes *fo-interpretation I*
assumes *orient-lit-inst l u v pos η*
assumes *validate-ground-lit I* (*subst-lit l σ*)
shows *I* (*subst u σ*) (*subst v σ*)
proof −
let *?u* = *subst u σ*
let *?v* = *subst v σ*
from *assms*(*2*) **have** *l* = (*Pos* (*Eq u v*)) ∨ *l* = (*Pos* (*Eq v u*)) **using** *orient-lit-inst-def* **by** *auto*
from *this* **and** *assms*(*3*) **have** *validate-ground-eq I* (*Eq ?u ?v*) ∨ *validate-ground-eq I* (*Eq ?v ?u*)
by *auto*
then have *I ?u ?v* ∨ *I ?v ?u* **by** *auto*
from *this* **and** ‹*fo-interpretation I*› **show** *I ?u ?v*
unfolding *fo-interpretation-def congruence-def equivalence-relation-def symmetric-def* **by** *blast*
qed

lemma *orient-lit-semantics-neg* :
assumes *fo-interpretation I*
assumes *orient-lit-inst l u v neg ϑ*
assumes *validate-ground-lit I* (*subst-lit l σ*)
shows ¬*I* (*subst u σ*) (*subst v σ*)
proof −
let *?u* = *subst u σ*
let *?v* = *subst v σ*

from *assms*(*2*) **have** $l = (Neg\ (Eq\ u\ v)) \vee l = (Neg\ (Eq\ v\ u))$ **using** *orient-lit-inst-def* **by** *auto*
from *this* **and** *assms*(*3*) **have** $\neg$*validate-ground-eq I* (*Eq ?u ?v*) $\vee$ $\neg$*validate-ground-eq I* (*Eq ?v ?u*)
by *auto*
then have $\neg$*I ?u ?v* $\vee$ $\neg$*I ?v ?u* **by** *auto*
from *this* **and** ‹*fo-interpretation I*› **show** $\neg$*I ?u ?v*
unfolding *fo-interpretation-def congruence-def equivalence-relation-def symmetric-def* **by** *blast*
qed

lemma *orient-lit-semantics-replacement* :
assumes *fo-interpretation I*
assumes *orient-lit-inst l u v polarity* ϑ
assumes *validate-ground-lit I* (*subst-lit l* σ)
assumes *I* (*subst u* σ) (*subst u'* σ)
shows *validate-ground-lit I* (*subst-lit* (*mk-lit polarity* (*Eq u' v*)) σ)
proof −
from *assms*(*2*) **obtain** *e* **where** $l = Pos\ e \vee l = Neg\ e$ **and** $e = Eq\ u\ v \vee e = Eq\ v\ u$
unfolding *orient-lit-inst-def* **by** *auto*
have $polarity = pos \vee polarity = neg$ **using** *sign.exhaust* **by** *blast*
then show *?thesis*
proof
assume $polarity = pos$
from *this* **and** *assms*(*1*) **and** *assms*(*2*) **and** ‹*validate-ground-lit I* (*subst-lit l* σ)› **have**
I (*subst u* σ) (*subst v* σ) **using** *orient-lit-semantics-pos* **by** *auto*
from *this* **and** *assms*(*1*) **and** ‹*I* (*subst u* σ) (*subst u'* σ)›
have *I* (*subst u'* σ) (*subst v* σ) **unfolding** *fo-interpretation-def*
congruence-def equivalence-relation-def symmetric-def transitive-def **by** *blast*
from *this* **and** ‹$polarity = pos$› **show** *?thesis* **by** *auto*
next
assume $polarity = neg$
from *this* **and** *assms*(*1*) **and** *assms*(*2*) **and** ‹*validate-ground-lit I* (*subst-lit l* σ)› **have**
$\neg$*I* (*subst u* σ) (*subst v* σ) **using** *orient-lit-semantics-neg*
by *blast*
from *this* **and** *assms*(*1*) **and** ‹*I* (*subst u* σ) (*subst u'* σ)›
have $\neg$*I* (*subst u'* σ) (*subst v* σ) **unfolding** *fo-interpretation-def*
congruence-def equivalence-relation-def symmetric-def transitive-def **by** *blast*
from *this* **and** ‹$polarity = neg$› **show** *?thesis* **by** *auto*
qed
qed

lemma *ground-factorization-is-sound* :
assumes *finite* (*cl-ecl C*)
assumes *factorization C D* σ *k C'*
assumes (*ground-clause* (*subst-cl* (*cl-ecl D*) ϑ))

shows *clause-entails-clause* (*subst-cl* (*subst-cl* (*cl-ecl* *C*) σ) ϑ)
(*subst-cl* (*cl-ecl* *D*) ϑ)
proof (*rule ccontr*)
let *?C* = (*cl-ecl* *C*)
let *?D* = (*cl-ecl* *D*)
assume ¬ *clause-entails-clause* (*subst-cl* (*subst-cl* (*cl-ecl* *C*) σ) ϑ)
(*subst-cl* (*cl-ecl* *D*) ϑ)
then obtain *I* **where**
validate-clause I (*subst-cl* (*subst-cl* (*cl-ecl* *C*) σ) ϑ) **and**
¬ (*validate-clause I* (*subst-cl* (*cl-ecl* *D*) ϑ)) **and** *fo-interpretation I*
unfolding *clause-entails-clause-def* **by** *auto*
from *assms*(*2*) **obtain** *L1 L2* L' *t s u v* **where**
orient-lit-inst L1 t s pos σ **and** *orient-lit-inst L2 u v pos* σ **and** *ck-unifier t u*
σ *k*
and L' = *Neg* (*Eq s v*)
and (*?D* = (*subst-cl* ((*?C* − { *L2* }) ∪ { L' })) σ)
and *L1* ≠ *L2*
and *L1* ∈ *?C*
using *factorization-def* **by** *auto*

from *assms*(*1*) **have** *finite* (*subst-cl* (*subst-cl* *?C* σ) ϑ) **by** *auto*
then obtain η **where** *i*: *ground-clause* (*subst-cl*
(*subst-cl* (*subst-cl* *?C* σ) ϑ) η)
using *ground-instance-exists* [*of* (*subst-cl* (*subst-cl* *?C* σ) ϑ)]
by *auto*
let *?CC* = (*subst-cl* (*subst-cl* (*subst-cl* *?C* σ) ϑ) η)
let $?\sigma''$ = *comp* σ ϑ
let $?\sigma'$ = *comp* $?\sigma''$ η
have *?CC* = (*subst-cl* (*subst-cl* *?C* $?\sigma''$) η)
using *composition-of-substs-cl* [*of ?C*] **by** *auto*
then have *?CC* = (*subst-cl* *?C* $?\sigma'$)
using *composition-of-substs-cl* [*of ?C*] **by** *auto*
from ‹*validate-clause I* (*subst-cl* (*subst-cl* (*cl-ecl* *C*) σ) ϑ)›
have *validate-ground-clause I ?CC* **using** *i validate-clause.simps* **by** *blast*
then obtain l' **where** l' ∈ *?CC* **and** *validate-ground-lit I* l' **by** *auto*
from ‹l' ∈ *?CC*› **and** ‹*?CC* = (*subst-cl* *?C* $?\sigma'$)› **obtain** *l* **where**
l ∈ *?C* **and** l' = (*subst-lit l* $?\sigma'$) **using** *subst-cl.simps* **by** *blast*
from ‹¬ *validate-clause I* (*subst-cl* (*cl-ecl* *D*) ϑ)›
have ¬ *validate-ground-clause I* (*subst-cl* *?D* ϑ)
using *assms*(*3*) *substs-preserve-ground-clause validate-clause.elims*(*3*) **by** *metis*

from ‹*ground-clause* (*subst-cl* *?D* ϑ)› **have**
(*subst-cl* *?D* ϑ) = (*subst-cl* (*subst-cl* *?D* ϑ) η)
using *substs-preserve-ground-clause* [*of* (*subst-cl* *?D* ϑ) η] **by** *blast*
from *this* **and** ‹¬ *validate-ground-clause I* (*subst-cl* *?D* ϑ)›
have ¬ *validate-ground-clause I* (*subst-cl* (*subst-cl* *?D* ϑ) η) **by** *force*
from ‹(*?D* = (*subst-cl* ((*?C* − { *L2* }) ∪ { L' })) σ)›
have (*subst-lit* L' σ) ∈ *?D* **by** *auto*
then have

(*subst-lit* (*subst-lit* (*subst-lit* L' σ) ϑ) η)
$\in$ (*subst-cl* (*subst-cl* *?D* ϑ) η)
by *auto*
from *this* **and** ‹¬ *validate-ground-clause* *I* (*subst-cl* (*subst-cl* *?D* ϑ) η)›
have ¬*validate-ground-lit* *I* (*subst-lit* (*subst-lit* (*subst-lit* L' σ) ϑ) η)
by *auto*
from *this* **and** ‹L'= *Neg* (*Eq* *s* *v*)› **have**
I (*subst* (*subst* (*subst* *s* σ) ϑ) η)
(*subst* (*subst* (*subst* *v* σ) ϑ) η) **by** *auto*
from *this* **have** *I* (*subst* *s* *?*σ') (*subst* *v* *?*σ') **by** *simp*

have *subst-lit* *l* σ $\in$ *?D*
proof (*rule* *ccontr*)
assume *subst-lit* *l* σ $\notin$ *?D*
from *this* **and** ‹(*?D* = (*subst-cl* ((*?C* − { *L2* }) $\cup$ { L' })) σ)› **and** ‹*l* $\in$ *?C*›
have *l* = *L2* **by** *auto*
from ‹*ck-unifier* *t* *u* σ *k*› **have** *subst* *t* σ = *subst* *u* σ
using *ck-unifier-thm* **by** *auto*
then have *subst* (*subst* (*subst* *t* σ) ϑ) η =
subst (*subst* (*subst* *u* σ) ϑ) η **by** *auto*
then have (*subst* *t* *?*σ') = *subst* *u* *?*σ' **by** *auto*
from ‹*validate-ground-lit* *I* l'› **and** ‹l' = (*subst-lit* *l* *?*σ')› **have**
validate-ground-lit *I* (*subst-lit* *l* *?*σ') **by** *auto*

from *this* **and** ‹*fo-interpretation* *I*› **and** ‹*l* = *L2*› **and** ‹*orient-lit-inst* *L2* *u* *v* *pos* σ›
have *I* (*subst* *u* *?*σ') (*subst* *v* *?*σ') **using** *orient-lit-semantics-pos*
by *blast*

from *this* **and** ‹*fo-interpretation* *I*› **and** ‹*I* (*subst* *s* *?*σ') (*subst* *v* *?*σ')›
have *I* (*subst* *u* *?*σ') (*subst* *s* *?*σ')
unfolding *fo-interpretation-def* *congruence-def* *equivalence-relation-def*
symmetric-def *transitive-def* **by** *blast*
from *this* **and** ‹(*subst* *t* *?*σ') = *subst* *u* *?*σ'›
have *I* (*subst* *t* *?*σ') (*subst* *s* *?*σ') **by** *auto*
from *this* **have** *validate-ground-eq* *I* (*subst-equation* (*Eq* *t* *s*) *?*σ')
by *auto*

from ‹*I* (*subst* *t* *?*σ') (*subst* *s* *?*σ')› **and** ‹*fo-interpretation* *I*›
have *I* (*subst* *s* *?*σ') (*subst* *t* *?*σ')
unfolding *fo-interpretation-def* *congruence-def* *equivalence-relation-def*
symmetric-def **by** *auto*
from *this* **have** *validate-ground-eq* *I* (*subst-equation* (*Eq* *s* *t*) *?*σ')
by *auto*

from ‹*orient-lit-inst* *L1* *t* *s* *pos* σ› **have** *L1* = (*Pos* (*Eq* *t* *s*)) $\lor$ *L1* = (*Pos* (*Eq* *s* *t*))
unfolding *orient-lit-inst-def* **by** *auto*

from *this* **and** ‹*validate-ground-eq I* (*subst-equation* (*Eq s t*) *?σ′*)› **and**
‹*validate-ground-eq I* (*subst-equation* (*Eq t s*) *?σ′*)›
have *validate-ground-lit I* (*subst-lit L1 ?σ′*)
by *auto*
from ‹*L1* ∈ *?C*› **and** ‹*?D* = (*subst-cl* ((*?C* − { *L2* }) ∪ { *L′* })) *σ*› **and**
‹*L1* ≠ *L2*›
have (*subst-lit L1 σ*) ∈ *?D*
by *auto*
then have
(*subst-lit* (*subst-lit* (*subst-lit L1 σ*) *ϑ*) *η*)
∈ (*subst-cl* (*subst-cl ?D ϑ*) *η*) **by** *auto*
then have (*subst-lit L1 ?σ′*) ∈ (*subst-cl* (*subst-cl ?D ϑ*) *η*)
using *composition-of-substs-lit* **by** *metis*
from *this* **and** ‹*validate-ground-lit I* (*subst-lit L1 ?σ′*)› **and**
‹¬ *validate-ground-clause I* (*subst-cl* (*subst-cl ?D ϑ*) *η*)›
show *False* **by** *auto*
qed

from ‹*subst-lit l σ* ∈ *?D*› **and** ‹*l′* = *subst-lit l ?σ′*›
have *l′* ∈ (*subst-cl* (*subst-cl ?D ϑ*) *η*)
using *subst-cl.simps composition-of-substs-lit mem-Collect-eq*
by (*metis* (*mono-tags*, *lifting*))
from *this* **and** ‹*validate-ground-lit I l′*› **have**
validate-ground-clause I (*subst-cl* (*subst-cl ?D ϑ*) *η*) **by** *auto*
from *this* **and** ‹¬ *validate-ground-clause I* (*subst-cl* (*subst-cl ?D ϑ*) *η*)›
show *False* **by** *blast*
qed

lemma *factorization-is-sound* :
assumes *finite* (*cl-ecl C*)
assumes *factorization C D σ k C′*
shows *clause-entails-clause* (*cl-ecl C*) (*cl-ecl D*)
proof (*rule ccontr*)
let *?C* = (*cl-ecl C*)
let *?D* = (*cl-ecl D*)
assume ¬ (*clause-entails-clause ?C ?D*)
then obtain *I* **where** *validate-clause I ?C* **and** ¬ (*validate-clause I ?D*) *fo-interpretation*
I
unfolding *clause-entails-clause-def* **by** *auto*
from ‹¬ (*validate-clause I ?D*)› **obtain** *ϑ*
where *D-false*: ¬ (*validate-ground-clause I* (*subst-cl ?D ϑ*))
and (*ground-clause* (*subst-cl ?D ϑ*)) **by** *auto*
have *validate-clause I* (*subst-cl* (*subst-cl ?C σ*) *ϑ*)
using ‹*validate-clause I* (*cl-ecl C*)› *instances-are-entailed* **by** *blast*
from *this* **and** *assms*(*1*) **and** *assms*(*2*) **have** *validate-clause I* (*subst-cl ?D ϑ*)
using *ground-factorization-is-sound* **unfolding** *clause-entails-clause-def*
using ‹*fo-interpretation I*› ‹*ground-clause* (*subst-cl* (*cl-ecl D*) *ϑ*)› **by** *blast*
from *this* **and** *D-false* **show** *False*
by (*metis* ‹*ground-clause* (*subst-cl* (*cl-ecl D*) *ϑ*)›

substs-preserve-ground-clause validate-clause.elims(*1*))
qed

lemma *ground-superposition-is-sound* :
assumes *finite* (*cl-ecl P1*)
assumes *finite* (*cl-ecl P2*)
assumes *superposition P1 P2 C* σ *k C*$'$
assumes (*ground-clause* (*subst-cl* (*cl-ecl C*) ϑ))
shows *set-entails-clause*
{ (*subst-cl* (*subst-cl* (*cl-ecl P1*) σ) ϑ),
(*subst-cl* (*subst-cl* (*cl-ecl P2*) σ) ϑ) }
(*subst-cl* (*cl-ecl C*) ϑ)

proof (*rule ccontr*)
let *?P1* = (*cl-ecl P1*)
let *?P2* = (*cl-ecl P2*)
let *?C* = (*cl-ecl C*)
assume ¬ *set-entails-clause*
{ (*subst-cl* (*subst-cl* (*cl-ecl P1*) σ) ϑ),
(*subst-cl* (*subst-cl* (*cl-ecl P2*) σ) ϑ) }
(*subst-cl* (*cl-ecl C*) ϑ)
then obtain *I*
where *validate-clause I* (*subst-cl* (*subst-cl* (*cl-ecl P1*) σ) ϑ)
and *validate-clause I* (*subst-cl* (*subst-cl* (*cl-ecl P2*) σ) ϑ)
and ¬ (*validate-clause I* (*subst-cl* (*cl-ecl C*) ϑ)) **and** *fo-interpretation I*
unfolding *set-entails-clause-def* **by** (*meson insert-iff validate-clause-set.elims*(*2*))
from *assms*(*3*) **obtain** *t s u v M p polarity t*$'$ *u*$'$ *L L*$'$ **where**
orient-lit-inst M u v pos σ
and *orient-lit-inst L t s polarity* σ
and *subterm t p u*$'$
and *ck-unifier u*$'$ *u* σ *k*
and *replace-subterm t p v t*$'$
and *L*$'$ = *mk-lit polarity* (*Eq t*$'$ *s*)
and *?C* = (*subst-cl* ((*?P1* − { *L* }) ∪ ((*?P2* − { *M* }) ∪ { *L*$'$ })) σ)
using *superposition-def* **by** *auto*

let *?P1*$'$ = (*subst-cl* (*subst-cl ?P1* σ) ϑ)
let *?P2*$'$ = (*subst-cl* (*subst-cl ?P2* σ) ϑ)
from *assms*(*1*) **have** *finite ?P1*$'$ **by** *simp*
from *assms*(*2*) **have** *finite ?P2*$'$ **by** *simp*

let *?vars* = (*vars-of-cl ?P1*$'$) ∪ (*vars-of-cl ?P2*$'$)
from ‹*finite ?P1*$'$› **have** *finite* (*vars-of-cl ?P1*$'$)
using *set-of-variables-is-finite-cl* [*of ?P1*$'$] **by** *auto*
from ‹*finite ?P2*$'$› **have** *finite* (*vars-of-cl ?P2*$'$)
using *set-of-variables-is-finite-cl* [*of ?P2*$'$] **by** *auto*
from ‹*finite* (*vars-of-cl ?P1*$'$)› **and** ‹*finite* (*vars-of-cl ?P2*$'$)› **have** *finite ?vars* **by** *auto*
then obtain η **where** *ground-on ?vars* η **using** *ground-subst-exists* **by** *blast*

then have *ground-on* (*vars-of-cl ?P1′*) η **unfolding** *ground-on-def* **by** *auto*
then have *ground-clause* (*subst-cl*
(*subst-cl* (*subst-cl ?P1* σ) ϑ) η)
using *ground-substs-yield-ground-clause*
[*of* (*subst-cl* (*subst-cl ?P1* σ) ϑ) η] **by** *auto*
from ‹*ground-on ?vars* η› **have** *ground-on* (*vars-of-cl ?P2′*) η **unfolding** *ground-on-def* **by** *auto*
then have *ground-clause* (*subst-cl*
(*subst-cl* (*subst-cl ?P2* σ) ϑ) η)
using *ground-substs-yield-ground-clause*
[*of* (*subst-cl* (*subst-cl ?P2* σ) ϑ) η] **by** *auto*

let *?P1″* = (*subst-cl ?P1′* η)
let *?P2″* = (*subst-cl ?P2′* η)
let *?*σ'' = *comp* σ ϑ
let *?*σ' = *comp ?*σ'' η
have *?P1″* = (*subst-cl* (*subst-cl ?P1 ?*σ'') η)
using *composition-of-substs-cl* [*of ?P1*] **by** *auto*
then have *?P1″* = (*subst-cl ?P1 ?*σ')
using *composition-of-substs-cl* [*of ?P1*] **by** *auto*
from ‹*ground-clause* (*subst-cl* (*subst-cl* (*subst-cl* (*cl-ecl P1*) σ) ϑ) η)›
and ‹*validate-clause I* (*subst-cl* (*subst-cl* (*cl-ecl P1*) σ) ϑ)›
have *validate-ground-clause I ?P1″* **using** *validate-clause.simps* **by** *blast*
then obtain *l1′* **where** *l1′* $\in$ *?P1″* **and** *validate-ground-lit I l1′* **by** *auto*

have *?P2″* = (*subst-cl* (*subst-cl ?P2 ?*σ'') η)
using *composition-of-substs-cl* [*of ?P2*] **by** *auto*
then have *?P2″* = (*subst-cl ?P2 ?*σ')
using *composition-of-substs-cl* [*of ?P2*] **by** *auto*
from ‹*ground-clause* (*subst-cl* (*subst-cl* (*subst-cl* (*cl-ecl P2*) σ) ϑ) η)› ‹*validate-clause I* (*subst-cl* (*subst-cl* (*cl-ecl P2*) σ) ϑ)›
have *validate-ground-clause I ?P2″* **using** *validate-clause.simps* **by** *blast*
then obtain *l2′* **where** *l2′* $\in$ *?P2″* **and** *validate-ground-lit I l2′* **by** *auto*

from ‹*l1′* $\in$ *?P1″*› **and** ‹*?P1″* = (*subst-cl ?P1 ?*σ')› **obtain** *l1* **where**
l1 $\in$ *?P1* **and** *l1′* = (*subst-lit l1 ?*σ') **using** *subst-cl.simps* **by** *blast*
from ‹*l2′* $\in$ *?P2″*› **and** ‹*?P2″* = (*subst-cl ?P2 ?*σ')› **obtain** *l2* **where**
l2 $\in$ *?P2* **and** *l2′* = (*subst-lit l2 ?*σ') **using** *subst-cl.simps* **by** *blast*

let *?C′* = (*subst-cl* (*subst-cl ?C* ϑ) η)

from ‹*ground-clause* (*subst-cl ?C* ϑ)› **have**
(*subst-cl ?C* ϑ) = (*subst-cl* (*subst-cl ?C* ϑ) η)
using *substs-preserve-ground-clause* [*of* (*subst-cl ?C* ϑ) η] **by** *blast*
from ‹$\neg$ *validate-clause I* (*subst-cl* (*cl-ecl C*) ϑ)›
have $\neg$ *validate-ground-clause I ?C′*
by (*metis assms*(*4*) *substs-preserve-ground-clause validate-clause.simps*)
have *l1* = *L*
proof (*rule ccontr*)

assume $l1 \neq L$
from *this* **and** ‹$l1 \in ?P1$› **and** ‹$?C = (subst\text{-}cl\ ((?P1 - \{ L \}) \cup ((?P2 - \{ M \}) \cup \{ L' \}))\ \sigma)$›
have $(subst\text{-}lit\ l1\ \sigma) \in ?C$ **by** *auto*
from *this* **have** $(subst\text{-}lit\ (subst\text{-}lit\ (subst\text{-}lit\ l1\ \sigma)\ \vartheta)\ \eta)$
$\in ?C'$ **by** *auto*
from *this* **and** ‹$l1' = (subst\text{-}lit\ l1\ ?\sigma')$› **have** $l1' \in ?C'$
by (*simp add*: *composition-of-substs-lit*)
from *this* **and** ‹*validate-ground-lit I l1'*› **have** *validate-ground-clause I ?C'* **by** *auto*
from *this* **and** ‹$\neg$ *validate-ground-clause I* (*subst-cl* (*subst-cl* (*cl-ecl C*) ϑ) η)›
show *False* **by** *auto*
qed

have $l2 = M$
proof (*rule ccontr*)
assume $l2 \neq M$
from *this* **and** ‹$l2 \in ?P2$› **and** ‹$?C = (subst\text{-}cl\ ((?P1 - \{ L \}) \cup ((?P2 - \{ M \}) \cup \{ L' \}))\ \sigma)$›
have $(subst\text{-}lit\ l2\ \sigma) \in ?C$ **by** *auto*
from *this* **have** $(subst\text{-}lit\ (subst\text{-}lit\ (subst\text{-}lit\ l2\ \sigma)\ \vartheta)\ \eta)$
$\in ?C'$ **by** *auto*
from *this* **and** ‹$l2' = (subst\text{-}lit\ l2\ ?\sigma')$› **have** $l2' \in ?C'$
by (*simp add*: *composition-of-substs-lit*)
from *this* **and** ‹*validate-ground-lit I l2'*› **have** *validate-ground-clause I ?C'* **by** *auto*
from *this* **and** ‹$\neg$ *validate-ground-clause I* (*subst-cl* (*subst-cl* (*cl-ecl C*) ϑ) η)›
show *False* **by** *auto*
qed

from ‹*orient-lit-inst M u v pos* σ› **and** ‹$l2 = M$› **and** ‹*fo-interpretation I*›
and ‹*validate-ground-lit I l2'*› **and** ‹$l2' = (subst\text{-}lit\ l2\ ?\sigma')$›
have $I\ (subst\ u\ ?\sigma')\ (subst\ v\ ?\sigma')$
using *orient-lit-semantics-pos* **by** *blast*

from ‹*subterm t p u'*› **have**
$subterm\ (subst\ t\ ?\sigma')\ p\ (subst\ u'\ ?\sigma')$
using *substs-preserve-subterms* [*of t p u'*] **by** *metis*

from ‹*ck-unifier u' u* σ *k*› **have** $(subst\ u\ \sigma) = (subst\ u'\ \sigma)$
using *ck-unifier-thm* [*of u' u* σ *k*] **by** *auto*
from *this* **have** $(subst\ (subst\ (subst\ u\ \sigma)\ \vartheta)\ \eta)$
$= (subst\ (subst\ (subst\ u'\ \sigma)\ \vartheta\ )\ \eta)$ **by** *auto*
from *this* **have** $(subst\ u\ ?\sigma') = (subst\ u'\ ?\sigma')$
using *composition-of-substs* **by** *auto*

from ‹$(subst\ u\ ?\sigma') = (subst\ u'\ ?\sigma')$›
and ‹$I\ (subst\ u\ ?\sigma')\ (subst\ v\ ?\sigma')$›
have $I\ (subst\ u'\ ?\sigma')\ (subst\ v\ ?\sigma')$

by *auto*
from ‹*subterm t p u′*›
and ‹*I* (*subst u′ ?σ′*) (*subst v ?σ′*)›
and ‹*fo-interpretation I*›
and ‹*replace-subterm t p v t′*›
have *I* (*subst t ?σ′*) (*subst t′ ?σ′*)
unfolding *fo-interpretation-def* **using** *replacement-preserves-congruences* [*of I u′ ?σ′ v t p t′*]
by *auto*

from ‹*l1* = *L*› **and** ‹*fo-interpretation I*› **and** ‹*validate-ground-lit I l1′*›
and ‹*l1′* = (*subst-lit l1 ?σ′*)›
and ‹*orient-lit-inst L t s polarity σ*›
and ‹*I* (*subst t ?σ′*) (*subst t′ ?σ′*)›
and ‹*L′* = *mk-lit polarity* (*Eq t′ s*)›
have *validate-ground-lit I* (*subst-lit L′ ?σ′*)
using *orient-lit-semantics-replacement* [*of I L t s polarity σ ?σ′ t′*] **by** *blast*

from ‹*?C* = (*subst-cl* ((*?P1* − { *L* }) ∪ ((*?P2* − { *M* }) ∪ { *L′* })) *σ*)›
have *subst-lit L′ σ* ∈ *?C* **by** *auto*
then have *subst-lit* (*subst-lit* (*subst-lit L′ σ*) *ϑ*) *η* ∈ *?C′*
by *auto*
then have *subst-lit L′ ?σ′* ∈ *?C′* **by** (*simp add*: *composition-of-substs-lit*)

from *this* **and** ‹*validate-ground-lit I* (*subst-lit L′ ?σ′*)› **and** ‹¬*validate-ground-clause I ?C′*›
show *False* **by** *auto*
qed

lemma *superposition-is-sound* :
assumes *finite* (*cl-ecl P1*)
assumes *finite* (*cl-ecl P2*)
assumes *superposition P1 P2 C σ k C′*
shows *set-entails-clause* { *cl-ecl P1*, *cl-ecl P2* } (*cl-ecl C*)
proof (*rule ccontr*)
let *?P1* = (*cl-ecl P1*)
let *?P2* = (*cl-ecl P2*)
let *?C* = (*cl-ecl C*)
assume ¬ *set-entails-clause* { *cl-ecl P1*, *cl-ecl P2* } (*cl-ecl C*)
then obtain *I*
where *validate-clause I ?P1* **and** *validate-clause I ?P2*
and ¬ (*validate-clause I ?C*) **and** *fo-interpretation I*
unfolding *set-entails-clause-def* **by** (*meson insert-iff validate-clause-set.elims*(*2*))
from ‹¬ (*validate-clause I ?C*)› **obtain** *ϑ*
where ¬ (*validate-ground-clause I* (*subst-cl ?C ϑ*))
and (*ground-clause* (*subst-cl ?C ϑ*)) **by** *auto*

have *P1-true*: *validate-clause I* (*subst-cl* (*subst-cl ?P1 σ*) *ϑ*)
using ‹*validate-clause I* (*cl-ecl P1*)› *instances-are-entailed* **by** *blast*

have *P2-true*: *validate-clause I* (*subst-cl* (*subst-cl ?P2* σ) ϑ)
using ‹*validate-clause I* (*cl-ecl P2*)› *instances-are-entailed* **by** *blast*
have ¬ (*validate-clause I* (*subst-cl ?C* ϑ))
by (*metis* ‹¬ *validate-ground-clause I* (*subst-cl* (*cl-ecl C*) ϑ)›
‹*ground-clause* (*subst-cl* (*cl-ecl C*) ϑ)›
substs-preserve-ground-clause validate-clause.elims(*1*))
let *?S* = { (*subst-cl* (*subst-cl* (*cl-ecl P1*) σ) ϑ),
(*subst-cl* (*subst-cl* (*cl-ecl P2*) σ) ϑ) }
from *P1-true* **and** *P2-true* **have** *validate-clause-set I ?S*
by (*metis insert-iff singletonD validate-clause-set.elims*(*3*))
from *this* **and** ‹¬ (*validate-clause I* (*subst-cl ?C* ϑ))› ‹*fo-interpretation I*›
have ¬ *set-entails-clause ?S* (*subst-cl* (*cl-ecl C*) ϑ)
using *set-entails-clause-def* **by** *blast*
from *this* **and** *assms*(*1*) **and** *assms*(*2*) **and** *assms*(*3*) **and**
‹(*ground-clause* (*subst-cl ?C* ϑ))›
show *False* **using** *ground-superposition-is-sound* **by** *auto*
qed

lemma *superposition-preserves-finiteness*:
assumes *finite* (*cl-ecl P1*)
assumes *finite* (*cl-ecl P2*)
assumes *superposition P1 P2 C σ k C′*
shows *finite* (*cl-ecl C*) ∧ (*finite C′*)
proof −
from *assms*(*3*) **obtain** *L M L′* **where**
def-C: (*cl-ecl C*) = (*subst-cl* (((*cl-ecl P1*) − { *L* }) ∪ (((*cl-ecl P2*) − { *M* }) ∪ { *L′* })) σ)
and *def-C′*: *C′* = (((*cl-ecl P1*) − { *L* }) ∪ (((*cl-ecl P2*) − { *M* }) ∪ { *L′* }))
using *superposition-def* **by** *auto*
from *assms*(*1*) **and** *assms*(*2*) **have** *finite* (((*cl-ecl P1*) − { *L* }) ∪ (((*cl-ecl P2*) − { *M* }) ∪ { *L′* }))
by *auto*
from *this* **and** *def-C def-C′* **show** *?thesis* **using** *substs-preserve-finiteness* **by** *auto*
qed

lemma *reflexion-preserves-finiteness*:
assumes *finite* (*cl-ecl P1*)
assumes *reflexion P1 C σ k C′*
shows *finite* (*cl-ecl C*) ∧ (*finite C′*)
proof −
from *assms*(*2*) **obtain** *L1* **where**
def-C: (*cl-ecl C*) = (*subst-cl* ((*cl-ecl P1*) − { *L1* }) σ)
and *def-C′*: *C′* = ((*cl-ecl P1*) − { *L1* })
using *reflexion-def* **by** *auto*
from *assms*(*1*) **have** *finite* ((*cl-ecl P1*) − { *L1* }) **by** *auto*
from *this* **and** *def-C def-C′* **show** *?thesis* **using** *substs-preserve-finiteness* **by** *auto*
qed

lemma *factorization-preserves-finiteness*:
assumes *finite* (*cl-ecl P1*)
assumes *factorization P1 C* σ *k* C'
shows *finite* (*cl-ecl C*) $\wedge$ (*finite* C')
proof −
from *assms*(*2*) **obtain** *L2* L' **where**
def-C: (*cl-ecl C*) = (*subst-cl* (((*cl-ecl P1*) − { *L2* }) $\cup$ { L' }) σ)
and *def-*C': C' = (((*cl-ecl P1*) − { *L2* }) $\cup$ { L' })
using *factorization-def* **by** *auto*
from *assms*(*1*) **have** (*finite* (((*cl-ecl P1*) − { *L2* }) $\cup$ { L' })) **by** *auto*
from *this* **and** *def-C* *def-*C' **show** *?thesis* **using** *substs-preserve-finiteness* **by** *auto*
qed

lemma *derivable-clauses-are-finite*:
assumes *derivable C P S* σ *k* C'
assumes $\forall$ *x* $\in$ *P*. (*finite* (*cl-ecl x*))
shows *finite* (*cl-ecl C*) $\wedge$ (*finite* C')
proof (*rule ccontr*)
assume *hyp*: $\neg$ (*finite* (*cl-ecl C*) $\wedge$ (*finite* C'))
have *not-sup*: $\neg$ ($\exists$ *P1 P2*. (*P1* $\in$ *P* $\wedge$ *P2* $\in$ *P* $\wedge$ *superposition P1 P2 C* σ *k* C'))
proof
assume ($\exists$ *P1 P2*. (*P1* $\in$ *P* $\wedge$ *P2* $\in$ *P* $\wedge$ *superposition P1 P2 C* σ *k* C'))
then obtain *P1 P2* **where** *P1* $\in$ *P* *P2* $\in$ *P* *superposition P1 P2 C* σ *k* C' **by** *auto*
from ‹*P1* $\in$ *P*› **and** *assms*(*2*) **have** *finite* (*cl-ecl P1*) **by** *auto*
from ‹*P2* $\in$ *P*› **and** *assms*(*2*) **have** *finite* (*cl-ecl P2*) **by** *auto*
from ‹(*finite* (*cl-ecl P1*))› **and** ‹(*finite* (*cl-ecl P2*))› **and** ‹*superposition P1 P2 C* σ *k* C'›
have *finite* (*cl-ecl C*) $\wedge$ (*finite* C') **using** *superposition-preserves-finiteness* [*of P1 P2 C* σ] **by** *auto*
then show *False* **using** *hyp* **by** *auto*
qed
have *not-ref*: $\neg$ ($\exists$ *P1*. (*P1* $\in$ *P* $\wedge$ *reflexion P1 C* σ *k* C'))
proof
assume ($\exists$ *P1*. (*P1* $\in$ *P* $\wedge$ *reflexion P1 C* σ *k* C'))
then obtain *P1* **where** *P1* $\in$ *P* *reflexion P1 C* σ *k* C' **by** *auto*
from ‹*P1* $\in$ *P*› **and** *assms*(*2*) **have** *finite* (*cl-ecl P1*) **by** *auto*
from ‹(*finite* (*cl-ecl P1*))› **and** ‹*reflexion P1 C* σ *k* C'›
have *finite* (*cl-ecl C*) $\wedge$ (*finite* C') **using** *reflexion-preserves-finiteness* [*of P1 C* σ] **by** *auto*
then show *False* **using** *hyp* **by** *auto*
qed
have *not-fact*: $\neg$ ($\exists$ *P1*. (*P1* $\in$ *P* $\wedge$ *factorization P1 C* σ *k* C'))
proof
assume ($\exists$ *P1*. (*P1* $\in$ *P* $\wedge$ *factorization P1 C* σ *k* C'))
then obtain *P1* **where** *P1* $\in$ *P* *factorization P1 C* σ *k* C' **by** *auto*
from ‹*P1* $\in$ *P*› **and** *assms*(*2*) **have** *finite* (*cl-ecl P1*) **by** *auto*

from ‹(*finite* (*cl-ecl P1*))› **and** ‹ *factorization P1 C σ k C′*›
have *finite* (*cl-ecl C*) ∧ (*finite C′*) **using** *factorization-preserves-finiteness* [*of P1 C σ*] **by** *auto*
then show *False* **using** *hyp* **by** *auto*
qed
from *not-sup not-ref not-fact* **and** *assms*(*1*) **show** *False* **unfolding** *derivable-def* **by** *blast*
qed

lemma *derivable-clauses-lemma*:
assumes *derivable C P S σ k C′*
shows ((*cl-ecl C*) = (*subst-cl C′ σ*))
proof (*rule ccontr*)
assume *hyp*: ¬ ((*cl-ecl C*) = (*subst-cl C′ σ*))
have *not-sup*: ¬ (∃ *P1 P2*. (*P1* ∈ *S* ∧ *P2* ∈ *S* ∧ *superposition P1 P2 C σ k C′*))
proof
assume (∃ *P1 P2*. (*P1* ∈ *S* ∧ *P2* ∈ *S* ∧ *superposition P1 P2 C σ k C′*))
then obtain *P1 P2* **where** *P1* ∈ *S P2* ∈ *S superposition P1 P2 C σ k C′* **by** *auto*
from ‹*superposition P1 P2 C σ k C′*› **obtain** *Cl-C Cl-P1 L Cl-P2 M L′ T*
where *Cl-C* = (*subst-cl* ((*Cl-P1* − { *L* }) ∪ ((*Cl-P2* − { *M* }) ∪ { *L′* })) *σ*)
(*C′* = (*Cl-P1* − { *L* }) ∪ ((*Cl-P2* − { *M* }) ∪ { *L′* }))
C = (*Ecl Cl-C T*)
unfolding *superposition-def* **by** *blast*
from ‹*Cl-C* = (*subst-cl* ((*Cl-P1* − { *L* }) ∪ ((*Cl-P2* − { *M* }) ∪ { *L′* })) *σ*)›
‹(*C′* = (*Cl-P1* − { *L* }) ∪ ((*Cl-P2* − { *M* }) ∪ { *L′* }))› ‹*C* = (*Ecl Cl-C T*)› *hyp* **show** *False* **by** *auto*
qed
have *not-ref*: ¬ (∃ *P1*. (*P1* ∈ *S* ∧ *reflexion P1 C σ k C′*))
proof
assume (∃ *P1*. (*P1* ∈ *S* ∧ *reflexion P1 C σ k C′*))
then obtain *P1* **where** *P1* ∈ *S reflexion P1 C σ k C′* **by** *auto*
from ‹*reflexion P1 C σ k C′*› **obtain** *T Cl-C Cl-P L1* **where**
C = (*Ecl Cl-C T*)
Cl-C = (*subst-cl* ((*Cl-P* − { *L1* }))) *σ*
(*C′* = ((*Cl-P* − { *L1* }))) **unfolding** *reflexion-def* **by** *blast*
from ‹*Cl-C* = (*subst-cl* ((*Cl-P* − { *L1* }))) *σ*›
‹(*C′* = ((*Cl-P* − { *L1* })))› ‹*C* = (*Ecl Cl-C T*)› *hyp* **show** *False* **by** *auto*
qed
have *not-fact*: ¬ (∃ *P1*. (*P1* ∈ *S* ∧ *factorization P1 C σ k C′*))
proof
assume (∃ *P1*. (*P1* ∈ *S* ∧ *factorization P1 C σ k C′*))
then obtain *P1* **where** *P1* ∈ *S factorization P1 C σ k C′* **by** *auto*
from ‹*factorization P1 C σ k C′*› **obtain** *T Cl-C Cl-P L′ L2* **where**
C = (*Ecl Cl-C T*)
Cl-C = (*subst-cl* ((*Cl-P* − { *L2* }) ∪ { *L′* })) *σ*
C′ = ((*Cl-P* − { *L2* }) ∪ { *L′* }) **unfolding** *factorization-def* **by** *blast*
from ‹*Cl-C* = (*subst-cl* ((*Cl-P* − { *L2* }) ∪ { *L′* })) *σ*›
‹*C′* = ((*Cl-P* − { *L2* }) ∪ { *L′* })› ‹*C* = (*Ecl Cl-C T*)› *hyp* **show** *False* **by**

auto
qed
from *not-sup not-ref not-fact* **and** *assms*(*1*) **show** *False* **unfolding** *derivable-def*
by *blast*
qed

lemma *substs-preserves-decompose-literal*:
assumes *decompose-literal L t s polarity*
shows *decompose-literal* (*subst-lit L* η) (*subst t* η) (*subst s* η) *polarity*
proof −
let *?L* = (*subst-lit L* η)
let *?t* = (*subst t* η)
let *?s* = (*subst s* η)

have *polarity* = *pos* ∨ *polarity* = *neg* **using** *sign.exhaust* **by** *auto*
then show *?thesis*
proof
assume *polarity* = *pos*
from *this* **and** *assms*(*1*) **have** *L* = *Pos* (*Eq t s*) ∨ *L* = *Pos* (*Eq s t*)
unfolding *decompose-literal-def decompose-equation-def* **by** *auto*
from ‹*L* = *Pos* (*Eq t s*) ∨ *L* = *Pos* (*Eq s t*)›
have *?L* = *Pos* (*Eq ?t ?s*) ∨ *?L* = *Pos* (*Eq ?s ?t*) **by** *auto*
from *this* ‹*polarity* = *pos*› **show** *?thesis* **unfolding** *decompose-literal-def*
decompose-equation-def **by** *auto*
next
assume *polarity* = *neg*
from *this* **and** *assms*(*1*) **have** *L* = *Neg* (*Eq t s*) ∨ *L* = *Neg* (*Eq s t*)
unfolding *decompose-literal-def decompose-equation-def* **by** *auto*
from *this* ‹*polarity* = *neg*› **show** *?thesis* **unfolding** *decompose-literal-def*
decompose-equation-def **by** *auto*
qed
qed

lemma *substs-preserve-dom-trm*:
assumes *dom-trm t C*
shows *dom-trm* (*subst t* σ) (*subst-cl C* σ)
proof −
let *?t* = (*subst t* σ)
from *assms*(*1*) **have** (∃ *L u v p*. (*L* ∈ *C* ∧ (*decompose-literal L u v p*)
∧ (((*p* = *neg* ∧ *t* = *u*) ∨ (*t*,*u*) ∈ *trm-ord*)))) **unfolding** *dom-trm-def* **by**
auto
from *this* **obtain** *L u v p* **where** *L* ∈ *C*
decompose-literal L u v p (((*p* = *neg* ∧ *t* = *u*) ∨ (*t*,*u*) ∈ *trm-ord*))
unfolding *dom-trm-def* **by** *blast*
let *?u* = (*subst u* σ)

from ‹*L* ∈ *C*› **have** (*subst-lit L* σ) ∈ (*subst-cl C* σ) **by** *auto*
from ‹*decompose-literal L u v p*›
have *decompose-literal* (*subst-lit L* σ) (*subst u* σ) (*subst v* σ) *p*

using *substs-preserves-decompose-literal* **by** *metis*
from ‹(((*p* = *neg* ∧ *t* = *u*) ∨ (*t*,*u*) ∈ *trm-ord*))›
have (((*p* = *neg* ∧ *?t* = *?u*) ∨ (*?t*,*?u*) ∈ *trm-ord*))
using *trm-ord-subst* **by** *auto*
from *this* ‹(*subst-lit L σ*) ∈ (*subst-cl C σ*)›
‹*decompose-literal* (*subst-lit L σ*) (*subst u σ*) (*subst v σ*) *p*›
show *dom-trm* (*subst t σ*) (*subst-cl C σ*)
unfolding *dom-trm-def* **by** *auto*
qed

lemma *substs-preserve-well-constrainedness*:
assumes *well-constrained C*
shows *well-constrained* (*subst-ecl C σ*)
proof (*rule ccontr*)
assume ¬*?thesis*
from *this* **obtain** *y* **where** *y* ∈ *trms-ecl* (*subst-ecl C σ*)
and ¬ *dom-trm y* (*cl-ecl* (*subst-ecl C σ*)) **unfolding** *well-constrained-def* **by** *auto*
obtain *Cl-C T* **where** *C* = (*Ecl Cl-C T*) **using** *eclause.exhaust* **by** *auto*
from *this* **have** (*subst-ecl C σ*)
= (*Ecl* (*subst-cl Cl-C σ*) (*subst-set T σ*)) **by** *auto*
from *this* **have** (*cl-ecl* (*subst-ecl C σ*) = (*subst-cl Cl-C σ*))
and *trms-ecl* (*subst-ecl C σ*) = (*subst-set T σ*)
by *auto*
from ‹(*cl-ecl* (*subst-ecl C σ*) = (*subst-cl Cl-C σ*))›
‹*C* = (*Ecl Cl-C T*)› **have** (*cl-ecl* (*subst-ecl C σ*) = (*subst-cl* (*cl-ecl C*) *σ*)) **by** *auto*
from ‹*y* ∈ *trms-ecl* (*subst-ecl C σ*)› ‹*C* = (*Ecl Cl-C T*)›
obtain *z* **where** *z* ∈ *T* **and** *y* = (*subst z σ*) **by** *auto*
from ‹*z* ∈ *T*› *assms(1)* ‹*C* = (*Ecl Cl-C T*)› **have** *dom-trm z* (*cl-ecl C*)
unfolding *well-constrained-def* **by** *auto*
from *this* **have** *dom-trm* (*subst z σ*) (*subst-cl* (*cl-ecl C*) *σ*)
using *substs-preserve-dom-trm* **by** *auto*
from *this* ‹*y* = (*subst z σ*)› **have** *dom-trm y* (*subst-cl* (*cl-ecl C*) *σ*)
by *auto*
from *this* ‹(*cl-ecl* (*subst-ecl C σ*) = (*subst-cl* (*cl-ecl C*) *σ*))›
‹¬ *dom-trm y* (*cl-ecl* (*subst-ecl C σ*))› **show** *False* **by** *auto*
qed

lemma *ck-trms-sound*:
assumes *T* = *get-trms D* (*dom-trms C E*) *k*
shows *T* ⊆ (*dom-trms C E*)
proof (*cases*)
assume *k* = *FirstOrder*
from *this* **and** *assms* **have** *T* = *filter-trms D* (*dom-trms C E*)
unfolding *get-trms-def* **by** *auto*
from *this* **show** *?thesis* **using** *filter-trms-inclusion* **by** *blast*
next
assume *k* ≠ *FirstOrder*

from *this* **and** *assms* **have** $T = (\mathit{dom\text{-}trms}\ C\ E)$
unfolding *get-trms-def* **by** *auto*
from *this* **show** *?thesis* **using** *filter-trms-inclusion* **by** *blast*
qed

lemma *derivable-clauses-are-well-constrained*:
assumes $\mathit{derivable}\ C\ P\ S\ \sigma\ k\ C'$
shows *well-constrained C*
proof (*rule ccontr*)
assume *hyp*: ¬ *well-constrained C*
then obtain *y* **where** $y \in \mathit{trms\text{-}ecl}\ C$ **and** ¬ *dom-trm y* (*cl-ecl C*)
unfolding *well-constrained-def* **by** *auto*
have *not-sup*: $\neg (\exists P1\ P2.\ (P1 \in S \wedge P2 \in S \wedge \mathit{superposition}\ P1\ P2\ C\ \sigma\ k\ C'))$
proof
assume $(\exists P1\ P2.\ (P1 \in S \wedge P2 \in S \wedge \mathit{superposition}\ P1\ P2\ C\ \sigma\ k\ C'))$
then obtain *P1 P2* **where** $P1 \in S$ $P2 \in S$ $\mathit{superposition}\ P1\ P2\ C\ \sigma\ k\ C'$ **by** *auto*
from ‹$\mathit{superposition}\ P1\ P2\ C\ \sigma\ k\ C'$› **obtain** *Cl-C T E*
where
$T = (\mathit{get\text{-}trms}\ \mathit{Cl\text{-}C}\ (\mathit{dom\text{-}trms}\ \mathit{Cl\text{-}C}\ (\mathit{subst\text{-}set}\ E\ \sigma))\ k)$
$\mathit{Cl\text{-}C} = (\mathit{subst\text{-}cl}\ C'\ \sigma)$
$C = (\mathit{Ecl}\ \mathit{Cl\text{-}C}\ T)$
unfolding *superposition-def* **by** *blast*
from ‹$T = (\mathit{get\text{-}trms}\ \mathit{Cl\text{-}C}\ (\mathit{dom\text{-}trms}\ \mathit{Cl\text{-}C}\ (\mathit{subst\text{-}set}\ E\ \sigma))\ k)$›
have $T \subseteq (\mathit{dom\text{-}trms}\ \mathit{Cl\text{-}C}\ (\mathit{subst\text{-}set}\ E\ \sigma))$
using *ck-trms-sound* **by** *metis*
from *this* **and** ‹$y \in \mathit{trms\text{-}ecl}\ C$› **and** ‹$C = (\mathit{Ecl}\ \mathit{Cl\text{-}C}\ T)$› **have**
$y \in (\mathit{dom\text{-}trms}\ (\mathit{cl\text{-}ecl}\ C)\ (\mathit{subst\text{-}set}\ E\ \sigma))$ **by** *auto*
from *this* **and** ‹¬ *dom-trm y* (*cl-ecl C*)› **show** *False* **unfolding** *dom-trms-def* **by** *auto*
qed
have *not-ref*: $\neg (\exists P1.\ (P1 \in S \wedge \mathit{reflexion}\ P1\ C\ \sigma\ k\ C'))$
proof
assume $(\exists P1.\ (P1 \in S \wedge \mathit{reflexion}\ P1\ C\ \sigma\ k\ C'))$
then obtain *P1* **where** $P1 \in S$ $\mathit{reflexion}\ P1\ C\ \sigma\ k\ C'$ **by** *auto*
from ‹$\mathit{reflexion}\ P1\ C\ \sigma\ k\ C'$› **obtain** *T Cl-C E* **where**
$T = (\mathit{get\text{-}trms}\ \mathit{Cl\text{-}C}\ (\mathit{dom\text{-}trms}\ \mathit{Cl\text{-}C}\ (\mathit{subst\text{-}set}\ E\ \sigma))\ k)$
$\mathit{Cl\text{-}C} = (\mathit{subst\text{-}cl}\ C'\ \sigma)$
$C = (\mathit{Ecl}\ \mathit{Cl\text{-}C}\ T)$
unfolding *reflexion-def* **by** *blast*
from ‹$T = (\mathit{get\text{-}trms}\ \mathit{Cl\text{-}C}\ (\mathit{dom\text{-}trms}\ \mathit{Cl\text{-}C}\ (\mathit{subst\text{-}set}\ E\ \sigma))\ k)$›
have $T \subseteq (\mathit{dom\text{-}trms}\ \mathit{Cl\text{-}C}\ (\mathit{subst\text{-}set}\ E\ \sigma))$
using *ck-trms-sound* **by** *metis*
from *this* **and** ‹$y \in \mathit{trms\text{-}ecl}\ C$› **and** ‹$C = (\mathit{Ecl}\ \mathit{Cl\text{-}C}\ T)$› **have**
$y \in (\mathit{dom\text{-}trms}\ (\mathit{cl\text{-}ecl}\ C)\ (\mathit{subst\text{-}set}\ E\ \sigma))$ **by** *auto*
from *this* **and** ‹¬ *dom-trm y* (*cl-ecl C*)› **show** *False* **unfolding** *dom-trms-def* **by** *auto*
qed
have *not-fact*: $\neg (\exists P1.\ (P1 \in S \wedge \mathit{factorization}\ P1\ C\ \sigma\ k\ C'))$

proof
assume (∃ *P1*. (*P1* ∈ *S* ∧ *factorization P1 C σ k C′*))
then obtain *P1* **where** *P1* ∈ *S factorization P1 C σ k C′* **by** *auto*
from ‹*factorization P1 C σ k C′*› **obtain** *T Cl-C E* **where**
T = (*get-trms Cl-C* (*dom-trms Cl-C* (*subst-set E σ*)) *k*)
Cl-C = (*subst-cl C′ σ*)
C = (*Ecl Cl-C T*)
unfolding *factorization-def* **by** *blast*
from ‹*T* = (*get-trms Cl-C* (*dom-trms Cl-C* (*subst-set E σ*)) *k*)›
have *T* ⊆(*dom-trms Cl-C* (*subst-set E σ*))
using *ck-trms-sound* **by** *metis*
from *this* **and** ‹*y* ∈ *trms-ecl C*› **and** ‹*C* = (*Ecl Cl-C T*)› **have**
y ∈ (*dom-trms* (*cl-ecl C*) (*subst-set E σ*)) **by** *auto*
from *this* **and** ‹¬ *dom-trm y* (*cl-ecl C*)› **show** *False* **unfolding** *dom-trms-def* **by** *auto*
qed
from *not-sup not-ref not-fact* **and** *assms*(*1*) **show** *False* **unfolding** *derivable-def* **by** *blast*
qed

lemma *derivable-clauses-are-entailed*:
assumes *derivable C P S σ k C′*
assumes *validate-clause-set I* (*cl-ecl* ‘ *P*)
assumes *fo-interpretation I*
assumes ∀ *x* ∈ *P*. (*finite* (*cl-ecl x*))
shows *validate-clause I* (*cl-ecl C*)
proof (*rule ccontr*)
assume ¬*validate-clause I* (*cl-ecl C*)
have *not-sup*: ¬ (∃ *P1 P2*. (*P1* ∈ *S* ∧ *P2* ∈ *S* ∧ *P* = { *P1*, *P2* } ∧ *superposition P1 P2 C σ k C′*))
proof
assume (∃ *P1 P2*. (*P1* ∈ *S* ∧ *P2* ∈ *S* ∧ *P* = { *P1*, *P2* } ∧ *superposition P1 P2 C σ k C′*))
from *this* **obtain** *P1 P2* **where** *P1* ∈ *P P2* ∈ *P* **and** *superposition P1 P2 C σ k C′* **by** *auto*
from ‹*P1* ∈ *P*› **and** *assms*(*2*) **have** *validate-clause I* (*cl-ecl P1*) **by** *auto*
from ‹*P2* ∈ *P*› **and** *assms*(*2*) **have** *validate-clause I* (*cl-ecl P2*) **by** *auto*
from *assms*(*4*) **and** ‹*P1* ∈ *P*› **have** *finite* (*cl-ecl P1*) **by** *auto*
from *assms*(*4*) **and** ‹*P2* ∈ *P*› **have** *finite* (*cl-ecl P2*) **by** *auto*
from *assms*(*3*) **and** ‹*finite* (*cl-ecl P1*)› **and** ‹*finite* (*cl-ecl P2*)›
and ‹*superposition P1 P2 C σ k C′*› **have** *set-entails-clause* { (*cl-ecl P1*), (*cl-ecl P2*) } (*cl-ecl C*)
using *superposition-is-sound* **by** *blast*
from *this* **and** *assms*(*3*) **and** ‹*validate-clause I* (*cl-ecl P1*)› **and** ‹*validate-clause I* (*cl-ecl P2*)›
have *validate-clause I* (*cl-ecl C*)
using *set-entails-clause-def* [*of* { (*cl-ecl P1*), (*cl-ecl P2*) } *cl-ecl C*] **by** *auto*
from *this* **and** ‹¬*validate-clause I* (*cl-ecl C*)› **show** *False* **by** *auto*
qed

have *not-fact*: ¬ (∃ *P1*. (*P1* ∈ *S* ∧ *P* = { *P1* } ∧ *factorization P1 C σ k C′*))
proof
assume (∃ *P1*. (*P1* ∈ *S* ∧ *P* = { *P1* } ∧ *factorization P1 C σ k C′*))
from *this* **obtain** *P1* **where** *P1* ∈ *P* **and** *factorization P1 C σ k C′* **by** *auto*
from ‹*P1* ∈ *P*› **and** *assms*(*2*) **have** *validate-clause I* (*cl-ecl P1*) **by** *auto*
from *assms*(*4*) **and** ‹*P1* ∈ *P*› **have** *finite* (*cl-ecl P1*) **by** *auto*
from *assms*(*3*) **and** ‹*finite* (*cl-ecl P1*)› **and**
‹*factorization P1 C σ k C′*› **have** *clause-entails-clause* (*cl-ecl P1*) (*cl-ecl C*)
using *factorization-is-sound* **by** *auto*
from *this* **and** *assms*(*3*) **and** ‹*validate-clause I* (*cl-ecl P1*)›
have *validate-clause I* (*cl-ecl C*) **unfolding** *clause-entails-clause-def* **by** *auto*
from *this* **and** ‹¬*validate-clause I* (*cl-ecl C*)› **show** *False* **by** *auto*
qed
have *not-ref*: ¬ (∃ *P1*. (*P1* ∈ *S* ∧ *P* = { *P1* } ∧ *reflexion P1 C σ k C′*))
proof
assume (∃ *P1*. (*P1* ∈ *S* ∧ *P* = { *P1* } ∧ *reflexion P1 C σ k C′*))
from *this* **obtain** *P1* **where** *P1* ∈ *P* **and** *reflexion P1 C σ k C′* **by** *auto*
from ‹*P1* ∈ *P*› **and** *assms*(*2*) **have** *validate-clause I* (*cl-ecl P1*) **by** *auto*
from *assms*(*4*) **and** ‹*P1* ∈ *P*› **have** *finite* (*cl-ecl P1*) **by** *auto*
from *assms*(*3*) **and** ‹*finite* (*cl-ecl P1*)› **and**
‹*reflexion P1 C σ k C′*› **have** *clause-entails-clause* (*cl-ecl P1*) (*cl-ecl C*)
using *reflexion-is-sound* **by** *auto*
from *this* **and** *assms*(*3*) **and** ‹*validate-clause I* (*cl-ecl P1*)›
have *validate-clause I* (*cl-ecl C*) **unfolding** *clause-entails-clause-def* **by** *auto*
from *this* **and** ‹¬*validate-clause I* (*cl-ecl C*)› **show** *False* **by** *auto*
qed
from *not-sup not-fact not-ref* **and** *assms*(*1*) **show** *False* **unfolding** *derivable-def* **by** *blast*
qed

