A Variant of the Superposition Calculus

Nicolas Peltier
CNRS/University of Grenoble (LIG)

May 26, 2024

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* $\subset\#$ (*mset-set E2*)
⟨*proof*⟩

lemma *mset-ordering-addition*:
assumes $A = B + C$
shows $B \subseteq\# A$
⟨*proof*⟩

lemma *equal-image-mset*:
assumes $\forall x \in E.\ (f\ x) = (g\ x)$
shows $\{\#\ (f\ x).\ x \in\#\ (\textit{mset-set}\ E)\ \#\} = \{\#\ (g\ x).\ x \in\#\ (\textit{mset-set}\ E)\ \ \#\}$
⟨*proof*⟩

lemma *multiset-order-inclusion*:
assumes *E* $\subset\#$ *F*
assumes *trans r*
shows (*E*,*F*) $\in$ (*mult r*)
⟨*proof*⟩

lemma *multiset-order-inclusion-eq*:
assumes *E* $\subseteq\#$ *F*
assumes *trans r*
shows *E* = *F* $\vee$ (*E*,*F*) $\in$ (*mult r*)
⟨*proof*⟩

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*⟩

lemma *image-mset-ordering-eq*:
assumes *M1* = {# (*f1 u*). *u* ∈# *L* #}
assumes *M2* = {# (*f2 u*). *u* ∈# *L* #}
assumes ∀ *u*. (*u* ∈# *L* ⟶ (((*f1 u*), (*f2 u*)) ∈ *r* ∨ (*f1 u*) = (*f2 u*)))
shows (*M1* = *M2*) ∨ ((*M1*,*M2*) ∈ (*mult r*))
⟨*proof*⟩

lemma *mult1-def-lemma* :
assumes *M* = *M0* + {#*a*#} ∧ *N* = *M0* + *K* ∧ (∀ *b*. *b* ∈# *K* ⟶ (*b*, *a*) ∈ *r*)
shows (*N*,*M*) ∈ (*mult1 r*)
⟨*proof*⟩

lemma *mset-ordering-add1*:
assumes (*E1*,*E2*) ∈ (*mult r*)
shows (*E1*,*E2* + {# *a* #}) ∈ (*mult r*)
⟨*proof*⟩

lemma *mset-ordering-singleton*:
assumes ∀ *x*. (*x* ∈# *E1* ⟶ (*x*,*a*) ∈ *r*)
shows (*E1*, {# *a* #}) ∈ (*mult r*)
⟨*proof*⟩

lemma *monotonic-fun-mult1*:
assumes ⋀ *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*⟩

lemma *monotonic-fun-mult*:
assumes ⋀ *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*⟩

lemma *mset-set-insert-eq*:
assumes *finite E*
shows *mset-set* (*E* ∪ { *x* }) ⊆# *mset-set E* + {# *x* #}
⟨*proof*⟩

lemma *mset-set-insert*:
assumes *x* ∉ *E*
assumes *finite E*
shows *mset-set* (*E* ∪ { *x* }) = *mset-set E* + {# *x* #}
⟨*proof*⟩

lemma *mset-image-comp*:

shows $\{\#\ (f\ x).\ x \in\#\ \{\#\ (g\ x).\ x \in\#\ E\ \#\}\ \#\} = \{\#\ (f\ (g\ x)).\ x \in\#\ E\ \#\}$
⟨*proof*⟩

lemma *mset-set-mset-image*:
shows $\bigwedge E.\ card\ E = N \Longrightarrow finite\ E \Longrightarrow mset\text{-}set\ (g\ `\ E) \subseteq\#\ \{\#\ (g\ x).\ x \in\#\ mset\text{-}set\ (E)\ \#\}$
⟨*proof*⟩

lemma *split-mset-set*:
assumes $C = C1 \cup C2$
assumes $C1 \cap C2 = \{\}$
assumes *finite C1*
assumes *finite C2*
shows $(mset\text{-}set\ C) = (mset\text{-}set\ C1) + (mset\text{-}set\ C2)$
⟨*proof*⟩

lemma *image-mset-thm*:
assumes $E = \{\#\ (f\ x).\ x \in\#\ E'\ \#\}$
assumes $x \in\#\ E$
shows $\exists\ y.\ ((y \in\#\ E') \wedge x = (f\ y))$
⟨*proof*⟩

lemma *split-image-mset*:
assumes $M = M1 + M2$
shows $\{\#\ (f\ x).\ x \in\#\ M\ \#\} = \{\#\ (f\ x).\ x \in\#\ M1\ \#\} + \{\#\ (f\ x).\ x \in\#\ M2\ \#\}$
⟨*proof*⟩