lemma *all-derived-clauses-are-finite*:
shows *derivable-ecl C S* ⟹ ∀ *x* ∈ *S*. (*finite* (*cl-ecl x*)) ⟹ *finite* (*cl-ecl C*)
proof (*induction rule*: *derivable-ecl.induct*)
fix *C* :: *′a eclause* **fix** *S* **assume** *C* ∈ *S*
assume ∀ *x* ∈ *S*. (*finite* (*cl-ecl x*))
from *this* ‹*C* ∈ *S*› **show** *finite* (*cl-ecl C*) **by** *auto*
next
fix *C S* **fix** *D* :: *′a eclause* **assume** *derivable-ecl C S*
assume ∀ *x* ∈ *S*. (*finite* (*cl-ecl x*)) **assume** *hyp-ind*: ∀ *x* ∈ *S*. (*finite* (*cl-ecl x*)) ⟹ *finite* (*cl-ecl C*)
(*renaming-cl C D*)
from ‹(*renaming-cl C D*)› **obtain** *η* **where** *D* = (*subst-ecl C η*)
unfolding *renaming-cl-def* **by** *auto*
obtain *C-Cl T* **where** *C* = (*Ecl C-Cl T*) **using** *eclause.exhaust* **by** *auto*
from *this* **and** ‹*D* = (*subst-ecl C η*)›
have (*cl-ecl D*) = (*subst-cl* (*cl-ecl C*) *η*) **by** *auto*
from *this hyp-ind* ‹∀ *x* ∈ *S*. (*finite* (*cl-ecl x*))› **show** *finite* (*cl-ecl D*)
using *substs-preserve-finiteness* **by** *auto*
next

fix $P\ S\ C\ S'\ \sigma\ C'$
assume h: $\forall\, x.\ x \in P \longrightarrow$ *derivable-ecl* $x\ S \wedge ((\forall\, x{\in}S.$ *finite* (*cl-ecl* x)) $\longrightarrow$ *finite* (*cl-ecl* x))
assume *derivable* $C\ P\ S'\ \sigma$ *FirstOrder* C'
assume $\forall\, x{\in}S.$ *finite* (*cl-ecl* x)
from h **and** ‹$\forall\, x{\in}S.$ *finite* (*cl-ecl* x)› **have** $\forall\, x \in P.$ (*finite* (*cl-ecl* x)) **by** *metis*
from *this* **and** ‹*derivable* $C\ P\ S'\ \sigma$ *FirstOrder* C'› **show** *finite* (*cl-ecl* C)
using *derivable-clauses-are-finite* **by** *auto*
qed

lemma *all-derived-clauses-are-wellconstrained*:
shows *derivable-ecl* $C\ S \Longrightarrow \forall\, x \in S.$ (*well-constrained* x) $\Longrightarrow$ *well-constrained* C
proof (*induction rule*: *derivable-ecl.induct*)
fix C :: *'a eclause* **fix** S **assume** $C \in S$
assume $\forall\, x \in S.$ (*well-constrained* x)
from *this* ‹$C \in S$› **show** *well-constrained* C **by** *auto*
next
fix $C\ S$ **fix** D :: *'a eclause* **assume** *derivable-ecl* $C\ S$
assume $\forall\, x \in S.$ (*well-constrained* x) **assume** *hyp-ind*: $\forall\, x \in S.$ (*well-constrained* x) $\Longrightarrow$ *well-constrained* C
(*renaming-cl* $C\ D$)
from ‹$\forall\, x \in S.$ (*well-constrained* x)› **and** *hyp-ind* **have** *well-constrained* C **by** *auto*
from ‹(*renaming-cl* $C\ D$)› **obtain** η **where** $D =$ (*subst-ecl* $C\ \eta$)
unfolding *renaming-cl-def* **by** *auto*
from *this* **and** ‹*well-constrained* C› **show** *well-constrained* D
using *substs-preserve-well-constrainedness* **by** *auto*
next
fix $P\ S\ C\ S'\ \sigma\ C'$
assume $\forall\, x.\ x \in P \longrightarrow$ *derivable-ecl* $x\ S \wedge$ (*Ball* S *well-constrained* $\longrightarrow$ *well-constrained* x)
assume *derivable* $C\ P\ S'\ \sigma$ *FirstOrder* C'
assume *Ball* S *well-constrained*
from ‹*derivable* $C\ P\ S'\ \sigma$ *FirstOrder* C'› **show** *well-constrained* C
using *derivable-clauses-are-well-constrained* **by** *auto*
qed

lemma *SOUNDNESS*:
shows *derivable-ecl* $C\ S \Longrightarrow \forall\, x \in S.$ (*finite* (*cl-ecl* x))
$\Longrightarrow$ *set-entails-clause* (*cl-ecl* ‘ S) (*cl-ecl* C)
proof (*induction rule*: *derivable-ecl.induct*)
fix C :: *'a eclause* **fix** S **assume** $C \in S$
assume $\forall\, x \in S.$ (*finite* (*cl-ecl* x))
from ‹$C \in S$› **show** *set-entails-clause* (*cl-ecl* ‘ S) (*cl-ecl* C)
unfolding *set-entails-clause-def* **by** *auto*
next
fix $C\ S$ **fix** D :: *'a eclause* **assume** *derivable-ecl* $C\ S$
assume $\forall\, x \in S.$ (*finite* (*cl-ecl* x))

assume *hyp-ind*: $\forall x \in S.$ (*finite* (*cl-ecl* x)) $\Longrightarrow$ *set-entails-clause* (*cl-ecl* ' S) (*cl-ecl* C)
assume (*renaming-cl* C D)
from ‹(*renaming-cl* C D)› **obtain** η **where** D = (*subst-ecl* C η)
unfolding *renaming-cl-def* **by** *auto*
obtain *C-Cl* T **where** C = (*Ecl C-Cl* T) **using** *eclause.exhaust* **by** *auto*
from *this* **and** ‹D = (*subst-ecl* C η)›
have (*cl-ecl* D) = (*subst-cl* (*cl-ecl* C) η) **by** *auto*
show *set-entails-clause* (*cl-ecl* ' S) (*cl-ecl* D)
proof (*rule ccontr*)
assume ¬*?thesis*
from *this* **obtain** I **where** *fo-interpretation* I **and** *i*: *validate-clause-set* I (*cl-ecl* 'S)
¬*validate-clause* I (*cl-ecl* D)
unfolding *set-entails-clause-def* **by** *auto*
from ‹¬*validate-clause* I (*cl-ecl* D)› **and** ‹(*cl-ecl* D) = (*subst-cl* (*cl-ecl* C) η)›
have ¬*validate-clause* I (*cl-ecl* C) **using** *instances-are-entailed* **by** *metis*
from *this* **and** ‹*fo-interpretation* I› *i* **have** ¬*set-entails-clause* (*cl-ecl* ' S) (*cl-ecl* C)
unfolding *set-entails-clause-def* **by** *auto*
from *this* **and** ‹$\forall x \in S.$ (*finite* (*cl-ecl* x))› *hyp-ind* **show** *False* **by** *auto*
qed
next
fix P S C S' σ C'
assume *h*: $\forall x.\ x \in P \longrightarrow$ *derivable-ecl* x S $\wedge$ (($\forall x \in S.$ *finite* (*cl-ecl* x)) $\longrightarrow$ *set-entails-clause* (*cl-ecl* ' S) (*cl-ecl* x))
assume *derivable* C P S' σ *FirstOrder* C'
assume $\forall x \in S.$ *finite* (*cl-ecl* x)
from *h* **and** ‹$\forall x \in S.$ *finite* (*cl-ecl* x)› **have** *i*: $\forall x \in P.$ *set-entails-clause* (*cl-ecl* ' S) (*cl-ecl* x)
by *metis*
show *set-entails-clause* (*cl-ecl* ' S) (*cl-ecl* C)
proof (*rule ccontr*)
assume ¬*?thesis*
from *this* **obtain** I **where** *fo-interpretation* I **and** *ii*: *validate-clause-set* I (*cl-ecl* 'S)
¬*validate-clause* I (*cl-ecl* C)
unfolding *set-entails-clause-def* **by** *auto*
from *h* ‹$\forall x \in S.$ *finite* (*cl-ecl* x)› **have** ($\forall x \in P.$ *finite* (*cl-ecl* x))
using *all-derived-clauses-are-finite* **by** *metis*
from ‹*fo-interpretation* I› *i* **and** *ii*
have $\forall x \in P.$ (*validate-clause* I (*cl-ecl* x)) **unfolding** *set-entails-clause-def* **by** *auto*
from *this* **have** *validate-clause-set* I (*cl-ecl* ' P) **by** *auto*
from *this* **and** ‹($\forall x \in P.$ *finite* (*cl-ecl* x))› ‹*fo-interpretation* I› ‹*derivable* C P S' σ *FirstOrder* C'›
have *validate-clause* I (*cl-ecl* C)
using *derivable-clauses-are-entailed* [*of* C P S' σ *FirstOrder* C' I] **by** *blast*
from *this* **and** ‹¬*validate-clause* I (*cl-ecl* C)› **show** *False* **by** *auto*

qed
qed

lemma *REFUTABLE-SETS-ARE-UNSAT*:
assumes $\forall x \in S.$ (*finite* (*cl-ecl x*))
assumes *derivable-ecl C S*
assumes (*cl-ecl C* = {})
shows ¬ (*satisfiable-clause-set* (*cl-ecl ‘ S*))
proof
assume (*satisfiable-clause-set* (*cl-ecl ‘ S*))
then obtain *I* **where** *fo-interpretation I* **and** *model*: *validate-clause-set I* (*cl-ecl ‘ S*)
unfolding *satisfiable-clause-set-def* [*of cl-ecl ‘ S*] **by** *blast*
from *assms*(*1*) *assms*(*2*) **have** *set-entails-clause* (*cl-ecl ‘ S*) (*cl-ecl C*)
using *SOUNDNESS* **by** *metis*
from *this* ‹*fo-interpretation I*› **and** *model* **have** *validate-clause I* (*cl-ecl C*)
unfolding *set-entails-clause-def* **by** *auto*
from *this* **and** *assms*(*3*) **show** *False* **by** *auto*
qed

6 Redundancy Criteria and Saturated Sets

We define redundancy criteria. We use similar notions as in the Bachmair and Ganzinger paper, the only difference is that we have to handle the sets of irreducible terms associated with the clauses. Indeed, to ensure completeness, we must guarantee that all the terms that are irreducible in the entailing clauses are also irreducible in the entailed one (otherwise some needed inferences could be blocked due the irreducibility condition, as in the basic superposition calculus). Of course, if the attached sets of terms are empty, then this condition trivially holds and the definition collapses to the usual one.

We introduce the following relation:

definition *subterms-inclusion* :: *'a trm set* ⇒ *'a trm set* ⇒ *bool*
where *subterms-inclusion E1 E2* = (∀ *x1* ∈ *E1*. ∃ *x2* ∈ *E2*. (*occurs-in x1 x2*))

lemma *subterms-inclusion-refl*:
shows *subterms-inclusion E E*
proof (*rule ccontr*)
assume ¬*subterms-inclusion E E*
from *this* **obtain** *x1* **where** *x1* ∈ *E* **and** ¬ *occurs-in x1 x1* **unfolding** *subterms-inclusion-def* **by** *force*
from ‹¬ *occurs-in x1 x1*› **have** ¬ (∃ *p*. *subterm x1 p x1*) **unfolding** *occurs-in-def* **by** *auto*
from *this* **have** ¬*subterm x1 Nil x1* **by** *metis*
from *this* **show** *False* **by** *auto*
qed

lemma *subterms-inclusion-subset*:
  assumes *subterms-inclusion E1 E2*
  assumes $E2 \subseteq E2'$
  shows *subterms-inclusion E1 E2'*
by (*meson assms*(*1*) *assms*(*2*) *basic-superposition.subterms-inclusion-def basic-superposition-axioms*

subsetD)

lemma *set-inclusion-preserve-normalization*:
  assumes *all-trms-irreducible E f*
  assumes $E' \subseteq E$
  shows *all-trms-irreducible E' f*
by (*meson all-trms-irreducible-def assms*(*1*) *assms*(*2*) *subsetD*)

lemma *subterms-inclusion-preserves-normalization*:
  assumes *all-trms-irreducible E f*
  assumes *subterms-inclusion E' E*
  shows *all-trms-irreducible E' f*
by (*meson all-trms-irreducible-def assms*(*1*) *assms*(*2*) *occur-in-subterm subterms-inclusion-def*)

We define two notions of redundancy, the first one is for inferences: any derivable clause must be entailed by a set of clauses that are strictly smaller than one of the premises.

definition *redundant-inference* ::
  'a eclause $\Rightarrow$ *'a eclause set* $\Rightarrow$ *'a eclause set* $\Rightarrow$ *'a subst* $\Rightarrow$ *bool*
  where *redundant-inference* $C\ S\ P\ \sigma \longleftrightarrow (\exists S' \subseteq$ *instances* S.
   set-entails-clause (*clset-instances* S') (*cl-ecl* C) $\wedge$
  $(\forall x \in S'.$ *subterms-inclusion* (*subst-set* (*trms-ecl* (*fst* x)) (*snd* x)) (*trms-ecl* C))
$\wedge$
   $(\forall x \in S'. \exists D' \in$ *cl-ecl* ' P. ((*cl-ecl* (*fst* x), *snd* x),(D',σ)) $\in$ *cl-ord*))

The second one is the usual notion for clauses: a clause is redundant if it is entailed by smaller (or equal) clauses.

definition *redundant-clause* ::
  'a eclause $\Rightarrow$ *'a eclause set* $\Rightarrow$ *'a subst* $\Rightarrow$ *'a clause* $\Rightarrow$ *bool*
  where (*redundant-clause* $C\ S\ \sigma\ C'$) =
   $(\exists S'.$ ($S' \subseteq$ (*instances* S) $\wedge$ (*set-entails-clause* (*clset-instances* S') (*cl-ecl* C))
$\wedge$
     $(\forall x \in S'.$ (*subterms-inclusion* (*subst-set* (*trms-ecl* (*fst* x)) (*snd* x))
      (*trms-ecl* C))) $\wedge$
     $(\forall x \in S'.$ (((*mset-ecl* ((*fst* x),(*snd* x))),(*mset-cl* (C',σ))) $\in$ (*mult* (*mult*
trm-ord))
      $\vee$ (*mset-ecl* ((*fst* x),(*snd* x))) = *mset-cl* (C',σ)))))

Note that according to the definition above, an extended clause is always redundant w.r.t. a clause obtained from the initial one by adding in the attached set of terms a subterm of a term that already occurs in this set. This remark is important because explicitly adding such subterms in the attached set may prune the search space, due to the fact that the containing term can be removed at some point when calling the function *dom-trm*. Adding the subterm explicitly is thus useful in this case. In practice, the simplest solution may be to assume that the set of irreducible terms is closed under subterm.

Of course, a clause is also redundant w.r.t. any clause obtained by removing terms in the attached set. In particular, terms can be safely removed from the set of irreducible terms of the entailing clauses if needed to make a given clause redundant.

```
lemma self-redundant-clause:
  assumes C ∈ S
  assumes C' = (cl-ecl C)
  assumes ground-clause (subst-cl (cl-ecl C) σ)
  shows redundant-clause (subst-ecl C σ) S σ C'
proof –
  obtain Cl-C and T where C = Ecl Cl-C T using eclause.exhaust by auto
  from this have cl-ecl C = Cl-C and trms-ecl C = T by auto
  let ?Cl-C = subst-cl Cl-C σ
  let ?T = subst-set T σ
  let ?C = subst-ecl C σ
  from ‹C = Ecl Cl-C T› have ?C = (Ecl ?Cl-C ?T) by auto
  from this have cl-ecl ?C = ?Cl-C and trms-ecl ?C = ?T by auto
  let ?S = { (C,σ) }
   from assms(1) assms(3) have i: ?S ⊆ (instances S) unfolding instances-def
by auto
 from ‹cl-ecl C = Cl-C› have clset-instances ?S = { ?Cl-C } unfolding clset-instances-def

    by auto
   from this and ‹cl-ecl ?C = ?Cl-C› have ii: set-entails-clause (clset-instances
?S) (cl-ecl ?C)
    using set-entails-clause-member by force
  have iii: (∀ x ∈ ?S. ( subterms-inclusion (subst-set (trms-ecl (fst x)) (snd x))
            (trms-ecl ?C)))
  proof
    fix x assume x ∈ ?S
    from this have x = (C,σ) by auto
    from this ‹C = Ecl Cl-C T›
      have subst-set (trms-ecl (fst x)) (snd x) = ?T by auto
    from this and ‹trms-ecl ?C = ?T›
      have subst-set (trms-ecl (fst x)) (snd x) = (trms-ecl ?C) by auto
    from this show ( subterms-inclusion (subst-set (trms-ecl (fst x)) (snd x))
            (trms-ecl ?C))
            using subterms-inclusion-refl by auto
  qed
  have iv: (∀ x ∈ ?S. ( ((mset-ecl ((fst x),(snd x))),(mset-cl (C',σ))) ∈ (mult (mult
trm-ord))
            ∨ (mset-ecl ((fst x),(snd x))) = mset-cl (C',σ)))
  proof
```

fix x **assume** $x \in ?S$
from *this* **have** $x = (C,\sigma)$ **by** *auto*
from *this* ‹$C = Ecl\ Cl\text{-}C\ T$› **have** $(mset\text{-}ecl\ ((fst\ x),(snd\ x))) = (mset\text{-}ecl\ (C,\sigma))$ **by** *auto*
from *this* ‹$C' = (cl\text{-}ecl\ C)$› **have** $(mset\text{-}ecl\ ((fst\ x),(snd\ x))) = mset\text{-}cl\ (C',\sigma)$ **by** *auto*
from *this* **show** $(\ ((mset\text{-}ecl\ ((fst\ x),(snd\ x))),(mset\text{-}cl\ (C',\sigma))) \in (mult\ (mult\ trm\text{-}ord))$
$\vee\ (mset\text{-}ecl\ ((fst\ x),(snd\ x))) = mset\text{-}cl\ (C',\sigma))$ **by** *auto*
qed
from *i ii iii iv* **show** *?thesis* **unfolding** *redundant-clause-def* **by** *metis*
qed

definition *trms-subsumes*
where $trms\text{-}subsumes\ C\ D\ \sigma$
$= (\ (subst\text{-}cl\ (cl\text{-}ecl\ C)\ \sigma) = (cl\text{-}ecl\ D)$
$\wedge\ ((subst\text{-}set\ (trms\text{-}ecl\ C)\ \sigma) \subseteq trms\text{-}ecl\ D))$

definition *inference-closed*
where $inference\text{-}closed\ S\ = (\forall\ P\ C'\ D\ \vartheta.$
$(derivable\ D\ P\ S\ \vartheta\ FirstOrder\ C') \longrightarrow (D \in S))$

Various notions of saturatedness are defined, depending on the kind of inferences that are considered and on the redundancy criterion.

The first definition is the weakest one: all ground inferences must be redundant (this definition is used for the completeness proof to make it the most general).

definition *ground-inference-saturated* :: *'a eclause set* $\Rightarrow$ *bool*
where $(ground\text{-}inference\text{-}saturated\ S) = (\forall\ C\ P\ \sigma\ C'.\ (derivable\ C\ P\ S\ \sigma\ Ground\ C') \longrightarrow$
$(ground\text{-}clause\ (cl\text{-}ecl\ C)) \longrightarrow (grounding\text{-}set\ P\ \sigma) \longrightarrow (redundant\text{-}inference\ C\ S\ P\ \sigma))$

The second one states that every ground instance of a first-order inference must be redundant.

definition *inference-saturated* :: *'a eclause set* $\Rightarrow$ *bool*
where $(inference\text{-}saturated\ S) = (\forall\ C\ P\ \sigma\ C'\ D\ \vartheta\ \eta.$
$(derivable\ C\ P\ S\ \sigma\ Ground\ C') \longrightarrow (ground\text{-}clause\ (cl\text{-}ecl\ C)) \longrightarrow (grounding\text{-}set\ P\ \sigma)$
$\longrightarrow (derivable\ D\ P\ S\ \vartheta\ FirstOrder\ C') \longrightarrow (trms\text{-}subsumes\ D\ C\ \eta)$
$\longrightarrow (\sigma \doteq \vartheta \lozenge \eta)$
$\longrightarrow (redundant\text{-}inference\ (subst\text{-}ecl\ D\ \eta)\ S\ P\ \sigma))$

The last definition is the most restrictive one: every derivable clause must be redundant.

definition *clause-saturated* :: *'a eclause set* $\Rightarrow$ *bool*
where $(clause\text{-}saturated\ S) = (\forall\ C\ P\ \sigma\ C'\ D\ \vartheta\ \eta.$

$(derivable\ C\ P\ S\ \sigma\ Ground\ C') \longrightarrow (ground\text{-}clause\ (cl\text{-}ecl\ C))$
$\longrightarrow (derivable\ D\ P\ S\ \vartheta\ FirstOrder\ C') \longrightarrow (trms\text{-}subsumes\ D\ C\ \eta)$
$\longrightarrow (\sigma \doteq \vartheta \lozenge \eta)$
$\longrightarrow (redundant\text{-}clause\ (subst\text{-}ecl\ D\ \eta)\ S\ \sigma\ C'))$

We now relate these various notions, so that the forthcoming completeness proof applies to all of them. To this purpose, we have to show that the conclusion of a (ground) inference rule is always strictly smaller than one of the premises.

lemma *conclusion-is-smaller-than-premisses*:
assumes $derivable\ C\ P\ S\ \sigma\ Ground\ C'$
assumes $\forall x \in S.\ (finite\ (cl\text{-}ecl\ x))$
assumes $grounding\text{-}set\ P\ \sigma$
shows $\exists\ D.\ (D \in P \land ((\ (mset\text{-}cl\ (C',\sigma)),\ (mset\text{-}ecl\ (D,\sigma))) \in (mult\ (mult\ trm\text{-}ord))))$
proof $(rule\ ccontr)$
assume hyp: $\neg\ (\exists\ D.\ (D \in P \land ((\ (mset\text{-}cl\ (C',\sigma)),\ (mset\text{-}ecl\ (D,\sigma))) \in (mult\ (mult\ trm\text{-}ord)))))$
from $assms(1)$ **have** $P \subseteq S$ **unfolding** *derivable-def* **by** *auto*
have $not\text{-}sup$: $\neg\ (\exists P1\ P2.\ (P1 \in P \land P2 \in P \land superposition\ P1\ P2\ C\ \sigma\ Ground\ C'))$
proof
assume $(\exists P1\ P2.\ (P1 \in P \land P2 \in P \land superposition\ P1\ P2\ C\ \sigma\ Ground\ C'))$
then obtain $P1\ P2$ **where** $P1 \in P\ P2 \in P\ superposition\ P1\ P2\ C\ \sigma\ Ground\ C'$ **by** *auto*
from ‹$superposition\ P1\ P2\ C\ \sigma\ Ground\ C'$› **obtain** $L\ t\ s\ u\ v\ M\ \ L'\ polarity\ u'\ p\ t'\ Cl\text{-}C\ NT$ **where**
$M \in (cl\text{-}ecl\ P2)\ L \in (cl\text{-}ecl\ P1)$
$orient\text{-}lit\text{-}inst\ M\ u\ v\ pos\ \sigma$
$orient\text{-}lit\text{-}inst\ L\ t\ s\ polarity\ \sigma$
$subterm\ t\ p\ u'$
$ck\text{-}unifier\ u'\ u\ \sigma\ Ground$
$replace\text{-}subterm\ t\ p\ v\ t'$
$L' = mk\text{-}lit\ polarity\ (Eq\ t'\ s)$
$(C = (Ecl\ Cl\text{-}C\ NT))$
$(subst\ u\ \sigma) \neq (subst\ v\ \sigma)$
$(\ (subst\text{-}lit\ M\ \sigma),(subst\text{-}lit\ L\ \sigma))$
$\in lit\text{-}ord$
$strictly\text{-}maximal\text{-}literal\ P2\ M\ \sigma$
$Cl\text{-}C = (subst\text{-}cl\ (((cl\text{-}ecl\ P1) - \{\ L\ \}) \cup (((cl\text{-}ecl\ P2) - \{\ M\ \}) \cup \{\ L'\ \}\ ))\ \sigma)$
$C' = (((cl\text{-}ecl\ P1) - \{\ L\ \}) \cup (((cl\text{-}ecl\ P2) - \{\ M\ \}) \cup \{\ L'\ \}\ ))$
unfolding *superposition-def* **by** *blast*
from ‹$P1 \in P$› **and** $assms(2)$ **and** ‹$P \subseteq S$› **have** $finite\ (cl\text{-}ecl\ P1)$ **by** *auto*
from ‹$P2 \in P$› **and** $assms(2)$ **and** ‹$P \subseteq S$› **have** $finite\ (cl\text{-}ecl\ P2)$ **by** *auto*

from $assms(3)$ **and** ‹$P2 \in P$› **have** $ground\text{-}clause\ (subst\text{-}cl\ (cl\text{-}ecl\ P2)\ \sigma)$
unfolding *grounding-set-def* **by** *auto*
from *this* **have** $vars\text{-}of\text{-}cl\ (subst\text{-}cl\ (cl\text{-}ecl\ P2)\ \sigma) = \{\}$ **by** *auto*

from ‹*M* ∈ (*cl-ecl P2*)›**have** (*subst-lit M σ*) ∈ (*subst-cl* (*cl-ecl P2*) *σ*) **by** *auto*
from *this* **and** ‹*vars-of-cl* (*subst-cl* (*cl-ecl P2*) *σ*) = {}› **have** *vars-of-lit* (*subst-lit M σ*) = {}
by *auto*
from ‹*orient-lit-inst M u v pos σ*› **have**
orient-lit (*subst-lit M σ*) (*subst u σ*) (*subst v σ*) *pos*
using *lift-orient-lit* **by** *auto*
from *this* **and** ‹*vars-of-lit* (*subst-lit M σ*) = {}› **have** *vars-of* (*subst u σ*) = {}
using *orient-lit-vars* **by** *blast*
from ‹*orient-lit* (*subst-lit M σ*) (*subst u σ*) (*subst v σ*) *pos*›
and ‹*vars-of-lit* (*subst-lit M σ*) = {}› **have** *vars-of* (*subst v σ*) = {}
using *orient-lit-vars* **by** *blast*
from ‹*orient-lit* (*subst-lit M σ*) (*subst u σ*) (*subst v σ*) *pos*›
have ((*subst u σ*),(*subst v σ*)) ∉ *trm-ord*
unfolding *orient-lit-def* **by** *auto*
from *this* **and** ‹(*subst u σ*) ≠ (*subst v σ*)›
and ‹*vars-of* (*subst u σ*) = {}› ‹*vars-of* (*subst v σ*) = {}›
have ((*subst v σ*),(*subst u σ*)) ∈ *trm-ord* **using** *trm-ord-ground-total*
unfolding *ground-term-def* **by** *blast*

from *assms(3)* **and** ‹*P1* ∈ *P*› **have** *ground-clause* (*subst-cl* (*cl-ecl P1*) *σ*)
unfolding *grounding-set-def* **by** *auto*
from *this* **have** *vars-of-cl* (*subst-cl* (*cl-ecl P1*) *σ*) = {} **by** *auto*
from ‹*L* ∈ (*cl-ecl P1*)›**have** (*subst-lit L σ*) ∈ (*subst-cl* (*cl-ecl P1*) *σ*) **by** *auto*
from *this* **and** ‹*vars-of-cl* (*subst-cl* (*cl-ecl P1*) *σ*) = {}› **have** *vars-of-lit* (*subst-lit L σ*) = {}
by *auto*
from ‹*orient-lit-inst L t s polarity σ*› **have**
orient-lit (*subst-lit L σ*) (*subst t σ*) (*subst s σ*) *polarity*
using *lift-orient-lit* **by** *auto*
from *this* **and** ‹*vars-of-lit* (*subst-lit L σ*) = {}› **have** *vars-of* (*subst t σ*) = {}
using *orient-lit-vars* **by** *blast*
from ‹*orient-lit* (*subst-lit L σ*) (*subst t σ*) (*subst s σ*) *polarity*›
and ‹*vars-of-lit* (*subst-lit L σ*) = {}› **have** *vars-of* (*subst s σ*) = {}
using *orient-lit-vars* **by** *blast*

let *?mC1* = *mset-ecl* (*P1*, *σ*)
let *?mC2* = *mset-ecl* (*C*, *σ*)

from ‹*L* ∈ (*cl-ecl P1*)› ‹*finite* (*cl-ecl P1*)›
have *mset-set* (*cl-ecl P1*) = *mset-set* ((*cl-ecl P1*)−{ *L* }) + *mset-set* { *L* }
using *split-mset-set* [*of cl-ecl P1 cl-ecl P1* − { *L* } { *L* }] **by** *blast*

from *this* **have** *d1*: {# (*mset-lit* (*subst-lit x σ*)). *x* ∈# (*mset-set* (*cl-ecl P1*)) #}
= {# (*mset-lit* (*subst-lit x σ*)). *x* ∈# (*mset-set* ((*cl-ecl P1*) − { *L* })) #}
+ {# (*mset-lit* (*subst-lit x σ*)). *x* ∈# (*mset-set* { *L* }) #}
using *split-image-mset* **by** *auto*

let *?C* = (((*cl-ecl P1*) − { *L* }) ∪ (((*cl-ecl P2*) − { *M* }) ∪ { *L′* }))
from ‹*finite* (*cl-ecl P1*)› ‹*finite* (*cl-ecl P2*)› **have** *finite ?C* **by** *auto*
let *?C′* = *?C* − ((*cl-ecl P1*) − { *L* })
from ‹*finite ?C*› **have** *finite ?C′* **by** *auto*
have *?C* = ((*cl-ecl P1*) − { *L* }) ∪ *?C′* **by** *auto*
from ‹*finite* (*cl-ecl P1*)› ‹*finite ?C′*›
  have *mset-set ?C* = *mset-set* ((*cl-ecl P1*)−{ *L* }) + *mset-set ?C′*
  using *split-mset-set* [*of ?C cl-ecl P1* − { *L* } *?C′*] **by** *blast*

from *this* **have** *d2*: {# (*mset-lit* (*subst-lit x* σ)). *x* ∈# (*mset-set ?C*) #}
  = {# (*mset-lit* (*subst-lit x* σ)). *x* ∈# (*mset-set* ((*cl-ecl P1*) − { *L* })) #}
    + {# (*mset-lit* (*subst-lit x* σ)). *x* ∈# (*mset-set ?C′*) #}
    using *split-image-mset* **by** *auto*

have {# (*mset-lit* (*subst-lit x* σ)). *x* ∈# (*mset-set* { *L* }) #} ≠ {#}
  by *auto*
let *?K* = {# (*mset-lit* (*subst-lit x* σ)). *x* ∈# (*mset-set ?C′*) #}
let *?J* = {# (*mset-lit* (*subst-lit x* σ)). *x* ∈# (*mset-set* { *L* }) #}
have (∀ *k* ∈ *set-mset ?K*. ∃ *j* ∈ *set-mset ?J*. (*k*, *j*) ∈ (*mult trm-ord*))
proof
fix *k* **assume** *k* ∈ *set-mset ?K*
from *this* **have** *k* ∈# *?K* **by** *auto*
from *this* **obtain** *M′* **where** *M′* ∈# (*mset-set ?C′*) **and** *k* = (*mset-lit* (*subst-lit*
M′ σ))
  using *image-mset-thm* [*of ?K* λx. (*mset-lit* (*subst-lit x* σ)) (*mset-set ?C′*)]
  by *metis*

from ‹*M′* ∈# (*mset-set ?C′*)›**and** ‹*finite ?C′*› **have** *M′* ∈ *?C′* **by** *auto*
have *L* ∈# (*mset-set* { *L* }) **by** *auto*
from *this* **have** (*mset-lit* (*subst-lit L* σ) ∈# *?J*) **by** *auto*
from *this* **have** (*mset-lit* (*subst-lit L* σ) ∈ *set-mset ?J*) **by** *auto*

have {# (*mset-lit* (*subst-lit x* σ)). *x* ∈# (*mset-set* { *L* }) #} ≠ {#} **by** *auto*

show ∃ *j* ∈ *set-mset ?J*. (*k*, *j*) ∈ (*mult trm-ord*)
proof (*cases*)
  assume *M′* ∈ (*cl-ecl P2*) − { *M* }
  from *this* **and** ‹*strictly-maximal-literal P2 M* σ›
    have ((*subst-lit M′* σ),(*subst-lit M* σ)) ∈ *lit-ord*
      unfolding *strictly-maximal-literal-def* **by** *metis*
  from *this* **and** ‹((*subst-lit M* σ),(*subst-lit L* σ)) ∈ *lit-ord*›
    have ((*subst-lit M′* σ),(*subst-lit L* σ)) ∈ *lit-ord*
    using *lit-ord-trans* **unfolding** *trans-def* **by** *metis*
  from *this* **have** ((*mset-lit* (*subst-lit M′* σ)),
      (*mset-lit* (*subst-lit L* σ))) ∈ (*mult trm-ord*)
      unfolding *lit-ord-def* **by** *auto*
  from ‹(*mset-lit* (*subst-lit L* σ) ∈ *set-mset ?J*)› *this* ‹((*mset-lit* (*subst-lit M′*

$\sigma)),$

$(mset\text{-}lit\ (subst\text{-}lit\ L\ \sigma))) \in\ (mult\ trm\text{-}ord)$›

and ‹$k = (mset\text{-}lit\ (subst\text{-}lit\ M'\ \sigma))$› **show** *?thesis* **by** *blast*

next assume $M' \notin (cl\text{-}ecl\ P2) - \{\ M\ \}$

from *this* **and** ‹$M' \in\ ?C'$› **have** $M' = L'$ **by** *auto*

from ‹$subterm\ t\ p\ u'$› **have** $subterm\ (subst\ t\ \sigma)\ p\ \ (subst\ u'\ \sigma)$

using *substs-preserve-subterms* **by** *blast*

from ‹$ck\text{-}unifier\ u'\ u\ \sigma\ Ground$› **have**

$(subst\ u\ \sigma) = (subst\ u'\ \sigma)$ **unfolding** *ck-unifier-def Unifier-def* **by** *auto*

from *this* **and** ‹$((subst\ v\ \sigma),(subst\ u\ \sigma)) \in trm\text{-}ord$›

have $((subst\ v\ \sigma),(subst\ u'\ \sigma)) \in trm\text{-}ord$ **by** *auto*

from *this* ‹$subterm\ t\ p\ u'$› ‹$replace\text{-}subterm\ t\ p\ v\ t'$›

have $((subst\ t'\ \sigma),(subst\ t\ \sigma)) \in trm\text{-}ord$

using *replacement-monotonic* **by** *auto*

have $polarity = pos \vee polarity = neg$ **using** *sign.exhaust* **by** *auto*

then have $((subst\text{-}lit\ L'\ \sigma),(subst\text{-}lit\ L\ \sigma)) \in lit\text{-}ord$

proof

assume $polarity = pos$

from *this* **and** ‹$orient\text{-}lit\text{-}inst\ L\ t\ s\ polarity\ \sigma$›

have *i*: $(mset\text{-}lit\ (subst\text{-}lit\ L\ \sigma)) = \{\#\ (subst\ s\ \sigma)\ \#\} + \{\#\ (subst\ t\ \sigma)\ \#\}$

unfolding *orient-lit-inst-def* **using** *add.commute* **by** *force*

from ‹$L' = mk\text{-}lit\ polarity\ (Eq\ t'\ s)$› ‹$polarity = pos$›

have *ii*: $(mset\text{-}lit\ (subst\text{-}lit\ L'\ \sigma)) = \{\#\ (subst\ s\ \sigma)\ \#\}$

$+ \{\#\ (subst\ t'\ \sigma)\ \#\}$

using *add.commute* **by** *force*

have $\{\#\ (subst\ t\ \sigma)\ \#\} \neq \{\#\}$ **by** *auto*

have $(\forall k' \in set\text{-}mset\ \{\#\ (subst\ t'\ \sigma)\ \#\}.\ \exists j' \in set\text{-}mset\ \{\#\ (subst\ t\ \sigma)$
$\#\}.\ (k', j') \in (trm\text{-}ord))$

proof

fix k' **assume** $k' \in set\text{-}mset\ \{\#\ (subst\ t'\ \sigma)\ \#\}$

from *this* **have** $k' = (subst\ t'\ \sigma)$ **by** *auto*

have $(subst\ t\ \sigma) \in set\text{-}mset\ \{\#\ (subst\ t\ \sigma)\ \#\}$ **by** *auto*

from *this* ‹$k' = (subst\ t'\ \sigma)$›

and ‹$((subst\ t'\ \sigma),(subst\ t\ \sigma)) \in trm\text{-}ord$›

show $\exists j' \in set\text{-}mset\ \{\#\ (subst\ t\ \sigma)\ \#\}.\ (k', j') \in (trm\text{-}ord)$

by *auto*

qed

from *i ii*

‹$((subst\ t'\ \sigma),(subst\ t\ \sigma)) \in trm\text{-}ord$›

have $(mset\text{-}lit\ (subst\text{-}lit\ L'\ \sigma),(mset\text{-}lit\ (subst\text{-}lit\ L\ \sigma)))$

$\in (mult\ trm\text{-}ord)$

by (*metis one-step-implies-mult empty-iff insert-iff set-mset-add-mset-insert*
set-mset-empty)

from *this* **show** *?thesis* **unfolding** *lit-ord-def* **by** *auto*

next

assume $polarity = neg$

from *this* **and** ‹$orient\text{-}lit\text{-}inst\ L\ t\ s\ polarity\ \sigma$›

```
have i: (mset-lit (subst-lit L σ)) = {# (subst s σ), (subst s σ) #}
  + {# (subst t σ), (subst t σ) #}
  unfolding orient-lit-inst-def by auto
from ‹L' = mk-lit polarity (Eq t' s)› ‹polarity = neg› have
  subst-lit L' σ = (Neg (Eq (subst t' σ) (subst s σ))) by auto
from this have (mset-lit (subst-lit L' σ))
  = {# (subst t' σ), (subst t' σ), (subst s σ), (subst s σ) #}
  by auto
from this have ii: (mset-lit (subst-lit L' σ))
  = {# (subst s σ), (subst s σ) #} + {# (subst t' σ), (subst t' σ) #}
  by (simp add: add.commute add.left-commute)

have {# (subst t σ), (subst t σ) #} ≠ {#} by auto

have (∀ k' ∈ set-mset {# (subst t' σ),(subst t' σ) #}.
  ∃ j' ∈ set-mset {# (subst t σ),(subst t σ) #}. (k', j') ∈ (trm-ord))
proof
  fix k' assume k' ∈set-mset {# (subst t' σ),(subst t' σ) #}
  from this have k' = (subst t' σ) by auto
  have (subst t σ) ∈ set-mset {# (subst t σ),(subst t σ) #} by auto
  from this ‹k' = (subst t' σ)›
    and ‹((subst t' σ),(subst t σ)) ∈ trm-ord›
    show ∃ j' ∈ set-mset {# (subst t σ),(subst t σ) #}. (k', j') ∈ (trm-ord)
      by auto
qed

from this i ii ‹{# (subst t σ), (subst t σ) #} ≠ {#}›
  have (mset-lit (subst-lit L' σ),
    (mset-lit (subst-lit L σ))) ∈ (mult trm-ord)
     using one-step-implies-mult [of {# (subst t σ), (subst t σ) #}
       {# (subst t' σ),(subst t' σ) #} trm-ord
       {# (subst s σ),(subst s σ) #}]
     trm-ord-trans by auto

  from this show ?thesis unfolding lit-ord-def by auto
 qed
from this and
  ‹(mset-lit (subst-lit L σ) ∈ set-mset ?J)›
  ‹k = (mset-lit (subst-lit M' σ))›
  ‹M' = L'› show ?thesis unfolding lit-ord-def by auto
qed
qed
from this d1 d2 have o:
  ({#mset-lit (subst-lit x σ). x ∈# mset-set ?C #},
  {#mset-lit (subst-lit x σ). x ∈# mset-set (cl-ecl P1)#})
  ∈ mult (mult trm-ord)
using mult-trm-ord-trans one-step-implies-mult [of {# (mset-lit (subst-lit x σ)).
x ∈# (mset-set { L }) #}
  {# (mset-lit (subst-lit x σ)). x ∈# (mset-set ?C') #} mult trm-ord
```

{# (*mset-lit* (*subst-lit* *x* σ)). *x* ∈# (*mset-set* ((*cl-ecl* *P1*) − { *L* })) #}] **by** *auto*

from *this* ‹C' = (((*cl-ecl* *P1*) − { *L* }) ∪ (((*cl-ecl* *P2*) − { *M* }) ∪ { L' }))› **and** ‹*P1* ∈ *P*›
  and *hyp* **show** *False* **by** *auto*
qed

have *not-ref*: ¬ (∃ *P1*. (*P1* ∈ *P* ∧ *reflexion* *P1* *C* σ *Ground* C'))
proof
  assume (∃ *P1*. (*P1* ∈ *P* ∧ *reflexion* *P1* *C* σ *Ground* C'))
  then obtain *P1* **where** *P1* ∈ *P* *reflexion* *P1* *C* σ *Ground* C' **by** *auto*
  from ‹*reflexion* *P1* *C* σ *Ground* C'› **obtain** *L1* *t* *s* *Cl-C* *Cl-P* **where**
    (*eligible-literal* *L1* *P1* σ)
    (*L1* ∈ (*cl-ecl* *P1*)) (*Cl-C* = (*cl-ecl* *C*)) (*Cl-P* = (*cl-ecl* *P1*))
    (*orient-lit-inst* *L1* *t* *s* *neg* σ)
    (*ck-unifier* *t* *s* σ *Ground*)
    (*Cl-C* = (*subst-cl* ((*Cl-P* − { *L1* }))) σ)
    (C' = ((*Cl-P* − { *L1* })))
    unfolding *reflexion-def* **by** *blast*
  from ‹*P1* ∈ *P*› **and** *assms*(*2*) **and** ‹*P* ⊆ *S*› **have** *finite* (*cl-ecl* *P1*) **by** *auto*

  let *?mC1* = *mset-ecl* (*P1*, σ)
  let *?mC2* = *mset-ecl* (*C*, σ)

  from ‹*L1* ∈ (*cl-ecl* *P1*)› ‹*finite* (*cl-ecl* *P1*)›
   have *mset-set* (*cl-ecl* *P1*) = *mset-set* ((*cl-ecl* *P1*)−{ *L1* }) + *mset-set* { *L1* }
    using *split-mset-set* [*of* *cl-ecl* *P1* *cl-ecl* *P1* − { *L1* } { *L1* }] **by** *blast*

  from *this* **have** *d1*: {# (*mset-lit* (*subst-lit* *x* σ)). *x* ∈# (*mset-set* (*cl-ecl* *P1*)) #}
    = {# (*mset-lit* (*subst-lit* *x* σ)). *x* ∈# (*mset-set* ((*cl-ecl* *P1*) − { *L1* })) #}
      + {# (*mset-lit* (*subst-lit* *x* σ)). *x* ∈# (*mset-set* { *L1* }) #}
      using *split-image-mset* **by** *auto*

  let *?C* = ((*cl-ecl* *P1*) − { *L1* })
  from ‹*finite* (*cl-ecl* *P1*)› **have** *finite* *?C* **by** *auto*
  let $?C'$ = {}
  have *finite* $?C'$ **by** *auto*
  have *?C* = ((*cl-ecl* *P1*) − { *L1* }) ∪ $?C'$ **by** *auto*
  from ‹*finite* (*cl-ecl* *P1*)› ‹*finite* $?C'$›
    have *mset-set* *?C* = *mset-set* ((*cl-ecl* *P1*)−{ *L1* }) + *mset-set* $?C'$
    using *split-mset-set* [*of* *?C* *cl-ecl* *P1* − { *L1* } $?C'$] **by** *blast*

  from *this* **have** *d2*: {# (*mset-lit* (*subst-lit* *x* σ)). *x* ∈# (*mset-set* *?C*) #}
    = {# (*mset-lit* (*subst-lit* *x* σ)). *x* ∈# (*mset-set* ((*cl-ecl* *P1*) − { *L1* })) #}
      + {# (*mset-lit* (*subst-lit* *x* σ)). *x* ∈# (*mset-set* $?C'$) #}
      using *split-image-mset* **by** *auto*

have $\{\#$ (*mset-lit* (*subst-lit* $x\ \sigma$)). $x \in\#$ (*mset-set* $\{$ *L1* $\}$) $\#\} \neq \{\#\}$
by *auto*
let *?K* = $\{\#$ (*mset-lit* (*subst-lit* $x\ \sigma$)). $x \in\#$ (*mset-set* *?C′*) $\#\}$
let *?J* = $\{\#$ (*mset-lit* (*subst-lit* $x\ \sigma$)). $x \in\#$ (*mset-set* $\{$ *L1* $\}$) $\#\}$
have ($\forall\, k \in$ *set-mset ?K*. $\exists\, j \in$ *set-mset ?J*. $(k, j) \in$ (*mult trm-ord*)) **by** *auto*

from *this d1 d2* ‹$\{\#$ (*mset-lit* (*subst-lit* $x\ \sigma$)). $x \in\#$ (*mset-set* $\{$ *L1* $\}$) $\#\} \neq \{\#\}$›
have *o*:
($\{\#$*mset-lit* (*subst-lit* $x\ \sigma$). $x \in\#$ *mset-set ?C* $\#\}$,
$\{\#$*mset-lit* (*subst-lit* $x\ \sigma$). $x \in\#$ *mset-set* (*cl-ecl P1*)$\#\}$)
$\in$ *mult* (*mult trm-ord*)
using *mult-trm-ord-trans one-step-implies-mult* [*of*
$\{\#$ (*mset-lit* (*subst-lit* $x\ \sigma$)). $x \in\#$ (*mset-set* $\{$ *L1* $\}$) $\#\}$
$\{\#$ (*mset-lit* (*subst-lit* $x\ \sigma$)). $x \in\#$ (*mset-set* *?C′*) $\#\}$ *mult trm-ord*
$\{\#$ (*mset-lit* (*subst-lit* $x\ \sigma$)). $x \in\#$ (*mset-set* ((*cl-ecl P1*) $-$ $\{$ *L1* $\}$)) $\#\}$]
by *auto*

from *this* ‹*Cl-P* = (*cl-ecl P1*)› ‹*C′* = ((*Cl-P* $-$ $\{$ *L1* $\}$))› **and** ‹*P1* $\in$ *P*›
and *hyp* **show** *False* **by** *auto*
qed

have *not-fact*: $\neg$ ($\exists$ *P1*. (*P1* $\in$ *P* $\wedge$ *factorization P1 C* σ *Ground C′*))
proof
assume ($\exists$ *P1*. (*P1* $\in$ *P* $\wedge$ *factorization P1 C* σ *Ground C′*))
then obtain *P1* **where** *P1* $\in$ *P factorization P1 C* σ *Ground C′* **by** *auto*
from ‹*factorization P1 C* σ *Ground C′*› **obtain** *L1 L2 L′ t s u v Cl-P Cl-C* **where**
(*eligible-literal L1 P1* σ)
(*L1* $\in$ (*cl-ecl P1*)) (*L2* $\in$ (*cl-ecl P1*) $-$ $\{$ *L1* $\}$) (*Cl-C* = (*cl-ecl C*)) (*Cl-P* = (*cl-ecl P1*))
(*orient-lit-inst L1 t s pos* σ)
(*orient-lit-inst L2 u v pos* σ)
((*subst t* σ) $\neq$ (*subst s* σ))
((*subst t* σ) $\neq$ (*subst v* σ))
(*ck-unifier t u* σ *Ground*)
(*L′* = *Neg* (*Eq s v*))
(*Cl-C* = (*subst-cl* ((*Cl-P* $-$ $\{$ *L2* $\}$) $\cup$ $\{$ *L′* $\}$)) σ)
(*C′* = ((*Cl-P* $-$ $\{$ *L2* $\}$) $\cup$ $\{$ *L′* $\}$))
unfolding *factorization-def* **by** *blast*
from ‹*P1* $\in$ *P*› **and** *assms*(*2*) **and** ‹*P* $\subseteq$ *S*› **have** *finite* (*cl-ecl P1*) **by** *auto*

from *assms*(*3*) **and** ‹*P1* $\in$ *P*› **have** *ground-clause* (*subst-cl* (*cl-ecl P1*) σ) **unfolding** *grounding-set-def* **by** *auto*
from *this* **have** *vars-of-cl* (*subst-cl* (*cl-ecl P1*) σ) = $\{\}$ **by** *auto*
from ‹*L1* $\in$ (*cl-ecl P1*)›**have** (*subst-lit L1* σ) $\in$ (*subst-cl* (*cl-ecl P1*) σ) **by** *auto*
from *this* **and** ‹*vars-of-cl* (*subst-cl* (*cl-ecl P1*) σ) = $\{\}$› **have** *vars-of-lit* (*subst-lit L1* σ) = $\{\}$
by *auto*

from ‹*orient-lit-inst L1 t s pos σ*› **have**
orient-lit (*subst-lit L1 σ*) (*subst t σ*) (*subst s σ*) *pos*
using *lift-orient-lit* **by** *auto*
from *this* **and** ‹*vars-of-lit* (*subst-lit L1 σ*) = {}› **have** *vars-of* (*subst t σ*) = {}
using *orient-lit-vars* **by** *blast*
from ‹*orient-lit* (*subst-lit L1 σ*) (*subst t σ*) (*subst s σ*) *pos*›
and ‹*vars-of-lit* (*subst-lit L1 σ*) = {}› **have** *vars-of* (*subst s σ*) = {}
using *orient-lit-vars* **by** *blast*

from ‹(*L2* ∈ (*cl-ecl P1*) − { *L1* })› **have** *L2* ∈ (*cl-ecl P1*) **by** *auto*
from ‹*L2* ∈ (*cl-ecl P1*)› **have** (*subst-lit L2 σ*) ∈ (*subst-cl* (*cl-ecl P1*) *σ*) **by** *auto*
from *this* **and** ‹*vars-of-cl* (*subst-cl* (*cl-ecl P1*) *σ*) = {}› **have** *vars-of-lit* (*subst-lit L2 σ*) = {}
by *auto*

from ‹*orient-lit-inst L2 u v pos σ*› **have**
orient-lit (*subst-lit L2 σ*) (*subst u σ*) (*subst v σ*) *pos*
using *lift-orient-lit* **by** *auto*
from *this* **and** ‹*vars-of-lit* (*subst-lit L2 σ*) = {}› **have** *vars-of* (*subst u σ*) = {}
using *orient-lit-vars* **by** *blast*
from ‹*orient-lit* (*subst-lit L2 σ*) (*subst u σ*) (*subst v σ*) *pos*›
and ‹*vars-of-lit* (*subst-lit L2 σ*) = {}› **have** *vars-of* (*subst v σ*) = {}
using *orient-lit-vars* **by** *blast*

from ‹*ck-unifier t u σ Ground*› **have** (*subst t σ*) = (*subst u σ*)
unfolding *ck-unifier-def Unifier-def* **by** *auto*

from ‹*orient-lit* (*subst-lit L1 σ*) (*subst t σ*) (*subst s σ*) *pos*›
have ((*subst t σ*),(*subst s σ*)) ∉ *trm-ord*
unfolding *orient-lit-def* **by** *auto*
from *this* **and** ‹(*subst t σ*) ≠ (*subst s σ*)›
and ‹*vars-of* (*subst t σ*) = {}› ‹*vars-of* (*subst s σ*) = {}›
have ((*subst s σ*),(*subst t σ*)) ∈ *trm-ord* **using** *trm-ord-ground-total*
unfolding *ground-term-def* **by** *blast*
from *this* **and** ‹(*subst t σ*) = (*subst u σ*)› **have**
((*subst s σ*),(*subst u σ*)) ∈ *trm-ord* **by** *auto*

from ‹*orient-lit* (*subst-lit L2 σ*) (*subst u σ*) (*subst v σ*) *pos*›
have ((*subst u σ*),(*subst v σ*)) ∉ *trm-ord*
unfolding *orient-lit-def* **by** *auto*
from *this* **and** ‹(*subst t σ*) ≠ (*subst v σ*)›
and ‹(*subst t σ*) = (*subst u σ*)›
and ‹*vars-of* (*subst u σ*) = {}› ‹*vars-of* (*subst v σ*) = {}›
have ((*subst v σ*),(*subst u σ*)) ∈ *trm-ord* **using** *trm-ord-ground-total*
unfolding *ground-term-def* **by** *metis*

let *?mC1* = *mset-ecl* (*P1*, σ)
let *?mC2* = *mset-ecl* (*C*, σ)

from ‹*L2* ∈ (*cl-ecl P1*)› ‹*finite* (*cl-ecl P1*)›
have *mset-set* (*cl-ecl P1*) = *mset-set* ((*cl-ecl P1*)−{ *L2* }) + *mset-set* { *L2* }
using *split-mset-set* [*of cl-ecl P1 cl-ecl P1* − { *L2* } { *L2* }] **by** *blast*

from *this* **have** *d1*: {# (*mset-lit* (*subst-lit x* σ)). *x* ∈# (*mset-set* (*cl-ecl P1*)) #}
= {# (*mset-lit* (*subst-lit x* σ)). *x* ∈# (*mset-set* ((*cl-ecl P1*) − { *L2* })) #}
+ {# (*mset-lit* (*subst-lit x* σ)). *x* ∈# (*mset-set* { *L2* }) #}
using *split-image-mset* **by** *auto*

let *?C* = (((*cl-ecl P1*) − { *L2* }) ∪ { *L′* })
from ‹*finite* (*cl-ecl P1*)› **have** *finite ?C* **by** *auto*
let *?C′* = *?C* − ((*cl-ecl P1*) − { *L2* })
from ‹*finite ?C*› **have** *finite ?C′* **by** *auto*
have *?C* = ((*cl-ecl P1*) − { *L2* }) ∪ *?C′* **by** *auto*
from ‹*finite* (*cl-ecl P1*)› ‹*finite ?C′*›
have *mset-set ?C* = *mset-set* ((*cl-ecl P1*)−{ *L2* }) + *mset-set ?C′*
using *split-mset-set* [*of ?C cl-ecl P1* − { *L2* } *?C′*] **by** *blast*

from *this* **have** *d2*: {# (*mset-lit* (*subst-lit x* σ)). *x* ∈# (*mset-set ?C*) #}
= {# (*mset-lit* (*subst-lit x* σ)). *x* ∈# (*mset-set* ((*cl-ecl P1*) − { *L2* })) #}
+ {# (*mset-lit* (*subst-lit x* σ)). *x* ∈# (*mset-set ?C′*) #}
using *split-image-mset* **by** *auto*

have {# (*mset-lit* (*subst-lit x* σ)). *x* ∈# (*mset-set* { *L2* }) #} ≠ {#}
by *auto*
let *?K* = {# (*mset-lit* (*subst-lit x* σ)). *x* ∈# (*mset-set ?C′*) #}
let *?J* = {# (*mset-lit* (*subst-lit x* σ)). *x* ∈# (*mset-set* { *L2* }) #}
have (∀ *k* ∈ *set-mset ?K*. ∃ *j* ∈ *set-mset ?J*. (*k*, *j*) ∈ (*mult trm-ord*))
proof
fix *k* **assume** *k* ∈ *set-mset ?K*
from *this* **have** *k* ∈# *?K* **by** *simp*
from *this* **obtain** *M′* **where** *M′* ∈# (*mset-set ?C′*) **and** *k* = (*mset-lit* (*subst-lit M′* σ))
using *image-mset-thm* [*of ?K* λ*x*. (*mset-lit* (*subst-lit x* σ)) (*mset-set ?C′*)]
by *metis*

from ‹*M′* ∈# (*mset-set ?C′*)›**and** ‹*finite ?C′*› **have** *M′* ∈ *?C′* **by** *auto*
have *L2* ∈# (*mset-set* { *L2* }) **by** *auto*
from *this* **have** (*mset-lit* (*subst-lit L2* σ) ∈# *?J*) **by** *auto*
from *this* **have** (*mset-lit* (*subst-lit L2* σ) ∈ *set-mset ?J*) **by** *auto*

have {# (*mset-lit* (*subst-lit x* σ)). *x* ∈# (*mset-set* { *L2* }) #} ≠ {#} **by** *auto*

show ∃ *j* ∈ *set-mset ?J*. (*k*, *j*) ∈ (*mult trm-ord*)
proof −

from ‹$M' \in$ *?C'*› **have** $M' = L'$ **by** *auto*
from ‹*orient-lit-inst L2 u v pos* σ›
have *i*: (*mset-lit* (*subst-lit L2* σ))
= {#} + {# (*subst u* σ), (*subst v* σ) #}
unfolding *orient-lit-inst-def* **using** *add.commute* **by** *force*
from ‹$L' =$ *Neg* (*Eq s v*)›
have *ii*: (*mset-lit* (*subst-lit* L' σ)) =
{#} + {# (*subst s* σ), (*subst s* σ), (*subst v* σ), (*subst v* σ) #}
by *force*
have {# (*subst u* σ), (*subst v* σ) #} $\neq$ {#} **by** *auto*
have ($\forall k' \in$ *set-mset* {# (*subst s* σ), (*subst s* σ), (*subst v* σ), (*subst v* σ) #}.

$\exists j' \in$ *set-mset* {# (*subst u* σ), (*subst v* σ) #}. (k', j') $\in$ (*trm-ord*))
proof
fix k' **assume** *nh*: $k' \in$*set-mset* {# (*subst s* σ), (*subst s* σ), (*subst v* σ), (*subst v* σ) #}
have (*subst u* σ) $\in$ *set-mset* {# (*subst u* σ), (*subst v* σ) #} **by** *auto*
from *nh* **have** $k' =$ (*subst s* σ) $\vee$ $k' =$ (*subst v* σ) **by** *auto*
then show $\exists j' \in$ *set-mset* {# (*subst u* σ), (*subst v* σ) #}. (k', j') $\in$ (*trm-ord*)
proof
assume $k' =$ (*subst s* σ)
from *this* **and** ‹((*subst s* σ),(*subst u* σ)) $\in$ *trm-ord*›
and ‹(*subst u* σ) $\in$ *set-mset* {# (*subst u* σ), (*subst v* σ) #}› **show** *?thesis* **by** *auto*
next
assume $k' =$ (*subst v* σ)
from *this* **and** ‹((*subst v* σ),(*subst u* σ)) $\in$ *trm-ord*›
and ‹(*subst u* σ) $\in$ *set-mset* {# (*subst u* σ), (*subst v* σ) #}› **show** *?thesis* **by** *auto*
qed
qed
from *this i ii* ‹{# (*subst u* σ), (*subst v* σ) #} $\neq$ {#}›
have (*mset-lit* (*subst-lit* L' σ),
(*mset-lit* (*subst-lit L2* σ))) $\in$ (*mult trm-ord*)
using *one-step-implies-mult* [*of* {# (*subst u* σ), (*subst v* σ) #}
{# (*subst s* σ),(*subst s* σ), (*subst v* σ),(*subst v* σ) #}
trm-ord {#}]
trm-ord-trans **by** *metis*
from *this* ‹$M' = L'$› ‹$k =$ (*mset-lit* (*subst-lit* M' σ))› **show** *?thesis* **by** *auto*
qed

qed
from *this d1 d2* ‹{# (*mset-lit* (*subst-lit x* σ)). $x \in$# (*mset-set* { *L2* }) #} $\neq$ {#}›
have *o*:
({#*mset-lit* (*subst-lit x* σ). $x \in$# *mset-set ?C* #},
{#*mset-lit* (*subst-lit x* σ). $x \in$# *mset-set* (*cl-ecl P1*)#})
$\in$ *mult* (*mult trm-ord*)

using *mult-trm-ord-trans one-step-implies-mult* [*of*
{# (*mset-lit* (*subst-lit* x σ)). x ∈# (*mset-set* { *L2* }) #}
{# (*mset-lit* (*subst-lit* x σ)). x ∈# (*mset-set* *?C′*) #} *mult trm-ord*
{# (*mset-lit* (*subst-lit* x σ)). x ∈# (*mset-set* ((*cl-ecl P1*) − { *L2* })) #}]
by *metis*

from *this* ‹(*Cl-P* = (*cl-ecl P1*))› ‹C' = ((*Cl-P* − { *L2* }) ∪ { L' })› **and** ‹*P1* ∈ P›
and *hyp* **show** *False* **by** *auto*
qed
from *not-sup not-ref not-fact* **and** *assms*(*1*) **show** *False* **unfolding** *derivable-def* **by** *blast*
qed

lemma *redundant-inference-clause*:
assumes *redundant-clause* E S σ C'
assumes *derivable* C P S σ *Ground* C'
assumes *grounding-set* P σ
assumes $\forall x \in S$. (*finite* (*cl-ecl* x))
shows *redundant-inference* E S P σ
proof −
from *assms*(*1*) **obtain** S' **where** $S' \subseteq$ (*instances* S)
(*set-entails-clause* (*clset-instances* S') (*cl-ecl* E))
($\forall x \in S'$. (*subterms-inclusion* (*subst-set* (*trms-ecl* (*fst* x)) (*snd* x))
(*trms-ecl* E))) **and**
ball-S′-C′-le: $\forall x \in S'$. (*mset-ecl* (*fst* x, *snd* x), *mset-cl* (C', σ)) ∈ *mult* (*mult* *trm-ord*) ∨
mset-ecl (*fst* x, *snd* x) = *mset-cl* (C', σ)
unfolding *redundant-clause-def* **by** *auto*
from *assms*(*3*) *assms*(*4*) ‹*derivable* C P S σ *Ground* C'›
obtain D **where** $D \in P$
(((*mset-cl* (C',σ)), (*mset-ecl* (D,σ))) ∈ (*mult* (*mult trm-ord*)))
using *conclusion-is-smaller-than-premisses* **by** *blast*

have $\forall x \in S'$. $\exists D' \in$ *cl-ecl* ‘ P. ((*cl-ecl* (*fst* x), *snd* x), (D', σ)) ∈ *cl-ord*
proof (*intro ballI*)
fix x **assume** $x \in S'$
have ((*cl-ecl* (*fst* x), *snd* x), (*cl-ecl* D, σ)) ∈ *cl-ord*
using *ball-S′-C′-le*[*rule-format*, *OF* ‹$x \in S'$›]
using ‹(*mset-cl* (C', σ), *mset-ecl* (D, σ)) ∈ *mult* (*mult trm-ord*)›
unfolding *cl-ord-def mem-Collect-eq prod.case mset-ecl-conv*
by (*metis mult-mult-trm-ord-trans*[*THEN transD*])
with ‹$D \in P$› **show** $\exists D' \in$ *cl-ecl* ‘ P. ((*cl-ecl* (*fst* x), *snd* x), (D', σ)) ∈ *cl-ord* **by** *auto*
qed
from *this* **and** ‹$S' \subseteq$ (*instances* S)› **and** ‹(*set-entails-clause* (*clset-instances* S') (*cl-ecl* E))›
and ‹($\forall x \in S'$. (*subterms-inclusion* (*subst-set* (*trms-ecl* (*fst* x)) (*snd* x))
(*trms-ecl* E)))›

show *?thesis* **unfolding** *redundant-inference-def*
by *auto*
qed

lemma *clause-saturated-and-inference-saturated*:
assumes *clause-saturated S*
assumes $\forall\, x \in S.\ (\mathit{finite}\ (\mathit{cl\text{-}ecl}\ x))$
shows *inference-saturated S*
proof (*rule ccontr*)
assume $\neg$ *inference-saturated S*
then obtain $C\ P\ \sigma\ C'\ D\ \ \vartheta\ \eta$
where *derivable C P S* σ *Ground* C' *ground-clause* (*cl-ecl C*)
derivable D P S ϑ *FirstOrder* C' *trms-subsumes D C* η
$\sigma \doteq \vartheta \lozenge \eta$ *grounding-set P* σ
$\neg$*redundant-inference* (*subst-ecl D* η) *S P* σ
unfolding *inference-saturated-def* **by** *blast*

from *assms*(*2*) ‹*grounding-set P* σ› ‹*derivable C P S* σ *Ground* C'›
‹$\neg$*redundant-inference* (*subst-ecl D* η) *S P* σ›
have $\neg$*redundant-clause* (*subst-ecl D* η) *S* σ C'
using *redundant-inference-clause* **by** *blast*

from *assms*(*1*) **have** $\bigwedge C\ P\ \sigma\ C'\ D\ \ \vartheta\ \eta.$
(*derivable C P S* σ *Ground* C') $\longrightarrow$ (*ground-clause* (*cl-ecl C*))
$\longrightarrow$ (*derivable D P S* ϑ *FirstOrder* C') $\longrightarrow$ (*trms-subsumes D C* η)
$\longrightarrow$ ($\sigma \doteq \vartheta \lozenge \eta$)
$\longrightarrow$ (*redundant-clause* (*subst-ecl D* η) *S* σ C') **unfolding** *clause-saturated-def*
by *blast*
from *this* **and** ‹*derivable C P S* σ *Ground* C'› ‹*ground-clause* (*cl-ecl C*)›
‹*derivable D P S* ϑ *FirstOrder* C'›
‹*trms-subsumes D C* η› ‹$\sigma \doteq \vartheta \lozenge \eta$› *assms*(*1*) **have** *redundant-clause* (*subst-ecl D* η) *S* σ C'
by *auto*
from *this* **and** ‹$\neg$*redundant-clause* (*subst-ecl D* η) *S* σ C'› **show** *False* **by** *auto*
qed

7 Refutational Completeness

We prove that our variant of the superposition calculus is complete under the redundancy criteria defined above. This is done as usual, by constructing a model of every saturated set not containing the empty clause.

7.1 Model Construction

We associate as usual every set of extended clauses with an interpretation. The interpretation is constructed in such a way that it is a model of the set of clauses if the latter is saturated and does not contain the empty clause. The interpretation is constructed by defining directly a normalization function mapping every term to its normal form, i.e., to the minimal equivalent term. Note that we do not consider sets of rewrite rules explicitly.

The next function associates every normalization function with the corresponding interpretation (two terms are in relation if they share the same normal form). The obtained relation is an interpretation if the normalization function is compatible with the term combination operator.

definition *same-values* :: $('a\ trm \Rightarrow 'a\ trm) \Rightarrow 'a\ trm \Rightarrow 'a\ trm \Rightarrow bool$
  where $(same\text{-}values\ f) =$
        $(\lambda x\ y.\ (f\ x) = (f\ y))$

definition *value-is-compatible-with-structure* :: $('a\ trm \Rightarrow 'a\ trm) \Rightarrow bool$
  where $(value\text{-}is\text{-}compatible\text{-}with\text{-}structure\ f) = (\forall\ t\ s.\ (f\ (Comb\ t\ s)) = (f\ (Comb\ (f\ t)\ (f\ s))))$

lemma *same-values-fo-int*:
  assumes *value-is-compatible-with-structure f*
  shows *fo-interpretation* (*same-values f*)
proof −
  let *?I* = (*same-values f*)
  have *ref*: *reflexive ?I* **unfolding** *same-values-def reflexive-def* **by** *simp*
  have *sym*: *symmetric ?I* **unfolding** *same-values-def symmetric-def* **by** *auto*
  have *trans*: *transitive ?I* **unfolding** *same-values-def transitive-def* **by** *auto*
  from *assms*(*1*) **have** *comp*: *compatible-with-structure ?I*
    unfolding *same-values-def*
      compatible-with-structure-def value-is-compatible-with-structure-def [*of f*]
   by *metis*
  from *ref trans sym comp* **have** *congruence ?I* **unfolding** *congruence-def equivalence-relation-def*
    by *auto*
  then show *?thesis* **unfolding** *fo-interpretation-def* **by** *auto*
qed

The normalization function is defined by mapping each term to a set of pairs. Intuitively, the second element of each pair represents the right hand side of a rule that can be used to rewrite the considered term, and the first element of the pair denotes its normal form. The value of the term is the first component of the pair with the smallest second component.

The following function returns the set of values for which the second component is minimal. We then prove that this set is non-empty and define a function returning an arbitrary chosen element.

definition *min-trms* :: $('a\ trm \times 'a\ trm)\ set \Rightarrow 'a\ trm\ set$
  where $(min\text{-}trms\ E) = (\{\ x.\ (\exists\ pair.\ (pair \in E$
  $\wedge (\forall pair' \in E.\ (snd\ pair', snd\ pair) \notin trm\text{-}ord)) \wedge x = fst\ pair)\ \})$

lemma *min-trms-not-empty*:

assumes $E \neq \{\}$
shows *min-trms* $E \neq \{\}$
proof −
from *assms(1)* **obtain** x **where** $x \in E$ **by** *auto*
let *?pair-ordering* = { (x,y). ((*snd* x),(*snd* y)) ∈ *trm-ord* }
from *trm-ord-wf* **have** *wf ?pair-ordering* **using** *measure-wf* **by** *auto*
from *this* ‹$x \in E$›
obtain y **where** $y \in E$ **and** ∀ z. (z,y) ∈ *?pair-ordering* ⟶ ($z \notin E$)
using *wfE-min* [*of ?pair-ordering*]
by *metis*
from *this* **have** *fst* y ∈ *min-trms* E **unfolding** *min-trms-def* **by** *blast*
then show *?thesis* **by** *auto*
qed

definition *get-min* :: *'a trm* ⇒ (*'a trm* × *'a trm*) *set* ⇒ *'a trm*
where (*get-min t E*) =
(*if* ((*min-trms E*) = {}) *then t else* (*SOME x*. (x ∈ *min-trms E*)))

We now define the normalization function. The definition is tuned to make the termination proof straightforward. We will reformulate it afterward to get a simpler definition.

We first test whether a subterm of the considered term is reducible. If this is the case then the value can be obtained by applying recursively the function on each subterm, and then again on the term obtained by combining the obtained normal forms. If not, then we collect all possible pairs (as explained above), and we use the one with the minimal second component. These pairs can be interpreted as rewrite rules, giving the value of the considered term: the second component is the right-hand side of the rule and the first component is the normal form of the right-hand side. As usual, such rewrite rules are obtained from ground clauses that have a strictly positive maximal literal, no selected literals, and that are not validated by the constructed interpretation.

function *trm-rep*:: *'a trm* ⇒ (*'a eclause set* ⇒ *'a trm*)
where
(*trm-rep t*) =
(λ*S*. (*if* ((*is-compound t*) ∧ ((*lhs t*),*t*) ∈ *trm-ord* ∧ ((*rhs t*),*t*) ∈ *trm-ord*
∧ (((*lhs t*,*t*) ∈ *trm-ord* ⟶ (*trm-rep* (*lhs t*) *S*) ≠ (*lhs t*))
∨ ((*rhs t*,*t*) ∈ *trm-ord* ⟶(*trm-rep* (*rhs t*) *S*) ≠ (*rhs t*))))
then (*if* ((*Comb* (*trm-rep* (*lhs t*) *S*) (*trm-rep* (*rhs t*) *S*)),*t*) ∈ *trm-ord*
then
(*trm-rep* (*Comb* (*trm-rep* (*lhs t*) *S*) (*trm-rep* (*rhs t*) *S*)) *S*)
else t)
else (*get-min t*
{ *pair*. ∃ *z CC C' C s L L'* σ *t' s'*.
pair = (*z*,*s*)
∧ *CC* ∈ *S* ∧ (*t* ∉ (*subst-set* (*trms-ecl CC*) σ))
∧ (∀ *x*. (∃ *x'* ∈ (*trms-ecl CC*). *occurs-in x* (*subst x'* σ))

$\longrightarrow$ ($(x,t) \in$ *trm-ord* $\longrightarrow$ (*trm-rep* x S) = x))
$\wedge$ (C' = (*cl-ecl CC*)) $\wedge$ $(s,t) \in$ *trm-ord* $\wedge$ (($s,t) \in$ *trm-ord* $\longrightarrow$ (z = *trm-rep* s S))
$\wedge$ (*orient-lit-inst* L' t' s' *pos* σ) $\wedge$ (*sel* C') = {} $\wedge$ ($L' \in C'$)
$\wedge$ (*maximal-literal* L C) $\wedge$ (L = (*subst-lit* L' σ)) $\wedge$ (C = (*subst-cl* C' σ))
$\wedge$ (*ground-clause* C) $\wedge$ (t = (*subst* t' σ)) $\wedge$ (s = (*subst* s' σ)) $\wedge$ (*finite* C')
$\wedge$
($\forall$ L u v.
($L \in C$ $\longrightarrow$ *orient-lit* L u v *pos*
$\longrightarrow$ $(u,t) \in$ *trm-ord* $\longrightarrow$ $(v,t) \in$ *trm-ord*
$\longrightarrow$ (*trm-rep* u S) $\neq$ (*trm-rep* v S))
$\wedge$
($L \in C$ $\longrightarrow$ *orient-lit* L u v *neg* $\longrightarrow$ $(u,t) \in$ *trm-ord* $\longrightarrow$ $(v,t) \in$ *trm-ord*
$\longrightarrow$ (*trm-rep* u S) = (*trm-rep* v S)))
$\wedge$ ($\forall$ s''. (
(*eq-occurs-in-cl* t s'' (C'− { L' }) σ) $\longrightarrow$ $(s'',t) \in$ *trm-ord* $\longrightarrow$ $(s,t) \in$ *trm-ord*
$\longrightarrow$ (*trm-rep* s S) $\neq$ (*trm-rep* s'' S))) })))
by *auto*
termination apply (*relation trm-ord*)
by *auto* (*simp add*: *trm-ord-wf*)

We now introduce a few shorthands and rewrite the previous definition into an equivalent simpler form. The key point is to prove that a term is always greater than its normal form.

definition *subterm-reduction-aux*:: $'a$ *eclause set* $\Rightarrow$ $'a$ *trm* $\Rightarrow$ $'a$ *trm*
where
subterm-reduction-aux S t =
(*if* ((*Comb* (*trm-rep* (*lhs* t) S) (*trm-rep* (*rhs* t) S)),t) $\in$ *trm-ord*
then (*trm-rep* (*Comb* (*trm-rep* (*lhs* t) S) (*trm-rep* (*rhs* t) S)) S)
else t)

definition *subterm-reduction*:: $'a$ *eclause set* $\Rightarrow$ $'a$ *trm* $\Rightarrow$ $'a$ *trm*
where
subterm-reduction S t =
(*trm-rep* (*Comb* (*trm-rep* (*lhs* t) S) (*trm-rep* (*rhs* t) S)) S)

definition *maximal-literal-is-unique*
where (*maximal-literal-is-unique* t s C' L' S σ) =
($\forall$ s''. (
(*eq-occurs-in-cl* t s'' (C'− { L' }) σ) $\longrightarrow$ $(s'',t) \in$ *trm-ord* $\longrightarrow$ $(s,t) \in$ *trm-ord*
$\longrightarrow$ (*trm-rep* s S) $\neq$ (*trm-rep* s'' S)))

definition *smaller-lits-are-false*
where (*smaller-lits-are-false* t C S) =
($\forall$ L u v.
($L \in C$ $\longrightarrow$ *orient-lit* L u v *pos*
$\longrightarrow$ $(u,t) \in$ *trm-ord* $\longrightarrow$ $(v,t) \in$ *trm-ord*

$\longrightarrow$ (*trm-rep u S*) $\neq$ (*trm-rep v S*))
$\wedge$
(*L* $\in$ *C* $\longrightarrow$ *orient-lit L u v neg* $\longrightarrow$ (*u*,*t*) $\in$ *trm-ord* $\longrightarrow$ (*v*,*t*) $\in$ *trm-ord*
$\longrightarrow$ (*trm-rep u S*) = (*trm-rep v S*)))

definition *int-clset*
where *int-clset S* = (*same-values* (λ*x*. (*trm-rep x S*)))

lemma *smaller-lits-are-false-if-cl-not-valid*:
assumes ¬(*validate-ground-clause* (*int-clset S*) *C*)
shows *smaller-lits-are-false t C S*
proof (*rule ccontr*)
let *?I* = *int-clset S*
assume ¬*smaller-lits-are-false t C S*
from *this* **obtain** *L u v* **where** *L* $\in$ *C*
and (*orient-lit L u v pos* $\wedge$ (*trm-rep u S*) = (*trm-rep v S*))
$\vee$ (*orient-lit L u v neg* $\wedge$ (*trm-rep u S*) $\neq$ (*trm-rep v S*))
unfolding *smaller-lits-are-false-def* **by** *blast*
then have (*orient-lit L u v pos* $\wedge$ (*trm-rep u S*) = (*trm-rep v S*))
$\vee$ (*orient-lit L u v neg* $\wedge$ (*trm-rep u S*) $\neq$ (*trm-rep v S*)) **by** *blast*
then show *False*
proof
assume *c-pos*: (*orient-lit L u v pos* $\wedge$ (*trm-rep u S*) = (*trm-rep v S*))
then have *orient-lit L u v pos* **by** *blast*
from *c-pos* **have** (*trm-rep u S*) = (*trm-rep v S*) **by** *blast*
from ‹*orient-lit L u v pos*› **have** *L* = (*Pos* (*Eq u v*)) $\vee$ *L* = (*Pos* (*Eq v u*))
unfolding *orient-lit-def* **by** *auto*
from *this* **and** ‹(*trm-rep u S*) = (*trm-rep v S*)› **have** *validate-ground-lit ?I L*
using *validate-ground-lit.simps*(*1*) *validate-ground-eq.simps*
unfolding *same-values-def int-clset-def*
by (*metis* (*mono-tags*, *lifting*))
from *this* **and** ‹*L* $\in$ *C*› **and** *assms* **show** *False* **unfolding** *int-clset-def*
using *validate-ground-clause.simps* **by** *blast*
next
assume *c-neg*: (*orient-lit L u v neg* $\wedge$ (*trm-rep u S*) $\neq$ (*trm-rep v S*))
then have *orient-lit L u v neg* **by** *blast*
from *c-neg* **have** (*trm-rep u S*) $\neq$ (*trm-rep v S*) **by** *blast*
from ‹*orient-lit L u v neg*› **have** *L* = (*Neg* (*Eq u v*)) $\vee$ *L* = (*Neg* (*Eq v u*))
unfolding *orient-lit-def* **by** *auto*
from *this* **and** ‹(*trm-rep u S*) $\neq$ (*trm-rep v S*)› **have** *validate-ground-lit ?I L*
using *validate-ground-lit.simps*(*2*) *validate-ground-eq.simps*
unfolding *same-values-def int-clset-def*
by (*metis* (*mono-tags*, *lifting*))
from *this* **and** ‹*L* $\in$ *C*› **and** *assms* **show** *False* **unfolding** *int-clset-def*
using *validate-ground-clause.simps* **by** *blast*
qed
qed

The following function states that all instances of the terms attached to a clause are in normal form w.r.t. the interpretation associated with S, up to some maximal term t

definition *trms-irreducible*
  where *trms-irreducible CC* σ *S t* =
    $(\forall x.\ (\exists x' \in (\textit{trms-ecl CC}).\ \textit{occurs-in}\ x\ (\textit{subst}\ x'\ \sigma)) \longrightarrow$
    $(\ (x,t) \in \textit{trm-ord} \longrightarrow (\textit{trm-rep}\ x\ S) = x))$

lemma *trms-irreducible-lemma*:
  assumes $\textit{all-trms-irreducible}\ (\textit{subst-set}\ (\textit{trms-ecl}\ C)\ \sigma)\ (\lambda t.\ \textit{trm-rep}\ t\ S)$
  shows $\textit{trms-irreducible}\ C\ \sigma\ S\ t$
proof (*rule ccontr*)
  assume $\neg\textit{trms-irreducible}\ C\ \sigma\ S\ t$
  from *this* **obtain** x **where** $\exists x' \in \textit{trms-ecl}\ C.\ \textit{occurs-in}\ x\ (\textit{subst}\ x'\ \sigma)$ **and**
    $\textit{trm-rep}\ x\ S \neq x$ **unfolding** *trms-irreducible-def* **by** *blast*
  from ‹$\exists x' \in \textit{trms-ecl}\ C.\ \textit{occurs-in}\ x\ (\textit{subst}\ x'\ \sigma)$› **obtain** x' **where**
    $x' \in \textit{trms-ecl}\ C$ **and** $\textit{occurs-in}\ x\ (\textit{subst}\ x'\ \sigma)$ **by** *blast*
  from ‹$x' \in \textit{trms-ecl}\ C$›
    have $(\textit{subst}\ x'\ \sigma) \in (\textit{subst-set}\ (\textit{trms-ecl}\ C)\ \sigma)$
    by *auto*
  from *this* **and** *assms(1)* ‹$\textit{occurs-in}\ x\ (\textit{subst}\ x'\ \sigma)$›
    have $\textit{trm-rep}\ x\ S = x$ **unfolding** *all-trms-irreducible-def* **by** *metis*
  from *this* **and** ‹$\textit{trm-rep}\ x\ S \neq x$› **show** *False* **by** *blast*
qed

The following predicate states that a term z is the normal form of the right-hand side of a rule of left-hand side t. It is used to define the set of possible values for term t. The actual value is that corresponding to the smallest right-hand side.

definition *candidate-values*
  where $(\textit{candidate-values}\ z\ CC\ C'\ C\ s\ L\ L'\ \sigma\ t'\ s'\ t\ S) =$
    $(CC \in S \wedge (t \notin (\textit{subst-set}\ (\textit{trms-ecl}\ CC)\ \sigma)) \wedge (\textit{trms-irreducible}\ CC\ \sigma\ S$
$t)$
    $\wedge\ (C' = (\textit{cl-ecl}\ CC)) \wedge (s,t) \in \textit{trm-ord} \wedge ((s,t) \in \textit{trm-ord} \longrightarrow (z = \textit{trm-rep}$
$s\ S))$
    $\wedge\ (\textit{orient-lit-inst}\ L'\ t'\ s'\ \textit{pos}\ \sigma) \wedge (\textit{sel}\ C' = \{\}) \wedge (L' \in C') \wedge (\textit{maximal-literal}$
$L\ C)$
    $\wedge\ (L = (\textit{subst-lit}\ L'\ \sigma)) \wedge (C = (\textit{subst-cl}\ C'\ \sigma)) \wedge (\textit{ground-clause}\ C)$
    $\wedge\ (t = (\textit{subst}\ t'\ \sigma)) \wedge\ (s = (\textit{subst}\ s'\ \sigma)) \wedge (\textit{finite}\ C')$
    $\wedge\ (\textit{smaller-lits-are-false}\ t\ C\ S)$
    $\wedge\ (\textit{maximal-literal-is-unique}\ t\ s\ C'\ L'\ S\ \sigma))$

definition *set-of-candidate-values*:: $'a$ *eclause set* $\Rightarrow$ $'a$ *trm* $\Rightarrow$ $('a\ \textit{trm} \times 'a\ \textit{trm})$
set
where *set-of-candidate-values S t* =
  $\{\ \textit{pair}.\ \exists z\ CC\ C'\ C\ s\ L\ L'\ \sigma\ t'\ s'.$
    $\textit{pair} = (z,s) \wedge (\textit{candidate-values}\ z\ CC\ C'\ C\ s\ L\ L'\ \sigma\ t'\ s'\ t\ S)\ \}$

definition *subterm-reduction-applicable-aux*:: $'a$ *eclause set* $\Rightarrow$ $'a$ *trm* $\Rightarrow$ *bool*

where *subterm-reduction-applicable-aux S t* =
(*is-compound t* ∧ (*lhs t*,*t*) ∈ *trm-ord* ∧ (*rhs t*,*t*) ∈ *trm-ord*
∧ (((*lhs t*,*t*) ∈ *trm-ord* ⟶ (*trm-rep* (*lhs t*) *S*) ≠ (*lhs t*))
∨ ((*rhs t*,*t*) ∈ *trm-ord* ⟶(*trm-rep* (*rhs t*) *S*) ≠ (*rhs t*))))

definition *subterm-reduction-applicable*:: *'a eclause set* ⇒ *'a trm* ⇒ *bool*
where *subterm-reduction-applicable S t* =
(*is-compound t* ∧ ((*trm-rep* (*lhs t*) *S*) ≠ (*lhs t*) ∨ (*trm-rep* (*rhs t*) *S*) ≠ (*rhs t*)))

lemma *trm-rep-is-lower-aux*:
assumes ∀ *y*. (*y*,*t*) ∈ *trm-ord* ⟶
(*y* ≠ (*trm-rep y S*) ⟶ ((*trm-rep y S*),*y*) ∈ *trm-ord*)
assumes (*subterm-reduction-applicable S t*)
shows ((*Comb* (*trm-rep* (*lhs t*) *S*) (*trm-rep* (*rhs t*) *S*)),*t*) ∈ *trm-ord*
proof −
have (*lhs t*,*t*) ∈ *trm-ord* **using** ‹*subterm-reduction-applicable S t*› *args-are-strictly-lower*

subterm-reduction-applicable-def
by *blast*
have (*rhs t*,*t*) ∈ *trm-ord* **using** ‹*subterm-reduction-applicable S t*› *args-are-strictly-lower*

subterm-reduction-applicable-def **by** *blast*
from *assms*(*1*) **and** ‹(*lhs t*,*t*) ∈ *trm-ord*› **have**
l: ((*lhs t* ≠ (*trm-rep* (*lhs t*) *S*)) ⟶ ((*trm-rep* (*lhs t*) *S*), (*lhs t*)) ∈ *trm-ord*)
by *metis*
from *assms*(*1*) **and** ‹(*rhs t*,*t*) ∈ *trm-ord*› **have**
r: (*rhs t* ≠ (*trm-rep* (*rhs t*) *S*) ⟶ ((*trm-rep* (*rhs t*) *S*), (*rhs t*)) ∈ *trm-ord*)
by *metis*
from ‹*subterm-reduction-applicable S t*›
have ((*trm-rep* (*lhs t*) *S*) ≠ (*lhs t*) ∨ (*trm-rep* (*rhs t*) *S*) ≠ (*rhs t*))
unfolding *subterm-reduction-applicable-def* [*of S t*] **by** *blast*
then show *?thesis*
proof
assume (*trm-rep* (*lhs t*) *S*) ≠ (*lhs t*)
from *this* **and** *l* **have** ((*trm-rep* (*lhs t*) *S*), (*lhs t*)) ∈ *trm-ord* **by** *metis*
from *this*
have *i*: ((*Comb* (*trm-rep* (*lhs t*) *S*) (*trm-rep* (*rhs t*) *S*)),(*Comb* (*lhs t*) (*trm-rep*
(*rhs t*) *S*)))
∈ *trm-ord* **using** *trm-ord-reduction-left* **by** *blast*
show ((*Comb* (*trm-rep* (*lhs t*) *S*) (*trm-rep* (*rhs t*) *S*)),*t*) ∈ *trm-ord*
proof (*cases*)
assume (*trm-rep* (*rhs t*) *S*) = (*rhs t*)
from *this*
and ‹((*Comb* (*trm-rep* (*lhs t*) *S*) (*trm-rep* (*rhs t*) *S*)),(*Comb* (*lhs t*) (*trm-rep*
(*rhs t*) *S*)))
∈ *trm-ord*›
show ((*Comb* (*trm-rep* (*lhs t*) *S*) (*trm-rep* (*rhs t*) *S*)),*t*) ∈ *trm-ord*
by (*metis assms*(*2*) *is-compound.elims*(*2*) *lhs.simps*(*1*)
rhs.simps(*1*) *subterm-reduction-applicable-def*)

next
  assume (*trm-rep* (*rhs t*) *S*) ≠ (*rhs t*)
  from *this* **and** *r* **have** ((*trm-rep* (*rhs t*) *S*), (*rhs t*)) ∈ *trm-ord* **by** *metis*
  from *this* **have** ((*Comb* (*lhs t*) (*trm-rep* (*rhs t*) *S*)), ((*Comb* (*lhs t*) (*rhs t*)))) ∈ *trm-ord*
    using *trm-ord-reduction-right* **by** *blast*
  from *this* **and** *i* **show** ((*Comb* (*trm-rep* (*lhs t*) *S*) (*trm-rep* (*rhs t*) *S*)),*t*) ∈ *trm-ord*
    by (*metis assms*(*2*) *is-compound.elims*(*2*) *lhs.simps*(*1*) *rhs.simps*(*1*)
      subterm-reduction-applicable-def trm-ord-trans transE)
qed
next
  assume (*trm-rep* (*rhs t*) *S*) ≠ (*rhs t*)
from *this* **and** *r* **have** ((*trm-rep* (*rhs t*) *S*), (*rhs t*)) ∈ *trm-ord* **by** *metis*
from *this*
  have *i*: ((*Comb* (*trm-rep* (*lhs t*) *S*) (*trm-rep* (*rhs t*) *S*)),(*Comb* (*trm-rep* (*lhs t*) *S*) (*rhs t*)))
      ∈ *trm-ord* **using** *trm-ord-reduction-right* **by** *blast*
show ((*Comb* (*trm-rep* (*lhs t*) *S*) (*trm-rep* (*rhs t*) *S*)),*t*) ∈ *trm-ord*
proof (*cases*)
  assume (*trm-rep* (*lhs t*) *S*) = (*lhs t*)
  from *this* **and**
  ‹((*Comb* (*trm-rep* (*lhs t*) *S*) (*trm-rep* (*rhs t*) *S*)),(*Comb* (*trm-rep* (*lhs t*) *S*) (*rhs t*))) ∈ *trm-ord*›
  show ((*Comb* (*trm-rep* (*lhs t*) *S*) (*trm-rep* (*rhs t*) *S*)),*t*) ∈ *trm-ord*
    by (*metis assms*(*2*) *basic-superposition.subterm-reduction-applicable-def*
      basic-superposition-axioms is-compound.elims(*2*) *lhs.simps*(*1*) *rhs.simps*(*1*))
next
  assume (*trm-rep* (*lhs t*) *S*) ≠ (*lhs t*)
  from *this* **and** *l* **have** ((*trm-rep* (*lhs t*) *S*), (*lhs t*)) ∈ *trm-ord* **by** *metis*
  from *this* **have** ((*Comb* (*trm-rep* (*lhs t*) *S*) (*rhs t*)), ((*Comb* (*lhs t*) (*rhs t*)))) ∈ *trm-ord*
    using *trm-ord-reduction-left* **by** *blast*
  from *this* **and** *i* **show** ((*Comb* (*trm-rep* (*lhs t*) *S*) (*trm-rep* (*rhs t*) *S*)),*t*) ∈ *trm-ord*
    by (*metis assms*(*2*) *basic-superposition.subterm-reduction-applicable-def*
      basic-superposition-axioms is-compound.elims(*2*) *lhs.simps*(*1*) *rhs.simps*(*1*)

      trm-ord-trans transE)
qed
qed
qed

The following lemma corresponds to the initial definition of the function *trm-rep*.

lemma *trm-rep-init-def*:
  shows (*trm-rep t*) = (λ*S*. (*if* (*subterm-reduction-applicable-aux S t*)
      then (*subterm-reduction-aux S t*)
      else (*get-min t* (*set-of-candidate-values S t*))))

unfolding *subterm-reduction-aux-def set-of-candidate-values-def candidate-values-def*

subterm-reduction-applicable-aux-def maximal-literal-is-unique-def smaller-lits-are-false-def

trms-irreducible-def
using *trm-rep.simps* [*of t*] **by** *force*

lemma *trm-rep-aux-def*:
assumes ∀ *y*. (*y*,*t*) ∈ *trm-ord* ⟶
(*y* ≠ (*trm-rep y S*) ⟶ ((*trm-rep y S*),*y*) ∈ *trm-ord*)
shows (*trm-rep t S*) = (*if* (*subterm-reduction-applicable S t*)
then (*subterm-reduction S t*)
else (*get-min t* (*set-of-candidate-values S t*)))
proof (*cases*)
assume *subterm-reduction-applicable S t*
then have *subterm-reduction-applicable-aux S t*
using *args-are-strictly-lower*
subterm-reduction-applicable-def subterm-reduction-applicable-aux-def **by** *blast*

from *this* **have** (*trm-rep t S*) = (*subterm-reduction-aux S t*)
using *trm-rep-init-def* [*of t*] **by** *meson*
then have ((*Comb* (*trm-rep* (*lhs t*) *S*) (*trm-rep* (*rhs t*) *S*)),*t*) ∈ *trm-ord*
using ‹*subterm-reduction-applicable S t*› *assms trm-rep-is-lower-aux* **by** *blast*
then show *?thesis*
by (*metis* ‹*trm-rep t S* = *subterm-reduction-aux S t*›
‹*subterm-reduction-applicable S t*›
subterm-reduction-def
subterm-reduction-aux-def)
next
assume ¬*subterm-reduction-applicable S t*
then have ¬*subterm-reduction-applicable-aux S t*
using *subterm-reduction-applicable-def subterm-reduction-applicable-aux-def* **by**
blast
from *this* **and** ‹¬*subterm-reduction-applicable S t*› **show** *?thesis*
by (*meson trm-rep-init-def*)
qed

lemma *trm-rep-is-lower*:
shows (*t* ≠ (*trm-rep t S*)) ⟶ (((*trm-rep t S*),*t*) ∈ *trm-ord*) (**is** *?P t*)
proof ((*rule wf-induct* [*of trm-ord ?P t*]),(*simp add*: *trm-ord-wf*))
next
fix *x* **assume** *hyp-ind*: ∀ *y*. (*y*,*x*) ∈ *trm-ord* ⟶ (*?P y*)
let *?v* = (*Comb* (*trm-rep* (*lhs x*) *S*) (*trm-rep* (*rhs x*) *S*))
show (*?P x*)
proof (*rule impI*)
assume *x* ≠ (*trm-rep x S*)
show ((*trm-rep x S*),*x*) ∈ *trm-ord*
proof *cases*
assume *c1*: *subterm-reduction-applicable S x*

from *this* **and** *hyp-ind*
have (*?v*,*x*) ∈ *trm-ord* **using** *trm-rep-is-lower-aux* **by** *metis*
from *c1* **and** *hyp-ind* **have** (*trm-rep x S*) = (*subterm-reduction S x*)
using *trm-rep-aux-def* [*of x S*] **by** *metis*
from *this* **have** (*trm-rep x S*) = (*trm-rep ?v S*)
unfolding *subterm-reduction-def* **by** *metis*
from ‹(*?v*,*x*) ∈ *trm-ord*› **and** *hyp-ind* **have** *?P ?v* **by** *metis*
from *this* **and** ‹(*trm-rep x S*) = (*trm-rep ?v S*)› **show** *?thesis*
by (*metis* ‹(*trm-rep* (*lhs x*) *S* · *trm-rep* (*rhs x*) *S*, *x*) ∈ *trm-ord*› *trm-ord-trans transE*)
next assume *c2*: ¬*subterm-reduction-applicable S x*
from *c2* **and** *hyp-ind* **have** (*trm-rep x S*) = (*get-min x* (*set-of-candidate-values S x*))
using *trm-rep-aux-def* [*of x S*] **by** *metis*
from *this* **and** ‹*x* ≠ (*trm-rep x S*)›
have (*trm-rep x S*) ∈ (*min-trms* (*set-of-candidate-values S x*))
unfolding *get-min-def* **by** (*metis* (*full-types*) *some-in-eq*)
then obtain *pair* **where** *pair* ∈ (*set-of-candidate-values S x*) (*trm-rep x S*) = *fst pair*
unfolding *min-trms-def* **by** *blast*
from ‹*pair* ∈ (*set-of-candidate-values S x*)›
have
∃ *CC C′ C L L′ σ t′ s′*. *candidate-values* (*fst pair*) *CC C′ C* (*snd pair*) *L L′ σ t′ s′ x S*
unfolding *set-of-candidate-values-def* **by** *fastforce*
from *this* **have** (*snd pair*,*x*) ∈ *trm-ord* **unfolding** *candidate-values-def* **by** *blast*
from
‹∃ *CC C′ C L L′ σ t′ s′*. *candidate-values* (*fst pair*) *CC C′ C* (*snd pair*) *L L′ σ t′ s′ x S*›
have ((*snd pair*, *x*) ∈ *trm-ord* ⟶ *fst pair* = *trm-rep* (*snd pair*) *S*)
unfolding *candidate-values-def* **by** *blast*
from ‹(*snd pair*,*x*) ∈ *trm-ord*› ‹((*snd pair*, *x*) ∈ *trm-ord* ⟶ *fst pair* = *trm-rep* (*snd pair*) *S*)›
have *fst pair* = *trm-rep* (*snd pair*) *S* **by** *blast*
from ‹(*snd pair*,*x*) ∈ *trm-ord*› **and** *hyp-ind* **have** (*?P* (*snd pair*)) **by** *blast*
from *this* **and** ‹*fst pair* = (*trm-rep* (*snd pair*) *S*)›
have *fst pair* = *snd pair* ∨ (*fst pair*,*snd pair*) ∈ *trm-ord*
by *metis*
from *this* **and** ‹(*trm-rep x S*) = *fst pair*› **and** ‹(*snd pair*,*x*) ∈ *trm-ord*› ‹*x* ≠ (*trm-rep x S*)›
show *?thesis* **by** (*metis trm-ord-trans transD*)
qed
qed
qed

lemma *trm-rep-is-lower-subt-red*:
assumes (*subterm-reduction-applicable S x*)
shows ((*trm-rep x S*),*x*) ∈ *trm-ord*

proof −
let *?v* = (*Comb* (*trm-rep* (*lhs x*) *S*) (*trm-rep* (*rhs x*) *S*))
from *assms*(*1*)
have (*?v*,*x*) ∈ *trm-ord* **using** *trm-rep-is-lower-aux trm-rep-is-lower* **by** *metis*
from *assms*(*1*) **have** (*trm-rep x S*) = (*subterm-reduction S x*)
using *trm-rep-aux-def* [*of x S*] *trm-rep-is-lower* **by** *metis*
from *this* **have** (*trm-rep x S*) = (*trm-rep ?v S*)
unfolding *subterm-reduction-def* **by** *metis*
have *?v* = *trm-rep ?v S* ∨ (*trm-rep ?v S*,*?v*) ∈ *trm-ord* **using** *trm-rep-is-lower* **by** *blast*
from *this* **and** ‹(*trm-rep x S*) = (*trm-rep ?v S*)› **show** (((*trm-rep x S*),*x*) ∈ *trm-ord*)
by (*metis* ‹(*trm-rep* (*lhs x*) *S* · *trm-rep* (*rhs x*) *S*, *x*) ∈ *trm-ord*› *trm-ord-trans transE*)
qed

lemma *trm-rep-is-lower-root-red*:
assumes ¬(*subterm-reduction-applicable S x*)
assumes *min-trms* (*set-of-candidate-values S x*) ≠ {}
shows (((*trm-rep x S*),*x*) ∈ *trm-ord*)
proof −
from *assms*(*1*) **have** (*trm-rep x S*) = (*get-min x* (*set-of-candidate-values S x*))
using *trm-rep-aux-def* [*of x S*] *trm-rep-is-lower* **by** *metis*
from *this* **and** *assms*(*2*) **have** (*trm-rep x S*) ∈ (*min-trms* (*set-of-candidate-values S x*))
unfolding *get-min-def* **by** (*metis* (*full-types*) *some-in-eq*)
then obtain *pair* **where** *pair* ∈ (*set-of-candidate-values S x*) **and** (*trm-rep x S*) = *fst pair*
unfolding *min-trms-def* **by** *blast*
from ‹*pair* ∈ (*set-of-candidate-values S x*)›
have ∃ *CC C′ C L L′ σ t′ s′*. *candidate-values* (*fst pair*) *CC C′ C* (*snd pair*) *L L′ σ t′ s′ x S*
unfolding *set-of-candidate-values-def* **by** *fastforce*
from *this* **have** (*snd pair*,*x*) ∈ *trm-ord* **unfolding** *candidate-values-def* **by** *blast*
from ‹∃ *CC C′ C L L′ σ t′ s′*. *candidate-values* (*fst pair*) *CC C′ C* (*snd pair*) *L L′ σ t′ s′ x S*›
have ((*snd pair*, *x*) ∈ *trm-ord* ⟶ *fst pair* = *trm-rep* (*snd pair*) *S*)
unfolding *candidate-values-def* **by** *blast*
from ‹(*snd pair*,*x*) ∈ *trm-ord*› **and** ‹((*snd pair*, *x*) ∈ *trm-ord* ⟶ *fst pair* = *trm-rep* (*snd pair*) *S*)›
have *fst pair* = *trm-rep* (*snd pair*) *S* **by** *blast*
have *snd pair* = *trm-rep* (*snd pair*) *S* ∨ (*trm-rep* (*snd pair*) *S*,*snd pair*) ∈ *trm-ord*
using *trm-rep-is-lower* **by** *blast*
from *this* **and** ‹(*snd pair*,*x*) ∈ *trm-ord*› **have** (*trm-rep* (*snd pair*) *S*,*x*) ∈ *trm-ord*
using *trm-ord-trans trans-def* **by** *metis*
from *this* **and** ‹(*trm-rep x S*) = *fst pair*› **and** ‹*fst pair* = *trm-rep* (*snd pair*) *S*› **show** *?thesis*
by *metis*

qed

Finally, the next lemma gives a simpler and more convenient definition of the function *trm-rep*.

lemma *trm-rep-simp-def*:
  shows (*trm-rep t S*) = (*if* (*subterm-reduction-applicable S t*)
                    then (*subterm-reduction S t*)
                    else (*get-min t* (*set-of-candidate-values S t*)))
using *trm-rep-is-lower trm-rep-aux-def* **by** *blast*

We now establish some useful properties of the normalization function.

lemma *trm-rep-involutive*:
  shows (*trm-rep* (*trm-rep t S*) *S*) = (*trm-rep t S*) (**is** *?P t*)
proof ((*rule wf-induct* [*of trm-ord ?P t*]),(*simp add*: *trm-ord-wf*))
next
  fix *x* **assume** *hyp-ind*: ∀ *y*. (*y*,*x*) ∈ *trm-ord* ⟶ (*?P y*)
  let *?v* = (*Comb* (*trm-rep* (*lhs x*) *S*) (*trm-rep* (*rhs x*) *S*))
  show (*?P x*)
   proof *cases*
    assume *c1*: *subterm-reduction-applicable S x*
    from *this* **and** *hyp-ind*
      have (*?v*,*x*) ∈ *trm-ord* **using** *trm-rep-is-lower-aux trm-rep-is-lower* **by** *metis*
    from *this hyp-ind* **have** (*trm-rep* (*trm-rep ?v S*) *S*) = (*trm-rep ?v S*)
     using *trm-rep-aux-def* [*of x S*] **by** *metis*
    from *c1* **have** *trm-rep x S* = *trm-rep ?v S*
     using *trm-rep-simp-def* [*of x S*] **unfolding** *subterm-reduction-def* **by** *metis*
    from *this* **and** ‹(*trm-rep* (*trm-rep ?v S*) *S*) = (*trm-rep ?v S*)› ‹*trm-rep x S* = *trm-rep ?v S*›
     show *?thesis* **by** *metis*
   next assume *c2*: ¬*subterm-reduction-applicable S x*
    from *c2* **have** (*trm-rep x S*) = (*get-min x* (*set-of-candidate-values S x*))
     using *trm-rep-simp-def* [*of x S*] **by** *metis*
    show *?thesis*
    proof (*rule ccontr*)
     assume (*trm-rep* (*trm-rep x S*) *S*) ≠ (*trm-rep x S*)
     from *this* **have** *x* ≠ (*trm-rep x S*) **by** *metis*
     from *c2* **and** ‹*x* ≠ (*trm-rep x S*)›
      have (*trm-rep x S*) ∈ (*min-trms* (*set-of-candidate-values S x*))
      using *trm-rep-simp-def* [*of x S*]
      unfolding *get-min-def* **by** (*metis* (*full-types*) *some-in-eq*)
     then obtain *pair* **where**
      pair ∈ (*set-of-candidate-values S x*) **and** (*trm-rep x S*) = *fst pair*
      unfolding *min-trms-def* **by** *blast*
     from ‹*pair* ∈ (*set-of-candidate-values S x*)›
      have *i*: ∃ *CC C′ C L L′ σ t′ s′*.
         candidate-values (*fst pair*) *CC C′ C* (*snd pair*) *L L′ σ t′ s′ x S*
      unfolding *set-of-candidate-values-def*
      by *fastforce*

from *this* **have** (*snd pair*,*x*) ∈ *trm-ord* **unfolding** *candidate-values-def* **by** *blast*
from *i* **have** ((*snd pair*, *x*) ∈ *trm-ord* ⟶ *fst pair* = *trm-rep* (*snd pair*) *S*)
unfolding *candidate-values-def* **by** *blast*
from ‹(*snd pair*,*x*) ∈ *trm-ord*›
and ‹((*snd pair*, *x*) ∈ *trm-ord* ⟶ *fst pair* = *trm-rep* (*snd pair*) *S*)›
have *fst pair* = *trm-rep* (*snd pair*) *S* **by** *blast*
from ‹(*snd pair*,*x*) ∈ *trm-ord*› **and** *hyp-ind* **have** (*?P* (*snd pair*)) **by** *blast*
from *this* **and** ‹*fst pair* = (*trm-rep* (*snd pair*) *S*)› **and** ‹(*trm-rep x S*) = *fst pair*›
and ‹(*trm-rep* (*trm-rep x S*) *S*) ≠ (*trm-rep x S*)›
show *False* **by** *metis*
qed
qed
qed

The following predicate is true if all proper subterms are in normal form.

definition *root-term* :: *'a eclause set* ⇒ *'a trm* ⇒ *bool*
where
(*root-term S t*) =
((*trm-rep t S*) = (*get-min t* (*set-of-candidate-values S t*)))

The following function checks that the considered term contains a subterm that can be reduced.

definition *st-red* :: *'a eclause set* ⇒ *'a trm* ⇒ *bool*
where
(*st-red S t*)
= (∃ *t' p*. ((*subterm t p t'*) ∧ (*root-term S t'*) ∧ (*trm-rep t' S* ≠ *t'*)))

lemma *red-arg-implies-red-trm* :
assumes *st-red S t1*
assumes *t* = (*Comb t1 t2*) ∨ *t* = (*Comb t2 t1*)
shows *st-red S t*
proof −
from *assms*(*1*) **obtain** *t' p* **where** *subterm t1 p t'* **and** *root-term S t'* **and** *trm-rep t' S* ≠ *t'*
unfolding *st-red-def* **by** *blast*
from ‹*subterm t1 p t'*› **and** *assms*(*2*) **obtain** *q* **where** *subterm t q t'*
by (*metis subterm.simps*(*4*) *subterm.simps*(*5*))
from *this* **and** ‹*root-term S t'*› **and** ‹*trm-rep t' S* ≠ *t'*›
show *?thesis* **unfolding** *st-red-def* **by** *blast*
qed

lemma *subterms-of-irred-trms-are-irred*:
(*trm-rep t S*) ≠ *t* ⟶ *st-red S t* (**is** (*?P t*))
proof ((*rule wf-induct* [*of trm-ord ?P t*]),(*simp add*: *trm-ord-wf*))
next
fix *x* **assume** *hyp-ind*: ∀ *y*. (*y*,*x*) ∈ *trm-ord* ⟶ (*?P y*)

show (*?P x*)
proof (*rule impI*)
assume (*trm-rep x S*) ≠ *x*
show *st-red S x*
proof (*rule ccontr*)
assume *neg-h*: ¬*st-red S x*
have *i*: ¬*subterm-reduction-applicable S x*
proof
assume *decomp-case*: *subterm-reduction-applicable S x*
then obtain *x1 x2* **where** *x* = (*Comb x1 x2*) **using** *is-compound.elims*(*2*)
unfolding *subterm-reduction-applicable-def* **by** *blast*
from *this* **and** *decomp-case* **have** ((*trm-rep x1 S*) ≠ *x1* ∨ (*trm-rep x2 S*) ≠ *x2*)
using *lhs.simps*(*1*) *rhs.simps*(*1*)
unfolding *subterm-reduction-applicable-def* **by** *metis*
then show *False*
proof
assume (*trm-rep x1 S*) ≠ *x1*
from ‹*x* = (*Comb x1 x2*)› **and** *trm-ord-subterm* **have** (*x1*,*x*) ∈ *trm-ord* **by** *auto*
from *this* **and** *hyp-ind* **and** ‹(*trm-rep x1 S*) ≠ *x1*›
have *st-red S x1* **by** *blast*
from *this* **and** *neg-h* **and** ‹*x* = (*Comb x1 x2*)› **show** *False*
using *red-arg-implies-red-trm* [*of S x1 x x2*] **by** *blast*
next
assume (*trm-rep x2 S*) ≠ *x2*
from ‹*x* = (*Comb x1 x2*)› **and** *trm-ord-subterm* **have** (*x2*,*x*) ∈ *trm-ord* **by** *auto*
from *this* **and** *hyp-ind* **and** ‹(*trm-rep x2 S*) ≠ *x2*›
have *st-red S x2* **by** *metis*
from *this* **and** *neg-h* **and** ‹*x* = (*Comb x1 x2*)› **show** *False*
using *red-arg-implies-red-trm* [*of S x2 x x1*] **by** *blast*
qed
qed
then have (*trm-rep x S*) = (*get-min x* (*set-of-candidate-values S x*))
using *trm-rep-simp-def* **by** *metis*
then have *root-term S x* **unfolding** *root-term-def* **by** *blast*
have *subterm x* [] *x* **by** *auto*
from *this* **and** ‹*root-term S x*› **and** ‹(*trm-rep x S*) ≠ *x*›**have**
st-red S x **unfolding** *st-red-def* **by** *blast*
from *this* **and** *neg-h* **show** *False* **by** *auto*
qed
qed
qed

lemma *trm-rep-compatible-with-structure*:
shows *value-is-compatible-with-structure* (λ*x*. *trm-rep x S*)
proof (*rule ccontr*)
assume ¬*value-is-compatible-with-structure* (λ*x*. *trm-rep x S*)

from *this* **obtain** $t\ s$
where *neg-h*:$trm\text{-}rep\ (Comb\ t\ s)\ S \neq (trm\text{-}rep\ (Comb\ (trm\text{-}rep\ t\ S)\ (trm\text{-}rep\ s\ S))\ S)$
unfolding *value-is-compatible-with-structure-def* **by** *blast*
from *this* **have** $(trm\text{-}rep\ t\ S) \neq t \vee (trm\text{-}rep\ s\ S) \neq s$ **by** *metis*
from *this* **have** *subterm-reduction-applicable* $S\ (Comb\ t\ s)$
unfolding *subterm-reduction-applicable-def*
by (*metis is-compound.simps*(*3*) *lhs.simps*(*1*) *rhs.simps*(*1*))
from *this* **have** $(trm\text{-}rep\ (Comb\ t\ s)\ S) = (subterm\text{-}reduction\ S\ (Comb\ t\ s))$
using *trm-rep-simp-def* **by** *metis*
from *this* **and** *neg-h* **show** *False* **unfolding** *subterm-reduction-def*
by (*metis lhs.simps*(*1*) *rhs.simps*(*1*))
qed

The following function checks that a position can be reduced, taking into account the order on positions associated with the considered clause and term. A term is reducible when all terms occurring at smaller positions are irreducible.

definition *minimal-redex*
where $minimal\text{-}redex\ p\ t\ C\ S\ t'$
$= (\forall\ q\ s.\ ((q,p) \in (pos\text{-}ord\ C\ t') \longrightarrow (subterm\ t\ q\ s) \longrightarrow (trm\text{-}rep\ s\ S = s)))$

The next function checks that a given clause contains two equations with the same left-hand side and whose right-hand sides are equivalent in a given interpretation. If no such equations exist then it is clear that the maximal literal is necessarily unique.