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*⟩

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*⟩

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
$(is\text{-}a\text{-}variable\ (Var\ x)) = True\ |$
$(is\text{-}a\text{-}variable\ (Const\ x)) = False\ |$
$(is\text{-}a\text{-}variable\ (Comb\ x\ y)) = False$

fun *is-a-constant*
where
$(is\text{-}a\text{-}constant\ (Var\ x)) = False\ |$
$(is\text{-}a\text{-}constant\ (Const\ x)) = True\ |$
$(is\text{-}a\text{-}constant\ (Comb\ x\ y)) = False$

fun *is-compound*
where
$(is\text{-}compound\ (Var\ x)) = False\ |$
$(is\text{-}compound\ (Const\ x)) = False\ |$
$(is\text{-}compound\ (Comb\ x\ y)) = True$

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

lemma *constants-are-not-variables* :
assumes *is-a-constant x*
shows ¬ (*is-a-variable x*)
⟨*proof*⟩

lemma *constants-are-ground* :
  assumes *is-a-constant x*
  shows *ground-term x*
⟨*proof*⟩

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* ≠ []) ∧ (*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* ⇒ *'a trm* ⇒ *bool*
  where
    occurs-in t s = (∃ *p*. *subterm s p t*)

definition *position-in* :: *position* ⇒ *'a trm* ⇒ *bool*
  where
    position-in p s = (∃ *t*. *subterm s p t*)

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

fun *proper-subterms-of*
where

proper-subterms-of t = { *s*. ∃ *p*. (*p* ≠ *Nil* ∧ (*subterm t p s*)) }

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

fun *replace-subterm* ::
'a trm ⇒ *position* ⇒ *'a trm* ⇒ *'a trm* ⇒ *bool*
where
(*replace-subterm T* [] *u S*) = (*S* = *u*) |
(*replace-subterm* (*Var x*) (*first* # *next*) *u S*) = *False* |
(*replace-subterm* (*Const c*) (*first* # *next*) *u S*) = *False* |
(*replace-subterm* (*Comb x y*) (*Left* # *next*) *u S*) =
(∃ *S1*. (*replace-subterm x next u S1*) ∧ (*S* = *Comb S1 y*)) |
(*replace-subterm* (*Comb x y*) (*Right* # *next*) *u S*) =
(∃ *S2*. (*replace-subterm y next u S2*) ∧ (*S* = *Comb x S2*))

lemma *replace-subterm-is-a-function*:
shows ⋀*t u v*. *subterm t p u* ⟹ ∃ *s*. *replace-subterm t p v s*
⟨*proof*⟩

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

lemma *root-subterm*:
shows *t* ∈ (*subterms-of t*)
⟨*proof*⟩

lemma *root-position*:
shows *Nil* ∈ (*pos-of t*)
⟨*proof*⟩

lemma *subterms-of-an-atomic-term*:
assumes *is-a-variable t* ∨ *is-a-constant t*
shows *subterms-of t* = { *t* }
⟨*proof*⟩

lemma *positions-of-an-atomic-term*:
assumes *is-a-variable t* ∨ *is-a-constant t*
shows *pos-of t* = { *Nil* }
⟨*proof*⟩