definition *equivalent-eq-exists*
where $equivalent\text{-}eq\text{-}exists\ t\ s\ C\ I\ \sigma\ L1 = (\exists\ L\in\ C - \{\ L1\ \}.\ \exists\ u\ v.$
$(orient\text{-}lit\text{-}inst\ L\ u\ v\ pos\ \sigma) \wedge ((subst\ t\ \sigma) = (subst\ u\ \sigma))$
$\wedge (I\ \ (subst\ s\ \sigma)\ (subst\ v\ \sigma)))$

lemma *maximal-literal-is-unique-lemma*:
assumes $\neg equivalent\text{-}eq\text{-}exists\ t\ s\ C\ (int\text{-}clset\ S)\ \sigma\ L1$
shows $maximal\text{-}literal\text{-}is\text{-}unique\ (subst\ t\ \sigma)\ (subst\ s\ \sigma)\ C\ L1\ S\ \sigma$
proof (*rule ccontr*)
let $?t = (subst\ t\ \sigma)$
let $?s = (subst\ s\ \sigma)$
let $?L = (subst\text{-}lit\ L\ \sigma)$
let $?C = (subst\text{-}cl\ C\ \sigma)$

assume $\neg(maximal\text{-}literal\text{-}is\text{-}unique\ ?t\ ?s\ C\ L1\ S\ \sigma)$
from *this* **obtain** s'' **where** $(eq\text{-}occurs\text{-}in\text{-}cl\ ?t\ s''\ (C- \{\ L1\ \})\ \sigma)$
and $(trm\text{-}rep\ ?s\ \ S) = (trm\text{-}rep\ s''\ S)$ **unfolding** *maximal-literal-is-unique-def* **by** *blast*
from ‹$(eq\text{-}occurs\text{-}in\text{-}cl\ ?t\ s''\ (C- \{\ L1\ \})\ \sigma)$›
obtain $L'\ t'\ s'$ **where** $L' \in (C- \{\ L1\ \})$
and $orient\text{-}lit\text{-}inst\ L'\ t'\ s'\ pos\ \sigma$ **and** $(subst\ t'\ \sigma) = ?t$
and $s'' = subst\ s'\ \sigma$

unfolding *eq-occurs-in-cl-def* **by** *auto*
from ‹$s'' = subst\ s'\ \sigma$› **and** ‹($trm\text{-}rep\ ?s\ \ S$) = ($trm\text{-}rep\ s''\ S$)›
have ($trm\text{-}rep\ ?s\ \ S$) = ($trm\text{-}rep$ ($subst\ s'\ \sigma$) S) **by** *blast*
from ‹$L' \in (C - \{ L1 \})$› ‹*orient-lit-inst* $L'\ t'\ s'\ pos\ \sigma$› ‹($subst\ t'\ \sigma$) = $?t$›
‹($trm\text{-}rep\ ?s\ \ S$) = ($trm\text{-}rep$ ($subst\ s'\ \sigma$) S)›
have *equivalent-eq-exists* $t\ s\ C$ (*int-clset* S) σ $L1$
unfolding *equivalent-eq-exists-def same-values-def int-clset-def*
by *metis*
from *this* **and** *assms*(*1*) **show** *False* **by** *blast*
qed

lemma *max-pos-lit-dominates-cl*:
assumes *maximal-literal* (*subst-lit* $L\ \sigma$) (*subst-cl* $C\ \sigma$)
assumes *orient-lit-inst* $L\ t\ s\ pos\ \sigma$
assumes $L' \in C - \{ L \}$
assumes ¬*equivalent-eq-exists* $t\ s\ C\ I\ \sigma\ L$
assumes *vars-of-lit* (*subst-lit* $L\ \sigma$) = {}
assumes *vars-of-lit* (*subst-lit* $L'\ \sigma$) = {}
assumes *fo-interpretation* I
shows ((*subst-lit* $L'\ \sigma$),(*subst-lit* $L\ \sigma$)) ∈ *lit-ord*
proof −
let $?L'$ = (*subst-lit* $L'\ \sigma$)
let $?L$ = (*subst-lit* $L\ \sigma$)
let $?t$ = (*subst* $t\ \sigma$)
let $?s$ = (*subst* $s\ \sigma$)

from *assms*(*2*) **have** ($?t$,$?s$) ∉ *trm-ord* **unfolding** *orient-lit-inst-def* **by** *auto*
obtain $u'\ v'$ **where** L' = (*Pos* (*Eq* $u'\ v'$)) ∨ L' = (*Neg* (*Eq* $u'\ v'$))
using *literal.exhaust equation.exhaust* **by** *metis*
from *this* **obtain** *polarity* $u\ v$ **where** *orient-lit-inst* $L'\ u\ v$ *polarity* σ
and ((*subst* $u\ \sigma$),(*subst* $v\ \sigma$)) ∉ *trm-ord* **using**
trm-ord-trans trm-ord-irrefl **unfolding** *trans-def irrefl-def orient-lit-inst-def* **by**
metis
let $?u$ = (*subst* $u\ \sigma$)
let $?v$ = (*subst* $v\ \sigma$)
from ‹*orient-lit-inst* $L'\ u\ v$ *polarity* σ› **have** *orient-lit* $?L'\ ?u\ ?v$ *polarity*
using *lift-orient-lit* **by** *auto*
from ‹*orient-lit-inst* $L\ t\ s\ pos\ \sigma$› **have** *orient-lit* $?L\ ?t\ ?s$ *pos*
using *lift-orient-lit* **by** *auto*

from *assms*(*6*) **and** ‹*orient-lit* $?L'\ ?u\ ?v$ *polarity*›
have *vars-of* $?u$ ⊆ {} **using** *orient-lit-vars* **by** *metis*
from *assms*(*6*) **and** ‹*orient-lit* $?L'\ ?u\ ?v$ *polarity*›
have *vars-of* $?v$ ⊆ {} **using** *orient-lit-vars* **by** *metis*

from *assms*(*5*) **and** ‹*orient-lit* $?L\ ?t\ ?s$ *pos*›
have *vars-of* $?t$ ⊆ {} **using** *orient-lit-vars* **by** *metis*
from *assms*(*5*) **and** ‹*orient-lit* $?L\ ?t\ ?s$ *pos*›
have *vars-of* $?s$ ⊆ {} **using** *orient-lit-vars* **by** *metis*

from *assms*(*1*) **and** ‹$L' \in C - \{ L \}$› **have** (*?L*,*?L'*) ∉ *lit-ord*
unfolding *maximal-literal-def* **by** *auto*
from *this* **and** ‹*orient-lit ?L ?t ?s pos*› ‹*orient-lit ?L' ?u ?v polarity*› **and** *assms*(*5*) *assms*(*6*)
have (*?t*,*?u*) ∉ *trm-ord* **using** *lit-ord-dominating-term* **by** *metis*
from *this* **and** ‹*vars-of ?t* ⊆ {}› ‹*vars-of ?u* ⊆ {}› **have** *?u* = *?t* ∨ (*?u*,*?t*) ∈ *trm-ord*
using *trm-ord-ground-total* **unfolding** *ground-term-def* **by** *blast*
from ‹(*?u*,*?v*) ∉ *trm-ord*› **and** ‹*vars-of ?u* ⊆ {}› ‹*vars-of ?v* ⊆ {}›
have *?u* = *?v* ∨ (*?v*,*?u*) ∈ *trm-ord*
using *trm-ord-ground-total* **unfolding** *ground-term-def* **by** *blast*
from ‹(*?t*,*?s*) ∉ *trm-ord*› **and** ‹*vars-of ?t* ⊆ {}› ‹*vars-of ?s* ⊆ {}›
have *?t* = *?s* ∨ (*?s*,*?t*) ∈ *trm-ord*
using *trm-ord-ground-total* **unfolding** *ground-term-def* **by** *blast*
from ‹*vars-of ?v* ⊆ {}› ‹*vars-of ?s* ⊆ {}› **have** *?v* = *?s* ∨ (*?v*,*?s*) ∈ *trm-ord* ∨ (*?s*,*?v*) ∈ *trm-ord*
using *trm-ord-ground-total* **unfolding** *ground-term-def* **by** *blast*

show *?thesis*
proof (*cases*)
assume (*?u*,*?t*) ∈ *trm-ord*
from *this* **and** ‹*?u* = *?v* ∨ (*?v*,*?u*) ∈ *trm-ord*› **have** (*?v*,*?t*) ∈ *trm-ord*
using *trm-ord-trans* **unfolding** *trans-def* **by** *auto*
from *this* **and** ‹(*?u*,*?t*) ∈ *trm-ord*› **and** ‹*orient-lit ?L ?t ?s pos*› ‹*orient-lit ?L' ?u ?v polarity*›
assms(*5*) *assms*(*6*) **show** *?thesis* **using** *lit-ord-dominating-term* **by** *metis*
next
assume (*?u*,*?t*) ∉ *trm-ord*
from *this* **and** ‹*?u* = *?t* ∨ (*?u*,*?t*) ∈ *trm-ord*› **have** *?u* = *?t* **by** *auto*
have *polarity* = *pos*
proof (*rule ccontr*)
assume *polarity* ≠ *pos*
then have *polarity* = *neg* **using** *sign.exhaust* **by** *auto*
from *this* **and** ‹*?u* = *?t*› **and** ‹*orient-lit ?L ?t ?s pos*›
‹*orient-lit ?L' ?u ?v polarity*› *assms*(*5*) *assms*(*6*)
have (*?L*,*?L'*) ∈ *lit-ord* **using** *lit-ord-neg-lit-lhs* **by** *auto*
from *this* **and** ‹(*?L*,*?L'*) ∉ *lit-ord*› **show** *False* **by** *auto*
qed
have *?v* ≠ *?s*
proof
assume *?v* = *?s*
from *this* *assms*(*7*) **have** *I ?s ?v* **unfolding** *fo-interpretation-def congruence-def*
equivalence-relation-def reflexive-def **by** *auto*
from *this* **and** ‹*orient-lit-inst L' u v polarity σ*› ‹*polarity* = *pos*› ‹*?u* = *?t*›
and *assms*(*3*) **have** *equivalent-eq-exists t s C I σ L*
unfolding *equivalent-eq-exists-def* **by** *metis*
from *this* **and** *assms*(*4*) **show** *False* **by** *auto*

qed
have $(?s,?v) \notin$ *trm-ord*
proof
assume $(?s,?v) \in$ *trm-ord*
from *this* **and** ‹$?u = ?t$› **and** ‹*orient-lit* $?L$ $?t$ $?s$ *pos*› ‹*orient-lit* $?L'$ $?u$ $?v$ *polarity*›
and ‹*polarity=pos*› *assms*(5) *assms*(6)
have $(?L,?L') \in$ *lit-ord* **using** *lit-ord-rhs* **by** *auto*
from *this* **and** ‹$(?L,?L') \notin$ *lit-ord*›**show** *False* **by** *auto*
qed
from *this* **and** ‹$?v \neq ?s$› **and** ‹$?v = ?s \vee (?v,?s) \in$ *trm-ord* $\vee (?s,?v) \in$ *trm-ord*›
have $(?v,?s) \in$ *trm-ord* **by** *metis*
from *this* **and** ‹$?u = ?t$› **and** ‹*orient-lit* $?L$ $?t$ $?s$ *pos*› ‹*orient-lit* $?L'$ $?u$ $?v$ *polarity*›
and ‹*polarity=pos*› *assms*(5) *assms*(6)
show $(?L',?L) \in$ *lit-ord* **using** *lit-ord-rhs* **by** *auto*
qed
qed

lemma *if-all-smaller-are-false-then-cl-not-valid*:
assumes (*smaller-lits-are-false* (*subst* t σ) (*subst-cl* C σ) S)
assumes (*fo-interpretation* (*same-values* ($\lambda t.$ (*trm-rep* t S))))
assumes *orient-lit-inst* $L1$ t s *pos* σ
assumes *maximal-literal* (*subst-lit* $L1$ σ) (*subst-cl* C σ)
assumes *ground-clause* (*subst-cl* C σ)
assumes (*subst-lit* $L1$ σ) $\in$ (*subst-cl* C σ)
assumes $\neg$*equivalent-eq-exists* t s C (*same-values* ($\lambda t.$ (*trm-rep* t S))) σ $L1$
assumes *trm-rep* (*subst* t σ) S = *trm-rep* (*subst* s σ) S
shows ($\neg$ *validate-ground-clause* (*same-values* ($\lambda t.$ (*trm-rep* t S))) (*subst-cl* ($C - \{ L1 \}$) σ))
proof
let $?I$ = (*same-values* ($\lambda t.$ (*trm-rep* t S)))
assume *validate-ground-clause* $?I$ (*subst-cl* ($C - \{ L1 \}$) σ)
then obtain L **where** $L \in$ (*subst-cl* ($C - \{ L1 \}$) σ) **and** *validate-ground-lit* $?I$ L
using *validate-ground-clause.simps* [*of* $?I$ (*subst-cl* ($C - \{ L1 \}$) σ)] **by** *blast*

from ‹$L \in$ (*subst-cl* ($C - \{ L1 \}$) σ)› **obtain** L' **where** $L' \in C - \{ L1 \}$ **and**

L = (*subst-lit* L' σ) **by** *auto*
from ‹$L' \in C - \{ L1 \}$› **and** ‹L = (*subst-lit* L' σ)›
have $L \in$(*subst-cl* C σ) **by** *auto*
from ‹$L \in$(*subst-cl* C σ)› **and** *assms*(5) **have** *vars-of-lit* $L = \{\}$ **by** *auto*
from *assms*(6) **and** *assms*(5) **have** *vars-of-lit* (*subst-lit* $L1$ σ) $= \{\}$ **by** *auto*

obtain u v *polarity* **where** o: *orient-lit-inst* L' u v *polarity* σ
and ((*subst* u σ), (*subst* v σ)) $\notin$ *trm-ord*
unfolding *orient-lit-inst-def* **using** *literal.exhaust equation.exhaust*

trm-ord-trans trm-ord-irrefl **unfolding** *trans-def irrefl-def* **by** *metis*
from *o* **and** ‹$L = (subst\text{-}lit\ L'\ \sigma)$›
have o' : *orient-lit L* (*subst u σ*) (*subst v σ*) *polarity*
using *lift-orient-lit* **by** *auto*

from o' **and** ‹*vars-of-lit L* = {}› **have** *vars-of* (*subst u σ*) = {}
using *orient-lit-vars* **by** *blast*
from o' **and** ‹*vars-of-lit L* = {}› **have** *vars-of* (*subst v σ*) = {}
using *orient-lit-vars* **by** *blast*

from *assms*(*3*)
have *o1* : *orient-lit* (*subst-lit L1 σ*) (*subst t σ*) (*subst s σ*) *pos*
using *lift-orient-lit* [*of L1 t s pos σ*] **by** *auto*
from *o1* **and** ‹*vars-of-lit* (*subst-lit L1 σ*) = {}› **have** *vars-of* (*subst t σ*) = {}
using *orient-lit-vars* **by** *blast*

have *polarity* = *pos* ∨ *polarity* = *neg* **using** *sign.exhaust* **by** *auto*
then show *False*
proof
assume *polarity* = *pos*
from *this* **and** *o* **and** *assms*(*2*) **and** ‹*validate-ground-lit ?I L*› **and** ‹$L = (subst\text{-}lit\ L'\ \sigma)$›
have (*trm-rep* (*subst u σ*) *S*) = (*trm-rep* (*subst v σ*) *S*)
using *orient-lit-semantics-pos* [*of ?I*] **unfolding** *same-values-def* **by** *metis*
from *assms*(*4*) **and** ‹*L* ∈(*subst-cl C σ*)›
have ((*subst-lit L1 σ*),*L*) ∉ *lit-ord* **unfolding** *maximal-literal-def*
by *blast*
from *this* **and** o' **and** *o1* **and** ‹*polarity*=*pos*› **and** ‹*vars-of-lit L* = {}› **and** ‹$L = (subst\text{-}lit\ L'\ \sigma)$›
and ‹*vars-of-lit* (*subst-lit L1 σ*) = {}›
have (*subst t σ*, *subst u σ*) ∉ *trm-ord*
and (*subst t σ*, *subst v σ*) ∉ *trm-ord*
using *lit-ord-dominating-term* [*of subst t σ subst u σ*
subst v σ subst-lit L1 σ subst s σ pos] **by** *auto*
show *?thesis*
proof (*cases*)
assume (*subst t σ*) = (*subst u σ*)
from *this* **and** *assms*(*8*) **and** ‹(*trm-rep* (*subst u σ*) *S*) = (*trm-rep* (*subst v σ*) *S*)›
have (*trm-rep* (*subst s σ*) *S*) = (*trm-rep* (*subst v σ*) *S*) **by** *metis*
from *this o* **and** ‹$L' \in C - \{ L1 \}$› ‹*polarity* = *pos*› ‹(*subst t σ*) = (*subst u σ*)› *assms*(*7*)
show *False* **unfolding** *equivalent-eq-exists-def same-values-def* **by** *blast*
next
assume (*subst t σ*) ≠ (*subst u σ*)
from *this* **and** ‹(*subst t σ*, *subst u σ*) ∉ *trm-ord*›
and ‹*vars-of* (*subst t σ*) = {}› **and** ‹*vars-of* (*subst u σ*) = {}›
have (*subst u σ*, *subst t σ*) ∈ *trm-ord*
using *trm-ord-ground-total* **unfolding** *ground-term-def* **by** *auto*

from *this* **and** ‹(*subst u σ*, *subst v σ*) $\notin$ *trm-ord*›
and ‹*vars-of* (*subst v σ*) = {}› **and** ‹*vars-of* (*subst t σ*) = {}›
have (*subst v σ*, *subst t σ*) ∈ *trm-ord*
using *trm-ord-ground-total trm-ord-trans*
unfolding *ground-term-def trans-def* **by** *metis*

from ‹*polarity* = *pos*› **and** *o′* **and** *assms*(*1*) **and** ‹*L* ∈(*subst-cl C σ*)› **and** ‹*L* = (*subst-lit L′ σ*)›
and ‹((*subst u σ*), *subst t σ*) ∈ *trm-ord*›
and ‹((*subst v σ*), *subst t σ*) ∈ *trm-ord*›
have *trm-rep* (*subst u σ*) *S* ≠ *trm-rep* (*subst v σ*) *S*
unfolding *smaller-lits-are-false-def* **by** *metis*
from *this* **and** ‹*trm-rep* (*subst u σ*) *S* = *trm-rep* (*subst v σ*) *S*›
show *False* **by** *blast*
qed
next assume *polarity* = *neg*
from *this* **and** *o* **and** *assms*(*2*) **and** ‹*validate-ground-lit ?I L*› **and** ‹*L* = (*subst-lit L′ σ*)›
have (*trm-rep* (*subst u σ*) *S*) ≠ (*trm-rep* (*subst v σ*) *S*)
using *orient-lit-semantics-neg* [*of ?I*] **unfolding** *same-values-def* **by** *metis*
from *assms*(*4*) **and** ‹*L* ∈(*subst-cl C σ*)›
have ((*subst-lit L1 σ*),*L*) $\notin$ *lit-ord* **unfolding** *maximal-literal-def*
by *blast*
from *this* **and** *o′* **and** *o1* **and** ‹*vars-of-lit L* = {}› **and** ‹*L* = (*subst-lit L′ σ*)›
and ‹*vars-of-lit* (*subst-lit L1 σ*) = {}›
have (*subst t σ*, *subst u σ*) $\notin$ *trm-ord*
and (*subst t σ*, *subst v σ*) $\notin$ *trm-ord*
using *lit-ord-dominating-term* [*of subst t σ subst u σ*
subst v σ subst-lit L1 σ subst s σ pos]
by *auto*

from ‹((*subst-lit L1 σ*),*L*) $\notin$ *lit-ord*› **and** *o′* **and** *o1* **and** ‹*polarity*=*neg*› **and** ‹*vars-of-lit L* = {}›
and ‹*L* = (*subst-lit L′ σ*)› **and** ‹*vars-of-lit* (*subst-lit L1 σ*) = {}›
have *subst t σ* ≠ *subst u σ*
using *lit-ord-neg-lit-lhs* **by** *auto*
from *this* **and** ‹(*subst t σ*, *subst u σ*) $\notin$ *trm-ord*› **and** ‹*vars-of* (*subst t σ*) = {}›
and ‹*vars-of* (*subst u σ*) = {}›
have (*subst u σ*, *subst t σ*) ∈ *trm-ord*
using *trm-ord-ground-total* **unfolding** *ground-term-def* **by** *auto*

from *this* **and** ‹(*subst u σ*, *subst v σ*) $\notin$ *trm-ord*› **and** ‹*vars-of* (*subst v σ*) = {}›
and ‹*vars-of* (*subst t σ*) = {}› **have** (*subst v σ*, *subst t σ*) ∈ *trm-ord*
using *trm-ord-ground-total trm-ord-trans* **unfolding** *ground-term-def trans-def* **by** *metis*

from ‹*polarity* = *neg*› **and** o' **and** *assms*(*1*) **and** ‹L ∈(*subst-cl* C σ)› **and** ‹L = (*subst-lit* L' σ)›
and ‹((*subst* u σ), *subst* t σ) ∈ *trm-ord*› **and** ‹((*subst* v σ), *subst* t σ) ∈ *trm-ord*›
have *trm-rep* (*subst* u σ) S = *trm-rep* (*subst* v σ) S
unfolding *smaller-lits-are-false-def* **by** *metis*
from *this* **and** ‹*trm-rep* (*subst* u σ) S ≠ *trm-rep* (*subst* v σ) S› **show** *False* **by** *blast*
qed
qed

We introduce the notion of a reduction, which is a ground superposition inference satisfying some additional conditions:

(i) the "from" clause is smaller than the "into" clause;

(ii) its "body" (i.e., the part of the clause without the equation involved in the rule) is false in a given interpretation and strictly smaller than the involved equation.

definition *reduction*
where (*reduction L1 C* σ' *t s polarity L2 u* u' *p v D I S* σ) =
((*D* ∈ *S*) ∧ (*eligible-literal L1 C* σ') ∧ (*eligible-literal L2 D* σ')
∧ *ground-clause* (*subst-cl* (*cl-ecl D*) σ')
∧ (*minimal-redex p* (*subst t* σ) *C S t*)
∧ (*coincide-on* σ σ' (*vars-of-cl* (*cl-ecl C*)))
∧ (*allowed-redex* u' *C* σ)
∧ (¬ *is-a-variable* u')
∧ (*L1* ∈ (*cl-ecl C*)) ∧ (*L2* ∈ (*cl-ecl D*))
∧ (*orient-lit-inst L1 t s polarity* σ')
∧ (*orient-lit-inst L2 u v pos* σ')
∧ (*subst u* σ') ≠ (*subst v* σ')
∧ (*subst* u' σ') = (*subst u* σ')
∧ (¬ *validate-ground-clause I* (*subst-cl* ((*cl-ecl D*) − { *L2* }) σ'))
∧ ((*subst-lit L2* σ'),(*subst-lit L1* σ')) ∈ *lit-ord*
∧ (∀ *x* ∈ (*cl-ecl D*) − { *L2* }. ((*subst-lit x* σ'),(*subst-lit L2* σ'))
∈ *lit-ord*)
∧ (*all-trms-irreducible* (*subst-set* (*trms-ecl D*) σ')
(λt. (*trm-rep t S*)))
∧ (*I* (*subst u* σ') (*subst v* σ'))
∧ (*subterm t p* u'))

The next lemma states that the rules used to evaluate terms can be renamed so that they share no variable with the clause in which the term occurs.

lemma *candidate-values-renaming*:
assumes (*candidate-values z CC* C' *C s L* L' σ t' s' *t S*)
assumes *finite* C'
assumes *finite* (*cl-ecl* (*D*:: *'a eclause*))
assumes *closed-under-renaming S*
assumes *Ball S well-constrained*
shows ∃ *CC-bis* C'*-bis* L'*-bis* σ*-bis* t'*-bis* s'*-bis t-bis*.

($candidate\text{-}values\ z\ CC\text{-}bis\ C'\text{-}bis\ C\ s\ L\ L'\text{-}bis\ \sigma\text{-}bis\ t'\text{-}bis\ s'\text{-}bis\ t\ S$)
$\wedge$ ($vars\text{-}of\text{-}cl$ ($cl\text{-}ecl\ D$)) $\cap$ $vars\text{-}of\text{-}cl$ ($cl\text{-}ecl\ CC\text{-}bis$) = {}

proof –
from *assms(2)* **have** *finite* (*vars-of-cl* C') **using** *set-of-variables-is-finite-cl* **by** *auto*
from *assms(3)* **have** *finite* (*vars-of-cl* (*cl-ecl D*)) **using** *set-of-variables-is-finite-cl*

by *auto*
from *infinite-vars* **have** ¬ (*finite vars*) **by** *auto*
from ‹*finite* (*vars-of-cl* C')› ‹*finite* (*vars-of-cl* (*cl-ecl D*))›
and ‹¬ (*finite vars*)›
obtain η **where** *renaming* η (*vars-of-cl* C')
and ((*subst-codomain* η (*vars-of-cl* C')) $\cap$ (*vars-of-cl* (*cl-ecl D*))) = {}
using *renaming-exists* [*of vars*] **by** *meson*
from *this* ‹*finite* (*vars-of-cl* C')› **obtain** η'
where *i*: ($\forall$ $x \in$ (*vars-of-cl* C'). (*subst* (*subst* (*Var x*) η) η')
= (*Var x*))
using *renamings-admit-inverse* **by** *blast*
obtain *CC-bis* **where** *CC-bis* = (*subst-ecl CC* η) **by** *auto*
obtain C'*-bis* **where** C'*-bis* = (*subst-cl* C' η) **by** *auto*
from *assms(1)* **have** C' = (*cl-ecl CC*) **unfolding** *candidate-values-def* **by** *metis*
from *this* **obtain** T **where** CC = (*Ecl* C' T)
by (*metis cl-ecl.simps trms-ecl.elims*)
from *this* **and** ‹*CC-bis* = (*subst-ecl CC* η)›
and ‹C'*-bis* = (*subst-cl* C' η)›
have C'*-bis* = (*cl-ecl CC-bis*) **by** *auto*
from *assms(1)* **have** $CC \in S$ **unfolding** *candidate-values-def* **by** *metis*
from *assms(1)* **have** $(s,t) \in$ *trm-ord* **unfolding** *candidate-values-def* **by** *metis*
from *assms(1)* **have** ((s,t) $\in$ *trm-ord* $\longrightarrow$ (z = *trm-rep s S*))
unfolding *candidate-values-def* **by** *metis*
from *assms(1)* **have** (*maximal-literal L C*) **unfolding** *candidate-values-def* **by** *metis*
from *assms(1)* **have** (*ground-clause C*) **unfolding** *candidate-values-def* **by** *metis*
from *assms(1)* **have** $L' \in C'$ **unfolding** *candidate-values-def* **by** *metis*
from *assms(1)* **have** L = (*subst-lit* L' σ)
unfolding *candidate-values-def* **by** *metis*
from *assms(1)* **have** (*smaller-lits-are-false t C S*)
unfolding *candidate-values-def* **by** *metis*
from *assms(1)* **have** C = (*subst-cl* C' σ)
unfolding *candidate-values-def* **by** *metis*
from *assms(1)* **have** (*orient-lit-inst* L' t' s' *pos* σ)
unfolding *candidate-values-def* **by** *metis*
from *assms(1)* **have** (*trms-irreducible CC* σ *S t*)
unfolding *candidate-values-def* **by** *metis*
from *assms(1)* **have** t = *subst* t' σ **unfolding** *candidate-values-def* **by** *metis*
from *assms(1)* **have** s = *subst* s' σ **unfolding** *candidate-values-def* **by** *metis*

from ‹$CC \in S$› **and** *assms(4)* **and** ‹*renaming* η (*vars-of-cl* C')› **and** ‹C' =

(*cl-ecl CC*)›
‹*CC-bis* = (*subst-ecl CC* η)› **have** *CC-bis* $\in$ *S*
unfolding *closed-under-renaming-def renaming-cl-def* **by** *auto*
from *assms*(*1*) **have** (*sel* C' = {}) **unfolding** *candidate-values-def* **by** *metis*
from *this* **and** ‹*renaming* η (*vars-of-cl* C')› ‹C' = (*cl-ecl CC*)›
‹C'*-bis* = (*subst-cl* C' η)› **have** *sel* C'*-bis* = {}
using *sel-renaming* **by** *auto*
obtain L'*-bis* **where** L'*-bis* = (*subst-lit* L' η) **by** *auto*
from *this* **and** ‹$L' \in C'$› ‹C'*-bis* = (*subst-cl* C' η)› **have** L'*-bis* $\in$ C'*-bis* **by** *auto*

let *?*ϑ = (*comp* (*comp* η η') σ)
let *?*ϑ' = (*comp* η' σ)
have *coincide-on* σ *?*ϑ (*vars-of-cl* C')
proof (*rule ccontr*)
assume $\neg$*coincide-on* σ *?*ϑ (*vars-of-cl* C')
from *this* **obtain** *x* **where** (*subst* (*Var x*) σ) $\neq$ (*subst* (*Var x*) *?*ϑ)
and $x \in$ *vars-of-cl* C' **unfolding** *coincide-on-def* **by** *auto*
from ‹$x \in$ *vars-of-cl* C'› *i*
have (*subst* (*subst* (*Var x*) η) η') = (*Var x*)
by *blast*
from *this* **and** ‹(*subst* (*Var x*) σ) $\neq$ (*subst* (*Var x*) *?*ϑ)›
show *False* **by** *simp*
qed
from ‹$L' \in C'$› **have** *vars-of-lit* $L' \subseteq$ *vars-of-cl* C' **by** *auto*
from *this* **and** ‹*coincide-on* σ *?*ϑ (*vars-of-cl* C')›
have *coincide-on* σ *?*ϑ (*vars-of-lit* L') **unfolding** *coincide-on-def* **by** *auto*
from *this* **and** ‹*L* = (*subst-lit* L' σ)›
have *L* = (*subst-lit* L' *?*ϑ) **using** *coincide-on-lit* **by** *auto*
have *subst-lit* L' *?*ϑ
= *subst-lit* (*subst-lit* L' η) *?*ϑ'
by (*simp add*: *coincide-on-def coincide-on-lit composition-of-substs-lit*)
from *this* **and** ‹*L* = (*subst-lit* L' *?*ϑ)› **and**
‹L'*-bis* = (*subst-lit* L' η)›
have *L* = (*subst-lit* L'*-bis* *?*ϑ')
by *auto*
from ‹*coincide-on* σ *?*ϑ (*vars-of-cl* C')› **and** ‹*C* = (*subst-cl* C' σ)›
have *C* = *subst-cl* C' *?*ϑ
using *coincide-on-cl* **by** *blast*
have *subst-cl* C' *?*ϑ
= *subst-cl* (*subst-cl* C' η) *?*ϑ'
by (*metis composition-of-substs-cl*)
from *this* **and** ‹*C* = *subst-cl* C' *?*ϑ› **and** ‹C'*-bis* = (*subst-cl* C' η)›
have *C* = *subst-cl* C'*-bis* *?*ϑ' **by** *auto*
from ‹(*finite* C')› **and** ‹C'*-bis* = (*subst-cl* C' η)› **have** *finite* C'*-bis* **by** *auto*
have $t \notin$ (*subst-set* (*trms-ecl CC-bis*) *?*ϑ')
proof
assume $t \in$ (*subst-set* (*trms-ecl CC-bis*) *?*ϑ')
from *this* **obtain** *u* **where** $u \in$ (*trms-ecl CC-bis*)
and *t* = (*subst u ?*ϑ') **by** *auto*

from ‹$u \in$ (*trms-ecl CC-bis*)› **and** ‹*CC-bis* = (*subst-ecl CC* η)›
obtain v **where** $v \in$ *trms-ecl CC* **and** u = (*subst v* η)
using ‹*CC* = *Ecl C' T*› **by** *auto*
from ‹u = (*subst v* η)› ‹t = (*subst u ?*ϑ')› **have** *subst v ?*ϑ = t **by** *simp*
from ‹$v \in$ *trms-ecl CC*› ‹*CC* $\in$ *S*› *assms*(*5*)
have *dom-trm v* (*cl-ecl CC*) **unfolding** *Ball-def well-constrained-def* **by** *metis*
from *this* **have** *vars-of v* $\subseteq$ *vars-of-cl* (*cl-ecl CC*) **using** *dom-trm-vars* **by** *auto*
from *this* **and** ‹C' = (*cl-ecl CC*)› ‹*coincide-on* σ *?*ϑ (*vars-of-cl* C')›
have *coincide-on* σ *?*ϑ (*vars-of v*) **unfolding** *coincide-on-def* **by** *auto*
from *this* **and** ‹*subst v ?*ϑ = t› **have** (*subst v* σ) = t
using *coincide-on-term* **by** *metis*
from *this* **and** ‹$v \in$ *trms-ecl CC*› **have**
($t \in$ (*subst-set* (*trms-ecl CC*) σ)) **by** *auto*
from *this* **and** *assms*(*1*) **show** *False* **unfolding** *candidate-values-def* **by** *metis*
qed
have (*trms-irreducible CC-bis ?*ϑ' *S t*)
proof (*rule ccontr*)
assume $\neg$(*trms-irreducible CC-bis ?*ϑ' *S t*)
then obtain x x' **where** $x' \in$ *trms-ecl CC-bis*
occurs-in x (*subst* x' *?*ϑ')
(x,t) $\in$ *trm-ord* (*trm-rep x S*) $\neq x$
using *trms-irreducible-def* **by** *blast*
from ‹$x' \in$ (*trms-ecl CC-bis*)› **and** ‹*CC-bis* = (*subst-ecl CC* η)›
obtain v **where** $v \in$ *trms-ecl CC* **and** x' = (*subst v* η)
using ‹*CC* = *Ecl C' T*› **by** *auto*
from ‹*occurs-in x* (*subst* x' *?*ϑ')› ‹x' = *subst v* η› **have** *occurs-in x* (*subst v ?*ϑ)
by *simp*
from ‹$v \in$ *trms-ecl CC*› ‹*CC* $\in$ *S*› *assms*(*5*)
have *dom-trm v* (*cl-ecl CC*) **unfolding** *Ball-def well-constrained-def* **by** *auto*
from *this* **have** *vars-of v* $\subseteq$ *vars-of-cl* (*cl-ecl CC*) **using** *dom-trm-vars* **by** *auto*
from *this* **and** ‹C' = (*cl-ecl CC*)› ‹*coincide-on* σ *?*ϑ (*vars-of-cl* C')›
have *coincide-on* σ *?*ϑ (*vars-of v*) **unfolding** *coincide-on-def* **by** *auto*
from *this* **have** (*subst v* σ) = (*subst v ?*ϑ)
using *coincide-on-term* **by** *metis*
from *this* **and** ‹*occurs-in x* (*subst v ?*ϑ)›
have *occurs-in x* (*subst v* σ) **by** *auto*
from *this* **and** ‹$v \in$ *trms-ecl CC*› **and** ‹(x,t) $\in$ *trm-ord*›
‹(*trm-rep x S*) $\neq x$› **and** ‹(*trms-irreducible CC* σ *S t*)› **show** *False*
unfolding *trms-irreducible-def* **by** *metis*
qed
obtain t'*-bis* **where** t'*-bis* = (*subst* t' η) **by** *auto*
obtain s'*-bis* **where** s'*-bis* = (*subst* s' η) **by** *auto*

from ‹(*orient-lit-inst* L' t' s' *pos* σ)› **have** *vars-of* $t' \subseteq$ *vars-of-lit* L'
using *orient-lit-inst-vars* **by** *auto*
from *this* ‹*coincide-on* σ *?*ϑ (*vars-of-lit* L')›
have *coincide-on* σ *?*ϑ (*vars-of* t') **unfolding** *coincide-on-def* **by** *blast*
from *this* **have** *subst* t' *?*ϑ = *subst* t' σ
using *coincide-on-term* **by** *metis*

from *this* ‹t'-*bis* = (*subst* t' η)› **have** *subst* t'-*bis* *?*ϑ' = *subst* t' σ **by** *simp*

from ‹(*orient-lit-inst* L' t' s' *pos* σ)› **have** *vars-of* s' $\subseteq$ *vars-of-lit* L'
  using *orient-lit-inst-vars* **by** *auto*
from *this* ‹*coincide-on* σ *?*ϑ (*vars-of-lit* L')›
  have *coincide-on* σ *?*ϑ (*vars-of* s') **unfolding** *coincide-on-def* **by** *blast*
from *this* **have** *subst* s' *?*ϑ = *subst* s' σ
  using *coincide-on-term* **by** *metis*
from *this* ‹s'-*bis* = (*subst* s' η)› **have** *subst* s'-*bis* *?*ϑ' = *subst* s' σ **by** *simp*

have (*orient-lit-inst* L'-*bis* t'-*bis* s'-*bis* *pos* *?*ϑ')
proof −
from ‹(*orient-lit-inst* L' t' s' *pos* σ)›
  have ((*subst* t' σ),(*subst* s' σ)) $\notin$ *trm-ord*
  using *orient-lit-inst-def* **by** *simp*

from ‹(*orient-lit-inst* L' t' s' *pos* σ)›
  have L' = (*Pos* (*Eq* t' s')) $\vee$ L' = (*Pos* (*Eq* s' t'))
  by (*simp add*: *orient-lit-inst-def*)
from *this*
  ‹L'-*bis* = (*subst-lit* L' η)›
  ‹t'-*bis* = (*subst* t' η)›
  ‹s'-*bis* = (*subst* s' η)›
  have L'-*bis* = (*Pos* (*Eq* t'-*bis* s'-*bis*)) $\vee$ L'-*bis* = (*Pos* (*Eq* s'-*bis* t'-*bis*))
  by *auto*
from ‹*subst* s'-*bis* *?*ϑ' = *subst* s' σ›
  and ‹*subst* t'-*bis* *?*ϑ' = *subst* t' σ›
  and ‹((*subst* t' σ),(*subst* s' σ)) $\notin$ *trm-ord*›
  have ((*subst* t'-*bis* *?*ϑ'),(*subst* s'-*bis* *?*ϑ')) $\notin$ *trm-ord*
  by *auto*
 from *this* **and** ‹L'-*bis* = (*Pos* (*Eq* t'-*bis* s'-*bis*)) $\vee$ L'-*bis* = (*Pos* (*Eq* s'-*bis*
t'-*bis*))›
  show *?thesis* **unfolding** *orient-lit-inst-def* **by** *auto*
qed
have (*maximal-literal-is-unique* t s C'-*bis* L'-*bis* S *?*ϑ')
proof (*rule ccontr*)
  assume $\neg$(*maximal-literal-is-unique* t s C'-*bis* L'-*bis* S *?*ϑ')
  from *this* **obtain** s'' **where** (*eq-occurs-in-cl* t s'' (C'-*bis*− { L'-*bis* }) *?*ϑ')
    (s'',t) $\in$ *trm-ord*
    (s,t) $\in$ *trm-ord*
    (*trm-rep* s S) = (*trm-rep* s'' S)
    unfolding *maximal-literal-is-unique-def*
    by *metis*
  from ‹(*eq-occurs-in-cl* t s'' (C'-*bis*− { L'-*bis* }) *?*ϑ')› **obtain** M u v **where**
    M $\in$ C'-*bis* − { L'-*bis* } *orient-lit-inst* M u v *pos* *?*ϑ'
    t = (*subst* u *?*ϑ') s'' = (*subst* v *?*ϑ')
    unfolding *eq-occurs-in-cl-def* **by** *blast*
  from ‹M $\in$ C'-*bis* − { L'-*bis* }›
    and ‹C'-*bis* = (*subst-cl* C' η)› **and** ‹L'-*bis* = (*subst-lit* L' η)›

obtain M' **where** $M' \in C' - \{ L' \}$ **and** *subst-lit* $M'\ \eta =\ M$ **by** *auto*
from ‹*orient-lit-inst M u v pos ?ϑ'*› **obtain** e **where** $M = (Pos\ e)$
unfolding *orient-lit-inst-def* **by** *auto*
from *this* **and** ‹*orient-lit-inst M u v pos ?ϑ'*› **have** $e = (Eq\ u\ v) \vee e = (Eq\ v\ u)$
unfolding *orient-lit-inst-def* **by** *auto*
from ‹*orient-lit-inst M u v pos ?ϑ'*› **have**
((*subst u ?ϑ'*),(*subst v ?ϑ'*)) $\notin$ *trm-ord*
unfolding *orient-lit-inst-def* **by** *auto*
from ‹$M = (Pos\ e)$› **and** ‹*subst-lit* $M'\ \eta =\ M$›
obtain e' **where** (*subst-equation* $e'\ \eta$) $= e$ **and** $M' = (Pos\ e')$
by (*metis* (*no-types, opaque-lifting*) *subst-lit.simps*(*1*) *subst-lit.simps*(*2*) *atom.simps*(*1*)

literal.distinct(*1*) *positive-literal.elims*(*1*))
from ‹$e = (Eq\ u\ v) \vee e = (Eq\ v\ u)$› **and** ‹(*subst-equation* $e'\ \eta$) $= e$›
obtain $u'\ v'$ **where** $e' = (Eq\ u'\ v') \vee (e' = (Eq\ v'\ u'))$ **and** (*subst* $u'\ \eta$) $= u$ **and**
(*subst* $v'\ \eta$) $= v$
by (*metis subst-equation.simps equation.inject subterms-of-eq.cases*)
from ‹((*subst u ?ϑ'*),(*subst v ?ϑ'*)) $\notin$ *trm-ord*›
‹(*subst* $u'\ \eta$) $= u$›
‹(*subst* $v'\ \eta$) $= v$›
have ((*subst* u' *?ϑ*),(*subst* v' *?ϑ*)) $\notin$ *trm-ord* **by** *simp*
from *this* **and** ‹$M' = (Pos\ e')$› **and** ‹$e' = (Eq\ u'\ v') \vee (e' = (Eq\ v'\ u'))$›
have *orient-lit-inst* $M'\ u'\ v'$ *pos ?ϑ*
unfolding *orient-lit-inst-def* **by** *auto*
from ‹$M' \in C' - \{ L' \}$› **have** *vars-of-lit* $M' \subseteq$ *vars-of-cl* C' **by** *auto*
from *this* **and** ‹*coincide-on* σ *?ϑ* (*vars-of-cl* C')› **have** *coincide-on* σ *?ϑ* (*vars-of-lit* M')
unfolding *coincide-on-def* **by** *auto*
from *this* **have** *coincide-on ?ϑ* σ (*vars-of-lit* M') **using** *coincide-sym* **by** *auto*
from *this* **and** ‹*orient-lit-inst* $M'\ u'\ v'$ *pos ?ϑ*› **have** *orient-lit-inst* $M'\ u'\ v'$ *pos* σ
using *orient-lit-inst-coincide* **by** *auto*
from ‹*orient-lit-inst* $M'\ u'\ v'$ *pos ?ϑ*› **have** *vars-of* $u' \subseteq$ *vars-of-lit* M' **and**
vars-of $v' \subseteq$ *vars-of-lit* M' **using** *orient-lit-inst-vars* **by** *auto*
from ‹*vars-of* $u' \subseteq$ *vars-of-lit* M'› **and** ‹*coincide-on ?ϑ* σ (*vars-of-lit* M')›
have *coincide-on ?ϑ* σ (*vars-of* u') **unfolding** *coincide-on-def* **by** *auto*
from *this* **have** *subst* u' *?ϑ* $=$ *subst* $u'\ \sigma$ **using** *coincide-on-term* **by** *metis*
from *this* **and** ‹(*subst* $u'\ \eta$) $= u$› **have** *subst u ?ϑ'* $=$ *subst* $u'\ \sigma$ **by** *simp*
from ‹*vars-of* $v' \subseteq$ *vars-of-lit* M'› **and** ‹*coincide-on ?ϑ* σ (*vars-of-lit* M')›
have *coincide-on ?ϑ* σ (*vars-of* v') **unfolding** *coincide-on-def* **by** *auto*
from *this* **have** *subst* v' *?ϑ* $=$ *subst* $v'\ \sigma$ **using** *coincide-on-term* **by** *metis*
from *this* **and** ‹(*subst* $v'\ \eta$) $= v$› **have** *subst v ?ϑ'* $=$ *subst* $v'\ \sigma$ **by** *simp*
from ‹*subst v ?ϑ'* $=$ *subst* $v'\ \sigma$› ‹$s'' =$ (*subst v ?ϑ'*)›
have $s'' =$ (*subst* $v'\ \sigma$) **by** *auto*
from ‹*subst u ?ϑ'* $=$ *subst* $u'\ \sigma$› ‹$t =$ (*subst u ?ϑ'*)›
have $t =$ (*subst* $u'\ \sigma$) **by** *auto*
from ‹$s'' =$ (*subst* $v'\ \sigma$)› ‹$t =$ (*subst* $u'\ \sigma$)›

‹*orient-lit-inst* M' u' v' *pos* σ› ‹$M' \in C' - \{ L' \}$›
have *eq-occurs-in-cl* t s'' $(C' - \{ L' \})$ σ
unfolding *eq-occurs-in-cl-def* **by** *auto*
from *this* **and** ‹$(s'',t) \in$ *trm-ord*› **and** ‹$(s,t) \in$ *trm-ord*› **and** ‹(*trm-rep* s S) = (*trm-rep* s'' S)›
have $\neg$(*maximal-literal-is-unique* t s C' L' S σ) **unfolding** *maximal-literal-is-unique-def*

by *blast*
from *this* **and** *assms*(*1*) **show** *False* **unfolding** *candidate-values-def* **by** *metis*
qed

from ‹t'-*bis* = (*subst* t' η)›
and ‹t = *subst* t' σ›
have t = *subst* t'-*bis* (*comp* η' σ)
using ‹*subst* t'-*bis* (*comp* η' σ) = *subst* t' σ› **by** *auto*

from ‹s'-*bis* = (*subst* s' η)›
and ‹s = *subst* s' σ›
have s = *subst* s'-*bis* (*comp* η' σ)
using ‹*subst* s'-*bis* (*comp* η' σ) = *subst* s' σ› **by** *auto*

from ‹*CC-bis* $\in S$› ‹$t \notin$ *subst-set* (*trms-ecl CC-bis*) (*comp* η' σ)›
‹*trms-irreducible CC-bis* (*comp* η' σ) S t›
‹C'-*bis* = *cl-ecl CC-bis*› ‹$(s, t) \in$ *trm-ord*› ‹($(s, t) \in$ *trm-ord* $\longrightarrow z$ = *trm-rep* s S)›
‹*orient-lit-inst* L'-*bis* t'-*bis* s'-*bis* *pos* (*comp* η' σ)›
‹*sel* C'-*bis* = {}› ‹L'-*bis* $\in C'$-*bis*› ‹*maximal-literal* L C›
‹L = *subst-lit* L'-*bis* (*comp* η' σ)›
‹C = *subst-cl* C'-*bis* (*comp* η' σ)›
‹*ground-clause* C› ‹t = *subst* t'-*bis* (*comp* η' σ)›
‹s = *subst* s'-*bis* (*comp* η' σ)›
‹*finite* C'-*bis*› ‹*smaller-lits-are-false* t C S›
‹*maximal-literal-is-unique* t s C'-*bis* L'-*bis* S (*comp* η' σ)›
have (*candidate-values* z *CC-bis* C'-*bis* C s L L'-*bis* ?ϑ' t'-*bis* s'-*bis* t S)
unfolding *candidate-values-def* **by** *metis*

have *vars-of-cl* (*cl-ecl* D) $\cap$ (*vars-of-cl* (*cl-ecl CC-bis*)) = {}
proof (*rule ccontr*)
assume *vars-of-cl* (*cl-ecl* D) $\cap$ (*vars-of-cl* (*cl-ecl CC-bis*)) $\neq$ {}
from *this* **and** ‹C'-*bis* = (*cl-ecl CC-bis*)›
obtain x **where** $x \in$ *vars-of-cl* C'-*bis* **and** $x \in$ *vars-of-cl* (*cl-ecl* D) **by** *auto*
from ‹$x \in$ *vars-of-cl* C'-*bis*›
obtain N **where** $N \in C'$-*bis* **and** $x \in$ *vars-of-lit* N **by** *auto*
from ‹$N \in C'$-*bis*› **and** ‹C'-*bis* = (*subst-cl* C' η)› **obtain** N' **where**
$N' \in C'$ **and** N = *subst-lit* N' η **by** *auto*
from ‹$x \in$ *vars-of-lit* N› **obtain** e **where** N = (*Pos* e) $\vee$ (N = (*Neg* e))
and $x \in$ *vars-of-eq* e
by (*metis negative-literal.elims*(*2*) *negative-literal.elims*(*3*) *vars-of-lit.simps*(*1*)

```
      vars-of-lit.simps(2))
    from ‹N = (Pos e) ∨ (N = (Neg e))› and ‹N = subst-lit N′ η› obtain e′
where
     N′ = (Pos e′) ∨ (N′ = (Neg e′)) and e = subst-equation e′ η
     by (metis subst-lit.elims atom.simps(1) atom.simps(2))

    from ‹x ∈ vars-of-eq e› obtain u v where e = (Eq u v) and x ∈ vars-of u
∪ vars-of v
     by (metis equation.exhaust vars-of-eq.simps)
    from ‹e = (Eq u v)› and ‹e = subst-equation e′ η› obtain u′ v′ where e′ =
(Eq u′ v′)
     u = (subst u′ η) and v = (subst v′ η)
     by (metis subst-equation.simps equation.exhaust equation.inject)
    from ‹x ∈ vars-of u ∪ vars-of v› have x ∈ vars-of u ∨ x ∈ vars-of v by auto
    then show False
    proof
     assume x ∈ vars-of u
     from this and ‹u = (subst u′ η)›
      obtain y where y ∈ vars-of u′ and x ∈ vars-of (subst (Var y) η)
      using vars-of-instances [of u′ η] by auto
     from ‹y ∈ vars-of u′› and ‹e′ = (Eq u′ v′)› have y ∈ vars-of-eq e′ by auto
     from this and ‹N′ = (Pos e′) ∨ (N′ = (Neg e′))› have y ∈ vars-of-lit N′
by auto
     from this and ‹N′ ∈ C′› have y ∈ vars-of-cl C′ by auto
     from this and ‹renaming η (vars-of-cl C′)›
      have is-a-variable (subst (Var y) η) unfolding renaming-def by auto
     from this and ‹x ∈ vars-of (subst (Var y) η)› have
       Var x = (subst (Var y) η)
      by (metis is-a-variable.elims(2) singletonD vars-of.simps(1))
     from this and ‹y ∈ vars-of-cl C′›
      have x ∈ (subst-codomain η (vars-of-cl C′)) unfolding subst-codomain-def
by auto
     from this and ‹x ∈ vars-of-cl (cl-ecl D)›
      and ‹((subst-codomain η (vars-of-cl C′)) ∩ (vars-of-cl (cl-ecl D))) = {}›
      show False by auto
    next
     assume x ∈ vars-of v
     from this and ‹v = (subst v′ η)›
      obtain y where y ∈ vars-of v′ and x ∈ vars-of (subst (Var y) η)
      using vars-of-instances [of v′ η] by auto
     from ‹y ∈ vars-of v′› and ‹e′ = (Eq u′ v′)› have y ∈ vars-of-eq e′ by auto
     from this and ‹N′ = (Pos e′) ∨ (N′ = (Neg e′))› have y ∈ vars-of-lit N′
by auto
     from this and ‹N′ ∈ C′› have y ∈ vars-of-cl C′ by auto
     from this and ‹renaming η (vars-of-cl C′)›
      have is-a-variable (subst (Var y) η) unfolding renaming-def by auto
     from this and ‹x ∈ vars-of (subst (Var y) η)› have
       Var x = (subst (Var y) η)
      by (metis is-a-variable.elims(2) singletonD vars-of.simps(1))
```

from *this* **and** ‹$y \in$ *vars-of-cl* C'›
have $x \in$ (*subst-codomain* η (*vars-of-cl* C')) **unfolding** *subst-codomain-def* **by** *auto*
from *this* **and** ‹$x \in$ *vars-of-cl* (*cl-ecl* D)›
and ‹((*subst-codomain* η (*vars-of-cl* C')) $\cap$ (*vars-of-cl* (*cl-ecl* D))) $= \{\}$›
show *False* **by** *auto*
qed
qed
from *this* **and** ‹(*candidate-values* z *CC-bis* C'*-bis* C s L L'*-bis* $?\vartheta'$ t'*-bis* s'*-bis* t S)›
show *?thesis* **by** *blast*
qed

lemma *pos-ord-acyclic*:
shows *acyclic* (*pos-ord* C t)
by (*simp add*: *acyclic-irrefl pos-ord-irrefl pos-ord-trans*)

definition *proper-subterm-red*
where *proper-subterm-red* t S σ =
($\exists\, p\ s.$ (*subterm* t p s $\wedge$ $p \neq$ *Nil* $\wedge$ (*trm-rep* (*subst* s σ) $S \neq$ (*subst* s σ))))

The following lemma states that if an eligible term in a clause instance is not in normal form, then the clause instance must be reducible (according to the previous definition of *reduction*). This is the key lemma for proving completeness. Note that we assume that the considered substitution is in normal form, so that the reduction cannot occur inside a variable. We also rename the clause used for the reduction, to ensure that it shares no variable with the provided clause. The proof requires an additional hypothesis in the case where the reducible term occurs at the root position in an eligible term of a positive literal, see the first hypothesis below and function *equivalent-eq-exists*.

lemma *reduction-exists*:
assumes *polarity* = *neg* $\vee$ $\neg$ *equivalent-eq-exists* t s (*cl-ecl* C) (*int-clset* S) σ *L1* $\vee$ *proper-subterm-red* t S σ
assumes $\forall\, x\ y.$ (($x \in$ *vars-of-cl* (*cl-ecl* C)) $\longrightarrow$ (*occurs-in* y (*subst* (*Var* x) σ)) $\longrightarrow$ *trm-rep* y $S = y$)
assumes *eligible-literal* *L1* C σ
assumes (*trm-rep* (*subst* t σ) S) $\neq$ (*subst* t σ)
assumes *L1* $\in$ (*cl-ecl* C)
assumes (*orient-lit-inst* *L1* t s *polarity* σ)
assumes $\forall\, x \in S.$ *finite* (*cl-ecl* x)
assumes *ground-clause* (*subst-cl* (*cl-ecl* C) σ)
assumes (*fo-interpretation* (*same-values* ($\lambda t.$ (*trm-rep* t S))))
assumes $C \in S$
assumes *Ball* S *well-constrained*
assumes *all-trms-irreducible* (*subst-set* (*trms-ecl* C) σ) ($\lambda t.$ *trm-rep* t S)
assumes $\neg$ *validate-ground-clause* (*int-clset* S) (*subst-cl* (*cl-ecl* C) σ)
assumes *closed-under-renaming* S

shows $\exists \sigma'$ *u* u' *p v D L2*.
((*reduction L1 C* σ' *t s polarity L2 u* u' *p v D* (*same-values* (λt. (*trm-rep t S*))) *S* σ)
$\wedge$ (*variable-disjoint C D*))

proof –

The first step is to get the minimal reducible position in $t \lhd \sigma$ and the corresponding subterm *v*.

let *?Redexes* = { *p*. $\exists$ *v*. *subterm* (*subst t* σ) *p v* $\wedge$ *root-term S v* $\wedge$ *trm-rep v S* $\neq$ *v* }
have *?Redexes* $\subseteq$ *pos-of* (*subst t* σ)
proof
fix *x* **assume** *x* $\in$ *?Redexes*
then have $\exists$ *v*. *subterm* (*subst t* σ) *x v* **by** *blast*
then have *position-in x* (*subst t* σ) **unfolding** *position-in-def* **by** *metis*
then show *x* $\in$ *pos-of* (*subst t* σ) **by** *auto*
qed
from *this* **have** *finite ?Redexes* **using** *set-of-positions-is-finite* [*of* (*subst t* σ)]
using *finite-subset* **by** *blast*
from *assms*(*4*) **have** *st-red S* (*subst t* σ) **using** *subterms-of-irred-trms-are-irred* **by** *blast*
from *this* **obtain** p' **where** p' $\in$ *?Redexes* **unfolding** *st-red-def* **by** *blast*
from ‹*finite ?Redexes*› *this* **obtain** *mp* **where** *mp* $\in$ *?Redexes*
and $\bigwedge p'$. (p', *mp*) $\in$ (*pos-ord C t*) $\Longrightarrow$ p' $\notin$ *?Redexes*
using *pos-ord-acyclic* [*of C t*] *finite-proj-wf* [*of ?Redexes* p' *pos-ord C t*] **by** *blast*
have *mr*: *minimal-redex mp* (*subst t* σ) *C S t*
proof (*rule ccontr*)
assume $\neg$*minimal-redex mp* (*subst t* σ) *C S t*
from *this* **obtain** p'' v' **where** (p'',*mp*) $\in$ (*pos-ord C t*) *subterm* (*subst t* σ) p'' v'
and *trm-rep* v' *S* $\neq$ v' **unfolding** *minimal-redex-def* **by** *blast*
show *False*
proof (*cases*)
assume (*root-term S* v')
from *this* **and** ‹*subterm* (*subst t* σ) p'' v'› ‹*trm-rep* v' *S* $\neq$ v'›
have p'' $\in$*?Redexes* **by** *blast*
from *this* **and** ‹$\bigwedge p'$. (p', *mp*) $\in$ (*pos-ord C t*) $\Longrightarrow$ p' $\notin$ *?Redexes*› **and** ‹(p'',*mp*) $\in$ (*pos-ord C t*)›
show *False* **by** *blast*
next assume $\neg$(*root-term S* v')
from ‹*trm-rep* v' *S* $\neq$ v'› **have** *st-red S* v'
using *subterms-of-irred-trms-are-irred* **by** *blast*
from *this* **obtain** p''' v'' **where** *subterm* v' p''' v'' *root-term S* v'' *trm-rep* v'' *S* $\neq$ v''
unfolding *st-red-def* **by** *blast*
from ‹*subterm* v' p''' v''› **and** ‹*subterm* (*subst t* σ) p'' v'›
have *subterm* (*subst t* σ) (*append* p'' p''') v''

using *subterm-of-a-subterm-is-a-subterm* **by** *metis*
from *this* **and** ‹*trm-rep* v'' *S* $\neq$ v''› ‹*root-term S* v''›
have (*append* p'' p''') $\in$ *?Redexes* **by** *blast*
from *this* **and** ‹$\bigwedge p'$. (p', *mp*) $\in$ (*pos-ord C t*) $\Longrightarrow$ $p' \notin$ *?Redexes*›
have (*append* p'' p''',*mp*) $\notin$ (*pos-ord C t*) **by** *blast*
from *this* **and** ‹(p'',*mp*) $\in$ (*pos-ord C t*)› **show** *False* **using** *pos-ord-prefix* **by** *auto*
qed
qed

from ‹*mp* $\in$ *?Redexes*› **obtain** *p v* **where** *mp=p subterm* (*subst t* σ) *p v* **and** *root-term S v*
and *trm-rep v S* $\neq$ *v* **unfolding** *st-red-def* **by** *blast*

Second, we find the clause *C2* and substitution η that are used to determine the value of *v* according to the definition of *trm-rep*, and we prove that they satisfy all the desired properties. In particular, clause *C2* is renamed to ensure that it shares no variable with *C*.

from ‹*subterm* (*subst t* σ) *p v*› **have**
si: ($\exists$ *x q1 q2*. (*is-a-variable x*) $\wedge$ (*subterm* (*subst x* σ) *q1 v*) $\wedge$
(*subterm t q2 x*) $\wedge$ (*p* = (*append q2 q1*))) $\vee$
(($\exists$ *u*. ($\neg$(*is-a-variable u*) $\wedge$ (*subterm t p u*) $\wedge$ (*v* = (*subst u* σ)))))
using *subterms-of-instances* **by** *metis*
let *?v* = *trm-rep v S*
from ‹*trm-rep v S* $\neq$ *v*› **and** ‹*root-term S v*› **have** *?v* $\in$ *min-trms* (*set-of-candidate-values S v*)
unfolding *root-term-def get-min-def* **by** (*metis some-in-eq*)
from ‹*?v* $\in$ *min-trms* (*set-of-candidate-values S v*)› **obtain** *pair* **where** *?v* = *fst pair*
and *pair* $\in$ (*set-of-candidate-values S v*) **and**
min-pair: ($\forall$ *pair'*$\in$*set-of-candidate-values S v*. (*snd pair'*, *snd pair*) $\notin$ *trm-ord*)

unfolding *min-trms-def* **by** *blast*

from ‹*pair* $\in$ (*set-of-candidate-values S v*)› **have**
$\exists$ *z CC C' C s L L'* σ *t' s'*. *pair* = (*z*, *s*) $\wedge$ (*candidate-values z CC C' C s L L'* σ *t' s' v S*)
unfolding *set-of-candidate-values-def* [*of S v*] **by** *blast*
from *this* **obtain** *zz C2-init Cl-C2-init gr-Cl-C2 gr-rhs gr-L2 L2-init* η*-init lhs-init rhs-init*
where *pair* = (*zz*, *gr-rhs*)
and (*candidate-values zz C2-init Cl-C2-init gr-Cl-C2 gr-rhs gr-L2 L2-init*
η*-init lhs-init rhs-init v S*)
by *blast*

from *assms*(*7*) **and** ‹*C* $\in$ *S*› **have** *finite* (*cl-ecl C*) **by** *auto*
from ‹(*candidate-values zz C2-init Cl-C2-init gr-Cl-C2 gr-rhs gr-L2 L2-init*
η*-init lhs-init rhs-init v S*)›
have *finite Cl-C2-init* **unfolding** *candidate-values-def* **by** *metis*

from *assms*(*11*) ‹*closed-under-renaming S*› ‹*finite Cl-C2-init*› ‹*finite* (*cl-ecl C*)›
‹(*candidate-values zz C2-init Cl-C2-init gr-Cl-C2 gr-rhs gr-L2 L2-init*
η*-init lhs-init rhs-init v S*)›
obtain *C2 Cl-C2* η *L2 lhs rhs* **where**
(*candidate-values zz C2 Cl-C2 gr-Cl-C2 gr-rhs gr-L2 L2* η *lhs rhs v S*)
and (*vars-of-cl* (*cl-ecl C*) $\cap$ *vars-of-cl* (*cl-ecl C2*)) = {}
using *candidate-values-renaming* [*of zz C2-init Cl-C2-init gr-Cl-C2 gr-rhs gr-L2 L2-init*
η*-init lhs-init rhs-init v S C*] **by** *auto*

from ‹(*candidate-values zz C2 Cl-C2 gr-Cl-C2 gr-rhs gr-L2 L2* η *lhs rhs v S*)›
and ‹*pair* = (*zz*, *gr-rhs*)› **and** ‹*?v* = *fst pair*›
have *cv*: (*candidate-values ?v C2 Cl-C2 gr-Cl-C2 gr-rhs gr-L2 L2* η *lhs rhs v S*)
by (*metis fst-conv*)
from *cv* **have** *C2* $\in$ *S*
unfolding *candidate-values-def* **by** *metis*
from *cv* **have** *ground-clause gr-Cl-C2*
unfolding *candidate-values-def* **by** *metis*

from *assms*(*7*) **and** *assms*(*10*) **have** *finite* (*vars-of-cl* (*cl-ecl C*))
using *set-of-variables-is-finite-cl* **by** *blast*

from *cv* **have** *smaller-lits-are-false v gr-Cl-C2 S*
unfolding *candidate-values-def* **by** *metis*

from *cv* **have** *gr-Cl-C2* = *subst-cl Cl-C2* η
unfolding *candidate-values-def* **by** *metis*
from *cv* **have** *orient-lit-inst L2 lhs rhs pos* η
unfolding *candidate-values-def* **by** *metis*
from *cv* **have** *maximal-literal gr-L2 gr-Cl-C2*
unfolding *candidate-values-def* **by** *metis*
from *cv* **have** *gr-L2* = *subst-lit L2* η
unfolding *candidate-values-def* **by** *metis*
from *cv* **have** *ground-clause gr-Cl-C2*
unfolding *candidate-values-def* **by** *metis*
from *cv* **have** *L2* $\in$ *Cl-C2*
unfolding *candidate-values-def* **by** *metis*
from *this* **and** ‹*gr-Cl-C2* = *subst-cl Cl-C2* η› **and** ‹*gr-L2* = *subst-lit L2* η›
have *gr-L2* $\in$ *gr-Cl-C2* **by** *auto*
from *cv* **have** *trm-rep v S* = *trm-rep gr-rhs S*
unfolding *candidate-values-def* **by** *metis*
from *cv* **have** (*gr-rhs*, *v*) $\in$ *trm-ord*
unfolding *candidate-values-def* **by** *metis*
from *cv* **have** *Cl-C2* = *cl-ecl C2*
unfolding *candidate-values-def* **by** *metis*
from *cv* **have** *v* $\notin$ *subst-set* (*trms-ecl C2*) η
unfolding *candidate-values-def* **by** *metis*

from *cv* **have** *sel* (*cl-ecl C2*) = {}
unfolding *candidate-values-def* **by** *metis*
from *this* **and** ‹*maximal-literal gr-L2 gr-Cl-C2*› **and** ‹*gr-Cl-C2* = *subst-cl Cl-C2* η›
and ‹*Cl-C2* = (*cl-ecl C2*)› **and** ‹*gr-L2* = *subst-lit L2* η› **have** *eligible-literal L2 C2* η
unfolding *eligible-literal-def* **by** *auto*
from *cv* **have** (*gr-rhs*, *v*) $\in$ *trm-ord*
unfolding *candidate-values-def* **by** *metis*

from *cv* **have** *norm*: ($\forall$ *x*. ($\exists$ *x'*$\in$ *trms-ecl C2*. *occurs-in x* (*subst x'* η)) $\longrightarrow$
(*x*, *v*) $\in$ *trm-ord* $\longrightarrow$ *trm-rep x S* = *x*)
unfolding *candidate-values-def trms-irreducible-def* **by** *metis*

from ‹*ground-clause gr-Cl-C2*› **and** ‹*gr-L2* $\in$ *gr-Cl-C2*› **have** *vars-of-lit gr-L2* = {}
by *auto*
from *cv* **have** *v* = *subst lhs* η **unfolding** *candidate-values-def* **by** *metis*
from *cv* **have** *gr-rhs* = *subst rhs* η **unfolding** *candidate-values-def* **by** *metis*

let *?I* = (*same-values* (λ*t*. (*trm-rep t S*)))

have *no-fact*: $\neg$ *equivalent-eq-exists lhs rhs Cl-C2* (*same-values* (λ*t*. *trm-rep t S*)) η *L2*
proof
assume *equivalent-eq-exists lhs rhs Cl-C2* (*same-values* (λ*t*. *trm-rep t S*)) η *L2*
from *this* **have** $\exists$ *L*$\in$*Cl-C2* − {*L2*}.$\exists$ *u v*. *orient-lit-inst L u v pos* η $\wedge$
subst lhs η = *subst u* η $\wedge$ *same-values* (λ*t*. *trm-rep t S*) (*subst rhs* η) (*subst v* η)
unfolding *equivalent-eq-exists-def*
by *blast*
from *this* **obtain** *M* **where** *M*$\in$*Cl-C2* − {*L2*} **and** *e*: $\exists$ *u v*. *orient-lit-inst M u v pos* η $\wedge$
subst lhs η = *subst u* η $\wedge$ *same-values* (λ*t*. *trm-rep t S*) (*subst rhs* η) (*subst v* η)
by *blast*
from *e* **obtain** *u' v'* **where** *orient-lit-inst M u' v' pos* η
and *i*: *subst lhs* η = *subst u'* η $\wedge$ *same-values* (λ*t*. *trm-rep t S*) (*subst rhs* η) (*subst v'* η)
by *blast*

from *i* **have** *subst lhs* η = *subst u'* η **by** *blast*
from *i* **have** *trm-rep* (*subst rhs* η) *S* = *trm-rep* (*subst v'* η) *S* **unfolding** *same-values-def* **by** *blast*
let *?u'* = (*subst u'* η)
let *?v'* = (*subst v'* η)
from ‹*orient-lit-inst M u' v' pos* η› **have** *orient-lit* (*subst-lit M* η) *?u' ?v' pos*
using *lift-orient-lit* **by** *auto*
from ‹*orient-lit-inst L2 lhs rhs pos* η› **have** *orient-lit* (*subst-lit L2* η) (*subst lhs*

η)
  (*subst rhs* η) *pos*
  using *lift-orient-lit* **by** *auto*
from ‹*orient-lit-inst M u′ v′ pos* η› **and** ‹*M* ∈ (*Cl-C2* − {*L2*})› **and** ‹*gr-Cl-C2* = *subst-cl Cl-C2* η›
  have *eq-occurs-in-cl ?u′ ?v′* (*Cl-C2* − {*L2*}) η **unfolding** *eq-occurs-in-cl-def* **by** *auto*

from ‹*M*∈*Cl-C2* − {*L2*}› **and** ‹*gr-Cl-C2* = *subst-cl Cl-C2* η›
  have (*subst-lit M* η) ∈ (*subst-cl* (*Cl-C2* − { *L2* }) η) **by** *auto*
from ‹*M*∈*Cl-C2* − {*L2*}› **and** ‹*gr-Cl-C2* = *subst-cl Cl-C2* η›
  have (*subst-lit M* η) ∈ *gr-Cl-C2* **by** *auto*

from ‹*vars-of-lit gr-L2* = {}› **and** ‹*gr-L2* = *subst-lit L2* η›
  ‹*orient-lit* (*subst-lit L2* η) (*subst lhs* η) (*subst rhs* η) *pos*›
  have *vars-of* (*subst rhs* η) = {} **using** *orient-lit-vars* **by** *blast*

from ‹*ground-clause gr-Cl-C2*› **and** ‹(*subst-lit M* η) ∈ *gr-Cl-C2*›
  have *vars-of-lit* (*subst-lit M* η) = {} **by** *auto*
from *this* **and** ‹*orient-lit* (*subst-lit M* η) *?u′ ?v′ pos*›
  have *vars-of ?v′* = {} **using** *orient-lit-vars* **by** *blast*

from ‹*maximal-literal gr-L2 gr-Cl-C2*› **and** ‹(*subst-lit M* η) ∈ *gr-Cl-C2*›
  have (*gr-L2*,(*subst-lit M* η)) ∉ *lit-ord*
  unfolding *maximal-literal-def* **by** *auto*

from *this* **and** ‹*orient-lit* (*subst-lit M* η) *?u′ ?v′ pos*›
  and ‹*orient-lit* (*subst-lit L2* η) (*subst lhs* η) (*subst rhs* η) *pos*›
  and ‹*subst lhs* η = *subst u′* η›
  and ‹*vars-of-lit gr-L2* = {}› **and** ‹*vars-of-lit* (*subst-lit M* η) = {}›
  and ‹*gr-L2* = *subst-lit L2* η› **have** ((*subst rhs* η),*?v′*) ∉ *trm-ord*
  using *lit-ord-rhs* **by** *auto*

from *this* **and** ‹*vars-of ?v′* = {}› **and** ‹*vars-of* (*subst rhs* η) = {}›
  have *?v′* = (*subst rhs* η) ∨ (*?v′*,(*subst rhs* η)) ∈ *trm-ord*
  using *trm-ord-ground-total* **unfolding** *ground-term-def* **by** *auto*
from *this* **and** ‹(*gr-rhs*,*v*) ∈ *trm-ord*› **and** ‹*gr-rhs* = *subst rhs* η› **have** (*?v′*,*v*) ∈ *trm-ord* **using** *trm-ord-trans* **unfolding** *trans-def* **by** *auto*
from *cv* **have** *maximal-literal-is-unique v gr-rhs Cl-C2 L2 S* η
  unfolding *candidate-values-def* **by** *metis*
from ‹*orient-lit-inst M u′ v′ pos* η› **have** ((*subst u′* η),(*subst v′* η)) ∉ *trm-ord*
  unfolding *orient-lit-inst-def* **by** *auto*

  have *trm-rep gr-rhs S* ≠ *trm-rep* (*subst v′* η) *S*
    by (*metis* ‹(*subst v′* η, *v*) ∈ *trm-ord*› ‹(*gr-rhs*, *v*) ∈ *trm-ord*›
      ‹*subst lhs* η = *subst u′* η›
      ‹*eq-occurs-in-cl* (*subst u′* η) (*subst v′* η) (*Cl-C2* − {*L2*}) η›
      ‹*maximal-literal-is-unique v gr-rhs Cl-C2 L2 S* η› ‹*v* = *subst lhs* η›
        maximal-literal-is-unique-def)

from *this* **and** ‹*trm-rep* (*subst rhs* η) *S* = *trm-rep* (*subst* v' η) *S*›
and ‹*gr-rhs* = (*subst rhs* η)› **show** *False* **by** *blast*
qed
from *this* ‹*gr-Cl-C2* = *subst-cl Cl-C2* η›
and ‹*gr-L2* = *subst-lit L2* η›
and ‹*smaller-lits-are-false v gr-Cl-C2 S*› **and** *assms*(*9*) **and** ‹*orient-lit-inst L2 lhs rhs pos* η›
and ‹*maximal-literal gr-L2 gr-Cl-C2*›
and ‹*ground-clause gr-Cl-C2*›
and ‹*gr-L2* $\in$ *gr-Cl-C2*› **and** ‹*v* = *subst lhs* η› ‹*gr-rhs* = *subst rhs* η›
and ‹*trm-rep v S* = *trm-rep gr-rhs S*›
have ($\neg$ *validate-ground-clause ?I* (*subst-cl* (*Cl-C2* $-$ { *L2* }) η))
using *if-all-smaller-are-false-then-cl-not-valid* [*of lhs* η *Cl-C2 S L2 rhs*] **by** *blast*

We fuse the substitutions σ and η so that the superposition rule can be applied:

from ‹*ground-clause* (*subst-cl* (*cl-ecl C*) σ)›
have *ground-on* (*vars-of-cl* (*cl-ecl C*)) σ **using** *ground-clauses-and-ground-substs* **by** *auto*
from ‹*finite* (*vars-of-cl* (*cl-ecl C*))› ‹(*vars-of-cl* (*cl-ecl C*) $\cap$ *vars-of-cl* (*cl-ecl C2*)) = {}›
‹*ground-on* (*vars-of-cl* (*cl-ecl C*)) σ› **obtain** σ' **where**
coincide-on σ' σ (*vars-of-cl* (*cl-ecl C*)) **and** *coincide-on* σ' η (*vars-of-cl* (*cl-ecl C2*))
using *combine-substs* [*of* (*vars-of-cl* (*cl-ecl C*)) (*vars-of-cl* (*cl-ecl C2*)) σ η] **by** *blast*

from ‹*coincide-on* σ' σ (*vars-of-cl* (*cl-ecl C*))› **have** *coincide-on* σ σ' (*vars-of-cl* (*cl-ecl C*))
using *coincide-sym* **by** *auto*
from ‹*coincide-on* σ' η (*vars-of-cl* (*cl-ecl C2*))› **have** *coincide-on* η σ' (*vars-of-cl* (*cl-ecl C2*))
using *coincide-sym* **by** *auto*

from ‹*eligible-literal L1 C* σ› ‹*L1* $\in$ (*cl-ecl C*)› ‹*coincide-on* σ σ' (*vars-of-cl* (*cl-ecl C*))›
have *eligible-literal L1 C* σ' **using** *eligible-literal-coincide* **by** *auto*

from ‹*eligible-literal L2 C2* η› ‹*L2* $\in$ *Cl-C2*› ‹*Cl-C2* = (*cl-ecl C2*)› ‹*coincide-on* η σ'
(*vars-of-cl* (*cl-ecl C2*))›
have *eligible-literal L2 C2* σ' **using** *eligible-literal-coincide* **by** *auto*

from ‹*ground-clause gr-Cl-C2*› **and** ‹*gr-Cl-C2* = (*subst-cl Cl-C2* η)›
have *ground-clause* (*subst-cl Cl-C2* σ')
by (*metis* ‹*Cl-C2* = *cl-ecl C2*› ‹*coincide-on* σ' η (*vars-of-cl* (*cl-ecl C2*))› *coincide-on-cl*)

from ‹*coincide-on* σ σ' (*vars-of-cl* (*cl-ecl C*))› ‹*L1* $\in$ (*cl-ecl C*)› **have**
coincide-on σ σ' (*vars-of-lit L1*) **unfolding** *coincide-on-def* **by** *auto*

from ‹*coincide-on* η σ' (*vars-of-cl* (*cl-ecl C2*))› ‹*L2* $\in$ *Cl-C2*› **and** ‹*Cl-C2* = (*cl-ecl C2*)›
have *coincide-on* η σ' (*vars-of-lit L2*) **unfolding** *coincide-on-def* **by** *auto*

from ‹(*orient-lit-inst L1 t s polarity* σ)› **and** ‹*coincide-on* σ σ' (*vars-of-lit L1*)›

have (*orient-lit-inst L1 t s polarity* σ')
using *orient-lit-inst-coincide* [*of L1 t s polarity* σ σ'] **by** *blast*

from ‹(*orient-lit-inst L2 lhs rhs pos* η)› **and** ‹*coincide-on* η σ' (*vars-of-lit L2*)›

have (*orient-lit-inst L2 lhs rhs pos* σ') **using** *orient-lit-inst-coincide* **by** *blast*

To prove that the superposition rule is applicable, we need to show that v does not occur inside a variable:

have $\neg$($\exists$ *x q1 q2*. (*is-a-variable x*) $\wedge$ (*subterm* (*subst x* σ) *q1 v*) $\wedge$
(*subterm t q2 x*) $\wedge$ (*p* = (*append q2 q1*)))
proof
assume ($\exists$ *x q1 q2*. (*is-a-variable x*) $\wedge$ (*subterm* (*subst x* σ) *q1 v*) $\wedge$
(*subterm t q2 x*) $\wedge$ (*p* = (*append q2 q1*)))
then obtain *x q1 q2* **where** *is-a-variable x subterm* (*subst x* σ) *q1 v*
(*subterm* (*subst x* σ) *q1 v*) (*subterm t q2 x*) **by** *auto*
from ‹(*subterm* (*subst x* σ) *q1 v*)› **have** *occurs-in v* (*subst x* σ)
unfolding *occurs-in-def* **by** *auto*
from ‹*is-a-variable x*› **obtain** x' **where** *x* = *Var* x' **using** *is-a-variable.elims*(*2*) **by** *blast*
from ‹*subterm t q2 x*› **have** *x* $\in$ *subterms-of t*
using *subterms-of.simps* **unfolding** *occurs-in-def* **by** *blast*
from *this* **have** *x* $\in$ *subterms-of-lit L1* **using** *assms*(*6*) **by** (*simp add*: *orient-lit-inst-subterms*)
from *this* ‹*L1* $\in$ (*cl-ecl C*)› **have** *x* $\in$ *subterms-of-cl* (*cl-ecl C*) **by** *auto*
from *this* **have** *vars-of x* $\subseteq$ *vars-of-cl* (*cl-ecl C*) **using** *subterm-vars* **by** *blast*
from *this* **and** ‹*x* = (*Var* x')› **have** x' $\in$ *vars-of-cl* (*cl-ecl C*) **by** *auto*
from ‹x' $\in$ *vars-of-cl* (*cl-ecl C*)› ‹*occurs-in v* (*subst x* σ)›
‹*x* = *Var* x'› *assms*(*2*) **have** *trm-rep v S* = *v* **by** *blast*
from *this* **and** ‹*trm-rep v S* $\neq$ *v*› **show** *False* **by** *blast*
qed
from *this* **and** *si* **obtain** *u* **where** $\neg$ (*is-a-variable u*) (*subterm t p u*) **and** *v* = (*subst u* σ)
by *auto*

from ‹*orient-lit-inst L1 t s polarity* σ› **have** *vars-of t* $\subseteq$ *vars-of-lit L1*
using *orient-lit-inst-vars* **by** *auto*
from ‹*subterm t p u*› **have** *vars-of u* $\subseteq$ *vars-of t* **using** *vars-subterm* **by** *auto*
from ‹*vars-of t* $\subseteq$ *vars-of-lit L1*› ‹*vars-of u* $\subseteq$ *vars-of t*› ‹*coincide-on* σ σ' (*vars-of-lit L1*)›

have *coincide-on* σ σ' (*vars-of u*) **unfolding** *coincide-on-def* **by** *blast*
from *this* **have** *subst u* σ = *subst u* σ' **using** *coincide-on-term* **by** *auto*

from ‹*orient-lit-inst L2 lhs rhs pos* η› **have** *vars-of lhs* ⊆ *vars-of-lit L2*
and *vars-of rhs* ⊆ *vars-of-lit L2* **using** *orient-lit-inst-vars* **by** *auto*
from ‹*vars-of lhs* ⊆ *vars-of-lit L2*› ‹*coincide-on* η σ' (*vars-of-lit L2*)›
have *coincide-on* η σ' (*vars-of lhs*) **unfolding** *coincide-on-def* **by** *blast*
from *this* **have** *subst lhs* η = *subst lhs* σ'
using *coincide-on-term* **by** *auto*

from ‹*vars-of rhs* ⊆ *vars-of-lit L2*› ‹*coincide-on* η σ' (*vars-of-lit L2*)›
have *coincide-on* η σ' (*vars-of rhs*) **unfolding** *coincide-on-def* **by** *blast*
from *this* **have** *subst rhs* η = *subst rhs* σ'
using *coincide-on-term* **by** *auto*

from ‹*trm-rep v S* = *trm-rep gr-rhs S*› **and** ‹*v*= *subst lhs* η› **and** ‹*gr-rhs* = (*subst rhs* η)›
have *trm-rep* (*subst rhs* η) *S* = *trm-rep* (*subst lhs* η) *S* **by** *metis*

from *this* **and** ‹*subst rhs* η = *subst rhs* σ'› ‹*subst lhs* η = *subst lhs* σ'›
have *trm-rep* (*subst rhs* σ') *S* = *trm-rep* (*subst lhs* σ') *S* **by** *metis*

from ‹*subst lhs* η = *subst lhs* σ'› ‹*subst u* σ = *subst u* σ'› ‹*v* = *subst u* σ› **and** ‹*v* = *subst lhs* η›
have *subst u* σ' = *subst lhs* σ' **by** *auto*

from ‹*coincide-on* σ' η (*vars-of-cl* (*cl-ecl C2*))› **and** ‹*Cl-C2* = (*cl-ecl C2*)›
have *coincide-on* σ' η (*vars-of-cl* (*Cl-C2* − { *L2* })) **unfolding** *coincide-on-def* **by** *auto*
from *this* **and** ‹(¬ *validate-ground-clause ?I* (*subst-cl* (*Cl-C2* − { *L2* }) η))›
have (¬ *validate-ground-clause ?I* (*subst-cl* (*Cl-C2* − { *L2* }) σ'))
using *coincide-on-cl* **by** *metis*

have (∀ *x*∈*cl-ecl C2* − {*L2*}. (*subst-lit x* σ', *subst-lit L2* σ') ∈ *lit-ord*)
proof
fix *x* **assume** *x* ∈*cl-ecl C2* − {*L2*}
from ‹*L2* ∈ *Cl-C2*› **and** ‹*gr-L2* = (*subst-lit L2* η)›
‹*gr-Cl-C2* = (*subst-cl Cl-C2* η)› **have** *gr-L2* ∈ *gr-Cl-C2* **by** *auto*
from *this* **and** ‹*ground-clause gr-Cl-C2*› **have** *vars-of-lit gr-L2* = {} **by** *auto*
from ‹*x* ∈ *cl-ecl C2* − {*L2*}› **and** ‹*Cl-C2* = (*cl-ecl C2*)› ‹*gr-Cl-C2* = (*subst-cl Cl-C2* η)›
have (*subst-lit x* η) ∈ *gr-Cl-C2* **by** *auto*
from *this* **and** ‹*ground-clause gr-Cl-C2*› **have** *vars-of-lit* (*subst-lit x* η) = {} **by** *auto*

from *this* ‹*x* ∈ *cl-ecl C2* − {*L2*}› ‹*maximal-literal gr-L2 gr-Cl-C2*› ‹*Cl-C2* = *cl-ecl C2*›
‹*gr-L2* = (*subst-lit L2* η)›
‹*gr-Cl-C2* = (*subst-cl Cl-C2* η)› ‹*orient-lit-inst L2 lhs rhs pos* η› *no-fact*

assms(*9*)
‹*vars-of-lit gr-L2* = {}› ‹*vars-of-lit* (*subst-lit x* η) = {}›
have (*subst-lit x* η, *subst-lit L2* η) $\in$ *lit-ord*
using *max-pos-lit-dominates-cl* [*of L2* η *Cl-C2 lhs rhs x ?I*] **by** *metis*

from ‹*L2* $\in$ *Cl-C2*› **have** *vars-of-lit L2* $\subseteq$ *vars-of-cl Cl-C2* **by** *auto*
from *this* **and** ‹*coincide-on* σ' η (*vars-of-cl* (*cl-ecl C2*))› **and** ‹*Cl-C2* = *cl-ecl C2*›
have *coincide-on* σ' η (*vars-of-lit L2*) **unfolding** *coincide-on-def* **by** *auto*
from *this* **have** *subst-lit L2* σ' = *subst-lit L2* η **using** *coincide-on-lit* **by** *auto*

from ‹$x \in$ (*cl-ecl C2*) $-$ {*L2*}› **have** $x \in$ *cl-ecl C2* **by** *auto*
from *this* **have** *vars-of-lit x* $\subseteq$ *vars-of-cl* (*cl-ecl C2*) **by** *auto*
from *this* **and** ‹*coincide-on* σ' η (*vars-of-cl* (*cl-ecl C2*))›
have *coincide-on* σ' η (*vars-of-lit x*) **unfolding** *coincide-on-def* **by** *auto*
from *this* **have** *subst-lit x* σ' = *subst-lit x* η **using** *coincide-on-lit* **by** *auto*

from ‹(*subst-lit x* η, *subst-lit L2* η) $\in$ *lit-ord*›
‹*subst-lit L2* σ' = *subst-lit L2* η›
‹*subst-lit x* σ' = *subst-lit x* η›
show (*subst-lit x* σ',*subst-lit L2* σ') $\in$ *lit-ord* **by** *metis*
qed

have *all-trms-irreducible* (*subst-set* (*trms-ecl C2*) σ') ($\lambda t.$ *trm-rep t S*)
proof (*rule ccontr*)
assume $\neg$*all-trms-irreducible* (*subst-set* (*trms-ecl C2*) σ') ($\lambda t.$ *trm-rep t S*)
from *this* **obtain** *x y* **where** $x \in$ (*subst-set* (*trms-ecl C2*) σ') **and** *occurs-in y x*
and *trm-rep y S* $\neq$ *y* **unfolding** *all-trms-irreducible-def* **by** *blast*
from ‹$x \in$ (*subst-set* (*trms-ecl C2*) σ')› **obtain** x' **where** $x' \in$ *trms-ecl C2*
and x = (*subst* x' σ') **by** *auto*
from *assms*(*11*) **and** ‹$x' \in$ (*trms-ecl C2*)› **and** ‹*C2* $\in$ *S*›
have *dom-trm* x' (*cl-ecl C2*) **unfolding** *Ball-def well-constrained-def* **by** *blast*
from *this* **obtain** x''
where $x'' \in$ *subterms-of-cl* (*cl-ecl C2*) **and** $x'' = x' \vee (x',x'') \in$ *trm-ord*
using *dom-trm-lemma* **by** *blast*
from ‹*dom-trm* x' (*cl-ecl C2*)› **have** *vars-of* x' $\subseteq$ *vars-of-cl* (*cl-ecl C2*)
using *dom-trm-vars* **by** *blast*
from *this* **and** ‹*coincide-on* σ' η (*vars-of-cl* (*cl-ecl C2*))› **have** *coincide-on* σ' η (*vars-of* x')
unfolding *coincide-on-def* **by** *auto*
from *this* **have** (*subst* x' η) = (*subst* x' σ') **using** *coincide-on-term* **by** *metis*
from *this* **and** ‹x = (*subst* x' σ')› **have** x = (*subst* x' η) **by** *auto*
from *this* **and** ‹$x' \in$ *trms-ecl C2*› **have** $x \in$(*subst-set* (*trms-ecl C2*) η)
by *auto*
from ‹$x'' \in$ (*subterms-of-cl* (*cl-ecl C2*))› **have**
(*subst* x'' η) $\in$ (*subterms-of-cl* (*subst-cl* (*cl-ecl C2*) η))
using *subterm-cl-subst* [*of* x'' *cl-ecl C2*] **by** *auto*

from ‹*orient-lit-inst L2 lhs rhs pos* η› ‹*gr-rhs* = (*subst rhs* η)›
‹*gr-L2* = (*subst-lit L2* η)›
have *orient-lit gr-L2* (*subst lhs* η) *gr-rhs pos*
using *lift-orient-lit*
by *auto*
from ‹*ground-clause gr-Cl-C2*› **have** *vars-of-cl gr-Cl-C2* = {} **by** *auto*
from ‹*vars-of-lit gr-L2* = {}› ‹*vars-of-cl gr-Cl-C2* = {}›
‹(*subst* x'' η) $\in$ (*subterms-of-cl* (*subst-cl* (*cl-ecl C2*) η))›
‹*orient-lit gr-L2* (*subst lhs* η) *gr-rhs pos*› ‹*maximal-literal gr-L2 gr-Cl-C2*›
‹*Cl-C2* = *cl-ecl C2*› ‹*gr-L2* = (*subst-lit L2* η)›
‹*gr-Cl-C2* = (*subst-cl Cl-C2* η)› ‹v = (*subst lhs* η)› ‹v = (*subst lhs* η)›
have (*subst* x'' η) = v $\vee$ (((*subst* x'' η),v) $\in$ *trm-ord*)
using *subterms-dominated* [*of gr-L2 gr-Cl-C2* (*subst lhs* η) *gr-rhs pos subst* x'' η]
by *metis*
from ‹$x'' = x' \vee (x',x'') \in$ *trm-ord*› ‹x = (*subst* x' η)› **have**
(*subst* x'' η) = x $\vee$ (x,(*subst* x'' η)) $\in$ *trm-ord*
using *trm-ord-subst* **by** *metis*
from *this* **and** ‹(*subst* x'' η) = v $\vee$ (((*subst* x'' η),v) $\in$ *trm-ord*)›
have $x = v \vee ((x,v) \in$ *trm-ord*) **using** *trm-ord-trans trans-def* **by** *metis*
then show *False*
proof
assume $x = v$
from *this* **and** ‹$v \notin$ *subst-set* (*trms-ecl C2*) η›
‹$x \in$ (*subst-set* (*trms-ecl C2*) η)› **show** *False* **by** *auto*
next
assume $(x,v) \in$ *trm-ord*
from ‹*occurs-in* y x› **have** $y = x \vee (y,x) \in$ *trm-ord* **unfolding** *occurs-in-def*
using *subterm-trm-ord-eq* **by** *auto*
from *this* **and** ‹$(x,v) \in$ *trm-ord*› **have** $(y,v) \in$ *trm-ord* **using** *trm-ord-trans*
unfolding *trans-def* **by** *metis*
from *this* **and** *norm* **and** ‹*trm-rep* y $S \neq y$› **and** ‹*occurs-in* y x› **and** ‹$x' \in$ *trms-ecl C2*›
and ‹x = (*subst* x' η)› **show** *False* **by** *metis*
qed
qed

from ‹*subterm* t p u› **have** $u = t \vee (u,t) \in$ *trm-ord* **using** *subterm-trm-ord-eq* **by** *auto*
from *assms*(*8*) **and** ‹*L1* $\in$ (*cl-ecl C*)› **have** *vars-of-lit* (*subst-lit L1* σ) = {}
by *auto*
from ‹*coincide-on* σ σ' (*vars-of-lit L1*)› **have** (*subst-lit L1* σ) = (*subst-lit L1* σ')
using *coincide-on-lit* **by** *metis*
from *this* **and** ‹*vars-of-lit* (*subst-lit L1* σ) = {}›
have *vars-of-lit* (*subst-lit L1* σ') = {} **by** *auto*

from ‹*coincide-on* η σ' (*vars-of-lit L2*)› **have** (*subst-lit L2* η) = (*subst-lit L2* σ')

```
    using coincide-on-lit by metis
  from ‹vars-of-lit gr-L2 = {}› ‹ground-clause gr-Cl-C2› ‹gr-Cl-C2 = (subst-cl
Cl-C2 η)›
    ‹L2 ∈ Cl-C2› have vars-of-lit (subst-lit L2 η) = {} by auto
  from this and ‹(subst-lit L2 η) = (subst-lit L2 σ′)›
   have vars-of-lit (subst-lit L2 σ′) = {} by auto
```

We now prove that the "into" clause is strictly smaller than the "from" clause. This is easy if the rewritten literal is negative or if the reduction does not occur at root level. Otherwise, we must use the fact that the function *trm-rep* selects the smallest right-hand side to compute the value of a term.

```
 have (subst-lit L2 σ′, subst-lit L1 σ′) ∈ lit-ord
  proof (rule ccontr)
   assume ¬(subst-lit L2 σ′, subst-lit L1 σ′) ∈ lit-ord
   from ‹orient-lit-inst L1 t s polarity σ′›
       have orient-lit (subst-lit L1 σ′)
         (subst t σ′) (subst s σ′) polarity
       using lift-orient-lit [of L1 t s polarity σ′] by auto
     from ‹orient-lit-inst L2 lhs rhs pos σ′›
       have orient-lit (subst-lit L2 σ′)
         (subst lhs σ′) (subst rhs σ′) pos
       using lift-orient-lit by auto
   have (u,t) ∉ trm-ord
   proof
     assume (u,t) ∈ trm-ord
     from this have (subst u σ′, subst t σ′) ∈ trm-ord
       using trm-ord-subst by auto
     have False subst u σ′ = subst lhs σ′
       using ‹(subst u σ′, subst t σ′) ∈ trm-ord›
       ‹(subst-lit L2 σ′, subst-lit L1 σ′) ∉ lit-ord› ‹subst lhs η = subst lhs σ′›
      ‹subst u σ = subst u σ′› ‹subst-lit L2 η = subst-lit L2 σ′› ‹gr-L2 = subst-lit
L2 η›
       ‹orient-lit (subst-lit L1 σ′) (subst t σ′) (subst s σ′) polarity›
        ‹orient-lit (subst-lit L2 σ′) (subst lhs σ′) (subst rhs σ′) pos› ‹v = subst lhs
η›
       ‹v = subst u σ› ‹vars-of-lit (subst-lit L1 σ′) = {}› ‹vars-of-lit gr-L2 = {}›
       lit-ord-dominating-term apply fastforce
       using ‹(subst u σ′, subst t σ′) ∈ trm-ord›
       ‹(subst-lit L2 σ′, subst-lit L1 σ′) ∉ lit-ord›
       ‹subst u σ′ = subst lhs σ′›
       ‹orient-lit (subst-lit L1 σ′) (subst t σ′) (subst s σ′) polarity›
       ‹orient-lit (subst-lit L2 σ′) (subst lhs σ′) (subst rhs σ′) pos›
       ‹vars-of-lit (subst-lit L1 σ′) = {}› ‹vars-of-lit (subst-lit L2 σ′) = {}›
       lit-ord-dominating-term by fastforce
     then show False by auto
   qed
   from this and ‹subterm t p u› have p = Nil using subterm-trm-ord by auto
```

have ¬ *proper-subterm-red t S σ*
proof
assume *proper-subterm-red t S σ*
from *this* **obtain** p' *s* **where** $p' \neq Nil$ **and** *subterm t* p' *s*
trm-rep (*subst s σ*) *S* ≠ (*subst s σ*)
unfolding *proper-subterm-red-def* **by** *blast*
from ‹*p* = *Nil*› **and** ‹$p' \neq Nil$› **have** $(p',p) \in$ (*pos-ord C t*)
using *pos-ord-nil* **by** *auto*
from ‹*subterm t* p' *s*› **have** *subterm* (*subst t σ*) p' (*subst s σ*)
by (*simp add*: *substs-preserve-subterms*)

from *this* **and** ‹$(p',p) \in$ (*pos-ord C t*)› *mr* **and** ‹*trm-rep* (*subst s σ*) *S* ≠ (*subst s σ*)› ‹*mp*=*p*›
show *False* **using** *minimal-redex-def* **by** *blast*
qed

from ‹(*u*,*t*) ∉ *trm-ord*› **and** ‹*u* = *t* ∨ (*u*,*t*) ∈ *trm-ord*› **have** *u* = *t* **by** *auto*
have *polarity* = *pos*
proof (*rule ccontr*)
assume *polarity* ≠ *pos*
then have *polarity* = *neg* **using** *sign.exhaust* **by** *auto*
from ‹*u* = *t*› **have** *subst t* σ' = *subst u* σ' **by** *auto*
from *this* **and** ‹*v* = (*subst u σ*)› **and** ‹*v* = (*subst lhs η*)›
and ‹*subst lhs η* = *subst lhs* σ'›
and ‹(*subst u σ*) = (*subst u* σ')›
have (*subst t* σ') = (*subst lhs* σ') **by** *auto*
from *this* **and** ‹*polarity* = *neg*› ‹*orient-lit* (*subst-lit L1* σ')
(*subst t* σ') (*subst s* σ') *polarity*›
and ‹*orient-lit* (*subst-lit L2* σ')
(*subst lhs* σ') (*subst rhs* σ') *pos*›
‹(*subst-lit L2* σ', *subst-lit L1* σ') ∉ *lit-ord*›
‹*vars-of-lit* (*subst-lit L1* σ') = {}›
‹*vars-of-lit* (*subst-lit L2* σ') = {}›
show *False* **using** *lit-ord-neg-lit-lhs* **by** *auto*
qed
from ‹*vars-of-lit* (*subst-lit L1 σ*) = {}› *assms*(*6*)
have *vars-of* (*subst t σ*) = {} **using** *lift-orient-lit orient-lit-vars*
by *blast*
from ‹*vars-of-lit* (*subst-lit L1 σ*) = {}› *assms*(*6*)
have *vars-of* (*subst s σ*) = {} **using** *lift-orient-lit orient-lit-vars*
by *blast*

have *trm-rep* (*subst t σ*) *S* ≠ *trm-rep* (*subst s σ*) *S*
proof
assume *trm-rep* (*subst t σ*) *S* = *trm-rep* (*subst s σ*) *S*
from *this* **have** *validate-ground-eq ?I* (*Eq* (*subst t σ*) (*subst s σ*))
unfolding *same-values-def* **using** *validate-ground-eq.simps* **by** (*metis* (*mono-tags*, *lifting*))
from ‹*trm-rep* (*subst t σ*) *S* = *trm-rep* (*subst s σ*) *S*›

have *validate-ground-eq ?I* (*Eq* (*subst s* σ) (*subst t* σ))
unfolding *same-values-def* **using** *validate-ground-eq.simps* **by** (*metis* (*mono-tags*, *lifting*))
from ‹*orient-lit-inst L1 t s polarity* σ› **and** ‹*polarity=pos*›
have *L1* = (*Pos* (*Eq t s*)) ∨ *L1* = (*Pos* (*Eq s t*))
unfolding *orient-lit-inst-def* **by** *auto*
from *this* **have** *subst-lit L1* σ = (*Pos* (*Eq* (*subst t* σ) (*subst s* σ))) ∨
subst-lit L1 σ = (*Pos* (*Eq* (*subst s* σ) (*subst t* σ))) **by** *auto*
from *this* **and** ‹*validate-ground-eq ?I* (*Eq* (*subst s* σ) (*subst t* σ))›
and ‹*validate-ground-eq ?I* (*Eq* (*subst t* σ) (*subst s* σ))›
have *validate-ground-lit ?I* (*subst-lit L1* σ) **using** *validate-ground-lit.simps*(*1*) **by** *metis*
from ‹*L1* ∈ (*cl-ecl C*)› **have** (*subst-lit L1* σ) ∈ (*subst-cl* (*cl-ecl C*) σ) **by** *auto*
from *this* **and** ‹*validate-ground-lit ?I* (*subst-lit L1* σ)›
have *validate-ground-clause ?I* (*subst-cl* (*cl-ecl C*) σ)
using *validate-ground-clause.simps* **by** *metis*
from *this* **and** ‹¬ *validate-ground-clause* (*int-clset S*) (*subst-cl* (*cl-ecl C*) σ)›
show *False* **unfolding** *int-clset-def* **by** *blast*
qed
have *cv*′: (*candidate-values* (*trm-rep* (*subst s* σ) *S*) *C* (*cl-ecl C*) (*subst-cl* (*cl-ecl C*) σ)
(*subst s* σ) (*subst-lit L1* σ) *L1* σ *t s v S*)
proof −
from ‹*polarity=pos*› **and** ‹*orient-lit-inst L1 t s polarity* σ› **have** ¬*negative-literal L1*
unfolding *orient-lit-inst-def* **by** *auto*
from *this* **and** ‹*eligible-literal L1 C* σ›
have *sel*(*cl-ecl C*) = {} **and** *maximal-literal* (*subst-lit L1* σ) (*subst-cl* (*cl-ecl C*) σ)
using *sel-neg* **unfolding** *eligible-literal-def* **by** *auto*
from ‹*v* = *subst u* σ› **and** ‹*u* = *t*› **have** *v* = *subst t* σ **by** *auto*
from *assms*(*7*) ‹*C* ∈ *S*› **have** *finite* (*cl-ecl C*) **by** *auto*
have *v* ∉ *subst-set* (*trms-ecl C*) σ
proof
assume *v* ∈ *subst-set* (*trms-ecl C*) σ
from *this* **and** *assms*(*12*) ‹*subterm* (*subst t* σ) *p v*› ‹*v* = *subst t* σ›
have *trm-rep v S* = *v* **unfolding** *all-trms-irreducible-def occurs-in-def* **by** *blast*
from *this* ‹*v* = *subst t* σ› ‹*trm-rep* (*subst t* σ) *S* ≠ (*subst t* σ)›
show *False* **by** *blast*
qed

from *assms*(*13*) **have** *smaller-lits-are-false v* (*subst-cl* (*cl-ecl C*) σ) *S*
using *smaller-lits-are-false-if-cl-not-valid* [*of S* (*subst-cl* (*cl-ecl C*) σ)] **by** *blast*
from *assms*(*1*) ‹¬ *proper-subterm-red t S* σ› ‹*polarity=pos*› ‹*v* = *subst t* σ› **have**
maximal-literal-is-unique v (*subst s* σ) (*cl-ecl C*) *L1 S* σ

using *maximal-literal-is-unique-lemma* [*of t s* (*cl-ecl C*) *S* σ *L1*] **by** *blast*
from ‹*all-trms-irreducible* (*subst-set* (*trms-ecl C*) σ) (λ*t. trm-rep t S*)›
have *trms-irreducible C* σ *S v* **using** *trms-irreducible-lemma* [*of C* σ *S v*]
by *blast*
have (*subst s* σ, *subst t* σ) $\in$ *trm-ord*
proof –
from ‹*orient-lit-inst L1 t s polarity* σ› **have** (*subst t* σ, *subst s* σ) $\notin$ *trm-ord*
unfolding *orient-lit-inst-def* **by** *auto*

from ‹*trm-rep* (*subst t* σ) *S* $\neq$ *trm-rep* (*subst s* σ) *S*›
have (*subst t* σ) $\neq$ (*subst s* σ) **by** *metis*
from *this* **and** ‹(*subst t* σ, *subst s* σ) $\notin$ *trm-ord*›
‹*vars-of* (*subst t* σ) = {}›
‹*vars-of* (*subst s* σ) = {}›
show (*subst s* σ, *subst t* σ) $\in$ *trm-ord*
using *trm-ord-ground-total* **unfolding** *ground-term-def* **by** *metis*
qed

from ‹*C* $\in$ *S*› ‹(*subst s* σ, *subst t* σ) $\in$ *trm-ord*›
and ‹*polarity=pos*› ‹*orient-lit-inst L1 t s polarity* σ› **and** ‹*sel* (*cl-ecl C*) =
{}›
and ‹*L1* $\in$ *cl-ecl C*›
and ‹*maximal-literal* (*subst-lit L1* σ) (*subst-cl* (*cl-ecl C*) σ)›
and ‹*ground-clause* (*subst-cl* (*cl-ecl C*) σ)› **and** ‹*v* = *subst t* σ›
and ‹*finite* (*cl-ecl C*)›
and ‹*v* $\notin$ *subst-set* (*trms-ecl C*) σ›
and ‹*smaller-lits-are-false v* (*subst-cl* (*cl-ecl C*) σ) *S*›
and ‹*maximal-literal-is-unique v* (*subst s* σ) (*cl-ecl C*) *L1 S* σ›
and ‹*trms-irreducible C* σ *S v*›
show *cv'*: (*candidate-values* (*trm-rep* (*subst s* σ) *S*) *C* (*cl-ecl C*) (*subst-cl*
(*cl-ecl C*) σ)
(*subst s* σ) (*subst-lit L1* σ) *L1* σ *t s v S*)
unfolding *candidate-values-def* **by** *blast*
qed

from *cv'* **have** (*trm-rep* (*subst s* σ) *S*,(*subst s* σ)) $\in$ *set-of-candidate-values S*
v
unfolding *set-of-candidate-values-def* **by** *blast*

from *this* **and** *min-pair* **and** ‹*pair* = (*zz*, *gr-rhs*)›
have ((*subst s* σ),*gr-rhs*) $\notin$ *trm-ord*
by (*metis snd-conv*)
have (*subst s* σ) $\neq$ *gr-rhs*
using ‹*trm-rep v S* = *trm-rep gr-rhs S*› ‹*u* = *t*› ‹*v* = *subst u* σ›
‹*trm-rep* (*subst t* σ) *S* $\neq$ *trm-rep* (*subst s* σ) *S*› **by** *blast*
have *vars-of gr-rhs* = {}
using ‹*subst rhs* η = *subst rhs* σ'›
‹*subst-lit L2* η = *subst-lit L2* σ'›
‹*gr-L2* = *subst-lit L2* η› ‹*gr-rhs* = *subst rhs* η›

‹*orient-lit* (*subst-lit L2* σ') (*subst lhs* σ') (*subst rhs* σ') *pos*›
‹*vars-of-lit gr-L2* = {}› *orient-lit-vars* **by** *fastforce*
from ‹(*subst s* σ) ≠ *gr-rhs*› **and** ‹*vars-of* (*subst s* σ) = {}› ‹*vars-of gr-rhs* = {}›
‹((*subst s* σ),*gr-rhs*) ∉ *trm-ord*›
have (*gr-rhs*,(*subst s* σ)) ∈ *trm-ord*
using *trm-ord-ground-total* **unfolding** *ground-term-def* **by** *blast*

have (*subst-lit L2* σ', *subst-lit L1* σ') ∈ *lit-ord*
using ‹(*gr-rhs*, *subst s* σ) ∈ *trm-ord*›
‹*subst lhs* η = *subst lhs* σ'› ‹*subst rhs* η = *subst rhs* σ'›
‹*subst-lit L1* σ = *subst-lit L1* σ'›
‹*gr-rhs* = *subst rhs* η›
‹*orient-lit* (*subst-lit L2* σ') (*subst lhs* σ') (*subst rhs* σ') *pos*›
‹*polarity* = *pos*› ‹*u* = *t*› ‹*v* = *subst lhs* η› ‹*v* = *subst u* σ›
‹*vars-of-lit* (*subst-lit L1* σ') = {}› ‹*vars-of-lit* (*subst-lit L2* σ') = {}›
assms(*6*) *lit-ord-rhs lift-orient-lit* **by** *fastforce*
from *this* **and** ‹(*subst-lit L2* σ', *subst-lit L1* σ') ∉ *lit-ord*› **show** *False* **by** *auto*
qed

have *trm-rep* (*subst u* σ) *S* ≠ (*subst u* σ)
using ‹*trm-rep v S* ≠ *v*› ‹*v* = *subst u* σ› **by** *blast*
have *allowed-redex u C* σ
proof (*rule ccontr*)
assume ¬*allowed-redex u C* σ
from *this* **obtain** *ss* **where** *ss* ∈ *trms-ecl C*
and *occurs-in* (*subst u* σ) (*subst ss* σ) **unfolding** *allowed-redex-def* **by** *auto*
from ‹*ss* ∈ *trms-ecl C*› **have** (*subst ss* σ) ∈ (*subst-set* (*trms-ecl C*) σ) **by** *auto*
from *this* **and** *assms*(*12*) **and** ‹*occurs-in* (*subst u* σ) (*subst ss* σ)›
‹*trm-rep* (*subst u* σ) *S* ≠ (*subst u* σ)›
show *False*
unfolding *all-trms-irreducible-def* **by** *blast*
qed
have *subst lhs* σ' ≠ *subst rhs* σ'
using ‹(*gr-rhs*, *v*) ∈ *trm-ord*›
‹*subst lhs* η = *subst lhs* σ'›
‹*subst rhs* η = *subst rhs* σ'›
‹*gr-rhs* = *subst rhs* η› ‹*v* = *subst lhs* η› *trm-ord-wf* **by** *auto*
from *this* ‹*mp*=*p*› ‹¬ (*is-a-variable u*)›
‹*all-trms-irreducible* (*subst-set* (*trms-ecl C2*) σ') (λ*t*. *trm-rep t S*)›
‹(*subst-lit L2* σ', *subst-lit L1* σ') ∈ *lit-ord*›
‹*all-trms-irreducible* (*subst-set* (*trms-ecl C2*) σ') (λ*t*. *trm-rep t S*)›
‹(∀ *x*∈*cl-ecl C2* − {*L2*}. (*subst-lit x* σ', *subst-lit L2* σ') ∈ *lit-ord*)›
‹*C2* ∈ *S*› ‹*eligible-literal L1 C* σ'› ‹*eligible-literal L2 C2* σ'›
‹*ground-clause* (*subst-cl Cl-C2* σ')› ‹*Cl-C2* = *cl-ecl C2*›
mr ‹*coincide-on* σ σ' (*vars-of-cl* (*cl-ecl C*))› ‹*L1* ∈ *cl-ecl C*› ‹*L2* ∈ *Cl-C2*›
‹*orient-lit-inst L1 t s polarity* σ'› ‹(*orient-lit-inst L2 lhs rhs pos* σ')›
‹(*subterm t p u*)› ‹*subst u* σ' = *subst lhs* σ'›
‹*trm-rep* (*subst rhs* σ') *S* = *trm-rep* (*subst lhs* σ') *S*›

‹(¬ *validate-ground-clause ?I* (*subst-cl* (*Cl-C2* − { *L2* }) σ'))›
‹*allowed-redex u C* σ›
have (*reduction L1 C* σ' *t s polarity L2 lhs u mp rhs C2* (*same-values* ($\lambda t.$ (*trm-rep t S*))) *S* σ)
unfolding *reduction-def same-values-def*
by *metis*
from ‹*vars-of-cl* (*cl-ecl C*) ∩ *vars-of-cl* (*cl-ecl C2*) = {}› **have** *variable-disjoint C C2*
unfolding *variable-disjoint-def* **by** *auto*
from *this* **and**
‹(*reduction L1 C* σ' *t s polarity L2 lhs u mp rhs C2* (*same-values* ($\lambda t.$ (*trm-rep t S*))) *S* σ)›
show *?thesis* **by** *blast*
qed

lemma *subts-of-irred-trms-are-irred*:
assumes *trm-rep y S* ≠ *y*
shows ⋀ *x*. *subterm x p y* ⟶ *trm-rep x S* ≠ *x*
proof (*induction p*)
case (*Nil*)
from *assms*(*1*) **show** *?case* **by** (*metis subterm.simps*(*1*))
next case (*Cons i p*)
show ⋀ *x*. *subterm x* (*Cons i p*) *y* ⟶ *trm-rep x S* ≠ *x*
proof
fix *x* **assume** *subterm x* (*Cons i p*) *y*
from *this* **obtain** *x1 x2* **where** *x* = *Comb x1 x2* **using** *subterm.elims*(*2*) **by** *blast*
have *i* = *Left* | *i* = *Right* **using** *indices.exhaust* **by** *auto*
then show *trm-rep x S* ≠ *x*
proof
assume *i* = *Left*
from *this* **and** ‹*subterm x* (*Cons i p*) *y*› ‹*x* = *Comb x1 x2*› **have** *subterm x1 p y* **by** *auto*
from *this* **and** *Cons.IH* **have** *trm-rep x1 S* ≠ *x1* **by** *blast*
from *this* **and** ‹*x* = *Comb x1 x2*› **have** *subterm-reduction-applicable S x*
unfolding *subterm-reduction-applicable-def*
by (*metis is-compound.simps*(*3*) *lhs.simps*(*1*))
from *this* **have** (*trm-rep x S*, *x*) ∈ *trm-ord* **using** *trm-rep-is-lower-subt-red* **by** *blast*
from *this* **show** *?thesis* **using** *trm-ord-irrefl* **unfolding** *irrefl-def* **by** *metis*
next
assume *i* = *Right*
from *this* **and** ‹*subterm x* (*Cons i p*) *y*› ‹*x* = *Comb x1 x2*› **have** *subterm x2 p y* **by** *auto*
from *this* **and** *Cons.IH* **have** *trm-rep x2 S* ≠ *x2* **by** *blast*
from *this* **and** ‹*x* = *Comb x1 x2*› **have** *subterm-reduction-applicable S x*
unfolding *subterm-reduction-applicable-def*
by (*metis is-compound.simps*(*3*) *rhs.simps*(*1*))
from *this* **have** (*trm-rep x S*, *x*) ∈ *trm-ord* **using** *trm-rep-is-lower-subt-red*

by *blast*
  from *this* **show** *?thesis* **using** *trm-ord-irrefl* **unfolding** *irrefl-def* **by** *metis*
  qed
 qed
qed

lemma *allowed-redex-coincide*:
 assumes *allowed-redex* t C σ
 assumes $t \in$ *subterms-of-cl* (*cl-ecl* C)
 assumes *coincide-on* σ σ' (*vars-of-cl* (*cl-ecl* C))
 assumes *well-constrained* C
 shows *allowed-redex* t C σ'
proof (*rule ccontr*)
 assume $\neg$*allowed-redex* t C σ'
 from *this* **obtain** s
  where $s \in$ *trms-ecl* C **and** *occurs-in* (*subst* t σ') (*subst* s σ')
  unfolding *allowed-redex-def* **by** *auto*
 from ‹$s \in$ *trms-ecl* C› **and** *assms*(*4*) **have** *vars-of* $s \subseteq$ *vars-of-cl* (*cl-ecl* C)
  using *dom-trm-vars* **unfolding** *well-constrained-def* **by** *blast*
 from *this* **have** *vars-of* $s \subseteq$ *vars-of-cl* (*cl-ecl* C) **using** *subterm-vars* **by** *blast*
 from *this* **and** *assms*(*3*) **have** *coincide-on* σ σ' (*vars-of* s) **unfolding** *coincide-on-def* **by** *auto*
 from *this* **have** (*subst* s σ) = (*subst* s σ') **using** *coincide-on-term* **by** *auto*
 from *assms*(*2*) **have** *vars-of* $t \subseteq$ *vars-of-cl* (*cl-ecl* C) **using** *subterm-vars* **by** *blast*
 from *this* **and** *assms*(*3*) **have** *coincide-on* σ σ' (*vars-of* t) **unfolding** *coincide-on-def* **by** *auto*
 from *this* **have** (*subst* t σ) = (*subst* t σ') **using** *coincide-on-term* **by** *auto*
 from *this* **and** ‹(*subst* s σ) = (*subst* s σ')› **and** ‹*occurs-in* (*subst* t σ') (*subst* s σ')›
  have *occurs-in* (*subst* t σ) (*subst* s σ) **by** *auto*
 from *this* **and** ‹$s \in$ *trms-ecl* C› **have** $\neg$*allowed-redex* t C σ **unfolding** *allowed-redex-def* **by** *auto*
 from *this* **and** *assms*(*1*) **show** *False* **by** *auto*
qed

The next lemma states that the irreducibility of an instance of a set of terms is preserved when the substitution is replaced by its equivalent normal form.

lemma *irred-terms-and-reduced-subst*:
 assumes $f = (\lambda t.$ (*trm-rep* t S))
 assumes $\eta =$ (*map-subst* f σ)
 assumes *all-trms-irreducible* (*subst-set* E σ) f
 assumes $I =$ *int-clset* S
 assumes *equivalent-on* σ η (*vars-of-cl* (*cl-ecl* C)) I
 assumes *lower-on* η σ (*vars-of-cl* (*cl-ecl* C))
 assumes $E =$ (*trms-ecl* C)
 assumes $\forall x \in S.\ \forall y.$ ($y \in$ *trms-ecl* $x \longrightarrow$ *dom-trm* y (*cl-ecl* x))
 assumes $C \in S$
 assumes *fo-interpretation* I

shows *all-trms-irreducible* (*subst-set E η*) *f*
proof (*rule ccontr*)
assume ¬*all-trms-irreducible* (*subst-set E η*) *f*
from *this* **obtain** *t y* **where** $y \in$ (*subst-set E η*) *occurs-in t y f t* $\neq$ *t*
unfolding *all-trms-irreducible-def* **by** *metis*
from ‹*occurs-in t y*› **obtain** *p* **where** *subterm y p t* **unfolding** *occurs-in-def* **by** *auto*
from *this* **and** ‹*f t* $\neq$ *t*› **and** *assms*(*1*) **have** *f y* $\neq$ *y* **using** *subts-of-irred-trms-are-irred* **by** *blast*
from ‹*y* ∈ (*subst-set E η*)› **obtain** *z* **where** *z* ∈ *E* **and** *y* = (*subst z η*)
by *auto*
from ‹*z* ∈ *E*› **have** (*subst z σ*) ∈ (*subst-set E σ*) **by** *auto*
have *subterm* (*subst z σ*) [] (*subst z σ*) **by** *auto*
then have *occurs-in* (*subst z σ*) (*subst z σ*) **unfolding** *occurs-in-def*
by *blast*
from *this* **and** *assms*(*3*) **and** ‹(*subst z σ*) ∈ (*subst-set E σ*)›
have *f* (*subst z σ*) = (*subst z σ*)
unfolding *all-trms-irreducible-def* **by** *metis*
from *this* **and** ‹*f y* ≠ *y*› **and** ‹*y* = (*subst z η*)›
have (*subst z σ*) ≠ (*subst z η*) **by** *metis*
from ‹*z* ∈ *E*› **and** *assms*(*7*) *assms*(*8*) *assms*(*9*) **have** *dom-trm z* (*cl-ecl C*) **by** *metis*
from *this* **have** *vars-of z* ⊆ *vars-of-cl* (*cl-ecl C*) **using** *dom-trm-vars* **by** *auto*
from *this* *assms*(*5*) **have** *equivalent-on σ η* (*vars-of z*) *I*
unfolding *equivalent-on-def* **by** *auto*
from ‹*vars-of z* ⊆ *vars-of-cl* (*cl-ecl C*)› *assms*(*6*) **have** *lower-on η σ* (*vars-of z*)

unfolding *lower-on-def* **by** *auto*
from ‹(*subst z σ*) ≠ (*subst z η*)›
‹*lower-on η σ* (*vars-of z*)›
have ((*subst z η*),(*subst z σ*)) ∈ *trm-ord*
using *lower-on-term* **unfolding** *lower-or-eq-def* **by** *metis*
from *this* **have** ((*subst z σ*),(*subst z η*)) ∉ *trm-ord*
using *trm-ord-trans trm-ord-irrefl irrefl-def trans-def* **by** *metis*
from *assms*(*10*) ‹*equivalent-on σ η* (*vars-of z*) *I*›
have (*I* (*subst z σ*) (*subst z η*)) **using** *equivalent-on-term*
unfolding *fo-interpretation-def* **by** *auto*
from *this* **and** *assms*(*4*) *assms*(*1*) ‹*f* (*subst z σ*) = (*subst z σ*)›
have (*subst z σ*) = *f* (*subst z η*) **unfolding** *same-values-def int-clset-def*
by *metis*
from *this* ‹((*subst z σ*),(*subst z η*)) ∉ *trm-ord*›
‹(*subst z σ*) ≠ (*subst z η*)› *assms*(*1*)
show *False* **using** *trm-rep-is-lower* **by** *metis*
qed

lemma *no-valid-literal*:
assumes *L* ∈ *C*
assumes *orient-lit-inst L t s pos σ*
assumes ¬(*validate-ground-clause* (*int-clset S*) (*subst-cl C σ*))

shows *trm-rep* (*subst t* σ) *S* $\neq$ *trm-rep* (*subst s* σ) *S*
proof
assume *neg-hyp*: *trm-rep* (*subst t* σ) *S* = *trm-rep* (*subst s* σ) *S*
let *?I* = *int-clset S*
from *neg-hyp* **have** *validate-ground-eq ?I* (*Eq* (*subst t* σ) (*subst s* σ))
unfolding *same-values-def int-clset-def* **using** *validate-ground-eq.simps*
by (*metis* (*mono-tags*, *lifting*))
from ‹*trm-rep* (*subst t* σ) *S* = *trm-rep* (*subst s* σ) *S*›
have *validate-ground-eq ?I* (*Eq* (*subst s* σ) (*subst t* σ))
unfolding *same-values-def int-clset-def* **using** *validate-ground-eq.simps*
by (*metis* (*mono-tags*, *lifting*))
from ‹*orient-lit-inst L t s pos* σ› **have** *L* = (*Pos* (*Eq t s*)) $\vee$ *L* = (*Pos* (*Eq s t*))
unfolding *orient-lit-inst-def* **by** *auto*
from *this* **have** *subst-lit L* σ = (*Pos* (*Eq* (*subst t* σ) (*subst s* σ))) $\vee$
subst-lit L σ = (*Pos* (*Eq* (*subst s* σ) (*subst t* σ))) **by** *auto*
from *this* **and** ‹*validate-ground-eq ?I* (*Eq* (*subst s* σ) (*subst t* σ))›
and ‹*validate-ground-eq ?I* (*Eq* (*subst t* σ) (*subst s* σ))›
have *validate-ground-lit ?I* (*subst-lit L* σ) **using** *validate-ground-lit.simps*(*1*) **by** *metis*
from *assms*(*1*) **have** (*subst-lit L* σ) $\in$ (*subst-cl C* σ) **by** *auto*
from ‹(*subst-lit L* σ) $\in$ (*subst-cl C* σ)›
and ‹*validate-ground-lit ?I* (*subst-lit L* σ)›
have *validate-ground-clause ?I* (*subst-cl C* σ)
using *validate-ground-clause.elims*(*3*) **by** *blast*
from *this* **and** ‹$\neg$ *validate-ground-clause ?I* (*subst-cl C* σ)› **show** *False* **by** *blast*
qed