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*⟩

lemma *occur-in-subterm*:
assumes *occurs-in u t*
assumes *occurs-in t s*

shows *occurs-in u s*
⟨*proof*⟩

lemma *vars-of-subterm* :
assumes $x \in$ *vars-of s*
shows $\bigwedge$ *t. subterm t p s* $\Longrightarrow x \in$ *vars-of t*
⟨*proof*⟩

lemma *vars-subterm* :
assumes *subterm t p s*
shows *vars-of s* $\subseteq$ *vars-of t*
⟨*proof*⟩

lemma *vars-subterms-of* :
assumes $s \in$ *subterms-of t*
shows *vars-of s* $\subseteq$ *vars-of t*
⟨*proof*⟩

lemma *subterms-of-a-non-atomic-term*:
shows *subterms-of* (*Comb t1 t2*) = (*subterms-of t1*) $\cup$ (*subterms-of t2*) $\cup$ { (*Comb t1 t2*) }
⟨*proof*⟩

lemma *positions-of-a-non-atomic-term*:
shows *pos-of* (*Comb t1 t2*) = (*left-app* ‘ (*pos-of t1*)) $\cup$ (*right-app* ‘ (*pos-of t2*)) $\cup$ { *Nil* }
⟨*proof*⟩

lemma *set-of-subterms-is-finite* :
shows (*finite* (*subterms-of* (*t* :: *'a trm*)))
⟨*proof*⟩

lemma *set-of-positions-is-finite* :
shows (*finite* (*pos-of* (*t* :: *'a trm*)))
⟨*proof*⟩

lemma *vars-of-instances*:
shows *vars-of* (*subst t* σ)
= $\bigcup$ { *V*. $\exists$ *x*. (*x* $\in$ (*vars-of t*)) $\wedge$ (*V* = *vars-of* (*subst* (*Var x*) σ)) }
⟨*proof*⟩

lemma *subterms-of-instances* :
$\forall$ *u v u' s*. (*u* = (*subst v s*) $\longrightarrow$ (*subterm u p u'*)
$\longrightarrow$ ($\exists$ *x q1 q2*. (*is-a-variable x*) $\wedge$ (*subterm* (*subst x s*) *q1 u'*) $\wedge$
(*subterm v q2 x*) $\wedge$ (*p* = (*append q2 q1*))) $\vee$
(($\exists$ *v'*. (($\neg$ *is-a-variable v'*) $\wedge$ (*subterm v p v'*) $\wedge$ (*u'* = (*subst v' s*)))))) (**is** *?prop p*)
⟨*proof*⟩

lemma *vars-of-replacement*:
shows $\bigwedge t\ s.\ x \in \mathit{vars\text{-}of}\ s \Longrightarrow \mathit{replace\text{-}subterm}\ t\ p\ v\ s \Longrightarrow x \in (\mathit{vars\text{-}of}\ t) \cup (\mathit{vars\text{-}of}\ v)$
⟨*proof*⟩

lemma *vars-of-replacement-set*:
assumes *replace-subterm t p v s*
shows $\mathit{vars\text{-}of}\ s \subseteq (\mathit{vars\text{-}of}\ t) \cup (\mathit{vars\text{-}of}\ v)$
⟨*proof*⟩

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
$(\mathit{subst\text{-}set}\ S\ \sigma) = \{\ u.\ \exists\, t.\ u = (\mathit{subst}\ t\ \sigma) \wedge t \in S\ \}$

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

lemma *subst-codomain-is-finite*:
assumes *finite A*
shows *finite* (*subst-codomain η A*)
⟨*proof*⟩

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 $\forall\, t1\ t2.\ \mathit{Comb}\ t1\ t2 \prec s \longrightarrow (t1 \prec s \wedge t2 \prec s)$
⟨*proof*⟩

lemma *subt-irrefl*:
shows $\neg\ (s \prec s)$
⟨*proof*⟩

lemma *MGU-exists*:
shows $\forall\, \sigma.\ ((\mathit{subst}\ t\ \sigma) = (\mathit{subst}\ s\ \sigma) \longrightarrow \mathit{unify}\ t\ s \neq \mathit{None})$
⟨*proof*⟩

We establish some useful properties of substitutions and instances.

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

lemma *comp-subst-terms*:
assumes $\sigma \doteq \vartheta \Diamond \eta$

shows $(subst\ t\ \sigma) = (subst\ (subst\ t\ \vartheta)\ \eta)$
⟨*proof*⟩

lemma *ground-instance*:
assumes *ground-on* $(vars\text{-}of\ t)\ \sigma$
shows *ground-term* $(subst\ t\ \sigma)$
⟨*proof*⟩

lemma *substs-preserve-groundness*:
assumes *ground-term* t
shows *ground-term* $(subst\ t\ \sigma)$
⟨*proof*⟩

lemma *ground-subst-exists* :
$finite\ V \Longrightarrow \exists \sigma.\ (ground\text{-}on\ V\ \sigma)$
⟨*proof*⟩

lemma *substs-preserve-ground-terms* :
assumes *ground-term* t
shows $subst\ t\ \sigma = t$
⟨*proof*⟩

lemma *substs-preserve-subterms* :
shows $\bigwedge t\ \ s.\ subterm\ t\ p\ s \Longrightarrow subterm\ (subst\ t\ \sigma)\ p\ (subst\ s\ \sigma)$
⟨*proof*⟩

lemma *substs-preserve-occurs-in*:
assumes *occurs-in* $s\ t$
shows *occurs-in* $(subst\ s\ \sigma)\ (subst\ t\ \sigma)$
⟨*proof*⟩

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

lemma *coincide-sym*:
assumes *coincide-on* $\sigma\ \eta\ V$
shows *coincide-on* $\eta\ \sigma\ V$
⟨*proof*⟩

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*⟩

lemma *ground-replacement*:
assumes *replace-subterm* $t\ p\ v\ s$
assumes *ground-term* $(subst\ t\ \sigma)$
assumes *ground-term* $(subst\ v\ \sigma)$
shows *ground-term* $(subst\ s\ \sigma)$
⟨*proof*⟩

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

lemma *combine-substs*:
assumes *finite V1*
assumes $V1 \cap V2 = \{\}$
assumes *ground-on V1 η1*
shows $\exists \sigma.$ (*coincide-on σ η1 V1*) $\wedge$ (*coincide-on σ η2 V2*)
⟨*proof*⟩

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-subst f l*)

lemma *map-subst-lemma*:
shows ((*subst* (*Var x*) *σ*) ≠ (*Var x*) ∨ (*subst* (*Var x*) *σ*) ≠ (*subst* (*Var x*) (*map-subst f σ*)))
⟶ ((*subst* (*Var x*) (*map-subst f σ*)) = (*f* (*subst* (*Var x*) *σ*)))
⟨*proof*⟩

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*
⟨*proof*⟩

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* ⇒ *'a trm* ⇒ *bool*

definition *reflexive* :: *'a binary-relation-on-trms* ⇒ *bool*
where
(*reflexive x*) = (∀ *y*. (*x y y*))

definition *symmetric* :: *'a binary-relation-on-trms* ⇒ *bool*
where
(*symmetric x*) = (∀ *y*. ∀ *z*. ((*x y z*) = (*x z y*)))

definition *transitive* :: *'a binary-relation-on-trms* ⇒ *bool*
where
(*transitive x*) = (∀ *y*. ∀ *z*. ∀ *u*. (*x y z*) ⟶ (*x z u*) ⟶ (*x y u*))

definition *equivalence-relation* :: *'a binary-relation-on-trms* ⇒ *bool*
where
(*equivalence-relation x*) = ((*reflexive x*) ∧ (*symmetric x*) ∧ (*transitive x*))

definition *compatible-with-structure* :: (*'a binary-relation-on-trms*) ⇒ *bool*
where
(*compatible-with-structure x*) = (∀ *t1 t2 s1 s2*.
(*x t1 s1*) ⟶ (*x t2 s2*) ⟶ (*x* (*Comb t1 t2*) (*Comb s1 s2*)))

definition *congruence* :: *'a binary-relation-on-trms* ⇒ *bool*
where
(*congruence x*) = ((*equivalence-relation x*) ∧ (*compatible-with-structure x*))

lemma *replacement-preserves-congruences* :
shows ⋀ *t s*. (*congruence I*) ⟹ (*I* (*subst u σ*) (*subst v σ*))
⟹ *subterm t p u* ⟹ *replace-subterm t p v s*
⟹ (*I* (*subst t σ*) (*subst s σ*))
⟨*proof*⟩