7.2 Lifting

This section contains all the necessary lemmata for transforming ground inferences into first-order inferences. We show that all the necessary properties can be lifted.

lemma *lift-orient-lit-inst*:
assumes *orient-lit-inst L t s polarity* ϑ
assumes *subst-eq* ϑ (*comp* σ η)
shows *orient-lit-inst L t s polarity* σ
proof −
let *?*ϑ = (*comp* σ η)
have *polarity* = *pos* $\vee$ *polarity* = *neg* **using** *sign.exhaust* **by** *auto*
then show *?thesis*
proof
assume *polarity* = *pos*
from *this* **and** *assms*(*1*) **have** *L* = *Pos* (*Eq t s*) $\vee$ *L* = *Pos* (*Eq s t*)
and ((*subst t* ϑ), (*subst s* ϑ)) $\notin$ *trm-ord*
unfolding *orient-lit-inst-def* **by** *auto*
from *assms*(*2*) **have** (*subst t* ϑ) = (*subst* (*subst t* σ) η)
by *auto*

from *assms*(*2*) **have** (*subst s* ϑ) = (*subst* (*subst s* σ) η)
by *auto*
from ‹(*subst t* ϑ) = (*subst* (*subst t* σ) η)›
‹(*subst s* ϑ) = (*subst* (*subst s* σ) η)›
‹((*subst t* ϑ), (*subst s* ϑ)) $\notin$ *trm-ord*›
have ((*subst* (*subst t* σ) η),(*subst* (*subst s* σ) η)) $\notin$ *trm-ord*
by *auto*
from *this* **have** ((*subst t* σ), (*subst s* σ)) $\notin$ *trm-ord*
using *trm-ord-subst* **by** *auto*
from *this* **and** ‹*polarity* = *pos*› ‹*L* = *Pos* (*Eq t s*) $\vee$ *L* = *Pos* (*Eq s t*)› **show** *?thesis*
unfolding *orient-lit-inst-def* **by** *blast*
next
assume *polarity* = *neg*
from *this* **and** *assms*(*1*) **have** *L* = *Neg* (*Eq t s*) $\vee$ *L* = *Neg* (*Eq s t*)
and ((*subst t* ϑ), (*subst s* ϑ)) $\notin$ *trm-ord*
unfolding *orient-lit-inst-def* **by** *auto*
from *assms*(*2*) **have** (*subst t* ϑ) = (*subst* (*subst t* σ) η)
by *auto*
from *assms*(*2*) **have** (*subst s* ϑ) = (*subst* (*subst s* σ) η)
by *auto*
from ‹(*subst t* ϑ) = (*subst* (*subst t* σ) η)›
‹(*subst s* ϑ) = (*subst* (*subst s* σ) η)›
‹((*subst t* ϑ), (*subst s* ϑ)) $\notin$ *trm-ord*›
have ((*subst* (*subst t* σ) η),(*subst* (*subst s* σ) η)) $\notin$ *trm-ord*
by *auto*
from *this* **have** ((*subst t* σ), (*subst s* σ)) $\notin$ *trm-ord*
using *trm-ord-subst* **by** *auto*
from *this* **and** ‹*polarity* = *neg*› ‹*L* = *Neg* (*Eq t s*) $\vee$ *L* = *Neg* (*Eq s t*)› **show** *?thesis*
unfolding *orient-lit-inst-def* **by** *blast*
qed
qed

lemma *lift-maximal-literal*:
assumes *maximal-literal* (*subst-lit L* σ) (*subst-cl C* σ)
shows *maximal-literal L C*
proof (*rule ccontr*)
assume $\neg$*maximal-literal L C*
then obtain *M* **where** *M* $\in$ *C* **and** (*L*,*M*) $\in$ *lit-ord* **unfolding** *maximal-literal-def* **by** *auto*
from ‹*M* $\in$ *C*› **have** (*subst-lit M* σ) $\in$ (*subst-cl C* σ) **by** *auto*
from ‹(*L*,*M*) $\in$ *lit-ord*› **have** ((*subst-lit L* σ),(*subst-lit M* σ)) $\in$ *lit-ord*
using *lit-ord-subst* **by** *auto*
from *this* **and** ‹(*subst-lit M* σ) $\in$ (*subst-cl C* σ)› **and** *assms*(*1*)
show *False* **unfolding** *maximal-literal-def* **by** *auto*
qed

lemma *lift-eligible-literal*:

assumes *eligible-literal* L C σ
assumes $\sigma \doteq \vartheta \lozenge \eta$
shows *eligible-literal* L C ϑ
proof –
from *assms*(*1*) **have** ($L \in$ *sel* (*cl-ecl* C) $\vee$
(*sel*(*cl-ecl* C) = {}
$\wedge$ (*maximal-literal* (*subst-lit* L σ) (*subst-cl* (*cl-ecl* C) σ))))
unfolding *eligible-literal-def* **by** *auto*
then show *?thesis*
proof
assume $L \in$ *sel* (*cl-ecl* C)
then show *?thesis* **unfolding** *eligible-literal-def* **by** *auto*
next
assume *sel*(*cl-ecl* C) = {}
$\wedge$ (*maximal-literal* (*subst-lit* L σ) (*subst-cl* (*cl-ecl* C) σ))
then have *sel* (*cl-ecl* C) = {} **and** *maximal-literal* (*subst-lit* L σ) (*subst-cl* (*cl-ecl* C) σ)
by *auto*
let *?*$\sigma = \vartheta \lozenge \eta$
from *assms*(*2*) **have** (*subst-lit* L σ) = (*subst-lit* L *?*σ)
using *subst-eq-lit* **by** *auto*
then have (*subst-lit* L σ) = (*subst-lit* (*subst-lit* L ϑ) η)
using *composition-of-substs-lit* [*of* L ϑ η] **by** *auto*

from *assms*(*2*) **have** (*subst-cl* (*cl-ecl* C) σ) = (*subst-cl* (*cl-ecl* C) *?*σ)
using *subst-eq-cl* [*of* σ *?*σ (*cl-ecl* C)] **by** *auto*
then have (*subst-cl* (*cl-ecl* C) σ) = (*subst-cl* (*subst-cl* (*cl-ecl* C) ϑ) η)
using *composition-of-substs-cl* [*of* *cl-ecl* C ϑ η] **by** *auto*

from ‹*maximal-literal* (*subst-lit* L σ) (*subst-cl* (*cl-ecl* C) σ)›
‹(*subst-lit* L σ) = (*subst-lit* (*subst-lit* L ϑ) η)›
‹(*subst-cl* (*cl-ecl* C) σ) = (*subst-cl* (*subst-cl* (*cl-ecl* C) ϑ) η)›
have *maximal-literal* (*subst-lit* (*subst-lit* L ϑ) η)
(*subst-cl* (*subst-cl* (*cl-ecl* C) ϑ) η) **by** *auto*
from *this* **have** *maximal-literal* (*subst-lit* L ϑ) (*subst-cl* (*cl-ecl* C) ϑ)
using *lift-maximal-literal* **by** *metis*
from *this* **and** ‹*sel* (*cl-ecl* C) = {}› **show** *?thesis* **unfolding** *eligible-literal-def* **by** *auto*
qed
qed

lemma *lift-allowed-redex*:
assumes $\sigma \doteq \vartheta \lozenge \eta$
assumes (*allowed-redex* u C σ)
shows (*allowed-redex* u C ϑ)
proof (*rule ccontr*)
assume $\neg$(*allowed-redex* u C ϑ)
from *this* **obtain** s **where** $s \in$ (*trms-ecl* C) **and** (*occurs-in* (*subst* u ϑ) (*subst* s ϑ))

unfolding *allowed-redex-def* **by** *metis*
from ‹(*occurs-in* (*subst u* ϑ) (*subst s* ϑ))›
have (*occurs-in* (*subst* (*subst u* ϑ) η) (*subst* (*subst s* ϑ) η))
using *substs-preserve-occurs-in* **by** *auto*
from ‹$\sigma \doteq \vartheta \lozenge \eta$› **have** (*subst u* σ) = (*subst* (*subst u* ϑ) η) **by** *auto*
from ‹$\sigma \doteq \vartheta \lozenge \eta$› **have** (*subst s* σ) = (*subst* (*subst s* ϑ) η) **by** *auto*
from ‹(*occurs-in* (*subst* (*subst u* ϑ) η) (*subst* (*subst s* ϑ) η))›
‹(*subst u* σ) = (*subst* (*subst u* ϑ) η)›
‹(*subst s* σ) = (*subst* (*subst s* ϑ) η)›
have (*occurs-in* (*subst u* σ) (*subst s* σ)) **by** *auto*
from *this* **and** ‹$s \in$ (*trms-ecl C*)› *assms*(*2*) **show** *False* **unfolding** *allowed-redex-def*
by *auto*
qed

lemma *lift-decompose-literal*:
assumes *decompose-literal* (*subst-lit L* σ) *t s polarity*
assumes *subst-eq* ϑ (*comp* σ η)
shows *decompose-literal* (*subst-lit L* ϑ) (*subst t* η) (*subst s* η) *polarity*
proof −
let *?L* = (*subst-lit L* σ)
let *?t*′ = (*subst t* η)
let *?s*′ = (*subst s* η)

let *?*ϑ = (*comp* σ η)
let *?L*′ = (*subst-lit ?L* η)

from *assms*(*2*) **have** (*subst-lit L* ϑ) = (*subst-lit L ?*ϑ) **using**
subst-eq-lit **by** *auto*
from *this* **have** (*subst-lit L* ϑ) = *?L*′
using *composition-of-substs-lit* **by** *metis*

have *polarity* = *pos* ∨ *polarity* = *neg* **using** *sign.exhaust* **by** *auto*
then show *?thesis*
proof
assume *polarity* = *pos*
from *this* **and** *assms*(*1*) **have** *?L* = *Pos* (*Eq t s*) ∨ *?L* = *Pos* (*Eq s t*)
unfolding *decompose-literal-def decompose-equation-def* **by** *auto*
from ‹*?L* = *Pos* (*Eq t s*) ∨ *?L* = *Pos* (*Eq s t*)›
have *?L*′ = *Pos* (*Eq ?t*′ *?s*′) ∨ *?L*′ = *Pos* (*Eq ?s*′ *?t*′) **by** *auto*
from *this* ‹(*subst-lit L* ϑ) = *?L*′›
have (*subst-lit L* ϑ) = *Pos* (*Eq ?t*′ *?s*′) ∨ (*subst-lit L* ϑ) = *Pos* (*Eq ?s*′ *?t*′)
by *auto*
from *this* ‹*polarity* = *pos*› **show** *?thesis* **unfolding** *decompose-literal-def*
decompose-equation-def **by** *auto*
next
assume *polarity* = *neg*
from *this* **and** *assms*(*1*) **have** *?L* = *Neg* (*Eq t s*) ∨ *?L* = *Neg* (*Eq s t*)
unfolding *decompose-literal-def decompose-equation-def* **by** *auto*
from ‹*?L* = *Neg* (*Eq t s*) ∨ *?L* = *Neg* (*Eq s t*)›

have *?L′* = *Neg* (*Eq ?t′ ?s′*) ∨ *?L′* = *Neg* (*Eq ?s′ ?t′*) **by** *auto*
from *this* **and** ‹(*subst-lit L* ϑ) = *?L′*›
have (*subst-lit L* ϑ) = *Neg* (*Eq ?t′ ?s′*) ∨ (*subst-lit L* ϑ) = *Neg* (*Eq ?s′ ?t′*)
by *auto*
from *this* ‹*polarity* = *neg*› **show** *?thesis* **unfolding** *decompose-literal-def*
decompose-equation-def **by** *auto*
qed
qed

lemma *lift-dom-trm*:
assumes *dom-trm* (*subst t* ϑ) (*subst-cl C* ϑ)
assumes $\sigma \doteq \vartheta \lozenge \eta$
shows *dom-trm* (*subst t* σ) (*subst-cl C* σ)
proof −
let *?t* = (*subst t* ϑ)
let *?t′* = (*subst ?t* η)
let *?t″* = (*subst t* σ)
have *?t′* = (*subst t* (ϑ ◊ η)) **by** *auto*
from *assms*(*2*) **have** *?t″* = (*subst t* (ϑ ◊ η)) **by** *auto*
from *this* **and** ‹*?t′* = (*subst t* (ϑ ◊ η))› **have** *?t′* = *?t″* **by** *metis*
from *assms*(*1*) **have** (∃ *L u v p*. (*L* ∈ (*subst-cl C* ϑ) ∧ (*decompose-literal L u v*
p)
∧ (((*p* = *neg* ∧ *?t* = *u*) ∨ (*?t*,*u*) ∈ *trm-ord*)))) **unfolding** *dom-trm-def* **by**
auto
from *this* **obtain** *L u v p* **where** *L* ∈ (*subst-cl C* ϑ)
decompose-literal L u v p (((*p* = *neg* ∧ *?t* = *u*) ∨ (*?t*,*u*) ∈ *trm-ord*))
unfolding *dom-trm-def* **by** *blast*

from ‹*L* ∈ (*subst-cl C* ϑ)› **obtain** *L′* **where** *L′* ∈ *C*
L = (*subst-lit L′* ϑ) **by** *auto*
from *this* **and** ‹*decompose-literal L u v p*› **have** *decompose-literal* (*subst-lit L′* ϑ)
u v p **by** *auto*
from *this assms*(*2*) ‹*L* = (*subst-lit L′* ϑ)›
have *decompose-literal* (*subst-lit L′* σ) (*subst u* η) (*subst v* η) *p*
using *lift-decompose-literal* [*of L′* ϑ *u v p* σ η] **by** *auto*
let *?u* = (*subst u* η)
from ‹*L′* ∈ *C*› **have** (*subst-lit L′* σ) ∈ (*subst-cl C* σ) **by** *auto*
from ‹(((*p* = *neg* ∧ *?t* = *u*) ∨ (*?t*,*u*) ∈ *trm-ord*))›
have (((*p* = *neg* ∧ *?t′* = *?u*) ∨ (*?t′*,*?u*) ∈ *trm-ord*))
using *trm-ord-subst* **by** *auto*
from *this* **and** ‹*?t′* = *?t″*› **have** (((*p* = *neg* ∧ *?t″* = *?u*) ∨ (*?t″*,*?u*) ∈ *trm-ord*))
by *auto*
from *this* ‹(*subst-lit L′* σ) ∈ (*subst-cl C* σ)›
‹*decompose-literal* (*subst-lit L′* σ) (*subst u* η) (*subst v* η) *p*›
show *dom-trm* (*subst t* σ) (*subst-cl C* σ)
unfolding *dom-trm-def* **by** *auto*
qed

lemma *lift-irreducible-terms*:

assumes $T = \textit{get-trms}\ C\ (\textit{dom-trms}\ (\textit{subst-cl}\ D\ \sigma)\ (\textit{subst-set}\ E\ \sigma))\ \textit{Ground}$
assumes $\sigma \doteq \vartheta \lozenge \eta$
shows $\exists\, T'.\ (\ (\textit{subst-set}\ T'\ \eta) \subseteq T \wedge T' = \textit{get-trms}\ C'$
$(\textit{dom-trms}\ (\textit{subst-cl}\ D\ \vartheta)\ (\textit{subst-set}\ E\ \vartheta))\ \textit{FirstOrder})$
proof −
let $?E = (\textit{dom-trms}\ (\textit{subst-cl}\ D\ \vartheta)\ (\textit{subst-set}\ E\ \vartheta))$
let $?E' = (\textit{dom-trms}\ (\textit{subst-cl}\ D\ \sigma)\ (\textit{subst-set}\ E\ \sigma))$
let $?T' = (\textit{filter-trms}\ C'\ ?E)$
have $?T' = \textit{get-trms}\ C'\ ?E\ \textit{FirstOrder}$
unfolding *get-trms-def* **by** *auto*
from *assms*(*1*) **have** $T = ?E'$ **unfolding** *get-trms-def* **by** *auto*
have $(\textit{subst-set}\ ?T'\ \eta) \subseteq ?E'$
proof
fix x **assume** $x \in (\textit{subst-set}\ ?T'\ \eta)$
from *this* **obtain** x' **where** $x = (\textit{subst}\ x'\ \eta)$ **and** $x' \in\ ?T'$ **by** *auto*
from ‹$x' \in\ ?T'$› **have** $x' \in\ ?E$ **using** *filter-trms-inclusion* **by** *auto*
from ‹$x' \in\ ?E$› **have** $x' \in (\textit{subst-set}\ E\ \vartheta)$
and *dom-trm* x' $(\textit{subst-cl}\ D\ \vartheta)$ **unfolding** *dom-trms-def* **by** *auto*
from ‹$x' \in (\textit{subst-set}\ E\ \vartheta)$› **obtain** y **where** $y \in E$
and $x' = (\textit{subst}\ y\ \vartheta)$ **by** *auto*
from ‹$x' = (\textit{subst}\ y\ \vartheta)$› **and** ‹*dom-trm* x' $(\textit{subst-cl}\ D\ \vartheta)$›
have *dom-trm* $(\textit{subst}\ y\ \vartheta)\ (\textit{subst-cl}\ D\ \vartheta)$ **by** *auto*
from *this* *assms*(*2*)
have *dom-trm* $(\textit{subst}\ y\ \sigma)\ (\textit{subst-cl}\ D\ \sigma)$
using *lift-dom-trm* **by** *auto*
from ‹$y \in E$› **have** $(\textit{subst}\ y\ \sigma) \in (\textit{subst-set}\ E\ \sigma)$ **by** *auto*
from *this* **and** ‹*dom-trm* $(\textit{subst}\ y\ \sigma)\ (\textit{subst-cl}\ D\ \sigma)$›
have $(\textit{subst}\ y\ \sigma) \in\ ?E'$ **unfolding** *dom-trms-def* **by** *auto*
from *assms*(*2*) **have** $(\textit{subst}\ y\ \sigma) = (\textit{subst}\ y\ (\vartheta \lozenge \eta))$ **by** *auto*
from *this* ‹$x = (\textit{subst}\ x'\ \eta)$› **and** ‹$x' = (\textit{subst}\ y\ \vartheta)$›
have $x = (\textit{subst}\ y\ \sigma)$ **by** *auto*
from *this* **and** ‹$(\textit{subst}\ y\ \sigma) \in\ ?E'$› **show** $x \in\ ?E'$ **by** *auto*
qed
from *this* **and** ‹$T = ?E'$› ‹$?T' = \textit{get-trms}\ C'\ ?E\ \textit{FirstOrder}$› **show** *?thesis* **by** *auto*
qed

We eventually deduce the following lemmas, which allows one to transform ground derivations into first-order derivations.

lemma *lifting-lemma-superposition*:
assumes *superposition P1 P2 C* σ *Ground* C'
shows $\exists\, D\ \vartheta.\ \textit{superposition P1 P2}\ D\ \vartheta\ \textit{FirstOrder}\ C' \wedge \sigma \doteq \vartheta \lozenge \sigma \wedge \textit{trms-subsumes}\ D\ C\ \sigma$
proof (*rule ccontr*)
assume *hyp*: $\nexists\, D\ \vartheta.\ \textit{superposition P1 P2}\ D\ \vartheta\ \textit{FirstOrder}\ C' \wedge \sigma \doteq \vartheta \lozenge \sigma \wedge \textit{trms-subsumes}\ D\ C\ \sigma$

have *not-sup*: $\neg$ *superposition P1 P2 C* σ *Ground* C'
proof (*rule notI*)

assume *superposition P1 P2 C σ Ground C′*
from *this* **obtain** *L t s u v M p Cl-P1 Cl-P2 Cl-C polarity t′ u′ L′ trms-C*
where *L* ∈ *Cl-P1* (*M* ∈ *Cl-P2*) (*eligible-literal L P1 σ*) (*eligible-literal M P2 σ*)
(*variable-disjoint P1 P2*)
(*Cl-P1* = (*cl-ecl P1*)) (*Cl-P2* = (*cl-ecl P2*))
(¬ *is-a-variable u′*)
(*allowed-redex u′ P1 σ*)
(*C* = (*Ecl Cl-C trms-C*))
(*orient-lit-inst M u v pos σ*)
(*orient-lit-inst L t s polarity σ*)
((*subst u σ*) ≠ (*subst v σ*))
(*subterm t p u′*)
(*ck-unifier u′ u σ Ground*)
(*replace-subterm t p v t′*)
(*L′* = *mk-lit polarity* (*Eq t′ s*))
(*trms-C* = *get-trms Cl-C* (*dom-trms Cl-C* (*subst-set*
((*trms-ecl P1*) ∪ (*trms-ecl P2*) ∪
{ *r*. ∃ *q*. (*q*,*p*) ∈ (*pos-ord P1 t*) ∧ (*subterm t q r*) }) *σ*)) *Ground*)
(*Cl-C* = (*subst-cl* ((*Cl-P1* − { *L* }) ∪ ((*Cl-P2* − { *M* }) ∪ { *L′* })) *σ*))
(*C′* = (*Cl-P1* − { *L* }) ∪ ((*Cl-P2* − { *M* }) ∪ { *L′* }))
unfolding *superposition-def get-trms-def* **by** *auto*
from ‹(*ck-unifier u′ u σ Ground*)› **have** *Unifier σ u′ u*
unfolding *ck-unifier-def* **by** *auto*
from *this* **have** (*subst u′ σ*) = (*subst u σ*) **unfolding** *Unifier-def* **by** *auto*
from *this* **have** *unify u′ u* ≠ *None* **using** *MGU-exists* **by** *auto*
from *this* **obtain** *ϑ* **where** *unify u′ u* = *Some ϑ* **by** *auto*
hence *min-IMGU ϑ u′ u* **by** (*rule unify-computes-min-IMGU*)
with ‹*Unifier σ u′ u*› **have** *σ* ≐ *ϑ* ◊ *σ*
unfolding *min-IMGU-def IMGU-def* **by** *simp*
with ‹(*eligible-literal L P1 σ*)› **have** *eligible-literal L P1 ϑ*
using *lift-eligible-literal* **by** *auto*
from ‹*σ* ≐ *ϑ* ◊ *σ*› **and** ‹(*eligible-literal M P2 σ*)› **have** *eligible-literal M P2 ϑ*
using *lift-eligible-literal* **by** *auto*
from ‹*min-IMGU ϑ u′ u*› **have** *ck-unifier u′ u ϑ FirstOrder* **unfolding** *ck-unifier-def* **by** *auto*
from ‹*σ* ≐ *ϑ* ◊ *σ*› **have** (*subst u σ*) = (*subst* (*subst u ϑ*) *σ*) **by** *auto*
from ‹*σ* ≐ *ϑ* ◊ *σ*› **have** (*subst v σ*) = (*subst* (*subst v ϑ*) *σ*) **by** *auto*
from ‹((*subst u σ*) ≠ (*subst v σ*))›
‹(*subst u σ*) = (*subst* (*subst u ϑ*) *σ*)›
‹(*subst v σ*) = (*subst* (*subst v ϑ*) *σ*)›
have (*subst* (*subst u ϑ*) *σ*) ≠ (*subst* (*subst v ϑ*) *σ*) **by** *auto*
from *this* **have** (*subst u ϑ*) ≠ (*subst v ϑ*) **by** *auto*

from ‹*σ* ≐ *ϑ* ◊ *σ*› ‹*allowed-redex u′ P1 σ*› **have** *allowed-redex u′ P1 ϑ*
using *lift-allowed-redex*[*of σ ϑ σ*] **by** *auto*
from ‹*σ* ≐ *ϑ* ◊ *σ*› ‹*orient-lit-inst M u v pos σ*› **have** *orient-lit-inst M u v pos ϑ*
using *lift-orient-lit-inst* **by** *auto*
from ‹*σ* ≐ *ϑ* ◊ *σ*› ‹*orient-lit-inst L t s polarity σ*› **have** *orient-lit-inst L t s*

polarity ϑ
using *lift-orient-lit-inst* **by** *auto*

from ‹($Cl\text{-}C$ = (*subst-cl* (($Cl\text{-}P1$ − { L }) ∪ (($Cl\text{-}P2$ − { M }) ∪ { L' })) σ))›
and ‹C' = ($Cl\text{-}P1$ − { L }) ∪ (($Cl\text{-}P2$ − { M }) ∪ { L' })›
have ($Cl\text{-}C$ = (*subst-cl* C' σ)) **by** *auto*

obtain E **where** E = ((*trms-ecl* $P1$) ∪ (*trms-ecl* $P2$) ∪
{ r. ∃ q. (q,p) ∈ (*pos-ord* $P1$ t) ∧ (*subterm* t q r) }) **by** *auto*
from *this* **and** ‹($Cl\text{-}C$ = (*subst-cl* C' σ))›
‹*trms-C* = (*get-trms* $Cl\text{-}C$ (*dom-trms* $Cl\text{-}C$ (*subst-set*
((*trms-ecl* $P1$) ∪ (*trms-ecl* $P2$) ∪
{ r. ∃ q. (q,p) ∈ (*pos-ord* $P1$ t) ∧ (*subterm* t q r) }) σ)) *Ground*)›
have *trms-C* = (*get-trms* $Cl\text{-}C$
(*dom-trms* (*subst-cl* C' σ) (*subst-set*
E σ)) *Ground*)
by *auto*

let *?Cl-C'* = (*subst-cl* C' ϑ)
from ‹$\sigma \doteq \vartheta \lozenge \sigma$› ‹*trms-C* = (*get-trms* $Cl\text{-}C$
(*dom-trms* (*subst-cl* C' σ) (*subst-set*
E σ)) *Ground*)›
obtain ∃ T'. ((*subst-set* T' σ) ⊆ *trms-C* ∧ T' = *get-trms* *?Cl-C'*
(*dom-trms* (*subst-cl* C' ϑ) (*subst-set* E ϑ)) *FirstOrder*)
using *lift-irreducible-terms* **by** *auto*
from *this* **obtain** T' **where** (*subst-set* T' σ) ⊆ *trms-C*
and T' = *get-trms* *?Cl-C'*
(*dom-trms* (*subst-cl* C' ϑ) (*subst-set* E ϑ)) *FirstOrder* **by** *auto*

obtain *C-fo* **where** *C-fo* = (*Ecl* *?Cl-C'* T') **by** *auto*
from ‹C' = ($Cl\text{-}P1$ − { L }) ∪ (($Cl\text{-}P2$ − { M }) ∪ { L' })›
have (*?Cl-C'* = (*subst-cl* (($Cl\text{-}P1$ − { L }) ∪ (($Cl\text{-}P2$ − { M }) ∪ { L' })) ϑ))
by *auto*
from ‹L ∈ $Cl\text{-}P1$› ‹(M ∈ $Cl\text{-}P2$)› ‹(*eligible-literal* L $P1$ ϑ)› ‹(*eligible-literal* M $P2$ ϑ)›
‹(*variable-disjoint* $P1$ $P2$)›
‹($Cl\text{-}P1$ = (*cl-ecl* $P1$))› ‹($Cl\text{-}P2$ = (*cl-ecl* $P2$))›
‹(¬ *is-a-variable* u')›
‹(*allowed-redex* u' $P1$ ϑ)›
‹(*C-fo* = (*Ecl* *?Cl-C'* T'))›
‹(*orient-lit-inst* M u v *pos* ϑ)›
‹(*orient-lit-inst* L t s *polarity* ϑ)›
‹((*subst* u ϑ) ≠ (*subst* v ϑ))›
‹(*subterm* t p u')›
‹(*ck-unifier* u' u ϑ *FirstOrder*)›
‹(*replace-subterm* t p v t')›
‹(L' = *mk-lit* *polarity* (*Eq* t' s))›

‹T' = (*get-trms ?Cl-C'* (*dom-trms* (*subst-cl C'* ϑ) (*subst-set E* ϑ)) *FirstOrder*)›
‹(*?Cl-C'* = (*subst-cl* ((*Cl-P1* − { *L* }) ∪ ((*Cl-P2* − { *M* }) ∪ { *L'* })) ϑ))›
‹(*C'* = (*Cl-P1* − { *L* }) ∪ ((*Cl-P2* − { *M* }) ∪ { *L'* }))›
‹*E* = ((*trms-ecl P1*) ∪ (*trms-ecl P2*) ∪
{ *r*. ∃ *q*. (*q*,*p*) ∈ (*pos-ord P1 t*) ∧ (*subterm t q r*) })›
have *superposition P1 P2 C-fo* ϑ *FirstOrder C'* **unfolding** *superposition-def* **by** *blast*

have *subst-cl ?Cl-C'* σ = *subst-cl* (*subst-cl* ((*Cl-P1* − { *L* }) ∪ (*Cl-P2* − { *M* }) ∪ { *L'* }) ϑ) σ
using ‹*?Cl-C'* = *subst-cl* ((*Cl-P1* − { *L* }) ∪ ((*Cl-P2* − { *M* }) ∪ { *L'* })) ϑ› **by** *auto*
also have ... = (*subst-cl* ((*Cl-P1* − { *L* }) ∪ ((*Cl-P2* − { *M* }) ∪ { *L'* })) (ϑ $\lozenge$ σ))
using *composition-of-substs-cl* [*of* ((*Cl-P1* − { *L* }) ∪ ((*Cl-P2* − { *M* }) ∪ { *L'* }))] **by** *auto*
also have ... = (*subst-cl* ((*Cl-P1* − { *L* }) ∪ ((*Cl-P2* − { *M* }) ∪ { *L'* })) σ)
using *subst-eq-cl*[*OF* ‹$\sigma \doteq \vartheta \lozenge \sigma$›] **by** *blast*
also have ... = *Cl-C*
using ‹*Cl-C* = (*subst-cl* ((*Cl-P1* − { *L* }) ∪ ((*Cl-P2* − { *M* }) ∪ { *L'* })) σ)› **by** *argo*
finally have *subst-cl* (*cl-ecl C-fo*) σ = *cl-ecl C*
using ‹*C* = *Ecl Cl-C trms-C*› ‹*C-fo* = *Ecl ?Cl-C' T'*› **by** *simp*
moreover have (*subst-set* (*trms-ecl C-fo*) σ) ⊆ (*trms-ecl C*)
using ‹*subst-set T'* σ ⊆ *trms-C*› ‹*C* = *Ecl Cl-C trms-C*› ‹*C-fo* = *Ecl ?Cl-C' T'*› **by** *auto*
ultimately have (*trms-subsumes C-fo C* σ)
unfolding *trms-subsumes-def* **by** *auto*
with ‹*superposition P1 P2 C-fo* ϑ *FirstOrder C'*› ‹$\sigma \doteq \vartheta \lozenge \sigma$› *hyp* **show** *False* **by** *auto*
qed

from *not-sup* **and** *assms*(*1*) **show** *False* **by** *blast*
qed

lemma *lifting-lemma-factorization*:
assumes *factorization P1 C* σ *Ground C'*
shows ∃ *D* ϑ. *factorization P1 D* ϑ *FirstOrder C'* ∧ $\sigma \doteq \vartheta \lozenge \sigma$ ∧ *trms-subsumes D C* σ
proof (*rule ccontr*)
assume *hyp*: ∄ *D* ϑ. *factorization P1 D* ϑ *FirstOrder C'* ∧ $\sigma \doteq \vartheta \lozenge \sigma$ ∧ *trms-subsumes D C* σ

have *not-fact*: ¬ *factorization P1 C* σ *Ground C'*
proof (*rule notI*)
assume *factorization P1 C* σ *Ground C'*
from *this* **obtain** *L1 L2 L' t s u v Cl-P Cl-C trms-C* **where**
eligible-literal L1 P1 σ

L1 ∈ (*cl-ecl P1*) *L2* ∈ (*cl-ecl P1*) − { *L1* } *Cl-C* = (*cl-ecl C*) (*Cl-P* = (*cl-ecl P1*))
(*orient-lit-inst L1 t s pos* σ)
(*orient-lit-inst L2 u v pos* σ)
((*subst t* σ) ≠ (*subst s* σ))
(*subst t* σ) ≠ (*subst v* σ)
(*ck-unifier t u* σ *Ground*)
(*L*′ = *Neg* (*Eq s v*))
C = (*Ecl Cl-C trms-C*)
trms-C = (*get-trms Cl-C*
(*dom-trms Cl-C* (*subst-set* ((*trms-ecl P1*) ∪ (*proper-subterms-of t*)) σ)))
Ground
(*Cl-C* = (*subst-cl* ((*Cl-P* − { *L2* }) ∪ { *L*′ })) σ)
(*C*′ = ((*Cl-P* − { *L2* }) ∪ { *L*′ }))
unfolding *factorization-def get-trms-def* **by** *auto*
from ‹(*ck-unifier t u* σ *Ground*)› **have** *Unifier* σ *t u*
unfolding *ck-unifier-def Unifier-def* **by** *auto*
from *this* **have** (*subst t* σ) = (*subst u* σ) **unfolding** *Unifier-def* **by** *auto*
from *this* **have** *unify t u* ≠ *None* **using** *MGU-exists* **by** *auto*
from *this* **obtain** ϑ **where** *unify t u* = *Some* ϑ **by** *auto*
hence *min-IMGU* ϑ *t u* **by** (*rule unify-computes-min-IMGU*)
with ‹*Unifier* σ *t u*› **have** σ ≐ ϑ ◊ σ
unfolding *min-IMGU-def IMGU-def* **by** *simp*
with ‹(*eligible-literal L1 P1* σ)› **have** *eligible-literal L1 P1* ϑ
using *lift-eligible-literal* **by** *auto*
from ‹*min-IMGU* ϑ *t u*› **have** *ck-unifier t u* ϑ *FirstOrder* **unfolding** *ck-unifier-def* **by** *auto*
from ‹σ ≐ ϑ ◊ σ› **have** (*subst t* σ) = (*subst* (*subst t* ϑ) σ) **by** *auto*
from ‹σ ≐ ϑ ◊ σ› **have** (*subst s* σ) = (*subst* (*subst s* ϑ) σ) **by** *auto*
from ‹σ ≐ ϑ ◊ σ› **have** (*subst v* σ) = (*subst* (*subst v* ϑ) σ) **by** *auto*

from ‹((*subst t* σ) ≠ (*subst s* σ))›
‹(*subst t* σ) = (*subst* (*subst t* ϑ) σ)›
‹(*subst s* σ) = (*subst* (*subst s* ϑ) σ)›
have (*subst* (*subst t* ϑ) σ) ≠ (*subst* (*subst s* ϑ) σ) **by** *auto*
from *this* **have** (*subst t* ϑ) ≠ (*subst s* ϑ) **by** *auto*

from ‹((*subst t* σ) ≠ (*subst v* σ))›
‹(*subst t* σ) = (*subst* (*subst t* ϑ) σ)›
‹(*subst v* σ) = (*subst* (*subst v* ϑ) σ)›
have (*subst* (*subst t* ϑ) σ) ≠ (*subst* (*subst v* ϑ) σ) **by** *auto*
from *this* **have** (*subst t* ϑ) ≠ (*subst v* ϑ) **by** *auto*

from ‹σ ≐ ϑ ◊ σ› ‹*orient-lit-inst L1 t s pos* σ› **have** *orient-lit-inst L1 t s pos* ϑ
using *lift-orient-lit-inst* **by** *auto*
from ‹σ ≐ ϑ ◊ σ› ‹*orient-lit-inst L2 u v pos* σ› **have** *orient-lit-inst L2 u v pos* ϑ
using *lift-orient-lit-inst* **by** *auto*

from ‹$(Cl\text{-}C = (subst\text{-}cl\ (\ (Cl\text{-}P\ -\ \{\ L2\ \})\ \cup\ \{\ L'\ \}\ ))\ \sigma)$›
and ‹$C' = (\ (Cl\text{-}P\ -\ \{\ L2\ \})\ \cup\ \{\ L'\ \}\ )$›
have $(Cl\text{-}C = (subst\text{-}cl\ C'\ \sigma))$ **by** *auto*

obtain E **where** $E = (trms\text{-}ecl\ P1)$ **by** *auto*
from *this* **and** ‹$(Cl\text{-}C = (subst\text{-}cl\ C'\ \sigma))$›
‹$trms\text{-}C = (get\text{-}trms\ \ Cl\text{-}C$
$(dom\text{-}trms\ Cl\text{-}C\ (subst\text{-}set\ (\ (trms\text{-}ecl\ P1)\ \cup\ (proper\text{-}subterms\text{-}of\ t)\ )\ \sigma)))$
$Ground$›
have $trms\text{-}C = (get\text{-}trms\ Cl\text{-}C$
$(dom\text{-}trms\ (subst\text{-}cl\ C'\ \sigma)\ (subst\text{-}set$
$(E \cup (proper\text{-}subterms\text{-}of\ t))\ \ \sigma))\ Ground)$
by *auto*

let $?Cl\text{-}C' = (subst\text{-}cl\ C'\ \vartheta)$
from ‹$\sigma \doteq \vartheta \lozenge \sigma$› ‹$trms\text{-}C = (get\text{-}trms\ Cl\text{-}C$
$(dom\text{-}trms\ (subst\text{-}cl\ C'\ \sigma)\ (subst\text{-}set$
$(E \cup (proper\text{-}subterms\text{-}of\ t))\ \ \sigma))\ Ground)$›
obtain T' **where** $(subst\text{-}set\ T'\ \sigma) \subseteq trms\text{-}C$
and $T' = get\text{-}trms\ ?Cl\text{-}C'$
$(dom\text{-}trms\ (subst\text{-}cl\ C'\ \vartheta)\ (subst\text{-}set\ (E \cup (proper\text{-}subterms\text{-}of\ t))\ \vartheta))\ FirstOrder$
using *lift-irreducible-terms* **by** *blast*

obtain $C\text{-}fo$ **where** $C\text{-}fo = (Ecl\ ?Cl\text{-}C'\ T')$ **by** *auto*
from ‹$C' = (\ (Cl\text{-}P\ -\ \{\ L2\ \})\ \cup\ \{\ L'\ \}\ )$›
have $(?Cl\text{-}C' = (subst\text{-}cl\ (\ (Cl\text{-}P\ -\ \{\ L2\ \})\ \cup\ \{\ L'\ \}\ )\ \vartheta))$
by *auto*
from ‹$C\text{-}fo = (Ecl\ ?Cl\text{-}C'\ T')$› **have** $?Cl\text{-}C' = (cl\text{-}ecl\ C\text{-}fo)$ **by** *auto*
have $?Cl\text{-}C' = (subst\text{-}cl\ C'\ \vartheta)$ **by** *auto*
from
‹$eligible\text{-}literal\ L1\ P1\ \vartheta$›
‹$L1 \in (cl\text{-}ecl\ P1)$› ‹$L2 \in (cl\text{-}ecl\ P1) - \{\ L1\ \}$› ‹$?Cl\text{-}C' = (cl\text{-}ecl\ C\text{-}fo)$› ‹$(Cl\text{-}P$
$= (cl\text{-}ecl\ P1))$›
‹$(orient\text{-}lit\text{-}inst\ L1\ t\ s\ pos\ \vartheta)$›
‹$(orient\text{-}lit\text{-}inst\ L2\ u\ v\ pos\ \vartheta)$›
‹$((subst\ t\ \vartheta) \neq (subst\ s\ \vartheta))$›
‹$(subst\ t\ \vartheta) \neq (subst\ v\ \vartheta)$›
‹$(ck\text{-}unifier\ t\ u\ \vartheta\ FirstOrder)$›
‹$(L' = Neg\ (Eq\ s\ v))$›
‹$C\text{-}fo = (Ecl\ ?Cl\text{-}C'\ T')$›
‹$T' = get\text{-}trms?Cl\text{-}C'$
$(dom\text{-}trms\ ?Cl\text{-}C'\ (subst\text{-}set\ (E \cup (proper\text{-}subterms\text{-}of\ t))\ \vartheta))\ FirstOrder$›
‹$(?Cl\text{-}C' = (subst\text{-}cl\ (\ (Cl\text{-}P\ -\ \{\ L2\ \})\ \cup\ \{\ L'\ \}\ )\ \vartheta))$›
‹$C' = (\ (Cl\text{-}P\ -\ \{\ L2\ \})\ \cup\ \{\ L'\ \}\ )$›
‹$E = (trms\text{-}ecl\ P1)$›
have *factorization P1 C-fo* ϑ *FirstOrder* C' **unfolding** *factorization-def* **by**
blast

have i: $subst\text{-}cl\ ?Cl\text{-}C'\ \sigma = subst\text{-}cl\ (subst\text{-}cl\ (Cl\text{-}P\ -\ \{\ L2\ \} \cup \{\ L'\ \})\ \vartheta)\ \sigma$

using ‹*?Cl-C*$'$ = *subst-cl* ((*Cl-P* $-$ { *L2* }) $\cup$ { *L*$'$ }) ϑ› **by** *auto*
have *ii*: *subst-cl* (*subst-cl* ((*Cl-P* $-$ { *L2* }) $\cup$ { *L*$'$ }) ϑ) σ
= *subst-cl* ((*Cl-P* $-$ { *L2* }) $\cup$ { *L*$'$ }) ($\vartheta \lozenge \sigma$)
using *composition-of-substs-cl* [*of* ((*Cl-P* $-$ { *L2* }) $\cup$ { *L*$'$ })] **by** *auto*
from ‹$\sigma \doteq \vartheta \lozenge \sigma$› **have** (*subst-cl* ((*Cl-P* $-$ { *L2* }) $\cup$ { *L*$'$ }) σ)
= (*subst-cl* ((*Cl-P* $-$ { *L2* }) $\cup$ { *L*$'$ }) ($\vartheta \lozenge \sigma$))
using *subst-eq-cl* [*of* σ $\vartheta \lozenge \sigma$ ((*Cl-P* $-$ { *L2* }) $\cup$ { *L*$'$ })] **by** *auto*
with *i ii* ‹*Cl-C* = (*subst-cl* ((*Cl-P* $-$ { *L2* }) $\cup$ { *L*$'$ }) σ)›
have (*subst-cl ?Cl-C*$'$ σ) = *Cl-C* **by** *metis*
with ‹*C* = *Ecl Cl-C trms-C*› ‹*C-fo* = *Ecl ?Cl-C*$'$ *T*$'$› **have** *subst-cl* (*cl-ecl C-fo*) σ = *cl-ecl C*
by *auto*
from ‹(*subst-set T*$'$ σ) $\subseteq$ *trms-C*›
and ‹(*C* = (*Ecl Cl-C trms-C*))› **and** ‹(*C-fo* = (*Ecl ?Cl-C*$'$ *T*$'$))›
have (*subst-set* (*trms-ecl C-fo*) σ) $\subseteq$ (*trms-ecl C*) **by** *auto*
from ‹(*subst-cl* (*cl-ecl C-fo*) σ) = (*cl-ecl C*)› ‹(*subst-set* (*trms-ecl C-fo*) σ) $\subseteq$ (*trms-ecl C*)›
have (*trms-subsumes C-fo C* σ)
unfolding *trms-subsumes-def* **by** *auto*
with ‹*factorization P1 C-fo* ϑ *FirstOrder C*$'$› ‹$\sigma \doteq \vartheta \lozenge \sigma$› *hyp* **show** *False* **by** *auto*
qed

from *not-fact* **and** *assms*(*1*) **show** *False* **by** *blast*
qed

lemma *lifting-lemma-reflexion*:
assumes *reflexion P1 C* σ *Ground C*$'$
shows $\exists$ *D* ϑ. *reflexion P1 D* ϑ *FirstOrder C*$'$ $\wedge$ $\sigma \doteq \vartheta \lozenge \sigma$ $\wedge$ *trms-subsumes D C* σ
proof (*rule ccontr*)
assume *hyp*: $\nexists$ *D* ϑ. *reflexion P1 D* ϑ *FirstOrder C*$'$ $\wedge$ $\sigma \doteq \vartheta \lozenge \sigma$ $\wedge$ *trms-subsumes D C* σ

have *not-ref*: $\neg$ *reflexion P1 C* σ *Ground C*$'$
proof (*rule notI*)
assume *reflexion P1 C* σ *Ground C*$'$
from *this* **obtain** *L1 t s Cl-P Cl-C trms-C* **where**
eligible-literal L1 P1 σ
L1 $\in$ *cl-ecl P1 Cl-C* = *cl-ecl C Cl-P* = *cl-ecl P1*
orient-lit-inst L1 t s neg σ
ck-unifier t s σ *Ground*
C = *Ecl Cl-C trms-C*
trms-C = *get-trms Cl-C*
(*dom-trms Cl-C* (*subst-set* (*trms-ecl P1* $\cup$ { *t* }) σ)) *Ground*
Cl-C = *subst-cl* (*Cl-P* $-$ { *L1* }) σ
C$'$ = *Cl-P* $-$ { *L1* }
unfolding *reflexion-def get-trms-def* **by** *auto*

from ‹*ck-unifier t s* σ *Ground*› **have** *Unifier* σ *t s*
unfolding *ck-unifier-def Unifier-def* **by** *auto*

hence *subst t* σ = *subst s* σ **unfolding** *Unifier-def* **by** *auto*
hence *unify t s* $\neq$ *None* **using** *MGU-exists* **by** *auto*
then obtain ϑ **where** *unify t s* = *Some* ϑ **by** *auto*
hence *min-IMGU* ϑ *t s* **by** (*rule unify-computes-min-IMGU*)
with ‹*Unifier* σ *t s*› **have** $\sigma \doteq \vartheta \lozenge \sigma$
unfolding *IMGU-def min-IMGU-def* **by** *simp*
with ‹*eligible-literal L1 P1* σ› **have** *eligible-literal L1 P1* ϑ
using *lift-eligible-literal* **by** *auto*

from ‹*min-IMGU* ϑ *t s*› **have** *ck-unifier t s* ϑ *FirstOrder* **unfolding** *ck-unifier-def* **by** *auto*

from ‹$\sigma \doteq \vartheta \lozenge \sigma$› ‹*orient-lit-inst L1 t s neg* σ› **have** *orient-lit-inst L1 t s neg* ϑ
using *lift-orient-lit-inst* **by** *auto*

from ‹*Cl-C* = *subst-cl* (*Cl-P* − { *L1* }) σ› **and** ‹C' = *Cl-P* − { *L1* }›
have *Cl-C* = *subst-cl* C' σ **by** *auto*

obtain *E* **where** *E* = (*trms-ecl P1*) **by** *auto*
with ‹*Cl-C* = *subst-cl* C' σ›
‹*trms-C* = *get-trms Cl-C* (*dom-trms Cl-C* (*subst-set* (*trms-ecl P1* $\cup$ { *t* }) σ)) *Ground*›

have *trms-C* = *get-trms Cl-C* (*dom-trms* (*subst-cl* C' σ) (*subst-set* (*E* $\cup$ { *t* }) σ)) *Ground*
by *auto*

let *?Cl-C'* = *subst-cl* C' ϑ

from ‹$\sigma \doteq \vartheta \lozenge \sigma$›
‹*trms-C* = *get-trms Cl-C* (*dom-trms* (*subst-cl* C' σ) (*subst-set* (*E* $\cup$ { *t* }) σ)) *Ground*›
obtain T' **where**
subst-set T' $\sigma \subseteq$ *trms-C* **and**
T' = *get-trms ?Cl-C'* (*dom-trms* (*subst-cl* C' ϑ) (*subst-set* (*E* $\cup$ { *t* }) ϑ)) *FirstOrder*
using *lift-irreducible-terms* **by** *blast*

obtain *C-fo* **where** *C-fo* = *Ecl ?Cl-C'* T' **by** *auto*

from ‹C' = *Cl-P* − { *L1* }›
have *?Cl-C'* = *subst-cl* (*Cl-P* − { *L1* }) ϑ **by** *auto*

from ‹*C-fo* = *Ecl ?Cl-C'* T'› **have** *?Cl-C'* = *cl-ecl C-fo* **by** *auto*

have *?Cl-C′* = (*subst-cl C′ ϑ*) **by** *auto*

from
‹*eligible-literal L1 P1 ϑ*›
‹*L1* ∈ *cl-ecl P1*› ‹*?Cl-C′* = *cl-ecl C-fo*› ‹*Cl-P* = *cl-ecl P1*›
‹*orient-lit-inst L1 t s neg ϑ*›
‹*ck-unifier t s ϑ FirstOrder*›
‹*C-fo* = *Ecl ?Cl-C′ T′*›
‹*T′* = *get-trms ?Cl-C′*
(*dom-trms* (*subst-cl C′ ϑ*) (*subst-set* (*E* ∪ { *t* }) *ϑ*)) *FirstOrder*›
‹*?Cl-C′* = *subst-cl* (*Cl-P* − { *L1* }) *ϑ*›
‹*C′* = *Cl-P* − { *L1* }›
‹*E* = *trms-ecl P1*›
have *reflexion P1 C-fo ϑ FirstOrder C′*
unfolding *reflexion-def* **by** *metis*

have *i*: *subst-cl ?Cl-C′ σ* = *subst-cl* (*subst-cl* (*Cl-P* − { *L1* }) *ϑ*) *σ*
using ‹*?Cl-C′* = *subst-cl* (*Cl-P* − { *L1* }) *ϑ*› **by** *auto*

have *ii*: *subst-cl* (*subst-cl* (*Cl-P* − { *L1* }) *ϑ*) *σ* = *subst-cl* (*Cl-P* − { *L1* }) (*ϑ* ◊ *σ*)
using *composition-of-substs-cl*[*of Cl-P* − { *L1* }] **by** *auto*

from ‹*σ* ≐ *ϑ* ◊ *σ*› **have** *subst-cl* (*Cl-P* − { *L1* }) *σ* = *subst-cl* (*Cl-P* − { *L1* }) (*ϑ* ◊ *σ*)
using *subst-eq-cl*[*of σ ϑ* ◊ *σ Cl-P* − { *L1* }] **by** *auto*
with *i ii* ‹*Cl-C* = (*subst-cl* ((*Cl-P* − { *L1* })) *σ*)›
have *subst-cl ?Cl-C′ σ* = *Cl-C* **by** *metis*
with ‹*C* = *Ecl Cl-C trms-C*› ‹*C-fo* = *Ecl ?Cl-C′ T′*›
have *subst-cl* (*cl-ecl C-fo*) *σ* = *cl-ecl C* **by** *auto*

from ‹*subst-set T′ σ* ⊆ *trms-C*› ‹*C* = *Ecl Cl-C trms-C*› ‹*C-fo* = *Ecl ?Cl-C′ T′*›
have *subst-set* (*trms-ecl C-fo*) *σ* ⊆ *trms-ecl C* **by** *auto*

from ‹*subst-cl* (*cl-ecl C-fo*) *σ* = *cl-ecl C*› ‹*subst-set* (*trms-ecl C-fo*) *σ* ⊆ *trms-ecl C*›
have *trms-subsumes C-fo C σ* **unfolding** *trms-subsumes-def* **by** *auto*
with ‹*reflexion P1 C-fo ϑ FirstOrder C′*› ‹*σ* ≐ *ϑ* ◊ *σ*› *hyp* **show** *False* **by** *auto*
qed

from *not-ref* **and** *assms*(*1*) **show** *False* **by** *blast*
qed

lemma *lifting-lemma*:
assumes *deriv*: *derivable C P S σ Ground C′*
shows ∃ *D ϑ*. ((*derivable D P S ϑ FirstOrder C′*) ∧ (*σ* ≐ *ϑ* ◊ *σ*) ∧ (*trms-subsumes D C σ*))
proof (*rule ccontr*)

assume *hyp*: $\neg$ ($\exists$ *D* ϑ. *derivable D P S* ϑ *FirstOrder* C' $\wedge$ $\sigma \doteq \vartheta \lozenge \sigma$ $\wedge$ *trms-subsumes D C* σ)

from *deriv* **have** $P \subseteq S$ **unfolding** *derivable-def* **by** *auto*

have *not-sup*: $\neg$ ($\exists$ *P1 P2*. P = { *P1*, *P2* } $\wedge$ *superposition P1 P2 C* σ *Ground* C')
using *lifting-lemma-superposition*
by (*metis* ‹$P \subseteq S$› *derivable-def hyp insert-subset*)

have *not-fact*: $\neg$ ($\exists$ *P1*. { *P1* } = P $\wedge$ *factorization P1 C* σ *Ground* C')
using *lifting-lemma-factorization*
by (*metis* ‹$P \subseteq S$› *derivable-def hyp insert-subset*)

have *not-ref*: $\neg$ ($\exists$ *P1*. { *P1* } = P $\wedge$ *reflexion P1 C* σ *Ground* C')
using *lifting-lemma-reflexion*
by (*metis* ‹$P \subseteq S$› *derivable-def hyp insert-subset*)

from *not-sup not-ref not-fact deriv* **show** *False* **unfolding** *derivable-def* **by** *blast*
qed

lemma *trms-subsumes-and-red-inf*:
assumes *trms-subsumes D C* η
assumes *redundant-inference* (*subst-ecl D* η) *S P* σ
assumes $\sigma \doteq \vartheta \lozenge \eta$
shows *redundant-inference C S P* σ
proof –
from *assms*(*2*) **obtain** S' **where** $S' \subseteq$ (*instances S*)
(*set-entails-clause* (*clset-instances* S') (*cl-ecl* (*subst-ecl D* η)))
($\forall x \in S'$. (*subterms-inclusion* (*subst-set* (*trms-ecl* (*fst x*)) (*snd x*))
(*trms-ecl* (*subst-ecl D* η)))) **and**
ball-S'*-le*: $\forall x \in S'$. $\exists D' \in$ *cl-ecl* ‘ *P*. ((*cl-ecl* (*fst x*), *snd x*), (D', σ)) $\in$ *cl-ord*
unfolding *redundant-inference-def* **by** *auto*
from *assms*(*1*) **have** (*subst-cl* (*cl-ecl D*) η) = (*cl-ecl C*)
unfolding *trms-subsumes-def* **by** *auto*
obtain *Cl-D T* **where** *D* = *Ecl Cl-D T* **using** *eclause.exhaust* **by** *auto*
from *this* **have** (*cl-ecl D*) = *Cl-D* **and** *trms-ecl D* = *T* **by** *auto*
from ‹*D* = *Ecl Cl-D T*› **have** *subst-ecl D* η = *Ecl* (*subst-cl Cl-D* η) (*subst-set T* η)
by *auto*
from *this* **have** (*cl-ecl* (*subst-ecl D* η)) = (*subst-cl Cl-D* η)
and *trms-ecl* (*subst-ecl D* η) = (*subst-set T* η) **by** *auto*
from ‹(*cl-ecl* (*subst-ecl D* η)) = (*subst-cl Cl-D* η)›
and ‹(*cl-ecl D*) = *Cl-D*› ‹(*subst-cl* (*cl-ecl D*) η) = (*cl-ecl C*)›
have (*cl-ecl* (*subst-ecl D* η)) = (*cl-ecl C*) **by** *auto*
from *this* **and** ‹(*set-entails-clause* (*clset-instances* S') (*cl-ecl* (*subst-ecl D* η)))›
have (*set-entails-clause* (*clset-instances* S') (*cl-ecl C*)) **by** *auto*

from ‹*trms-ecl D* = *T*› **and** ‹*trms-ecl* (*subst-ecl D* η) = (*subst-set T* η)›

have *trms-ecl* (*subst-ecl D* η) = (*subst-set* (*trms-ecl D*) η) **by** *auto*

from *assms*(*1*) **have** (*subst-set* (*trms-ecl D*) η) $\subseteq$ (*trms-ecl C*)
unfolding *trms-subsumes-def* **by** *auto*
have *ii*: ($\forall x \in S'$. (*subterms-inclusion* (*subst-set* (*trms-ecl* (*fst x*)) (*snd x*))
(*trms-ecl C*)))
proof (*rule ccontr*)
assume $\neg$ ($\forall x \in S'$. (*subterms-inclusion* (*subst-set* (*trms-ecl* (*fst x*)) (*snd x*))
(*trms-ecl C*)))
from *this* **obtain** x **where** $x \in S'$ **and**
$\neg$(*subterms-inclusion* (*subst-set* (*trms-ecl* (*fst x*)) (*snd x*)) (*trms-ecl C*))
by *auto*
from ‹$x \in S'$› **and** ‹($\forall x \in S'$. (*subterms-inclusion* (*subst-set* (*trms-ecl* (*fst x*))
(*snd x*))
(*trms-ecl* (*subst-ecl D* η)))))›
have (*subterms-inclusion* (*subst-set* (*trms-ecl* (*fst x*)) (*snd x*))
(*trms-ecl* (*subst-ecl D* η))) **by** *auto*

obtain *E1* **where** *E1* = (*subst-set* (*trms-ecl* (*fst x*)) (*snd x*)) **by** *auto*
obtain *E2* **where** *E2* = (*subst-set* (*trms-ecl D*) η) **by** *auto*
obtain *E2'* **where** *E2'* = (*trms-ecl C*) **by** *auto*
from ‹*E2* = (*subst-set* (*trms-ecl D*) η)› ‹*E2'* = (*trms-ecl C*)›
‹(*subst-set* (*trms-ecl D*) η) $\subseteq$ (*trms-ecl C*)›
have *E2* $\subseteq$ *E2'* **by** *auto*
from ‹*E1* = (*subst-set* (*trms-ecl* (*fst x*)) (*snd x*))›
‹*E2* = (*subst-set* (*trms-ecl D*) η)›
‹*trms-ecl* (*subst-ecl D* η) = (*subst-set* (*trms-ecl D*) η)›
‹(*subterms-inclusion* (*subst-set* (*trms-ecl* (*fst x*)) (*snd x*))
(*trms-ecl* (*subst-ecl D* η)))› **have** *subterms-inclusion E1 E2* **by** *auto*
from *this* **and** ‹*E2* $\subseteq$ *E2'*› **have** *subterms-inclusion E1 E2'*
using *subterms-inclusion-subset* **by** *auto*
from *this* **and** ‹*E1* = (*subst-set* (*trms-ecl* (*fst x*)) (*snd x*))› ‹*E2'* = (*trms-ecl*
C)›
and ‹$\neg$(*subterms-inclusion* (*subst-set* (*trms-ecl* (*fst x*)) (*snd x*))
(*trms-ecl C*))› **show** *False* **by** *auto*
qed
from *this* **and** ‹(*set-entails-clause* (*clset-instances S'*) (*cl-ecl C*))›
and *ball-S'-le*
and ‹$S' \subseteq$ (*instances S*)›
show *redundant-inference C S P* σ **unfolding** *redundant-inference-def* **by** *auto*
qed

lemma *lift-inference*:
assumes *inference-saturated S*
shows *ground-inference-saturated S*
proof (*rule ccontr*)
assume $\neg$ (*ground-inference-saturated S*)
from *this* **obtain** *C P* σ *C'* **where** *derivable C P S* σ *Ground C' ground-clause*
(*cl-ecl C*)

grounding-set P σ ¬redundant-inference C S P σ **unfolding** *ground-inference-saturated-def*

by *blast*
from ‹*derivable C P S σ Ground C′*› **obtain** *D ϑ η* **where** *derivable D P S ϑ FirstOrder C′*
$\sigma \doteq \vartheta \lozenge \eta$ *trms-subsumes D C η* **using** *lifting-lemma* **by** *blast*
from ‹*trms-subsumes D C η*› **and** ‹*¬redundant-inference C S P σ*› ‹$\sigma \doteq \vartheta \lozenge \eta$›
have ¬ *redundant-inference* (*subst-ecl D η*) *S P σ*
using *trms-subsumes-and-red-inf* **by** *auto*
from *this* **and** ‹*derivable C P S σ Ground C′*› ‹*derivable D P S ϑ FirstOrder C′*› ‹*trms-subsumes D C η*›
‹$\sigma \doteq \vartheta \lozenge \eta$› ‹*ground-clause* (*cl-ecl C*)› ‹*grounding-set P σ*›
‹¬ *redundant-inference* (*subst-ecl D η*) *S P σ*›
assms(*1*) **show** *False* **unfolding** *inference-saturated-def* **by** *auto*
qed

lemma *lift-redundant-cl* :
assumes *C′* = *subst-cl D ϑ*
assumes *redundant-clause C S η C′*
assumes $\sigma \doteq \vartheta \lozenge \eta$
assumes *finite D*
shows *redundant-clause C S σ D*
proof −
from *assms*(*2*) **have**
(∃ *S′*. (*S′* ⊆ (*instances S*) ∧ (*set-entails-clause* (*clset-instances S′*) (*cl-ecl C*)) ∧
(∀ *x* ∈ *S′*. (*subterms-inclusion* (*subst-set* (*trms-ecl* (*fst x*)) (*snd x*)) (*trms-ecl C*))) ∧
(∀ *x* ∈ *S′*. (((*mset-ecl* ((*fst x*),(*snd x*))),(*mset-cl* (*C′*,*η*))) ∈ (*mult* (*mult trm-ord*))
∨ (*mset-ecl* ((*fst x*),(*snd x*))) = *mset-cl* (*C′*,*η*))))))
unfolding *redundant-clause-def* **by** *auto*
from *this* **obtain** *S′* **where** *i*: *S′* ⊆ (*instances S*)
and *ii*: (*set-entails-clause* (*clset-instances S′*) (*cl-ecl C*))
and *iii*: (∀ *x* ∈ *S′*. (*subterms-inclusion* (*subst-set* (*trms-ecl* (*fst x*)) (*snd x*)) (*trms-ecl C*)))
and
iv: (∀ *x* ∈ *S′*. (((*mset-ecl* ((*fst x*),(*snd x*))),(*mset-cl* (*C′*,*η*))) ∈ (*mult* (*mult trm-ord*))
∨ (*mset-ecl* ((*fst x*),(*snd x*))) = *mset-cl* (*C′*,*η*)))
by *auto*
let *?m1* = *mset-cl* (*C′*,*η*)
let *?m2* = *mset-cl* (*D*,*σ*)
from *assms*(*1*) *assms*(*3*) *assms*(*4*)
have *mset-cl* (*C′*,*η*) = *mset-cl* (*D*,*σ*) ∨ (*mset-cl* (*C′*,*η*),*mset-cl* (*D*,*σ*)) ∈ (*mult* (*mult trm-ord*))
using *mset-subst* **by** *auto*
from *this iv*
have (∀ *x* ∈ *S′*. (((*mset-ecl* ((*fst x*),(*snd x*))),(*mset-cl* (*D*,*σ*))) ∈ (*mult* (*mult*

trm-ord))
  ∨ (*mset-ecl* ((*fst x*),(*snd x*))) = *mset-cl* (*D*,σ)))
  using *mult-mult-trm-ord-trans* **unfolding** *trans-def* **by** *metis*
 from *this* **and** *i ii iii* **show** *?thesis* **unfolding** *redundant-clause-def* **by** *meson*
qed