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*⟩

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 = (∀ *x* ∈ *V*. (*is-a-variable* (*subst* (*Var x*) *σ*))
∧ (∀ *x y*. ((*x* ∈ *V*) ⟶ (*y* ∈ *V*) ⟶ *x* ≠ *y* ⟶ (*subst* (*Var x*) *σ*) ≠ (*subst* (*Var y*) *σ*))))

lemma *renamings-admit-inverse*:
shows *finite V* ⟹ *renaming σ V* ⟹ ∃ *ϑ*. (∀ *x* ∈ *V*. (*subst* (*subst* (*Var x*) *σ*) *ϑ*) = (*Var x*))
∧ (∀ *x*. (*x* ∉ (*subst-codomain σ V*) ⟶ (*subst* (*Var x*) *ϑ*) = (*Var x*)))
∧ (∀ *x*. *is-a-variable* (*subst* (*Var x*) *ϑ*))
⟨*proof*⟩

lemma *renaming-exists*:
assumes ¬ *finite* (*Vars* :: (*'a set*))

shows *finite* $V \Longrightarrow (\forall V'::'a\ set.\ finite\ V' \longrightarrow (\exists \eta.\ ((renaming\ \eta\ V) \wedge ((subst\text{-}codomain\ \eta\ V) \cap V') = \{\})))$
⟨*proof*⟩

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
    (*positive-literal* (*Pos x*)) = *True* |
    (*positive-literal* (*Neg x*)) = *False*

fun *negative-literal* :: *'a literal* ⇒ *bool*
  where
    (*negative-literal* (*Pos x*)) = *False* |
    (*negative-literal* (*Neg x*)) = *True*

fun *mk-lit* :: *sign* ⇒ *'a equation* ⇒ *'a literal*
  where
    (*mk-lit pos x*) = (*Pos x*) |
    (*mk-lit neg x*) = (*Neg x*)

definition *decompose-equation*
  where *decompose-equation e t s* = (*e* = (*Eq t s*) ∨ (*e* = (*Eq s t*)))

definition *decompose-literal*
  where *decompose-literal L t s p* =
        (∃ *e*. ((*p* = *pos* ∧ (*L* = (*Pos e*)) ∧ *decompose-equation e t s*)
        ∨ (*p* = *neg* ∧ (*L* = (*Neg e*)) ∧ *decompose-equation e t s*)))

fun *subterms-of-eq*
  where *subterms-of-eq* (*Eq t s*) = (*subterms-of t* ∪ *subterms-of s*)

fun *vars-of-eq*
  where *vars-of-eq* (*Eq t s*) = (*vars-of t* ∪ *vars-of s*)

lemma *decompose-equation-vars*:
  assumes *decompose-equation e t s*
  shows *vars-of t* ⊆ *vars-of-eq e*
⟨*proof*⟩

fun *subterms-of-lit*
  where
    subterms-of-lit (*Pos e*) = (*subterms-of-eq e*) |
    subterms-of-lit (*Neg e*) = (*subterms-of-eq e*)

fun *vars-of-lit*
  where
    vars-of-lit (*Pos e*) = (*vars-of-eq e*) |
    vars-of-lit (*Neg e*) = (*vars-of-eq e*)

fun *vars-of-cl*
  where *vars-of-cl C* = { *x*. ∃ *L*. *x* ∈ (*vars-of-lit L*) ∧ *L* ∈ *C* }

fun *subterms-of-cl*
  where *subterms-of-cl C* = { *x*. ∃ *L*. *x* ∈ (*subterms-of-lit L*) ∧ *L* ∈ *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*
⟨*proof*⟩

lemma *vars-of-cl-lem*:
assumes *L* ∈ *C*
shows *vars-of-lit L* ⊆ *vars-of-cl C*
⟨*proof*⟩

lemma *set-of-variables-is-finite-eq*:
shows *finite* (*vars-of-eq e*)
⟨*proof*⟩

lemma *set-of-variables-is-finite-lit*:
shows *finite* (*vars-of-lit l*)
⟨*proof*⟩

lemma *set-of-variables-is-finite-cl*:
assumes *finite C*
shows *finite* (*vars-of-cl C*)
⟨*proof*⟩

lemma *subterm-lit-vars* :
assumes $u \in$ *subterms-of-lit* L
shows *vars-of* $u \subseteq$ *vars-of-lit* L
⟨*proof*⟩

lemma *subterm-vars* :
assumes $u \in$ *subterms-of-cl* C
shows *vars-of* $u \subseteq$ *vars-of-cl* C
⟨*proof*⟩

We establish some basic properties of substitutions.

lemma *subterm-cl-subst*:
assumes $x \in$ (*subterms-of-cl* C)
shows (*subst* x σ) $\in$ (*subterms-of-cl* (*subst-cl* C σ))
⟨*proof*⟩

lemma *ground-substs-yield-ground-clause*:
assumes *ground-on* (*vars-of-cl* C) σ
shows *ground-clause* (*subst-cl* C σ)
⟨*proof*⟩

lemma *ground-clauses-and-ground-substs*:
assumes *ground-clause* (*subst-cl* C σ)
shows *ground-on* (*vars-of-cl* C) σ
⟨*proof*⟩

lemma *ground-instance-exists*:
assumes *finite* C
shows $\exists\sigma$. (*ground-clause* (*subst-cl* C σ))
⟨*proof*⟩

lemma *composition-of-substs* :
shows (*subst* (*subst* t σ) η)
= (*subst* t (*comp* σ η))
⟨*proof*⟩

lemma *composition-of-substs-eq* :
shows (*subst-equation* (*subst-equation* e σ) η)
= (*subst-equation* e (*comp* σ η))
⟨*proof*⟩

lemma *composition-of-substs-lit* :
shows (*subst-lit* (*subst-lit* l σ) η)
= (*subst-lit* l (*comp* σ η))
⟨*proof*⟩

lemma *composition-of-substs-cl* :
shows (*subst-cl* (*subst-cl* C σ) η)
= (*subst-cl* C (*comp* σ η))

⟨*proof*⟩

lemma *substs-preserve-ground-lit* :
assumes *ground-clause C*
assumes $y \in C$
shows *subst-lit y* $\sigma = y$
⟨*proof*⟩

lemma *substs-preserve-ground-clause* :
assumes *ground-clause C*
shows *subst-cl C* $\sigma = C$
⟨*proof*⟩

lemma *substs-preserve-finiteness* :
assumes *finite C*
shows *finite* (*subst-cl C* σ)
⟨*proof*⟩

We prove that two equal substitutions yield the same objects.

lemma *subst-eq-eq* :
assumes *subst-eq* σ η
shows *subst-equation e* σ = *subst-equation e* η
⟨*proof*⟩

lemma *subst-eq-lit* :
assumes *subst-eq* σ η
shows *subst-lit l* σ = *subst-lit l* η
⟨*proof*⟩

lemma *subst-eq-cl*:
assumes *subst-eq* σ η
shows *subst-cl C* σ = *subst-cl C* η
⟨*proof*⟩

lemma *coincide-on-eq* :
assumes *coincide-on* σ η (*vars-of-eq e*)
shows *subst-equation e* σ = *subst-equation e* η
⟨*proof*⟩

lemma *coincide-on-lit* :
assumes *coincide-on* σ η (*vars-of-lit l*)
shows *subst-lit l* σ = *subst-lit l* η
⟨*proof*⟩

lemma *coincide-on-cl* :
assumes *coincide-on* σ η (*vars-of-cl C*)
shows *subst-cl C* σ = *subst-cl C* η
⟨*proof*⟩

3.2 Semantics

Interpretations are congruences on the set of terms.