We deduce the following (trivial) lemma, stating that sets that are closed under all inferences are also saturated.

lemma *inference-closed-sets-are-saturated*:
 assumes *inference-closed S*
 assumes $\forall x \in S$. (*finite* (*cl-ecl x*))
 shows *clause-saturated S*
proof (*rule ccontr*)
 assume ¬*?thesis*
 from *this* **obtain** $C\ P\ \sigma\ C'\ D\ \vartheta\ \eta$
  where
  (*derivable C P S* σ *Ground* C') (*ground-clause* (*cl-ecl C*))
  (*derivable D P S* ϑ *FirstOrder* C') (*trms-subsumes D C* η)
  ($\sigma \doteq \vartheta \lozenge \eta$)
  ¬(*redundant-clause* (*subst-ecl D* η) *S* σ C')
   unfolding *clause-saturated-def* **by** *blast*
 from ‹(*derivable D P S* ϑ *FirstOrder* C')› *assms*(*1*) **have** $D \in S$
  unfolding *inference-closed-def* **by** *auto*
 from ‹*derivable D P S* ϑ *FirstOrder* C'› **have** (*cl-ecl D*) = (*subst-cl* C' ϑ)
  using *derivable-clauses-lemma* **by** *auto*
 from ‹*trms-subsumes D C* η› **have** (*cl-ecl C*) = (*subst-cl* (*cl-ecl D*) η)
  unfolding *trms-subsumes-def* **by** *blast*
 from ‹$\sigma \doteq \vartheta \lozenge \eta$› **have** *subst-cl* (*cl-ecl D*) σ = *subst-cl* (*cl-ecl D*) ($\vartheta \lozenge \eta$)
  using *subst-eq-cl* **by** *blast*
 then have (*subst-cl* (*cl-ecl D*) σ) = (*subst-cl* (*subst-cl* (*cl-ecl D*) ϑ) η)
   using *composition-of-substs-cl* [*of cl-ecl D* ϑ η] **by** *auto*
 from *this* **and** ‹(*cl-ecl C*) = (*subst-cl* (*cl-ecl D*) η)›
  ‹(*ground-clause* (*cl-ecl C*))› **have** *ground-clause* (*subst-cl* (*cl-ecl D*) η)
  by *auto*
 from *this* ‹$D \in S$› ‹(*cl-ecl D*) = (*subst-cl* C' ϑ)›
  ‹(*cl-ecl D*) = (*subst-cl* C' ϑ)›
  have *redundant-clause* (*subst-ecl D* η) *S* η (*subst-cl* C' ϑ)
  using *self-redundant-clause* **by** *metis*
 from ‹*derivable D P S* ϑ *FirstOrder* C'› **have** $P \subseteq S$ **unfolding** *derivable-def* **by** *auto*
 from *this* *assms*(*2*) **have** $\forall x \in P$. (*finite* (*cl-ecl x*)) **by** *auto*
 from *this* ‹*derivable D P S* ϑ *FirstOrder* C'› **have** *finite* C'
  using *derivable-clauses-are-finite* **by** *auto*
 from *this* ‹(*cl-ecl D*) = *subst-cl* C' ϑ›
   ‹*redundant-clause* (*subst-ecl D* η) *S* η (*subst-cl* C' ϑ)› ‹($\sigma \doteq \vartheta \lozenge \eta$)›
  have *redundant-clause* (*subst-ecl D* η) *S* σ C'
  using *lift-redundant-cl* **by** *metis*
 from *this* **and** ‹¬(*redundant-clause* (*subst-ecl D* η) *S* σ C')› **show** *False* **by** *auto*
qed

7.3 Satisfiability of Saturated Sets with No Empty Clause

We are now in the position to prove that the previously constructed interpretation is indeed a model of the set of extended clauses, if the latter is saturated and does not contain an extended clause with empty clausal part. More precisely, the constructed interpretation satisfies the clausal part of every extended clause whose attached set of terms is in normal form. This is the case in particular if this set is empty, hence if the clause is an input clause.

Note that we do not provide any function for explicitly constructing such saturated sets, except by generating all derivable clauses (see below).

lemma *int-clset-is-a-model*:
  assumes *ground-inference-saturated S*
  assumes *all-finite*: $\forall x \in S$. (*finite* (*cl-ecl* x))
  assumes *Ball S well-constrained*
  assumes *all-non-empty*: $\forall x \in S$. (*cl-ecl* x) $\neq \{\}$
  assumes *closed-under-renaming S*
  shows $\forall$ C σ. (*fst pair* = C) $\longrightarrow$ σ = (*snd pair*) $\longrightarrow$ $C \in S$ $\longrightarrow$
    ground-clause (*subst-cl* (*cl-ecl* C) σ)
    $\longrightarrow$ (*all-trms-irreducible* (*subst-set* (*trms-ecl* C) σ) (λt. (*trm-rep* t S)))
      $\longrightarrow$ *validate-ground-clause* (*same-values* (λt. (*trm-rep* t S))) (*subst-cl* (*cl-ecl* C) σ)
        (**is** *?P pair*)
proof ((*rule wf-induct* [*of ecl-ord ?P pair*]),(*simp add*: *wf-ecl-ord*))
next

The proof is by induction and contradiction. We consider a minimal instance that is not true in the interpretation and we derive a contradiction.

  fix *pair* **assume** *hyp-ind*: $\forall y$. (y,*pair*) $\in$ *ecl-ord* $\longrightarrow$ (*?P y*)
  let *?I* = (*int-clset S*)
  have *fo-interpretation ?I*
  unfolding *int-clset-def* **using** *trm-rep-compatible-with-structure same-values-fo-int*

    by *metis*
  show (*?P pair*)
  proof (*rule ccontr*)
    assume $\neg$(*?P pair*)
    then obtain C σ **where** C = (*fst pair*) **and** σ = (*snd pair*) **and** $C \in S$
      and *ground-clause* (*subst-cl* (*cl-ecl* C) σ)
      and (*all-trms-irreducible* (*subst-set* (*trms-ecl* C) σ)
        (λt. (*trm-rep* t S)))
      and *cm*: $\neg$*validate-ground-clause* (*int-clset S*) (*subst-cl* (*cl-ecl* C) σ)
    unfolding *int-clset-def* **by** *metis*

First, we prove that no reduction is possible (otherwise the superposition rule applies).

  let *?nored* = ($\forall$ *L1 L2 D t s* u' *u v p polarity* σ'.

$\neg$ ((*reduction L1 C* σ' *t s polarity L2 u u'* *p v D ?I S* σ) $\wedge$ (*variable-disjoint C D*)))
have *?nored*
proof (*rule ccontr*)
assume $\neg$ *?nored*
then obtain *L1 L2 D t s u' u v p polarity* σ'
where *red*: *reduction L1 C* σ' *t s polarity L2 u u' p v D ?I S* σ
and *variable-disjoint C D*
by *blast*
from *red* **have** (*subst u* σ') $\neq$ (*subst v* σ')
unfolding *reduction-def* **by** *blast*
from *red* **have** *ground-clause* (*subst-cl* (*cl-ecl D*) σ')
unfolding *reduction-def* **by** *blast*
from *red* **have** (*coincide-on* σ σ' (*vars-of-cl* (*cl-ecl C*)))
unfolding *reduction-def* **by** *blast*
from *red* **have** $\neg$ *is-a-variable u'* **unfolding** *reduction-def* **by** *blast*
from *red* **have** $D \in S$ **unfolding** *reduction-def* **by** *blast*
from *red* **have** *el1*: (*eligible-literal L1 C* σ') **unfolding** *reduction-def* **by** *blast*
from *red* **have** *el2*: (*eligible-literal L2 D* σ') **unfolding** *reduction-def* **by** *blast*
from *red* **have** $D \in S$ **unfolding** *reduction-def* **by** *blast*
from *red* **have** (*minimal-redex p* (*subst t* σ) *C S t*)
unfolding *reduction-def* **by** *blast*
from *red* **have** *l1*: *L1* $\in$ (*cl-ecl C*) **unfolding** *reduction-def* **by** *blast*
from *red* **have** *l2*: *L2* $\in$ (*cl-ecl D*) **unfolding** *reduction-def* **by** *blast*
from *red* **have** *o1*: (*orient-lit-inst L1 t s polarity* σ') **unfolding** *reduction-def* **by** *blast*
from *red* **have** *o2*: (*orient-lit-inst L2 u v pos* σ') **unfolding** *reduction-def* **by** *blast*
from *red* **have** *e*:(*subst u'* σ') = (*subst u* σ')
unfolding *reduction-def* **by** *blast*
from *red* **have** ($\neg$ *validate-ground-clause ?I* (*subst-cl* ((*cl-ecl D*) $-$ { *L2* }) σ'))
unfolding *reduction-def* **by** *blast*
from *red* **have** ($\forall$ *x* $\in$ (*cl-ecl D*) $-$ { *L2* }. ((*subst-lit x* σ'),(*subst-lit L2* σ'))
$\in$ *lit-ord*)
unfolding *reduction-def* **by** *blast*
from *red* **have** *st*: (*subterm t p u'*) **unfolding** *reduction-def* **by** *blast*
from *red* **have** (*allowed-redex u' C* σ) **unfolding** *reduction-def* **by** *blast*
from *st* **have** *u'* $\in$ (*subterms-of t*) **using** *occurs-in-def* **by** *auto*
from *this* **and** *o1* **have** *u'* $\in$ (*subterms-of-lit L1*) **using** *orient-lit-inst-subterms* **by** *auto*
from *this* **and** ‹*L1* $\in$ (*cl-ecl C*)› **have** *u'* $\in$ (*subterms-of-cl* (*cl-ecl C*)) **by** *auto*
from *this* **and** ‹(*allowed-redex u' C* σ)› **and** ‹$C \in S$›
and ‹(*coincide-on* σ σ' (*vars-of-cl* (*cl-ecl C*)))›
assms(*3*)

have *rte*: (*allowed-redex* u' *C* σ') **using** *allowed-redex-coincide* [*of* u' *C* σ σ']
by *metis*

from *red* **have** ((*subst-lit L2* σ'),(*subst-lit L1* σ'))
$\in$ *lit-ord* **unfolding** *reduction-def* **by** *blast*
from *red* **have** (*all-trms-irreducible* (*subst-set* (*trms-ecl D*) σ')
(λt. (*trm-rep t S*))) **unfolding** *reduction-def* **by** *blast*
from *red* **have** *?I* (*subst u* σ') (*subst v* σ')
unfolding *reduction-def* **by** *blast*

from *e* **have** *t*: *ck-unifier* u' *u* σ' *Ground* **unfolding** *ck-unifier-def Unifier-def* **by** *auto*

have $\forall x \in$ (*cl-ecl D*). ((*mset-lit* (*subst-lit x* σ')),(*mset-lit* (*subst-lit L1* σ')))

$\in$ (*mult trm-ord*)
proof (*rule ccontr*)
assume $\neg(\forall x \in$ (*cl-ecl D*). ((*mset-lit* (*subst-lit x* σ')),(*mset-lit* (*subst-lit L1* σ')))
$\in$ (*mult trm-ord*))
from *this* **obtain** *x* **where** $x \in$ (*cl-ecl D*)
and ((*mset-lit* (*subst-lit x* σ')),(*mset-lit* (*subst-lit L1* σ')))
$\notin$ (*mult trm-ord*)
by *auto*
show *False*
proof (*cases*)
assume $x = L2$
from *this* **and** ‹((*subst-lit L2* σ'),(*subst-lit L1* σ'))
$\in$ *lit-ord*› **and** ‹((*mset-lit* (*subst-lit x* σ')),(*mset-lit* (*subst-lit L1* σ')))
$\notin$ (*mult trm-ord*)›
show *False* **unfolding** *lit-ord-def* **by** *auto*
next
assume $x \neq L2$
from *this* **and** ‹$x \in$ (*cl-ecl D*)› **have** $x \in$ (*cl-ecl D*) $-$ { *L2* } **by** *auto*
from *this* **and** ‹($\forall x \in$ (*cl-ecl D*) $-$ { *L2* }. ((*subst-lit x* σ'),(*subst-lit L2* σ'))
$\in$ *lit-ord*)›
have ((*subst-lit x* σ'),(*subst-lit L2* σ'))
$\in$ *lit-ord* **by** *auto*
from ‹((*mset-lit* (*subst-lit x* σ')),(*mset-lit* (*subst-lit L1* σ')))
$\notin$ (*mult trm-ord*)›
have ((*subst-lit x* σ'),(*subst-lit L1* σ'))
$\notin$ *lit-ord*
unfolding *lit-ord-def* **by** *auto*

from *this* **and** ‹((*subst-lit x* σ'),(*subst-lit L2* σ'))
$\in$ *lit-ord*›
and ‹((*subst-lit L2* σ'),(*subst-lit L1* σ'))

∈ *lit-ord*›
show *False* **using** *lit-ord-trans* **unfolding** *trans-def* **by** *blast*
qed
qed
from *all-finite* **and** ‹*C* ∈ *S*› **have** *finite* (*cl-ecl C*) **by** *auto*
from *this* **and** ‹*L1* ∈ (*cl-ecl C*)›
have (*mset-lit* (*subst-lit L1 σ′*)) ∈# *mset-ecl* (*C*,*σ′*)
using *mset-ecl-and-mset-lit* **by** *auto*
from *this* **have** (*mset-lit* (*subst-lit L1 σ′*)) ∈ (*set-mset* (*mset-ecl* (*C*,*σ′*)))
by *simp*
have ∀ *x*. (*x* ∈ (*set-mset* (*mset-ecl* (*D*,*σ′*)))
⟶ (∃ *y* ∈ *set-mset* (*mset-ecl* (*C*,*σ′*)). (*x*,*y*) ∈ (*mult trm-ord*)))
proof ((*rule allI*),(*rule impI*))
fix *x* **assume** *x* ∈ (*set-mset* (*mset-ecl* (*D*,*σ′*)))

then have *x* ∈# *mset-ecl* (*D*,*σ′*) **by** *simp*
from ‹*x* ∈# *mset-ecl* (*D*,*σ′*)› **obtain** *x′*
where *x′* ∈# (*mset-set* (*cl-ecl D*))
and *x* = (*mset-lit* (*subst-lit x′ σ′*)) **by** *auto*

from ‹*x′* ∈# (*mset-set* (*cl-ecl D*))› **have** *x′* ∈ (*cl-ecl D*)
using *count-mset-set*(*3*) **by** (*fastforce simp*: *count-eq-zero-iff*)
from *this*
and ‹∀ *x* ∈ (*cl-ecl D*).
((*mset-lit* (*subst-lit x σ′*)),(*mset-lit* (*subst-lit L1 σ′*)))
∈ (*mult trm-ord*)›
and ‹*x* = (*mset-lit* (*subst-lit x′ σ′*))›
have (*x*,(*mset-lit* (*subst-lit L1 σ′*))) ∈ (*mult trm-ord*)
by *auto*

from ‹(*mset-lit* (*subst-lit L1 σ′*)) ∈ (*set-mset* (*mset-ecl* (*C*,*σ′*)))›
and ‹(*x*,(*mset-lit* (*subst-lit L1 σ′*))) ∈ (*mult trm-ord*)›
show
(∃ *y* ∈ *set-mset* (*mset-ecl* (*C*,*σ′*)). (*x*,*y*) ∈ (*mult trm-ord*))
by *auto*
qed

from *this* **have**
dom: ⋀*I J K*. *J* ≠ {#} ∧ (∀ *k*∈*set-mset K*. ∃ *j*∈*set-mset J*. (*k*, *j*) ∈ (*mult trm-ord*)) ⟶
(*I* + *K*, *I* + *J*) ∈ *mult* (*mult trm-ord*)
by (*blast intro*: *one-step-implies-mult*)
from ‹(*mset-lit* (*subst-lit L1 σ′*)) ∈# *mset-ecl* (*C*,*σ′*)›
have *mset-ecl* (*C*,*σ′*) ≠ {#} **by** *auto*
from ‹∀ *x*. (*x* ∈ (*set-mset* (*mset-ecl* (*D*,*σ′*)))
⟶ (∃ *y* ∈ *set-mset* (*mset-ecl* (*C*,*σ′*)). (*x*,*y*) ∈ (*mult trm-ord*)))›
and ‹*mset-ecl* (*C*,*σ′*) ≠ {#}›
have ({#} + *mset-ecl* (*D*, *σ′*), {#} + *mset-ecl* (*C*, *σ′*)) ∈ *mult* (*mult trm-ord*)

using *dom* [*of* (*mset-ecl* (*C*,σ')) *mset-ecl* (*D*,σ') {#}] **by** *auto*
from *this* **have** (*mset-ecl* (*D*, σ'), *mset-ecl* (*C*, σ')) $\in$ *mult* (*mult trm-ord*)
by *auto*
from *this* **have** ((*D*,σ'), (*C*,σ')) $\in$ *ecl-ord*
unfolding *ecl-ord-def* **by** *auto*

from *st* **obtain** t' **where** *rt*: (*replace-subterm t p v* t')
using *replace-subterm-is-a-function* **by** *blast*

from *st* **obtain** *R Cl-R nt-R* L' *Cl-C Cl-D* **where**
ntr: *nt-R* = (*dom-trms Cl-R* (*subst-set*
((*trms-ecl C*) $\cup$ (*trms-ecl D*) $\cup$
{ *r*. $\exists$ *q*. (*q*,*p*) $\in$ (*pos-ord C t*) $\wedge$ (*subterm t q r*) }) σ'))
and *r*: *R* = *Ecl Cl-R nt-R*
and *clc*: *Cl-C* = (*cl-ecl C*)
and *cld*: *Cl-D* = (*cl-ecl D*)
and *clr*: *Cl-R* = (*subst-cl* ((*Cl-C* $-$ { *L1* }) $\cup$ ((*Cl-D* $-$ { *L2* }) $\cup$ { L'
})) σ')
and l': L' = *mk-lit polarity* (*Eq* t' *s*)
by *auto*

from ‹*orient-lit-inst L1 t s polarity* σ'› **have** *vars-of t* $\subseteq$ *vars-of-lit L1*
using *orient-lit-inst-vars* **by** *auto*
from ‹*L1* $\in$ (*cl-ecl C*)› **have** *vars-of-lit L1* $\subseteq$ *vars-of-cl* (*cl-ecl C*) **by** *auto*
from *this* **and** ‹*vars-of t* $\subseteq$ *vars-of-lit L1*› **have** *vars-of t* $\subseteq$*vars-of-cl* (*cl-ecl*
C) **by** *auto*
from *this* **and** ‹*coincide-on* σ σ' (*vars-of-cl* (*cl-ecl C*))›
have *coincide-on* σ σ' (*vars-of t*) **unfolding** *coincide-on-def* **by** *auto*
from *this* **have** *subst t* σ = *subst t* σ' **using** *coincide-on-term* **by** *auto*

from ‹($\forall$ *x* $\in$ (*cl-ecl D*) $-$ { *L2* }. ((*subst-lit x* σ'),(*subst-lit L2* σ'))
$\in$ *lit-ord*)›
have *strictly-maximal-literal D L2* σ' **unfolding** *strictly-maximal-literal-def*
by *metis*
from *ntr* **have** *nt-R* = *get-trms Cl-R* (*dom-trms Cl-R* (*subst-set*
((*trms-ecl C*) $\cup$ (*trms-ecl D*) $\cup$
{ *r*. $\exists$ *q*. (*q*,*p*) $\in$ (*pos-ord C t*) $\wedge$ (*subterm t q r*) }) σ')) *Ground*
unfolding *get-trms-def* **by** *auto*
from *this* ‹(*subst u* σ') $\neq$ (*subst v* σ')› ‹$\neg$ *is-a-variable* u'› *l1 l2 el1 el2*
‹*variable-disjoint C D*› *rte r o1 o2 t st rt* l' *clr ntr clr clc cld* ‹*R* = *Ecl Cl-R*
nt-R›
‹((*subst-lit L2* σ'),(*subst-lit L1* σ'))
$\in$ *lit-ord*›
‹*strictly-maximal-literal D L2* σ' ›
have *superposition C D R* σ' *Ground* ((*Cl-C* $-$ { *L1* }) $\cup$ ((*Cl-D* $-$ { *L2*
}) $\cup$ { L' }))
unfolding *superposition-def* **by** *blast*
from *l2* **have** (*subst-lit L2* σ') $\in$ (*subst-cl* (*cl-ecl D*) σ') **by** *auto*
from *this* **and** ‹*ground-clause* (*subst-cl* (*cl-ecl D*) σ')›

have *vars-of-lit* (*subst-lit L2* σ') = {}
by *auto*
from *this* **and** *o2* **have** *vars-of* (*subst v* σ') = {}
unfolding *orient-lit-inst-def* **using** *vars-of-lit.simps vars-of-eq.simps* **by** *force*

from ‹(*coincide-on* σ σ' (*vars-of-cl* (*cl-ecl C*)))›
have (*subst-cl* (*cl-ecl C*) σ') = (*subst-cl* (*cl-ecl C*) σ)
using *coincide-on-cl* [*of* σ σ' (*cl-ecl C*)] **by** *auto*

from *this* **and** ‹*ground-clause* (*subst-cl* (*cl-ecl C*) σ)›
have *ground-clause* (*subst-cl* (*cl-ecl C*) σ') **using**
coincide-on-cl **by** *auto*

from *l1* **have** (*subst-lit L1* σ') $\in$ (*subst-cl* (*cl-ecl C*) σ') **by** *auto*
from *this* **and** ‹*ground-clause* (*subst-cl* (*cl-ecl C*) σ')›
have *vars-of-lit* (*subst-lit L1* σ') = {} **by** *auto*
from *this* **and** *o1* **have** *vars-of* (*subst t* σ') = {}
unfolding *orient-lit-inst-def* **using** *vars-of-lit.simps vars-of-eq.simps* **by** *force*
from ‹*vars-of-lit* (*subst-lit L1* σ') = {}›
and *o1* **have** *vars-of* (*subst s* σ') = {}
unfolding *orient-lit-inst-def* **using** *vars-of-lit.simps vars-of-eq.simps* **by** *force*

from ‹*vars-of* (*subst t* σ') = {}› **and** ‹*vars-of* (*subst v* σ') = {}›
and *rt* **have** *vars-of* (*subst* t' σ') = {} **using** *ground-replacement* [*of t p v* t' σ']
unfolding *ground-term-def* **by** *blast*

from ‹*vars-of* (*subst* t' σ') = {}› **and** ‹*vars-of* (*subst s* σ') = {}›
have *vars-of-eq* (*subst-equation* (*Eq* t' *s*) σ') = {} **by** *auto*
from l' **have** L' = (*Pos* (*Eq* t' *s*)) $\vee$ L' = (*Neg* (*Eq* t' *s*)) **using** *mk-lit.elims* **by** *auto*
from *this* **and** ‹*vars-of-eq* (*subst-equation* (*Eq* t' *s*) σ') = {}›
have *vars-of-lit* (*subst-lit* L' σ') = {} **by** *auto*

from ‹$C \in S$› **and** ‹$D \in S$› **and**
‹*superposition C D R* σ' *Ground* ((*Cl-C* − { *L1* }) $\cup$ ((*Cl-D* − { *L2* }) $\cup$ { L' }))›
have *derivable R* { *C*,*D* } *S* σ' *Ground* ((*Cl-C* − { *L1* }) $\cup$ ((*Cl-D* − { *L2* }) $\cup$ { L' }))
unfolding *derivable-def* **by** *auto*

have *ground-clause Cl-R*
proof (*rule ccontr*)
assume $\neg$*ground-clause Cl-R*
then have *vars-of-cl Cl-R* $\neq$ {} **by** *auto*
then obtain *M* **where** *M* $\in$ *Cl-R* **and** *vars-of-lit M* $\neq$ {} **by** *auto*

from ‹$M \in$ *Cl-R*› **and** *clr* **obtain** M' **where** $M =$ (*subst-lit* M' σ')
and $M' \in$ ((*Cl-C* $-$ { *L1* }) $\cup$ ((*Cl-D* $-$ { *L2* }) $\cup$ { L' }))
by *auto*
show *False*
proof (*cases*)
assume $M' = L'$
from *this* **and** ‹*vars-of-lit* (*subst-lit* L' σ') $=$ {}› **and** ‹*vars-of-lit* $M \neq$
{}›
and ‹$M =$ (*subst-lit* M' σ')› **show** *False* **by** *auto*
next
assume $M' \neq L'$
from *this* **and** *l1 clc cld* **and** ‹$M' \in$(*Cl-C* $-$ { *L1* }) $\cup$ ((*Cl-D* $-$ { *L2*
}) $\cup$ { L' })›
have $M' \in$ (*cl-ecl C*) $\vee$ $M' \in$ (*cl-ecl D*)
by *auto*
then show *False*
proof
assume $M' \in$ (*cl-ecl C*)
from *this* **and** ‹*ground-clause* (*subst-cl* (*cl-ecl C*) σ')› **have**
vars-of-lit (*subst-lit* M' σ') $=$ {} **by** *auto*
from *this* **and** ‹$M =$ (*subst-lit* M' σ')› **and**
‹*vars-of-lit* $M \neq$ {}› **show** *False* **by** *auto*
next
assume $M' \in$ (*cl-ecl D*)
from *this* **and** ‹*ground-clause* (*subst-cl* (*cl-ecl D*) σ')› **have**
vars-of-lit (*subst-lit* M' σ') $=$ {} **by** *auto*
from *this* **and** ‹$M =$ (*subst-lit* M' σ')› **and**
‹*vars-of-lit* $M \neq$ {}› **show** *False* **by** *auto*
qed
qed
qed
from ‹*ground-clause* (*subst-cl* (*cl-ecl C*) σ')› **and** ‹*ground-clause* (*subst-cl*
(*cl-ecl D*) σ')›
have *grounding-set* { *C*,*D* } σ' **unfolding** *grounding-set-def* **by** *auto*
from ‹*ground-clause Cl-R*› **and** ‹$R =$ *Ecl Cl-R nt-R*› **have** *ground-clause*
(*cl-ecl R*) **by** *auto*

from *this* **and** ‹*derivable R* { *C*,*D* } *S* σ' *Ground* ((*Cl-C* $-$ { *L1* }) $\cup$
((*Cl-D* $-$ { *L2* }) $\cup$ { L' }))›
and ‹*ground-inference-saturated S*› ‹*grounding-set* { *C*,*D* } σ'›
have (*redundant-inference R S* { *C*,*D* } σ') **unfolding** *ground-inference-saturated-def*

by *blast*
from *this* **obtain** S' **where** $S' \subseteq$ (*instances S*) **and**
(*set-entails-clause* (*clset-instances* S') (*cl-ecl R*))
and *order*: $\forall x \in S'$. ((*cl-ecl* (*fst x*), *snd x*), *cl-ecl C*, σ') $\in$ *cl-ord* $\vee$
((*cl-ecl* (*fst x*), *snd x*), *cl-ecl D*, σ') $\in$ *cl-ord*
and *all-normalized-term-included*: ($\forall x \in S'$.
(*subterms-inclusion* (*subst-set* (*trms-ecl* (*fst x*)) (*snd x*))

($trms$-ecl R)))
unfolding *redundant-inference-def* **by** *auto*

have *all-smaller*: ($\forall x \in S'$. (((fst x),(snd x)),(C,σ)) $\in$ *ecl-ord*)
proof (*rule ccontr*)
assume $\neg$($\forall x \in S'$. (((fst x),(snd x)),(C,σ)) $\in$ *ecl-ord*)
then obtain x **where** $x \in S'$ **and** (((fst x),(snd x)),(C,σ)) $\notin$ *ecl-ord*
by *auto*

from *order*[*rule-format*, *OF* ‹$x \in S'$›]
have ((*cl-ecl* (*fst* x), *snd* x), *cl-ecl* C, σ') $\in$ *cl-ord*
proof (*elim disjE*)
assume ((*cl-ecl* (*fst* x), *snd* x), *cl-ecl* C, σ') $\in$ *cl-ord*
thus *?thesis* **by** *assumption*
next
assume ((*cl-ecl* (*fst* x), *snd* x), *cl-ecl* D, σ') $\in$ *cl-ord*
thus ((*cl-ecl* (*fst* x), *snd* x), *cl-ecl* C, σ') $\in$ *cl-ord*
using ‹((D, σ'), (C, σ')) $\in$ *ecl-ord*›[*unfolded member-ecl-ord-iff*]
by (*rule cl-ord-trans*[*THEN transD*])
qed
from ‹(*coincide-on* σ σ' (*vars-of-cl* (*cl-ecl* C)))›
have (*mset-ecl* (C,σ')) = (*mset-ecl* (C,σ))
using *ecl-ord-coincide* [*of* σ σ' C] **by** *auto*
with ‹((*cl-ecl* (*fst* x), *snd* x), *cl-ecl* C, σ') $\in$ *cl-ord*›
have ((*mset-ecl* x), (*mset-ecl* (C,σ))) $\in$ (*mult* (*mult trm-ord*))
by (*metis* (*no-types*, *lifting*) *CollectD case-prodD cl-ord-def mset-ecl-conv prod.collapse*)
from *this* **and** ‹$\neg$(((fst x),(snd x)),(C,σ)) $\in$ *ecl-ord*› **show** *False*
unfolding *ecl-ord-def* **by** *auto*
qed

have *validate-clause-set ?I* (*clset-instances* S')
proof (*rule ccontr*)
assume $\neg$ *validate-clause-set ?I* (*clset-instances* S')
then obtain x **where** $x \in$(*clset-instances* S') **and** $\neg$*validate-clause ?I* x
using *validate-clause-set.simps* **by** *blast*
from ‹$x \in$(*clset-instances* S')› **obtain** *pair*$'$ **where** *pair*$' \in S'$
and x = (*subst-cl* (*cl-ecl* (*fst pair*$'$)) (*snd pair*$'$))
unfolding *clset-instances-def*
by *auto*
from *all-smaller* **and** ‹*pair*$' \in S'$› **have** (*pair*$'$,(C,σ)) $\in$ *ecl-ord*
by *auto*
from *this* **and** ‹C = *fst pair*› **and** ‹σ = *snd pair*› **have** (*pair*$'$,*pair*) $\in$ *ecl-ord*
by *auto*
from *this* **and** *hyp-ind* **have** *?P pair*$'$ **by** *blast*

from ‹*pair*$' \in S'$› **and** *all-normalized-term-included* **have**
(*subterms-inclusion* (*subst-set* (*trms-ecl* (*fst pair*$'$)) (*snd pair*$'$))

$(trms\text{-}ecl\ R))$ **by** *auto*
have i: $(all\text{-}trms\text{-}irreducible\ (subst\text{-}set\ (trms\text{-}ecl\ (fst\ pair')) (snd\ pair'))$
$(\lambda t.\ (trm\text{-}rep\ t\ S)))$
proof $(rule\ ccontr)$
assume $\neg(all\text{-}trms\text{-}irreducible\ (subst\text{-}set\ (trms\text{-}ecl\ (fst\ pair')) (snd\ pair'))$
$(\lambda t.\ (trm\text{-}rep\ t\ S)))$
then obtain $w\ w'$ **where** $w \in subst\text{-}set\ (trms\text{-}ecl\ (fst\ pair'))\ (snd\ pair')$
and $occurs\text{-}in\ w'\ w$
$trm\text{-}rep\ w'\ S \neq w'$
unfolding *all-trms-irreducible-def* **by** *blast*
from ‹$w \in subst\text{-}set\ (trms\text{-}ecl\ (fst\ pair'))\ (snd\ pair')$› **and**
‹$(subterms\text{-}inclusion\ (subst\text{-}set\ (trms\text{-}ecl\ (fst\ pair'))\ (snd\ pair'))$
$(trms\text{-}ecl\ R))$› **obtain** w'' **where** $w'' \in trms\text{-}ecl\ R$ **and** $occurs\text{-}in\ w\ w''$
unfolding *subterms-inclusion-def* **by** *auto*
from ‹$occurs\text{-}in\ w\ w''$› **and** ‹$occurs\text{-}in\ w'\ w$› **have** $occurs\text{-}in\ w'\ w''$
using *occur-in-subterm* **by** *blast*
from *ntr* **have** $nt\text{-}R \subseteq (subst\text{-}set\ ((trms\text{-}ecl\ C) \cup (trms\text{-}ecl\ D) \cup$
$\{\ r.\ \exists\ q.\ (q,p) \in (pos\text{-}ord\ C\ t) \wedge (subterm\ t\ q\ r)\ \})\ \sigma')$
using *dom-trms-subset* $[of\ Cl\text{-}R\ (subst\text{-}set\ ((trms\text{-}ecl\ C) \cup (trms\text{-}ecl\ D) \cup$
$\{\ r.\ \exists\ q.\ (q,p) \in (pos\text{-}ord\ C\ t) \wedge (subterm\ t\ q\ r)\ \})\ \sigma')]$ **by** *blast*
from *this* **and** r **have** $trms\text{-}ecl\ R \subseteq (subst\text{-}set\ ((trms\text{-}ecl\ C) \cup (trms\text{-}ecl$
$D) \cup$
$\{\ r.\ \exists\ q.\ (q,p) \in (pos\text{-}ord\ C\ t) \wedge (subterm\ t\ q\ r)\ \})\ \sigma')$ **by** *auto*
from *this* **and** ‹$w'' \in trms\text{-}ecl\ R$› **have**
$w'' \in (subst\text{-}set\ ((trms\text{-}ecl\ C) \cup (trms\text{-}ecl\ D) \cup$
$\{\ r.\ \exists\ q.\ (q,p) \in (pos\text{-}ord\ C\ t) \wedge (subterm\ t\ q\ r)\ \})\ \sigma')$ **by** *blast*
from *this* **obtain** w'''
where $w''' \in ((trms\text{-}ecl\ C) \cup (trms\text{-}ecl\ D) \cup$
$\{\ r.\ \exists\ q.\ (q,p) \in (pos\text{-}ord\ C\ t) \wedge (subterm\ t\ q\ r)\ \})$ **and** $w'' = subst$
$w'''\ \sigma'$
by *auto*
from *this* **and** ‹$occurs\text{-}in\ w'\ w''$› **have** $occurs\text{-}in\ w'\ (subst\ w'''\ \sigma')$ **by** *auto*

have $\neg\ (w''' \in trms\text{-}ecl\ C)$
proof
assume $w''' \in trms\text{-}ecl\ C$
from *this* **and** ‹$occurs\text{-}in\ w'\ w''$› **and** ‹$w'' = subst\ w'''\ \sigma'$› **have**
$occurs\text{-}in\ w'\ (subst\ w'''\ \sigma')$ **by** *auto*
from $assms(3)$ **and** ‹$C \in S$› **and** ‹$w''' \in trms\text{-}ecl\ C$›
have $vars\text{-}of\ w''' \subseteq vars\text{-}of\text{-}cl\ (cl\text{-}ecl\ C)$ **using** *dom-trm-vars*
unfolding *Ball-def well-constrained-def* **by** *blast*
from *this* **and** ‹$coincide\text{-}on\ \sigma\ \sigma'\ (vars\text{-}of\text{-}cl\ (cl\text{-}ecl\ C))$› **have** $coincide\text{-}on$
$\sigma\ \sigma'\ (vars\text{-}of\ w''')$
unfolding *coincide-on-def* **by** *auto*
from *this* **have** $subst\ w'''\ \sigma = subst\ w'''\ \sigma'$
using *coincide-on-term* **by** *auto*
from *this*
and ‹$occurs\text{-}in\ w'\ (subst\ w'''\ \sigma')$›
have $occurs\text{-}in\ w'\ (subst\ w'''\ \sigma)$ **by** *auto*

from *this* **and** ‹$w''' \in$ *trms-ecl C*›
‹(*all-trms-irreducible* (*subst-set* (*trms-ecl C*) σ)
($\lambda t.$ (*trm-rep t S*)))›
have *trm-rep* w' $S = w'$
unfolding *all-trms-irreducible-def* **using** *subst-set.simps* **by** *blast*
from *this* **and** ‹*trm-rep* w' $S \neq w'$› **show** *False* **by** *blast*
qed

have $\neg$ ($w''' \in$ *trms-ecl D*)
proof
assume $w''' \in$ *trms-ecl D*
from *this* **and** ‹*occurs-in* w' w''› **and** ‹$w'' =$ *subst* w''' σ'› **have**
occurs-in w' (*subst* w''' σ') **by** *auto*
from *this* **and** ‹$w''' \in$ *trms-ecl D*›
‹(*all-trms-irreducible* (*subst-set* (*trms-ecl D*) σ')
($\lambda t.$ (*trm-rep t S*)))›
have *trm-rep* w' $S = w'$
unfolding *all-trms-irreducible-def* **using** *subst-set.simps* **by** *blast*
from *this* **and** ‹*trm-rep* w' $S \neq w'$› **show** *False* **by** *blast*
qed

from *this* **and**
‹$w''' \in$ ((*trms-ecl C*) $\cup$ (*trms-ecl D*)
$\cup$ { $r.$ $\exists$ $q.$ (q,p) $\in$ (*pos-ord C t*) $\wedge$ (*subterm t q r*) })›
and ‹$\neg$ ($w''' \in$ *trms-ecl C*)›
obtain *q-w* **where** (*subterm t q-w* w''') **and** (*q-w*,p) $\in$ (*pos-ord C t*)
by *auto*

from ‹*subterm t q-w* w'''› **have** *subterm* (*subst t* σ') *q-w* (*subst* w''' σ')
using *substs-preserve-subterms* **by** *auto*
from ‹*occurs-in* w' (*subst* w''' σ')› **obtain** q
where (*subterm* (*subst* w''' σ') q w') **unfolding** *occurs-in-def* **by** *blast*
from *this* **and** ‹*subterm* (*subst t* σ') *q-w* (*subst* w''' σ')›
have *subterm* (*subst t* σ') (*append q-w q*) w' **using** *subterm-of-a-subterm-is-a-subterm*
by *blast*
from *this* **and** ‹(*subst t* σ) = (*subst t* σ')›
have *subterm* (*subst t* σ) (*append q-w q*) w' **by** *auto*
from ‹(*q-w*,p) $\in$ (*pos-ord C t*)› **have** ((*append q-w q*),p) $\in$ (*pos-ord C t*)
using *pos-ord-prefix* **by** *blast*
from *this* **and** ‹*minimal-redex p* (*subst t* σ) *C S t*›
and ‹*subterm* (*subst t* σ) (*append q-w q*) w'›
have *trm-rep* w' $S = w'$
unfolding *minimal-redex-def* **by** *blast*
from *this* **and** ‹*trm-rep* w' $S \neq w'$› **show** *False* **by** *blast*
qed
from ‹$S' \subseteq$ (*instances S*)› **and** ‹*pair'* $\in S'$› **have**
ii: *ground-clause* (*subst-cl* (*cl-ecl* (*fst pair'*)) (*snd pair'*))
unfolding *instances-def* [*of S*] **by** *fastforce*
from ‹$S' \subseteq$ (*instances S*)› **and** ‹*pair'* $\in S'$› **have**

iii: (*fst pair'*) ∈ *S* **unfolding** *instances-def* [*of S*] **by** *fastforce*
from ‹*?P pair'*› **and** *i ii iii* **have** *validate-ground-clause ?I*
(*subst-cl* (*cl-ecl* (*fst pair'*)) (*snd pair'*)) **unfolding** *int-clset-def* **by** *blast*
from *this* **and** ‹x = (*subst-cl* (*cl-ecl* (*fst pair'*)) (*snd pair'*))›
and ‹¬*validate-clause ?I x*› **show** *False*
by (*metis ii substs-preserve-ground-clause validate-clause.simps*)
qed
from *this* **and** ‹*fo-interpretation ?I*› **and**
‹(*set-entails-clause* (*clset-instances S'*) (*cl-ecl R*))›
have *validate-clause ?I* (*cl-ecl R*) **unfolding** *set-entails-clause-def* **by** *blast*
from *this* **have** *validate-ground-clause ?I* (*cl-ecl R*)
by (*metis* ‹*R* = *Ecl Cl-R nt-R*› ‹*ground-clause Cl-R*›
cl-ecl.simps substs-preserve-ground-clause validate-clause.simps)
from *this* **obtain** L'' **where** L'' ∈ (*cl-ecl R*) **and** *validate-ground-lit ?I* L''
using *validate-ground-clause.simps* **by** *blast*
from ‹L'' ∈ (*cl-ecl R*)› **and** ‹*R* = *Ecl Cl-R nt-R*› **and**
‹*Cl-R* = (*subst-cl* ((*Cl-C* − { *L1* }) ∪ ((*Cl-D* − { *L2* }) ∪ { *L'* })) σ')›
obtain *M* **where** *m*: *M* ∈ ((*Cl-C* − { *L1* }) ∪ ((*Cl-D* − { *L2* }) ∪ { *L'*
}))
and L'' = *subst-lit M* σ' **by** *auto*
have *M* ∉ *cl-ecl C*
proof
assume *M* ∈ *cl-ecl C*
from *this* **have** *vars-of-lit M* ⊆ *vars-of-cl* (*cl-ecl C*) **by** *auto*
from *this* **and** ‹*coincide-on* σ σ' (*vars-of-cl* (*cl-ecl C*))›
have *coincide-on* σ σ' (*vars-of-lit M*) **unfolding** *coincide-on-def* **by** *auto*
from *this* **have** *subst-lit M* σ = *subst-lit M* σ' **using** *coincide-on-lit* **by**
auto
from *this* **and** ‹L'' = *subst-lit M* σ'›**have** L'' = *subst-lit M* σ **by** *auto*
from ‹*M* ∈ *cl-ecl C*› **and** ‹L'' = *subst-lit M* σ›**and** ‹*validate-ground-lit ?I*
L''›
have *validate-ground-clause ?I* (*subst-cl* (*cl-ecl C*) σ)
by (*metis* (*mono-tags*, *lifting*) *subst-cl.simps mem-Collect-eq*
validate-ground-clause.simps)
from *this* **and** *cm* **show** *False* **by** *blast*
qed
have *M* ∉ *Cl-D* − { *L2* }
proof
assume *M* ∈ *Cl-D* − { *L2* }
from *this* **and** *cld* **and** ‹L'' = *subst-lit M* σ'› **and** ‹*validate-ground-lit ?I*
L''›
have *validate-ground-clause ?I* (*subst-cl* ((*cl-ecl D*) − { *L2* }) σ')
by (*metis* (*mono-tags*, *lifting*) *subst-cl.simps mem-Collect-eq vali-
date-ground-clause.simps*)
from *this* **and** ‹¬*validate-ground-clause ?I* (*subst-cl* ((*cl-ecl D*) − { *L2* }
) σ')›
show *False* **by** *blast*
qed
have *M* ≠ *L'*

proof
assume $M = L'$
from ‹*?I* (*subst* u σ') (*subst* v σ')›
and e **have** *?I* (*subst* u' σ') (*subst* v σ') **by** *metis*
from *rt* **and** *st* ‹*fo-interpretation ?I*› ‹*?I* (*subst* u' σ') (*subst* v σ')›
have *?I* (*subst* t σ') (*subst* t' σ')
using *replacement-preserves-congruences* [*of ?I* u' σ' v t p t']
unfolding *fo-interpretation-def* **by** *metis*
from *l1* **and** *cm* **have** ¬ (*validate-ground-lit ?I* (*subst-lit L1* σ'))
using ‹*subst-cl* (*cl-ecl C*) σ' = *subst-cl* (*cl-ecl C*) σ›
‹*subst-lit L1* σ' ∈ *subst-cl* (*cl-ecl C*) σ'›
validate-ground-clause.simps **by** *blast*
from *this* **and** ‹*?I* (*subst* t σ') (*subst* t' σ')› **and** ‹*fo-interpretation ?I*›
and l' ‹*orient-lit-inst L1 t s polarity* σ'›
have ¬*validate-ground-lit ?I* (*subst-lit* L' σ')
using *trm-rep-preserves-lit-semantics* [*of ?I t* σ' t' *L1 s polarity* σ'] **by** *metis*
from *this* **and** ‹$M = L'$› **and** ‹*validate-ground-lit ?I* L''› **and** ‹L'' = *subst-lit M* σ'›
show *False* **by** *blast*
qed
from *this* **and** ‹$M \notin$ *Cl-D* − { *L2* }› ‹$M \notin$ (*cl-ecl C*)› **and** *m clc* **show** *False* **by** *auto*
qed

Second, we show that the clause contains no contradictory literal (otherwise the reflexion rule applies).

let *?no-cont* = ∀ *L t s*. (*L* ∈ (*cl-ecl C*)) ⟶ (*eligible-literal L C* σ)
⟶ (*orient-lit-inst L t s neg* σ) ⟶ (*trm-rep* (*subst t* σ) *S*) = (*subst t* σ)
⟶ (*subst t* σ) ≠ (*subst s* σ)
have *?no-cont*
proof (*rule ccontr*)
assume ¬ *?no-cont*
then obtain *L t s* **where** *l*: *L* ∈ (*cl-ecl C*) **and** *e*: (*eligible-literal L C* σ)
and *o*: *orient-lit-inst L t s neg* σ
and (*trm-rep* (*subst t* σ) *S*) = (*subst t* σ)
and (*subst t* σ) = (*subst s* σ) **by** *blast*
from ‹(*subst t* σ) = (*subst s* σ)›
have *t*: *ck-unifier t s* σ *Ground* **unfolding** *ck-unifier-def Unifier-def* **by** *auto*
from *l* **and** *e* **and** *o* **and** *t* **obtain** *R Cl-R nt-R* **where**
R = *Ecl Cl-R nt-R* **and** *Cl-R* = (*subst-cl* ((*cl-ecl C*) − { *L* }) σ) **and**
reflexion C R σ *Ground* ((*cl-ecl C*) − { *L* })
and *trms-ecl R* = (*dom-trms* (*cl-ecl R*) (*subst-set* ((*trms-ecl C*) ∪ { *t* }) σ))
unfolding *reflexion-def get-trms-def*
by *fastforce*
from ‹*C* ∈ *S*› **and** ‹*reflexion C R* σ *Ground* ((*cl-ecl C*) − { *L* })›
have *derivable R* { *C* } *S* σ *Ground* ((*cl-ecl C*) − { *L* })

unfolding *derivable-def* **by** *auto*
from ‹*ground-clause* (*subst-cl* (*cl-ecl* C) σ)›
and ‹*Cl-R* = (*subst-cl* ((*cl-ecl* C) − { L }) σ)›
have *ground-clause Cl-R* **using** *ground-clause.simps* **by** *auto*
from *this* **and** ‹R = *Ecl Cl-R nt-R*› **have** *ground-clause* (*cl-ecl* R) **by** *auto*
from ‹*ground-clause* (*subst-cl* (*cl-ecl* C) σ)›
have *grounding-set* { C } σ **unfolding** *grounding-set-def* **by** *auto*
from *this* **and** ‹*ground-clause* (*cl-ecl* R)›
‹*derivable* R { C } S σ *Ground* ((*cl-ecl* C) − { L })› **and** ‹*ground-inference-saturated* S›
have (*redundant-inference* R S { C } σ) **unfolding** *ground-inference-saturated-def*

by *blast*
from *this* **obtain** S' **where** $S' \subseteq$ (*instances* S) **and**
(*set-entails-clause* (*clset-instances* S') (*cl-ecl* R))
and *all-smaller*: $\forall\, x \in S'$. ((*cl-ecl* (*fst* x), *snd* x), *cl-ecl* C, σ) $\in$ *cl-ord*
and *all-normalized-term-included*: ($\forall\, x \in S'$.
(*subterms-inclusion* (*subst-set* (*trms-ecl* (*fst* x)) (*snd* x))
(*trms-ecl* R)))
unfolding *redundant-inference-def* **by** *auto*
have *validate-clause-set ?I* (*clset-instances* S')
proof (*rule ccontr*)
assume ¬ *validate-clause-set ?I* (*clset-instances* S')
then obtain x **where** $x \in$(*clset-instances* S') **and** ¬*validate-clause ?I* x
using *validate-clause-set.simps* **by** *blast*
from ‹$x \in$(*clset-instances* S')› **obtain** *pair'* **where** *pair'* $\in S'$
and x = (*subst-cl* (*cl-ecl* (*fst pair'*)) (*snd pair'*))
unfolding *clset-instances-def*
by *auto*
from *all-smaller* **and** ‹*pair'* $\in S'$› **have** (*pair'*,(C,σ)) $\in$ *ecl-ord*
by (*metis member-ecl-ord-iff prod.collapse*)
from *this* **and** ‹C = *fst pair*› **and** ‹σ = *snd pair*› **have** (*pair'*,*pair*) $\in$ *ecl-ord*
by *auto*
from *this* **and** *hyp-ind* **have** *?P pair'* **by** *blast*

from ‹*trms-ecl* R = (*dom-trms* (*cl-ecl* R) (*subst-set* ((*trms-ecl* C) $\cup$ { t }) σ))›
have *trms-ecl* $R \subseteq$ (*subst-set* ((*trms-ecl* C) $\cup$ { t }) σ)
using *dom-trms-subset* **by** *metis*
from ‹*pair'* $\in S'$› **and** *all-normalized-term-included* **have**
(*subterms-inclusion* (*subst-set* (*trms-ecl* (*fst pair'*)) (*snd pair'*))
(*trms-ecl* R)) **by** *auto*
have i: (*all-trms-irreducible* (*subst-set* (*trms-ecl* (*fst pair'*)) (*snd pair'*))
(λt. (*trm-rep* t S)))
proof (*rule ccontr*)
assume ¬(*all-trms-irreducible* (*subst-set* (*trms-ecl* (*fst pair'*)) (*snd pair'*))
(λt. (*trm-rep* t S)))
then obtain t' t'' **where** $t' \in$ (*subst-set* (*trms-ecl* (*fst pair'*)) (*snd pair'*))

occurs-in t'' t' **and** *trm-rep* t'' $S \neq t''$ **unfolding** *all-trms-irreducible-def* **by** *blast*
from ‹$t' \in$ (*subst-set* (*trms-ecl* (*fst pair'*)) (*snd pair'*))›
and ‹(*subterms-inclusion* (*subst-set* (*trms-ecl* (*fst pair'*)) (*snd pair'*)) (*trms-ecl* R))›
obtain s' **where** $s' \in$ (*trms-ecl* R) **and** *occurs-in* t' s'
unfolding *subterms-inclusion-def* **by** *auto*
from ‹$s' \in$ (*trms-ecl* R)› **and** ‹*trms-ecl* $R \subseteq$ (*subst-set* ((*trms-ecl* C) $\cup$ { t }) σ)›
obtain s'' **where** $s' =$ *subst* s'' σ **and** $s'' \in$ ((*trms-ecl* C) $\cup$ { t }) **by** *auto*
from ‹$s'' \in$ ((*trms-ecl* C) $\cup$ { t })› **have** $s'' \in$ *trms-ecl* $C \vee s'' = t$ **by** *auto*
thus *False*
proof
assume $s'' \in$ *trms-ecl* C
from *this* **and** ‹$s' =$ *subst* s'' σ› **have** $s' \in$ (*subst-set* (*trms-ecl* C) σ) **by** *auto*
from *this* **and** ‹(*all-trms-irreducible* (*subst-set* (*trms-ecl* C) σ)
($\lambda t.$ (*trm-rep* t S)))› **and** ‹*occurs-in* t' s'› **have** *trm-rep* t' $S = t'$
unfolding *all-trms-irreducible-def* **by** *blast*
from *this* **and** ‹*occurs-in* t'' t'› **and** ‹*trm-rep* t'' $S \neq t''$›**show** *False*
using *occurs-in-def subts-of-irred-trms-are-irred* **by** *blast*
next
assume $s'' = t$
from *this* **and** ‹$s' =$ *subst* s'' σ› **have** $s' =$ *subst* t σ **by** *auto*
from *this* **and** ‹(*trm-rep* (*subst* t σ) S) = (*subst* t σ)›
have *trm-rep* s' $S = s'$ **by** *blast*
from ‹*trm-rep* s' $S = s'$› ‹*trm-rep* t'' $S \neq t''$› ‹*occurs-in* t' s'› ‹*occurs-in* t'' t'›
show *False* **using** *occurs-in-def subts-of-irred-trms-are-irred* **by** *blast*
qed
qed
from ‹$S' \subseteq$ (*instances* S)› **and** ‹*pair'* $\in S'$› **have**
ii: *ground-clause* (*subst-cl* (*cl-ecl* (*fst pair'*)) (*snd pair'*))
unfolding *instances-def* [*of S*] **by** *fastforce*
from ‹$S' \subseteq$ (*instances* S)› **and** ‹*pair'* $\in S'$› **have**
iii: (*fst pair'*) $\in S$ **unfolding** *instances-def* [*of S*] **by** *fastforce*
from ‹*?P pair'*› **and** *i ii iii* **have** *validate-ground-clause ?I*
(*subst-cl* (*cl-ecl* (*fst pair'*)) (*snd pair'*)) **unfolding** *int-clset-def* **by** *blast*
from *this* **and** ‹$x =$ (*subst-cl* (*cl-ecl* (*fst pair'*)) (*snd pair'*))›
and ‹¬*validate-clause ?I x*› **show** *False*
by (*metis ii substs-preserve-ground-clause validate-clause.simps*)
qed
from *this* **and** ‹*fo-interpretation ?I*› **and**
‹(*set-entails-clause* (*clset-instances* S') (*cl-ecl* R))›
have *validate-clause ?I* (*cl-ecl* R) **unfolding** *set-entails-clause-def* **by** *blast*
from *this* **have** *validate-ground-clause ?I* (*cl-ecl* R)
by (*metis* ‹$R =$ *Ecl Cl-R nt-R*› ‹*ground-clause Cl-R*›

cl-ecl.simps substs-preserve-ground-clause validate-clause.simps)
from *this* **obtain** L' **where** $L' \in$ (*cl-ecl R*) **and** *validate-ground-lit ?I* L'
using *validate-ground-clause.simps* **by** *blast*
from ‹$L' \in$ (*cl-ecl R*)› **and** ‹*R* = *Ecl Cl-R nt-R*› **and**
‹*Cl-R* = (*subst-cl* ((*cl-ecl C*) − { *L* }) σ)›
obtain L'' **where** $L'' \in$ *cl-ecl C* **and** L' = *subst-lit* L'' σ
by *auto*
from ‹$L'' \in$ *cl-ecl C*› **and** ‹L' = *subst-lit* L'' σ›**and** ‹*validate-ground-lit ?I*
L'›
have *validate-ground-clause ?I* (*subst-cl* (*cl-ecl C*) σ)
by (*metis* (*mono-tags*, *lifting*) *subst-cl.simps mem-Collect-eq*
validate-ground-clause.simps)
from *this* **and** *cm* **show** *False* **unfolding** *int-clset-def* **by** *blast*
qed