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

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

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

fun *validate-ground-clause* :: *'a binary-relation-on-trms* ⇒ *'a clause* ⇒ *bool*
where
validate-ground-clause I C = (∃ *L*.(*L* ∈ *C*) ∧ (*validate-ground-lit I L*))

fun *validate-clause* :: *'a binary-relation-on-trms* ⇒ *'a clause* ⇒ *bool*
where
validate-clause I C = (∀ *s*. (*ground-clause* (*subst-cl C s*))
⟶ (*validate-ground-clause I* (*subst-cl C s*)))

fun *validate-clause-set* :: *'a binary-relation-on-trms* ⇒ *'a clause set* ⇒ *bool*
where
validate-clause-set I S = (∀ *C*. (*C* ∈ *S* ⟶ (*validate-clause I C*)))

definition *clause-entails-clause* :: *'a clause* ⇒ *'a clause* ⇒ *bool*
where
clause-entails-clause C D = (∀ *I*. (*fo-interpretation I* ⟶ *validate-clause I C* ⟶
validate-clause I D))

definition *set-entails-clause* :: *'a clause set* ⇒ *'a clause* ⇒ *bool*
where
set-entails-clause S C = (∀ *I*. (*fo-interpretation I* ⟶ *validate-clause-set I S* ⟶
validate-clause I C))

definition *satisfiable-clause-set* :: *'a clause set* ⇒ *bool*
where
(*satisfiable-clause-set S*) = (∃ *I*. (*fo-interpretation I*) ∧ (*validate-clause-set I S*))

We state basic properties of the entailment relation.

lemma *set-entails-clause-member*:
assumes *C* ∈ *S*
shows *set-entails-clause S C*
⟨*proof*⟩

lemma *instances-are-entailed* :
assumes *validate-clause I C*
shows *validate-clause I* (*subst-cl C σ*)
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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
(*subst-ecl* (*Ecl C S*) σ) =
(*Ecl* (*subst-cl C* σ) { *s*. ($\exists$ *t*. (*s* = (*subst t* σ) $\wedge$ *t* $\in$ *S*)) })

fun *cl-ecl*
where
(*cl-ecl* (*Ecl C X*)) = *C*

fun *trms-ecl*
where
(*trms-ecl* (*Ecl C X*)) = *X*

definition *renaming-cl*
where *renaming-cl C D* = ($\exists$ η. (*renaming* η (*vars-of-cl* (*cl-ecl C*))) $\wedge$ *D* = (*subst-ecl C* η))

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

definition *variable-disjoint*
where (*variable-disjoint C D*) = ((*vars-of-cl* (*cl-ecl C*)) $\cap$ (*vars-of-cl* (*cl-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-irefl* : *irefl 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-irefl* : (∀ *x y*. (*irefl* (*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*) =

$(((\ (L = (Pos\ (Eq\ u\ v)))) \vee (L = (Pos\ (Eq\ v\ u))))$
$\quad \wedge (((subst\ u\ \sigma),(subst\ v\ \sigma)) \notin trm\text{-}ord) \wedge (s = pos))$
$\vee$
$((\ (L = (Neg\ (Eq\ u\ v)))) \vee (L = (Neg\ (Eq\ v\ u)))) \wedge (((subst\ u\ \sigma),(subst\ v\ \sigma))$
$\notin trm\text{-}ord) \wedge (s = neg)))$

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*
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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,σ) = {# (mset-lit (subst-lit x σ)). x ∈# (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)
⟨*proof*⟩

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*⟩

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*⟩

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

definition *lit-ord* :: ('*a literal* × '*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*)
⟨*proof*⟩

lemma *lit-ord-trans*:
shows *trans* *lit-ord*
⟨*proof*⟩

lemma *lit-ord-wf*:
shows *wf lit-ord*
⟨*proof*⟩

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*, σ_C), (*D*, σ_D)) ∈ *ecl-ord* ⟷ ((*cl-ecl C*, σ_C), (*cl-ecl D*, σ_D)) ∈ *cl-ord*
⟨*proof*⟩

lemma *mult-mult-trm-ord-trans*:
shows *trans* (*mult* (*mult trm-ord*))
⟨*proof*⟩