Third, we prove that the clause contains no pair of equations with the same left-hand side and equivalent right-hand sides (otherwise the factorization rule applies and a smaller false clause is derived).

let *?no-fact* = ∀ *L1 L2 t s u v*. (*L1* ∈ (*cl-ecl C*)) ⟶ (*eligible-literal L1 C* σ)
⟶ (*L2* ∈ (*cl-ecl C*) − { *L1* }) ⟶ (*orient-lit-inst L1 t s pos* σ)
⟶ (*orient-lit-inst L2 u v pos* σ) ⟶ (*subst t* σ) = (*subst u* σ)
⟶ (¬ (*proper-subterm-red t S* σ))
⟶ (*trm-rep* ((*subst s*) σ) *S*) ≠ (*trm-rep* ((*subst v*) σ) *S*)
have *?no-fact*
proof (*rule ccontr*)
assume ¬ *?no-fact*
then obtain *L1 L2 t s u v* **where** *l1*: *L1* ∈ (*cl-ecl C*) **and** *l2*: *L2* ∈ (*cl-ecl*
C) − { *L1* }
and *e1*: (*eligible-literal L1 C* σ) **and** *o1*: (*orient-lit-inst L1 t s pos* σ)
and *o2*: (*orient-lit-inst L2 u v pos* σ) **and** *e*: (*subst t* σ) = (*subst u* σ)
and (¬ (*proper-subterm-red t S* σ))
and *i*: (*trm-rep* ((*subst s*) σ) *S*) = (*trm-rep* ((*subst v*) σ) *S*)
by *blast*
from *e* **have** *t*: *ck-unifier t u* σ *Ground* **unfolding** *ck-unifier-def Unifier-def*

using *inferences.distinct* **by** *metis*
from ‹*L1* ∈ (*cl-ecl C*)› *o1* ‹¬(*validate-ground-clause* (*int-clset S*) (*subst-cl*
(*cl-ecl C*) σ))›
have *trm-rep* (*subst t* σ) *S* ≠ *trm-rep* (*subst s* σ) *S*
using *no-valid-literal* **by** *metis*
then have *subst t* σ ≠ *subst s* σ **by** *metis*

from ‹*L2* ∈ (*cl-ecl C*) − { *L1* }› **have** *L2* ∈ (*cl-ecl C*) **by** *auto*
from *this o2* ‹¬(*validate-ground-clause* (*int-clset S*) (*subst-cl* (*cl-ecl C*) σ))›
have *trm-rep* (*subst u* σ) *S* ≠ *trm-rep* (*subst v* σ) *S*
using *no-valid-literal* **by** *metis*
from *this* **and** *e* **have** *subst t* σ ≠ *subst v* σ **by** *metis*

obtain *R Cl-R nt-R* L' **where**

ntr: *nt-R* = (*dom-trms Cl-R* (*subst-set* ((*trms-ecl C*) ∪ (*proper-subterms-of t*)) σ))
and *r*: *R* = *Ecl Cl-R nt-R*
and *clr*: *Cl-R* = (*subst-cl* (((*cl-ecl C*) − { *L2* }) ∪ { *L'* }) σ)
and *l'*: *L'* = *Neg* (*Eq s v*) **by** *auto*
from *ntr r l' clr l1 l2 o1 o2 e1 t*
‹*subst t σ ≠ subst s σ*› ‹*subst t σ ≠ subst v σ*›
have *factorization C R σ Ground* (((*cl-ecl C*) − { *L2* }) ∪ { *L'* })
unfolding *factorization-def get-trms-def* **using** *inferences.distinct*
by (*metis cl-ecl.simps*)

from *l2* **have** (*subst-lit L2 σ*) ∈ (*subst-cl* (*cl-ecl C*) σ) **by** *auto*
from *this* **and** ‹*ground-clause* (*subst-cl* (*cl-ecl C*) σ)›
have *vars-of-lit* (*subst-lit L2 σ*) = {}
by *auto*
from *this* **and** *o2* **have** *vars-of* (*subst v σ*) = {}
unfolding *orient-lit-inst-def* **using** *vars-of-lit.simps vars-of-eq.simps* **by** *force*

from *l1* **have** (*subst-lit L1 σ*) ∈ (*subst-cl* (*cl-ecl C*) σ) **by** *auto*
from *this* **and** ‹*ground-clause* (*subst-cl* (*cl-ecl C*) σ)›
have *vars-of-lit* (*subst-lit L1 σ*) = {} **by** *auto*
from *this* **and** *o1* **have** *vars-of* (*subst s σ*) = {}
unfolding *orient-lit-inst-def* **using** *vars-of-lit.simps vars-of-eq.simps* **by** *force*
from ‹*vars-of* (*subst v σ*) = {}› **and** ‹*vars-of* (*subst s σ*) = {}›
and *l'* **have** *vars-of-lit* (*subst-lit L' σ*) = {} **by** *auto*

from ‹*C* ∈ *S*› **and** ‹*factorization C R σ Ground* (((*cl-ecl C*) − { *L2* }) ∪ { *L'* })›
have *derivable R* { *C* } *S σ Ground* (((*cl-ecl C*) − { *L2* }) ∪ { *L'* })
unfolding *derivable-def* **by** *auto*
have *ground-clause Cl-R*
proof (*rule ccontr*)
assume ¬*ground-clause Cl-R*
then have *vars-of-cl Cl-R* ≠ {} **by** *auto*
then obtain *M* **where** *M* ∈ *Cl-R* **and** *vars-of-lit M* ≠ {} **by** *auto*
from ‹*M* ∈ *Cl-R*› **and** *clr* **obtain** *M'* **where** *M* = (*subst-lit M' σ*)
and *M'* ∈((*cl-ecl C*) − { *L2*, *L1* }) ∪ { *L'*, *L1* }
by *auto*
show *False*
proof (*cases*)
assume *M'* = *L'*
from *this* **and** ‹*vars-of-lit* (*subst-lit L' σ*) = {}› **and** ‹*vars-of-lit M* ≠ {}›
and ‹*M* = (*subst-lit M' σ*)› **show** *False* **by** *auto*
next
assume *M'* ≠ *L'*
from *this* **and** *l1* **and** ‹*M'* ∈((*cl-ecl C*) − { *L2*, *L1* }) ∪ { *L'*, *L1* }›

have $M' \in (cl\text{-}ecl\ C)$
by *auto*
from *this* **and** ‹*ground-clause* (*subst-cl* (*cl-ecl C*) σ)› **have**
vars-of-lit (*subst-lit* M' σ) = {} **by** *auto*
from *this* **and** ‹M = (*subst-lit* M' σ)› **and**
‹*vars-of-lit* $M \neq$ {}› **show** *False* **by** *auto*
qed
qed

from ‹*ground-clause Cl-R*› **and** ‹R = *Ecl Cl-R nt-R*› **have** *ground-clause* (*cl-ecl R*) **by** *auto*
from ‹*ground-clause* (*subst-cl* (*cl-ecl C*) σ)›
have *grounding-set* { C } σ **unfolding** *grounding-set-def* **by** *auto*
from *this* ‹*ground-clause* (*cl-ecl R*)›
and ‹*derivable R* { C } S σ *Ground* (((*cl-ecl C*) − { *L2* }) ∪ { L' })›
and ‹*ground-inference-saturated S*›
have (*redundant-inference R S* { C } σ) **unfolding** *ground-inference-saturated-def*

by *blast*
from *this* **obtain** S' **where** $S' \subseteq$ (*instances S*) **and**
(*set-entails-clause* (*clset-instances* S') (*cl-ecl R*))
and *all-smaller*: $\forall x \in S'$. ((*cl-ecl* (*fst x*), *snd x*), *cl-ecl C*, σ) ∈ *cl-ord*
and *all-normalized-term-included*: ($\forall x \in S'$.
(*subterms-inclusion* (*subst-set* (*trms-ecl* (*fst x*)) (*snd x*))
(*trms-ecl R*)))
unfolding *redundant-inference-def* **by** *auto*
have *validate-clause-set ?I* (*clset-instances* S')
proof (*rule ccontr*)
assume ¬ *validate-clause-set ?I* (*clset-instances* S')
then obtain x **where** x ∈(*clset-instances* S') **and** ¬*validate-clause ?I x*
using *validate-clause-set.simps* **by** *blast*
from ‹x ∈(*clset-instances* S')› **obtain** *pair'* **where** *pair'* ∈ S'
and x = (*subst-cl* (*cl-ecl* (*fst pair'*)) (*snd pair'*))
unfolding *clset-instances-def*
by *auto*
from *all-smaller* **and** ‹*pair'* ∈ S'› **have** (*pair'*,(C,σ)) ∈ *ecl-ord*
by (*metis member-ecl-ord-iff prod.collapse*)
from *this* **and** ‹C = *fst pair*› **and** ‹σ = *snd pair*› **have** (*pair'*,*pair*) ∈ *ecl-ord*
by *auto*
from *this* **and** *hyp-ind* **have** *?P pair'* **by** *blast*

from *r ntr* **have** *trms-ecl R* = (*dom-trms* (*cl-ecl R*)
(*subst-set* ((*trms-ecl C*) ∪ (*proper-subterms-of t*)) σ))
by *auto*

from *this* **have** *trms-ecl R* ⊆ (*subst-set* ((*trms-ecl C*) ∪ (*proper-subterms-of t*)) σ)
using *dom-trms-subset* **by** *metis*

from ‹*pair*$'$ $\in$ S'› **and** *all-normalized-term-included* **have**
(*subterms-inclusion* (*subst-set* (*trms-ecl* (*fst pair*$'$)) (*snd pair*$'$))
(*trms-ecl* R)) **by** *auto*

have *i*: (*all-trms-irreducible* (*subst-set* (*trms-ecl* (*fst pair*$'$)) (*snd pair*$'$))
($\lambda t.$ (*trm-rep* t S)))
proof (*rule ccontr*)
assume $\neg$(*all-trms-irreducible* (*subst-set* (*trms-ecl* (*fst pair*$'$)) (*snd pair*$'$))
($\lambda t.$ (*trm-rep* t S)))
then obtain t' t'' **where** $t' \in$ (*subst-set* (*trms-ecl* (*fst pair*$'$)) (*snd pair*$'$))
occurs-in t'' t' **and** *trm-rep* t'' $S \neq t''$ **unfolding** *all-trms-irreducible-def*
by *blast*
from ‹$t' \in$ (*subst-set* (*trms-ecl* (*fst pair*$'$)) (*snd pair*$'$))›
and ‹(*subterms-inclusion* (*subst-set* (*trms-ecl* (*fst pair*$'$)) (*snd pair*$'$))
(*trms-ecl* R))›
obtain s' **where** $s' \in$ (*trms-ecl* R) **and** *occurs-in* t' s'
unfolding *subterms-inclusion-def* **by** *auto*
from ‹$s' \in$ (*trms-ecl* R)› **and** ‹*trms-ecl* $R \subseteq$ (*subst-set* ((*trms-ecl* C) $\cup$
(*proper-subterms-of* t)) σ)›
have $s' \in$(*subst-set* ((*trms-ecl* C) $\cup$ (*proper-subterms-of* t)) σ) **by** *auto*
then obtain s''
where $s' =$ *subst* s'' σ **and** $s'' \in$ ((*trms-ecl* C) $\cup$ (*proper-subterms-of* t))
by *auto*
from ‹$s'' \in$ ((*trms-ecl* C) $\cup$ (*proper-subterms-of* t))› **have** $s'' \in$ *trms-ecl*
C
$\vee$ $s'' \in$ (*proper-subterms-of* t) **by** *auto*
thus *False*
proof
assume $s'' \in$ *trms-ecl* C
from *this* **and** ‹$s' =$ *subst* s'' σ› **have** $s' \in$ (*subst-set* (*trms-ecl* C) σ) **by**
auto
from *this* **and** ‹(*all-trms-irreducible* (*subst-set* (*trms-ecl* C) σ)
($\lambda t.$ (*trm-rep* t S)))› **and** ‹*occurs-in* t' s'› **have** *trm-rep* t' $S = t'$
unfolding *all-trms-irreducible-def* **by** *blast*
from *this* **and** ‹*occurs-in* t'' t'› **and** ‹*trm-rep* t'' $S \neq t''$›**show** *False*
using *occurs-in-def subts-of-irred-trms-are-irred* **by** *blast*
next
assume $s'' \in$ (*proper-subterms-of* t)
from ‹*occurs-in* t' s'› ‹*occurs-in* t'' t'› ‹$s' = s'' \lhd \sigma$› ‹*trm-rep* t'' $S \neq t''$›
have *trm-rep* (*subst* s'' σ) $S \neq$ (*subst* s'' σ)
using *occurs-in-def subts-of-irred-trms-are-irred* **by** *blast*
from *this* **and** ‹$s'' \in$ (*proper-subterms-of* t)› **and** ‹$\neg$ (*proper-subterm-red*
t S σ)›
show *False* **using** *proper-subterm-red-def proper-subterms-of.simps* **by**
blast
qed
qed
from ‹$S' \subseteq$ (*instances* S)› **and** ‹*pair*$'$ $\in$ S'› **have**

```
      ii: ground-clause (subst-cl (cl-ecl (fst pair′)) (snd pair′))
      unfolding instances-def [of S] by fastforce
     from ‹S′ ⊆ (instances S)› and ‹pair′ ∈ S′› have
      iii: (fst pair′) ∈ S unfolding instances-def [of S] by fastforce
     from ‹?P pair′› and i ii iii have validate-ground-clause ?I
      (subst-cl (cl-ecl (fst pair′)) (snd pair′))
      unfolding int-clset-def by blast
     from this and ‹x = (subst-cl (cl-ecl (fst pair′)) (snd pair′))›
      and ‹¬validate-clause ?I x› show False
      by (metis ii substs-preserve-ground-clause validate-clause.simps)
    qed
    from this and ‹fo-interpretation ?I› and
    ‹(set-entails-clause (clset-instances S′) (cl-ecl R))›
     have validate-clause ?I (cl-ecl R) unfolding set-entails-clause-def by blast
    from this have validate-ground-clause ?I (cl-ecl R)
      by (metis ‹R = Ecl Cl-R nt-R› ‹ground-clause Cl-R›
        cl-ecl.simps substs-preserve-ground-clause validate-clause.simps)
    from this obtain L′′ where L′′ ∈ (cl-ecl R) and validate-ground-lit ?I L′′
      using validate-ground-clause.simps by blast
    from ‹L′′ ∈ (cl-ecl R)› and ‹R = Ecl Cl-R nt-R› and
      ‹Cl-R = (subst-cl ( ((cl-ecl C) − { L2 }) ∪ { L′ } ) σ)›
      obtain M where m: M ∈ ( ((cl-ecl C) − { L2, L1 }) ∪ { L′, L1 } )
      and L′′ = subst-lit M σ
      by auto
    have M ∈ cl-ecl C
    proof (rule ccontr)
      assume M ∉ cl-ecl C
      from this and m and l1 have M = L′ by auto
      from this and ‹L′′ = subst-lit M σ› and ‹L′ = (Neg (Eq s v))›
        have L′′ = (Neg (Eq (subst s σ)  (subst v σ))) by auto
      from this and ‹validate-ground-lit ?I L′′›
        have ¬(?I (subst s σ) (subst v σ))
           using validate-ground-lit.simps(2) validate-ground-eq.simps by metis
       from this and i show False unfolding same-values-def int-clset-def by blast
    qed
     from ‹M ∈ cl-ecl C› and ‹L′′ = subst-lit M σ›and ‹validate-ground-lit ?I L′′›
      have validate-ground-clause ?I (subst-cl (cl-ecl C) σ)
        by (metis (mono-tags, lifting) subst-cl.simps mem-Collect-eq
          validate-ground-clause.simps)
    from this and cm show False by blast
   qed
```

Now, it remains to prove that the considered clause yields a rule which can be used to reduce the left-hand side of the maximal equation, which (since no reduction is possible) entails that the left-hand side must be equivalent to the right-hand side (thus contradicting the fact that the clause is false).

```
   have (finite (cl-ecl C)) by (simp add: ‹C ∈ S› all-finite)
```

have (*cl-ecl C*) ≠ {} **by** (*simp add*: ‹*C* ∈ *S*› *all-non-empty*)
from ‹*finite* (*cl-ecl C*)› ‹(*cl-ecl C*) ≠ {}› ‹*ground-clause* (*subst-cl* (*cl-ecl C*) *σ*)›
obtain *L* **where** *L* ∈ (*cl-ecl C*) *eligible-literal L C σ* **using** *eligible-lit-exists* **by** *metis*
obtain *t s p* **where** *orient-lit-inst L t s p σ* **using** *literal.exhaust equation.exhaust*
using *trm-ord-irrefl trm-ord-trans*
unfolding *orient-lit-inst-def irrefl-def trans-def* **by** *metis*

We first show that the terms occurring inside variables are irreducible. To this aim, we need to consider the normal form of the substitution *σ*, obtained by replacing the image of each variable by its normal form.

have ∀ *x y*.
((*x* ∈ *vars-of-cl* (*cl-ecl C*)) ⟶ *occurs-in y* (*subst* (*Var x*) *σ*) ⟶ *trm-rep y S* = *y*)
proof (*rule ccontr*)
assume ¬(∀ *x y*. (*x* ∈ *vars-of-cl* (*cl-ecl C*)) ⟶ *occurs-in y* (*subst* (*Var x*) *σ*) ⟶ *trm-rep y S* = *y*)
then obtain *x y* **where** (*x* ∈ *vars-of-cl* (*cl-ecl C*)) **and**
occurs-in y (*subst* (*Var x*) *σ*) **and** *trm-rep y S* ≠ *y* **by** *blast*
from ‹*occurs-in y* (*subst* (*Var x*) *σ*)› **obtain** *p* **where** *subterm* (*subst* (*Var x*) *σ*) *p y*
unfolding *occurs-in-def* **by** *auto*
from ‹*subterm* (*subst* (*Var x*) *σ*) *p y*› **and** ‹*trm-rep y S* ≠ *y*›
have *trm-rep* (*subst* (*Var x*) *σ*) *S* ≠ (*subst* (*Var x*) *σ*)
using *subts-of-irred-trms-are-irred* **by** *blast*

let *?ϑ* = *map-subst* (*λx*. (*trm-rep x S*)) *σ*
have *equivalent-on σ ?ϑ* (*vars-of-cl* (*cl-ecl C*)) *?I*
proof (*rule ccontr*)
assume ¬*equivalent-on σ ?ϑ* (*vars-of-cl* (*cl-ecl C*)) *?I*
then obtain *z* **where** *z* ∈ *vars-of-cl* (*cl-ecl C*)
and ¬ (*?I* (*subst* (*Var z*) *σ*) (*subst* (*Var z*) *?ϑ*))
unfolding *equivalent-on-def* **by** *blast*
from ‹¬ (*?I* (*subst* (*Var z*) *σ*) (*subst* (*Var z*) *?ϑ*))›
have *trm-rep* (*subst* (*Var z*) *σ*) *S* ≠ *trm-rep* (*subst* (*Var z*) *?ϑ*) *S*
unfolding *same-values-def int-clset-def* **by** *blast*
from *this* **have** *trm-rep* (*trm-rep* (*subst* (*Var z*) *σ*) *S*) *S* ≠ *trm-rep* (*subst* (*Var z*) *?ϑ*) *S*
using *trm-rep-involutive* **by** *metis*
from *this* **have** (*subst* (*Var z*) *σ*) = (*subst* (*Var z*) *?ϑ*)
using *map-subst-lemma* [*of z σ λx*. (*trm-rep x S*)] **by** *metis*
from *this* **and** ‹¬ (*?I* (*subst* (*Var z*) *σ*) (*subst* (*Var z*) *?ϑ*))›
show *False* **using** ‹*fo-interpretation ?I*›
unfolding *fo-interpretation-def congruence-def equivalence-relation-def reflexive-def*
by *metis*
qed

from *this* **and** ‹¬ *validate-ground-clause ?I* (*subst-cl* (*cl-ecl C*) σ)›
‹*fo-interpretation ?I*›
have ¬ *validate-ground-clause ?I* (*subst-cl* (*cl-ecl C*) *?ϑ*)
using *equivalent-on-cl* **by** *metis*
have *lower-on ?ϑ σ* (*vars-of-cl* (*cl-ecl C*))
proof (*rule ccontr*)
assume ¬*lower-on ?ϑ σ* (*vars-of-cl* (*cl-ecl C*))
then obtain *z* **where** *z* ∈ *vars-of-cl* (*cl-ecl C*)
and (*subst* (*Var z*) σ) ≠ (*subst* (*Var z*) *?ϑ*)
and ((*subst* (*Var z*) *?ϑ*),(*subst* (*Var z*) σ)) ∉ *trm-ord*
unfolding *lower-on-def lower-or-eq-def* **by** *metis*
from ‹(*subst* (*Var z*) σ) ≠ (*subst* (*Var z*) *?ϑ*)› **have**
(*trm-rep* (*subst* (*Var z*) σ) *S*) = (*subst* (*Var z*) *?ϑ*)
using *map-subst-lemma* [*of z σ λx.* (*trm-rep x S*)] **by** *metis*
from *this* **and** ‹(*subst* (*Var z*) σ) ≠ (*subst* (*Var z*) *?ϑ*)›
and ‹((*subst* (*Var z*) *?ϑ*),(*subst* (*Var z*) σ)) ∉ *trm-ord*›
show *False* **using** *trm-rep-is-lower* **by** *metis*
qed
have *subst* (*Var x*) σ ≠ (*Var x*)
proof
assume *subst* (*Var x*) σ = (*Var x*)
from *this* **and** ‹*x* ∈ *vars-of-cl* (*cl-ecl C*)› **have** ¬ (*ground-on* (*vars-of-cl*
(*cl-ecl C*)) σ)
unfolding *ground-on-def ground-term-def* **by** *auto*
from *this* **and** ‹*ground-clause* (*subst-cl* (*cl-ecl C*) σ)›
show *False* **using** *ground-clauses-and-ground-substs* **by** *metis*
qed
from ‹*subst* (*Var x*) σ ≠ (*Var x*)›
have (*trm-rep* (*subst* (*Var x*) σ) *S*) = (*subst* (*Var x*) *?ϑ*)
using *map-subst-lemma* [*of x σ λx.* (*trm-rep x S*)] **by** *metis*
from *this* **and** ‹*trm-rep* (*subst* (*Var x*) σ) *S* ≠ (*subst* (*Var x*) σ)›
have ((*subst* (*Var x*) *?ϑ*),(*subst* (*Var x*) σ)) ∈ *trm-ord*
using *trm-rep-is-lower* **by** *metis*
from ‹*lower-on ?ϑ σ* (*vars-of-cl* (*cl-ecl C*))› **and** ‹*x* ∈ *vars-of-cl* (*cl-ecl C*)›
‹*finite* (*cl-ecl C*)›
and ‹((*subst* (*Var x*) *?ϑ*),(*subst* (*Var x*) σ)) ∈ *trm-ord*›
have ((*C*,*?ϑ*), (*C*, σ)) ∈ *ecl-ord*
using *lower-on-cl* **by** *blast*
from ‹*C* = *fst pair*› ‹σ = *snd pair*› **have** *pair* = (*C*,σ) **by** *auto*
from *this* **and** ‹((*C*,*?ϑ*), (*C*, σ)) ∈ *ecl-ord*› **have**
((*C*,*?ϑ*),*pair*) ∈ *ecl-ord*
by *metis*
from *this* **and** *hyp-ind* **have** *?P* (*C*,*?ϑ*) **by** *blast*
from ‹(*all-trms-irreducible* (*subst-set* (*trms-ecl C*) σ)
(λ*t.* (*trm-rep t S*)))›
‹*lower-on ?ϑ σ* (*vars-of-cl* (*cl-ecl C*))› ‹*C* ∈ *S*› ‹*fo-interpretation ?I*›
‹*equivalent-on σ ?ϑ* (*vars-of-cl* (*cl-ecl C*)) *?I*› *assms*(*3*)
have (*all-trms-irreducible* (*subst-set* (*trms-ecl C*) *?ϑ*)
(λ*t.* (*trm-rep t S*)))

using *irred-terms-and-reduced-subst* **unfolding** *Ball-def well-constrained-def*

by *metis*
have *ground-clause* (*subst-cl* (*cl-ecl* C) *?ϑ*)
proof −
from ‹*ground-clause* (*subst-cl* (*cl-ecl* C) σ)›
have *ground-on* (*vars-of-cl* (*cl-ecl* C)) σ **using** *ground-clauses-and-ground-substs*
by *auto*
from *this* **and** ‹*lower-on ?ϑ* σ (*vars-of-cl* (*cl-ecl* C))›
have *ground-on* (*vars-of-cl* (*cl-ecl* C)) *?ϑ*
using *lower-on-ground* **by** *meson*
from *this* **show** *?thesis* **using** *ground-substs-yield-ground-clause* **by** *metis*
qed
from *this*
‹(*all-trms-irreducible* (*subst-set* (*trms-ecl* C) *?ϑ*) ($\lambda t.$ (*trm-rep* t S)))›
‹*?P* (C,*?ϑ*)› ‹¬ *validate-ground-clause ?I* (*subst-cl* (*cl-ecl* C) *?ϑ*)›
‹$C \in S$› **show** *False* **unfolding** *int-clset-def* **by** (*metis fst-conv snd-conv*)
qed

Next, we show that the eligible term t is in normal form. We first need to establish the result for proper subterms of t before considering the general case.

have ¬(*proper-subterm-red* t S σ)
proof
assume (*proper-subterm-red* t S σ)
from *this* **have** *trm-rep* (*subst* t σ) $S \neq$ *subst* t σ
using *proper-subterm-red-def substs-preserve-subterms subts-of-irred-trms-are-irred*

by *blast*

from ‹(*proper-subterm-red* t S σ)›
‹$\forall$ x y.
(($x \in$ *vars-of-cl* (*cl-ecl* C)) $\longrightarrow$ *occurs-in* y (*subst* (*Var* x) σ) $\longrightarrow$ *trm-rep* y
S = y)›
‹*eligible-literal* L C σ›
‹*trm-rep* (*subst* t σ) $S \neq$ *subst* t σ› ‹$L \in$ *cl-ecl* C›
‹*orient-lit-inst* L t s p σ› ‹$\forall x \in S$. *finite* (*cl-ecl* x)›
‹*ground-clause* (*subst-cl* (*cl-ecl* C) σ)›
‹*fo-interpretation* (*int-clset* S)›
‹*Ball* S *well-constrained*› ‹$C \in S$›
‹*all-trms-irreducible* (*subst-set* (*trms-ecl* C) σ) ($\lambda t.$ *trm-rep* t S)›
‹¬ *validate-ground-clause* (*int-clset* S) (*subst-cl* (*cl-ecl* C) σ)›
‹*closed-under-renaming* S›
have
$\exists \sigma''$ u u' *pa* v D *L2*. (*reduction* L C σ'' t s p *L2* u u' *pa* v D
(*same-values* ($\lambda t.$ *trm-rep* t S)) S σ $\wedge$ *variable-disjoint* C D)
using *reduction-exists* [*of* p t s C S σ L] **unfolding** *int-clset-def* **by**
blast
from *this* **and** ‹*?nored*› **show** *False* **unfolding** *int-clset-def* **by** *blast*

qed

have $p = neg \vee \neg$ *equivalent-eq-exists t s* (*cl-ecl C*) (*same-values* (λx. *trm-rep x S*)) σ
L
proof (*rule ccontr*)
assume *neg*: $\neg$ ($p = neg \vee \neg$ *equivalent-eq-exists t s* (*cl-ecl C*)
(*same-values* (λx. *trm-rep x S*)) σ *L*)
then have $p \neq neg$ **by** *metis*
from *neg* **have** *equivalent-eq-exists t s* (*cl-ecl C*) (*same-values* (λx. *trm-rep x S*)) σ *L*
by *metis*
from ‹$p \neq neg$› **have** $p = pos$ **using** *sign.exhaust* **by** *auto*
from ‹*equivalent-eq-exists t s* (*cl-ecl C*) (*same-values* (λx. *trm-rep x S*)) σ *L*›
obtain *L2* **where** *L2* $\in$ (*cl-ecl C*) $-$ { *L* } **and** *f*:$\exists$ *u v*. *orient-lit-inst L2 u v pos* σ $\wedge$
subst t σ = *subst u* σ $\wedge$ *trm-rep* (*subst s* σ) *S* = *trm-rep* (*subst v* σ) *S*
unfolding *equivalent-eq-exists-def* **unfolding** *same-values-def* **by** *metis*
from *f* **obtain** *u v* **where** *f'*: *orient-lit-inst L2 u v pos* σ $\wedge$ *subst t* σ = *subst u* σ
$\wedge$ *trm-rep* (*subst s* σ) *S* = *trm-rep* (*subst v* σ) *S*
by *blast*
from *f'* **have** *orient-lit-inst L2 u v pos* σ **by** *metis*
from *f'* **have** *subst t* σ = *subst u* σ **by** *metis*
from *f'* **have** *trm-rep* (*subst s* σ) *S* = *trm-rep* (*subst v* σ) *S* **by** *metis*
from ‹*orient-lit-inst L2 u v pos* σ› ‹*subst t* σ = *subst u* σ›
‹*trm-rep* (*subst s* σ) *S* = *trm-rep* (*subst v* σ) *S*›
‹*orient-lit-inst L t s p* σ› ‹$p = pos$› ‹$L \in$ (*cl-ecl C*)› ‹*L2* $\in$ (*cl-ecl C*) $-$ { *L* }›
‹*eligible-literal L C* σ›
‹$\neg$(*proper-subterm-red t S* σ)›
and ‹*?no-fact*› **show** *False* **by** *blast*
qed

have (*trm-rep* (*subst t* σ) *S*) = (*subst t* σ)
proof (*rule ccontr*)
assume (*trm-rep* (*subst t* σ) *S*) $\neq$ (*subst t* σ)

from ‹$p = neg \vee \neg$ *equivalent-eq-exists t s* (*cl-ecl C*) (*same-values* (λx. *trm-rep x S*)) σ *L*›
‹$\forall$ *x y*.
(($x \in$ *vars-of-cl* (*cl-ecl C*)) $\longrightarrow$ *occurs-in y* (*subst* (*Var x*) σ) $\longrightarrow$ *trm-rep y S* = *y*)›
‹*eligible-literal L C* σ›
‹*trm-rep* (*subst t* σ) *S* $\neq$ *subst t* σ› ‹$L \in$ *cl-ecl C*›
‹*orient-lit-inst L t s p* σ› ‹$\forall$ *x*$\in$*S*. *finite* (*cl-ecl x*)›
‹*ground-clause* (*subst-cl* (*cl-ecl C*) σ)›

‹*fo-interpretation* (*int-clset S*)›
‹*Ball S well-constrained*› ‹$C \in S$›
‹*all-trms-irreducible* (*subst-set* (*trms-ecl C*) σ) (λt. *trm-rep t S*)›
‹$\neg$ *validate-ground-clause* (*int-clset S*) (*subst-cl* (*cl-ecl C*) σ)›
‹*closed-under-renaming S*›
have
$\exists \sigma''$ *u u′ pa v D L2*. (*reduction L C* σ'' *t s p L2 u u′ pa v D*
(*same-values* (λt. *trm-rep t S*)) *S* σ $\wedge$ *variable-disjoint C D*)
using *reduction-exists* [*of p t s C S* σ *L*] **unfolding** *int-clset-def* **by**
blast
from *this* **and** ‹*?nored*› **show** *False* **unfolding** *int-clset-def* **by** *blast*
qed

from ‹*orient-lit-inst L t s p* σ› **have** ((*subst t* σ),(*subst s* σ)) $\notin$ *trm-ord*
unfolding *orient-lit-inst-def* **by** *auto*
from ‹*ground-clause* (*subst-cl* (*cl-ecl C*) σ)›
have *vars-of-cl* (*subst-cl* (*cl-ecl C*) σ) = {} **by** *auto*
from *this* **and** ‹$L \in$ (*cl-ecl C*)› **have** *vars-of-lit* (*subst-lit L* σ) = {} **by** *auto*
from ‹*orient-lit-inst L t s p* σ›
have *orient-lit* (*subst-lit L* σ) (*subst t* σ) (*subst s* σ) *p*
using *lift-orient-lit* **by** *auto*
from ‹*orient-lit* (*subst-lit L* σ) (*subst t* σ) (*subst s* σ) *p*›
have *vars-of* (*subst t* σ) $\subseteq$ *vars-of-lit* (*subst-lit L* σ)
using *orient-lit-vars* **by** *auto*
from *this* **and** ‹*vars-of-lit* (*subst-lit L* σ) = {}› **have** *vars-of* (*subst t* σ) =
{}
by *auto*
from ‹*orient-lit* (*subst-lit L* σ) (*subst t* σ) (*subst s* σ) *p*›
have *vars-of* (*subst s* σ) $\subseteq$ *vars-of-lit* (*subst-lit L* σ)
using *orient-lit-vars* **by** *auto*
from *this* **and** ‹*vars-of-lit* (*subst-lit L* σ) = {}› **have** *vars-of* (*subst s* σ) =
{}
by *auto*
from
‹((*subst t* σ),(*subst s* σ)) $\notin$ *trm-ord*›
‹*vars-of* (*subst t* σ) = {}› ‹*vars-of* (*subst s* σ) = {}›
have (*subst t* σ) = (*subst s* σ) $\vee$ ((*subst s* σ),(*subst t* σ)) $\in$ *trm-ord*
using *trm-ord-ground-total* **unfolding** *ground-term-def* **by** *blast*

from ‹$L \in$ (*cl-ecl C*)› **have** (*subst-lit L* σ) $\in$ (*subst-cl* (*cl-ecl C*) σ) **by** *auto*

Using the fact that the eligible term is in normal form and that the eligible literal is false in the considered interpretation but is not a contradiction, we deduce that this literal must be positive.

have *p* = *pos*
proof (*rule ccontr*)
assume *p* $\neq$ *pos*
from *this* **have** *p* = *neg* **using** *sign.exhaust* **by** *auto*
from ‹*trm-rep* (*subst t* σ) *S* = (*subst t* σ)› ‹$L \in$ (*cl-ecl C*)› ‹*eligible-literal*

$L\ C\ \sigma$›
and ‹*orient-lit-inst* $L\ t\ s\ p\ \sigma$› ‹$p = neg$› ‹*?no-cont*›
have (*subst* $t\ \sigma$) $\neq$ (*subst* $s\ \sigma$) by *blast*
from *this* and
‹(*subst* $t\ \sigma$) = (*subst* $s\ \sigma$) $\vee$ ((*subst* $s\ \sigma$),(*subst* $t\ \sigma$)) $\in$ *trm-ord*›
and ‹*trm-rep* (*subst* $t\ \sigma$) S = *subst* $t\ \sigma$›
have ((*trm-rep* (*subst* $s\ \sigma$) S),(*trm-rep* (*subst* $t\ \sigma$) S)) $\in$ *trm-ord*
using *trm-rep-is-lower* [*of* (*subst* $s\ \sigma$) S] *trm-ord-trans* unfolding *trans-def*
by *metis*
from *this* have (*trm-rep* (*subst* $s\ \sigma$) S) $\neq$ (*trm-rep* (*subst* $t\ \sigma$) S)
using *trm-ord-irrefl irrefl-def* by *metis*

from *this* have $\neg$*validate-ground-eq ?I* (*Eq* (*subst* $t\ \sigma$) (*subst* $s\ \sigma$))
unfolding *same-values-def int-clset-def* using *validate-ground-eq.simps*
by (*metis* (*mono-tags*, *lifting*))
from ‹(*trm-rep* (*subst* $s\ \sigma$) S) $\neq$ (*trm-rep* (*subst* $t\ \sigma$) S)›
have $\neg$*validate-ground-eq ?I* (*Eq* (*subst* $s\ \sigma$) (*subst* $t\ \sigma$))
unfolding *same-values-def int-clset-def* using *validate-ground-eq.simps*
by (*metis* (*mono-tags*, *lifting*))
from ‹*orient-lit-inst* $L\ t\ s\ p\ \sigma$› and ‹p=*neg*› have L = (*Neg* (*Eq* $t\ s$)) $\vee$ L = (*Neg* (*Eq* $s\ t$))
unfolding *orient-lit-inst-def* by *auto*
from *this* have *subst-lit* $L\ \sigma$ = (*Neg* (*Eq* (*subst* $t\ \sigma$) (*subst* $s\ \sigma$)))
$\vee$ *subst-lit* $L\ \sigma$ = (*Neg* (*Eq* (*subst* $s\ \sigma$) (*subst* $t\ \sigma$))) by *auto*
from *this* and ‹$\neg$*validate-ground-eq ?I* (*Eq* (*subst* $s\ \sigma$) (*subst* $t\ \sigma$))›
and ‹$\neg$*validate-ground-eq ?I* (*Eq* (*subst* $t\ \sigma$) (*subst* $s\ \sigma$))›
have *validate-ground-lit ?I* (*subst-lit* $L\ \sigma$) using *validate-ground-lit.simps*(*2*) by *metis*

from ‹(*subst-lit* $L\ \sigma$) $\in$ (*subst-cl* (*cl-ecl* C) σ)›
and ‹*validate-ground-lit ?I* (*subst-lit* $L\ \sigma$)›
have *validate-ground-clause ?I* (*subst-cl* (*cl-ecl* C) σ)
using *validate-ground-clause.elims*(*3*) by *blast*

from *this* and ‹$\neg$ *validate-ground-clause ?I* (*subst-cl* (*cl-ecl* C) σ)› show *False* by *blast*
qed

This entails that the right-hand side of the eligible literal occurs in the set of possible values for the left-hand side t, which is impossible since this term is irreducible.

from ‹$L \in$ (*cl-ecl* C)› ‹*orient-lit-inst* $L\ t\ s\ p\ \sigma$› ‹p = *pos*›
‹$\neg$ *validate-ground-clause ?I* (*subst-cl* (*cl-ecl* C) σ)›
have *trm-rep* (*subst* $t\ \sigma$) $S \neq$ *trm-rep* (*subst* $s\ \sigma$) S
using *no-valid-literal* by *metis*

from *this* have (*subst* $t\ \sigma$) $\neq$ (*subst* $s\ \sigma$) by *metis*
from *this* and ‹(*subst* $t\ \sigma$) = (*subst* $s\ \sigma$) $\vee$ ((*subst* $s\ \sigma$),(*subst* $t\ \sigma$)) $\in$ *trm-ord*›
have ((*subst* $s\ \sigma$),(*subst* $t\ \sigma$)) $\in$ *trm-ord*

using *trm-ord-ground-total* **unfolding** *ground-term-def* **by** *blast*
from ‹*p=pos*› **and** ‹*orient-lit-inst L t s p σ*› **have** ¬*negative-literal L*
unfolding *orient-lit-inst-def* **by** *auto*
from *this* **and** ‹*eligible-literal L C σ*›
have *sel*(*cl-ecl C*) = {} **and** *maximal-literal* (*subst-lit L σ*) (*subst-cl* (*cl-ecl C*) *σ*)
using *sel-neg* **unfolding** *eligible-literal-def* **by** *auto*

from ‹¬ *validate-ground-clause ?I* (*subst-cl* (*cl-ecl C*) *σ*)›
have *smaller-lits-are-false* (*subst t σ*) (*subst-cl* (*cl-ecl C*) *σ*) *S*
using *smaller-lits-are-false-if-cl-not-valid* [*of S* (*subst-cl* (*cl-ecl C*) *σ*)] **by** *blast*
from ‹*p* = *pos*› **and** ‹*p* = *neg* ∨ ¬ *equivalent-eq-exists t s* (*cl-ecl C*)
(*same-values* (*λx. trm-rep x S*)) *σ L*›
have ¬ *equivalent-eq-exists t s* (*cl-ecl C*) (*int-clset S*) *σ L* **unfolding** *int-clset-def*
using *sign.distinct* **by** *metis*
from *this* ‹*p=pos*› **have** *maximal-literal-is-unique* (*subst t σ*) (*subst s σ*) (*cl-ecl C*) *L S σ*
using *maximal-literal-is-unique-lemma* [*of t s* (*cl-ecl C*) *S σ L*] **by** *blast*
from ‹*all-trms-irreducible* (*subst-set* (*trms-ecl C*) *σ*) (*λt. trm-rep t S*)›
have *trms-irreducible C σ S* (*subst t σ*)
using *trms-irreducible-lemma* **by** *blast*

have (*subst t σ*) ∉ *subst-set* (*trms-ecl C*) *σ*
proof
assume (*subst t σ*) ∈ *subst-set* (*trms-ecl C*) *σ*
from *this* **obtain** *t′* **where** *t′* ∈ *trms-ecl C* **and** (*subst t′ σ*) = (*subst t σ*) **by** *auto*
from ‹*t′* ∈ *trms-ecl C*› **and** *assms*(*3*) **and** ‹*C* ∈ *S*› **have** *dom-trm t′* (*cl-ecl C*)
unfolding *Ball-def well-constrained-def* **by** *auto*
from *this* **obtain** *M u v q* **where** *M* ∈ (*cl-ecl C*) *decompose-literal M u v q* **and**
q = *neg* ∧ (*u* = *t′*) ∨ ((*t′*,*u*) ∈ *trm-ord*) **unfolding** *dom-trm-def* **by** *blast*
obtain *u′ v′ q′* **where** *orient-lit-inst M u′ v′ q′ σ* **using** *literal.exhaust equation.exhaust*
using *trm-ord-irrefl trm-ord-trans*
unfolding *orient-lit-inst-def irrefl-def trans-def* **by** *metis*
from ‹*decompose-literal M u v q*› **and** ‹*orient-lit-inst M u′ v′ q′ σ*›
have *u* = *u′* ∨ *u* = *v′*
unfolding *decompose-literal-def orient-lit-inst-def*
by (*metis atom.simps*(*2*) *decompose-equation-def equation.inject literal.distinct*(*1*)
literal.inject(*1*))
from ‹*decompose-literal M u v q*› **and** ‹*orient-lit-inst M u′ v′ q′ σ*›
have *q* = *q′*
unfolding *decompose-literal-def orient-lit-inst-def* **by** *auto*
from ‹*vars-of-cl* (*subst-cl* (*cl-ecl C*) *σ*) = {}› **and** ‹*M* ∈ (*cl-ecl C*)›

have *vars-of-lit* (*subst-lit* M σ) = {} **by** *auto*
from ‹*orient-lit-inst* M u' v' q' σ› **have**
orient-lit (*subst-lit* M σ) (*subst* u' σ) (*subst* v' σ) q'
using *lift-orient-lit* **by** *auto*
from ‹*orient-lit-inst* L t s p σ› **have**
orient-lit (*subst-lit* L σ) (*subst* t σ) (*subst* s σ) p
using *lift-orient-lit* **by** *auto*
have (t',u) $\notin$ *trm-ord*
proof
assume (t',u) $\in$ *trm-ord*
then have ((*subst* t' σ),(*subst* u σ)) $\in$ *trm-ord*
using *trm-ord-subst* **by** *auto*
from *this* **and** ‹(*subst* t' σ) = (*subst* t σ)› **have**
((*subst* t σ),(*subst* u σ)) $\in$ *trm-ord* **by** *auto*
from ‹*orient-lit* (*subst-lit* M σ) (*subst* u' σ) (*subst* v' σ) q'›
and ‹*orient-lit* (*subst-lit* L σ) (*subst* t σ) (*subst* s σ) p›
and ‹((*subst* t σ),(*subst* u σ)) $\in$ *trm-ord*›
and ‹*vars-of-lit* (*subst-lit* M σ) = {}›
and ‹*vars-of-lit* (*subst-lit* L σ) = {}›
and ‹$u = u' \vee u = v'$›
have ((*subst-lit* L σ),(*subst-lit* M σ)) $\in$ *lit-ord*
using *lit-ord-dominating-term* **by** *metis*
from *this* **and** ‹*maximal-literal* (*subst-lit* L σ) (*subst-cl* (*cl-ecl* C) σ)›
and ‹$M \in$ (*cl-ecl* C)› **show** *False* **using** *maximal-literal-def* **by** *auto*
qed
have $\neg$ (q = *neg* $\wedge$ ($u = t'$))
proof
assume q = *neg* $\wedge$ ($u = t'$)
then have q = *neg* **and** $u = t'$ **by** *auto*
from ‹*orient-lit* (*subst-lit* M σ) (*subst* u' σ) (*subst* v' σ) q'›
and ‹*orient-lit* (*subst-lit* L σ) (*subst* t σ) (*subst* s σ) p›
and ‹$u = t'$›
and ‹(*subst* t' σ) = (*subst* t σ)›
and ‹q = *neg*› **and** ‹$q = q'$›
and ‹p = *pos*›
and ‹*vars-of-lit* (*subst-lit* M σ) = {}›
and ‹*vars-of-lit* (*subst-lit* L σ) = {}›
and ‹$u = u' \vee u = v'$›
have ((*subst-lit* L σ),(*subst-lit* M σ)) $\in$ *lit-ord*
using *lit-ord-neg-lit-lhs lit-ord-neg-lit-rhs* **by** *metis*
from *this* **and** ‹*maximal-literal* (*subst-lit* L σ) (*subst-cl* (*cl-ecl* C) σ)›
and ‹$M \in$ (*cl-ecl* C)› **show** *False* **using** *maximal-literal-def* **by** *auto*
qed
from *this* **and** ‹(t',u) $\notin$ *trm-ord*› **and** ‹q = *neg* $\wedge$ ($u = t'$) $\vee$ ((t',u) $\in$ *trm-ord*)›
show *False* **by** *auto*
qed

from ‹$C \in S$› ‹(*subst* s σ, *subst* t σ) $\in$ *trm-ord*›

and ‹p=pos› ‹*orient-lit-inst* L t s p σ› **and** ‹*sel* (*cl-ecl* C) = {}›
and ‹L ∈ *cl-ecl* C›
and ‹*maximal-literal* (*subst-lit* L σ) (*subst-cl* (*cl-ecl* C) σ)›
and ‹*ground-clause* (*subst-cl* (*cl-ecl* C) σ)›
and ‹*finite* (*cl-ecl* C)›
and ‹*smaller-lits-are-false* (*subst* t σ) (*subst-cl* (*cl-ecl* C) σ) S›
and ‹*maximal-literal-is-unique* (*subst* t σ) (*subst* s σ) (*cl-ecl* C) L S σ›
and ‹*trms-irreducible* C σ S (*subst* t σ)›
and ‹(*subst* t σ) ∉ *subst-set* (*trms-ecl* C) σ›
have *cv*: (*candidate-values* (*trm-rep* (*subst* s σ) S) C (*cl-ecl* C)
(*subst-cl* (*cl-ecl* C) σ) (*subst* s σ) (*subst-lit* L σ) L σ t s (*subst* t σ) S)
unfolding *candidate-values-def* **by** *blast*
from *cv* **have** (*trm-rep* (*subst* s σ) S,(*subst* s σ)) ∈ *set-of-candidate-values* S (*subst* t σ)
unfolding *set-of-candidate-values-def* **by** *blast*

from ‹*trm-rep* (*subst* t σ) S = (*subst* t σ)›
have ¬(*subterm-reduction-applicable* S (*subst* t σ))
using *trm-rep-is-lower-subt-red trm-ord-irrefl irrefl-def*
by *metis*

from ‹(*trm-rep* (*subst* s σ) S, *subst* s σ)
∈ *set-of-candidate-values* S (*subst* t σ)›
have *set-of-candidate-values* S (*subst* t σ) ≠ {} **by** *blast*

from ‹(*trm-rep* (*subst* s σ) S, *subst* s σ)
∈ *set-of-candidate-values* S (*subst* t σ)›
have *min-trms* (*set-of-candidate-values* S (*subst* t σ)) ≠ {}
using *min-trms-not-empty* **by** *blast*

from ‹¬(*subterm-reduction-applicable* S (*subst* t σ))›
‹*min-trms* (*set-of-candidate-values* S (*subst* t σ)) ≠ {}›
have (*trm-rep* (*subst* t σ) S,(*subst* t σ)) ∈ *trm-ord*
using *trm-rep-is-lower-root-red* [*of* S *subst* t σ] **by** *blast*
from *this* **and** ‹(*trm-rep* (*subst* t σ) S) = (*subst* t σ)›
show *False* **using** *trm-ord-irrefl irrefl-def* **by** *metis*
qed
qed

As an immediate consequence of the previous lemma, we show that the set of clauses that are derivable from an unsatisfiable clause set must contain an empty clause (since this set is trivially saturated).

lemma *COMPLETENESS*:
assumes ∀ x. (x ∈ S ⟶ (*trms-ecl* x = {}))
assumes (∀ x∈S. *finite* (*cl-ecl* x))
assumes ¬ (*satisfiable-clause-set* (*cl-ecl* ‘ S))
shows ∃ x. (*derivable-ecl* x S) ∧ *cl-ecl* x = {}
proof (*rule ccontr*)
assume ¬ (∃ x. (*derivable-ecl* x S) ∧ *cl-ecl* x = {})

let *?S* = { *y*. (*derivable-ecl y S*) }
let *?I* = *same-values* (λx. (*trm-rep x ?S*))
have *fo-interpretation ?I* **using** *trm-rep-compatible-with-structure same-values-fo-int*

by *metis*

have $\forall x \in$ *?S*. (*cl-ecl x*) $\neq$ {}
proof (*rule ccontr*)
assume ¬ *?thesis*
then obtain *x* **where** *x* ∈ *?S* **and** *cl-ecl x* = {} **by** *blast*
from ‹*x* ∈ *?S*› **have** *derivable-ecl x S* **by** (*meson CollectD*)
from *this* ‹*cl-ecl x* = {}› ‹¬ (∃ *x*. (*derivable-ecl x S*) ∧ *cl-ecl x* = {})›
show *False* **by** *metis*
qed
have *all-finite*: ∀ *x* ∈ *?S*. (*finite* (*cl-ecl x*))
proof (*rule ccontr*)
assume ¬ *?thesis*
then obtain *x* **where** *x* ∈ *?S* **and** ¬ *finite* (*cl-ecl x*) **by** *blast*
from ‹*x* ∈ *?S*› **have** *derivable-ecl x S* **by** (*meson CollectD*)
from *this assms*(*2*) ‹¬ *finite* (*cl-ecl x*)› **show** *False* **using** *all-derived-clauses-are-finite*
by *metis*
qed
have *Ball S well-constrained*
proof
fix *x* **assume** *x* ∈ *S*
from *this assms*(*1*) **have** *trms-ecl x* = {} **by** *auto*
from *this* **show** *well-constrained x* **unfolding** *well-constrained-def* **by** *blast*
qed
have *Ball ?S well-constrained*
proof
fix *x* **assume** *x* ∈ *?S*
from *this* **have** *derivable-ecl x S* **by** (*meson CollectD*)
from *this assms*(*2*) ‹*Ball S well-constrained*› **show** *well-constrained x*
using *all-derived-clauses-are-wellconstrained*
by *metis*
qed
have *closed-under-renaming ?S*
proof (*rule ccontr*)
assume ¬ *?thesis*
then obtain *C D* **where** *C* ∈ *?S renaming-cl C D D* ∉ *?S*
unfolding *closed-under-renaming-def* **by** *metis*
from ‹*C* ∈ *?S*› **have** *derivable-ecl C S* **by** (*meson CollectD*)
from ‹*derivable-ecl C S*› ‹*renaming-cl C D*› **have** (*derivable-ecl D S*)
using *derivable-ecl.intros*(*2*) **by** *metis*
from *this* **and** ‹*D* ∉ *?S*› **show** *False* **by** *blast*
qed
have *inference-closed ?S*
proof (*rule ccontr*)
assume ¬ *?thesis*

then obtain $D\ P\ \vartheta\ C'$ **where** (*derivable* $D\ P$ *?S* ϑ *FirstOrder* C') $D \notin$ *?S*
unfolding *inference-closed-def* **by** *metis*
from ‹*derivable* $D\ P$ *?S* ϑ *FirstOrder* C'› **have** $P \subseteq$ *?S* **using** *derivable-premisses*
by *metis*
have $\forall x.\ x \in P \longrightarrow$ *derivable-ecl* $x\ S$
proof ((*rule allI*),(*rule impI*))
fix x **assume** $x \in P$
from *this* **and** ‹$P \subseteq$ *?S*› **have** $x \in$ *?S* **by** (*meson rev-subsetD*)
from *this* **show** *derivable-ecl* $x\ S$ **by** (*meson CollectD*)
qed
from *this* **and** ‹(*derivable* $D\ P$ *?S* ϑ *FirstOrder* C')› **have** *derivable-ecl* $D\ S$
using *derivable-ecl.intros*(*3*) [*of* $P\ S\ D$ *?S* $\vartheta\ C'$] **by** *meson*
from *this* **and** ‹$D \notin$ *?S*› **show** *False* **by** *blast*
qed
from *this all-finite* **have** *clause-saturated ?S*
using *inference-closed-sets-are-saturated* **by** *meson*
from *this all-finite* **have** *inference-saturated ?S*
using *clause-saturated-and-inference-saturated* **by** *meson*
from *this* **have** *ground-inference-saturated ?S*
using *lift-inference* **by** *metis*
have *validate-clause-set ?I* (*cl-ecl* ‘ S)
proof (*rule ccontr*)
assume $\neg$ *?thesis*
from *this* **obtain** *Cl-C* **where** *clc*: *Cl-C* $\in$ (*cl-ecl* ‘ S) **and** $\neg$ (*validate-clause*
?I Cl-C)
using *validate-clause-set.simps* **by** *metis*
from *clc* **obtain** C **where** $C \in S$ **and** *Cl-C* $=$ (*cl-ecl* C) **by** *blast*
from ‹$C \in S$› **have** *derivable-ecl* $C\ S$
using *derivable-ecl.intros*(*1*) **by** *metis*
from *this* **have** $C \in$ *?S* **by** *blast*
from ‹$\neg$ (*validate-clause ?I Cl-C*)› **obtain** σ
where $\neg$ (*validate-ground-clause ?I* (*subst-cl Cl-C* σ))
and *ground-clause* (*subst-cl Cl-C* σ)
using *validate-clause.simps* **by** *metis*
let *?pair* $= (C,\sigma)$
have *fst ?pair* $= C$ **by** *auto*
have *snd ?pair* $= \sigma$ **by** *auto*

from ‹$C \in S$› *assms*(*1*) **have** *trms-ecl* $C = \{\}$ **by** *auto*
then have (*subst-set* (*trms-ecl* C) σ) $= \{\}$ **by** *auto*
then have n: *all-trms-irreducible* (*subst-set* (*trms-ecl* C) σ)
($\lambda t.$ *trm-rep* t $\{y.$ *derivable-ecl* $y\ S\}$)
unfolding *all-trms-irreducible-def* **by** *blast*

from ‹*ground-inference-saturated ?S*› *all-finite* ‹*Ball ?S well-constrained*›
‹*closed-under-renaming ?S*› ‹$\forall x \in$ *?S*. (*cl-ecl* x) $\neq \{\}$›
have $\forall C\ \sigma.$ *fst ?pair* $= C \longrightarrow$
$\sigma =$ *snd ?pair* $\longrightarrow$
$C \in \{y.$ *derivable-ecl* $y\ S\} \longrightarrow$

ground-clause (*subst-cl* (*cl-ecl* *C*) σ) ⟶
all-trms-irreducible (*subst-set* (*trms-ecl* *C*) σ) (λ*t*. *trm-rep* *t* {*y*. *derivable-ecl* *y* *S*})
⟶ *validate-ground-clause* *?I* (*subst-cl* (*cl-ecl* *C*) σ)
using *int-clset-is-a-model* [*of* *?S* *?pair*] **by** *blast*
from *this* ‹*fst* *?pair* = *C*› ‹*C* ∈ *?S*› ‹*snd* *?pair* = σ› ‹*Cl-C* = (*cl-ecl* *C*)›
‹*ground-clause* (*subst-cl* *Cl-C* σ)› *n*
have *validate-ground-clause* *?I* (*subst-cl* (*cl-ecl* *C*) σ) **by** *metis*
from *this* **and** ‹¬ (*validate-ground-clause* *?I* (*subst-cl* *Cl-C* σ))› ‹*Cl-C* = (*cl-ecl* *C*)›
show *False* **by** *metis*
qed
from *this* **and** *assms*(*3*) ‹*fo-interpretation* *?I*› **show** *False* **using** *satisfiable-clause-set-def* **by** *metis*
qed

end

end

References

[1] L. Bachmair and H. Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, 3(4):217–247, 1994.

[2] L. Bachmair, H. Ganzinger, C. Lynch, and W. Snyder. Basic paramodulation and superposition. In D. Kapur, editor, *Automated Deduction - CADE-11, 11th International Conference on Automated Deduction, Saratoga Springs, NY, USA, June 15-18, 1992, Proceedings*, volume 607 of *Lecture Notes in Computer Science*, pages 462–476. Springer, 1992.

[3] L. Bachmair, H. Ganzinger, C. Lynch, and W. Snyder. Basic paramodulation. *Inf. Comput.*, 121(2):172–192, 1995.

[4] R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 371–443. Elsevier and MIT Press, 2001.