lemma *ecl-ord-trans*:
shows *trans ecl-ord*
⟨*proof*⟩

lemma *cl-ord-trans*:
shows *trans cl-ord*
⟨*proof*⟩

lemma *cl-ord-eq-trans*:
shows *trans cl-ord-eq*
⟨*proof*⟩

lemma *wf-ecl-ord*:
shows *wf ecl-ord*
⟨*proof*⟩

definition *maximal-literal* :: *'a literal* ⇒ *'a clause* ⇒ *bool*
where
(*maximal-literal L C*) = (∀ *x*. (*x* ∈ *C* ⟶ (*L*,*x*) ∉ *lit-ord*))

definition *eligible-literal*
where
(*eligible-literal L C* σ) = (*L* ∈ *sel* (*cl-ecl C*) ∨
(*sel*(*cl-ecl C*) = {}
∧ (*maximal-literal* (*subst-lit L* σ) (*subst-cl* (*cl-ecl C*) σ))))

definition *strictly-maximal-literal*
where *strictly-maximal-literal C L* σ =
(∀ *x* ∈ (*cl-ecl C*) − { *L* }. ((*subst-lit x* σ),(*subst-lit L* σ))
∈ *lit-ord*)

definition *lower-or-eq*
where *lower-or-eq t s* = ((*t* = *s*) ∨ ((*t*,*s*) ∈ *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*⟩

The next definition extends the ordering to substitutions.

definition *lower-on*
where *lower-on* σ η $V = (\forall x \in V.$
(*lower-or-eq* (*subst* (*Var* x) σ) ((*subst* (*Var* x) η))))

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-or-eq* (*Comb t1 t2*) (*Comb s1 s2*)
⟨*proof*⟩

lemma *lower-on-term*:
shows $\bigwedge \sigma\ \eta$. *lower-on* σ η (*vars-of* t) $\Longrightarrow$
(*lower-or-eq* (*subst* t σ) (*subst* t η))
⟨*proof*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

lemma *subterm-trm-ord* :
shows $\bigwedge$ t s.
subterm t p s $\Longrightarrow$ $p \neq []$
$\Longrightarrow$ (s,t) $\in$ *trm-ord*
⟨*proof*⟩

lemma *subterm-trm-ord-eq* :
assumes *subterm* t p s
shows $s = t \vee$ (s,t) $\in$ *trm-ord*
⟨*proof*⟩

lemma *subterms-of-trm-ord-eq* :
assumes $s \in$ *subterms-of* t
shows $s = t \vee$ (s,t) $\in$ *trm-ord*
⟨*proof*⟩

lemma *subt-trm-ord*:
shows $t \prec s \longrightarrow$ (t,s) $\in$ *trm-ord*
⟨*proof*⟩

lemma *trm-ord-vars*:
assumes (t,s) $\in$ *trm-ord*
shows *vars-of* $t \subseteq$ *vars-of* s
⟨*proof*⟩

lemma *lower-on-ground*:
assumes *lower-on* σ η V
assumes *ground-on* V η
shows *ground-on* V σ
⟨*proof*⟩

lemma *replacement-monotonic* :
shows $\bigwedge$ t s. ((*subst* v σ), (*subst* u σ)) $\in$ *trm-ord*
$\Longrightarrow$ *subterm* t p u $\Longrightarrow$ *replace-subterm* t p v s
$\Longrightarrow$ ((*subst* s σ), (*subst* t σ)) $\in$ *trm-ord*
⟨*proof*⟩

lemma *mset-lit-subst*:
shows (*mset-lit* (*subst-lit* L σ)) =

$\{\#\ (subst\ x\ \sigma).\ x \in\#\ (mset\text{-}lit\ L)\ \#\}$
⟨*proof*⟩

lemma *lit-ord-irrefl*:
shows $(L,L) \notin lit\text{-}ord$
⟨*proof*⟩

lemma *lit-ord-subst*:
assumes $(L,M) \in lit\text{-}ord$
shows $((subst\text{-}lit\ L\ \sigma), (subst\text{-}lit\ M\ \sigma)) \in lit\text{-}ord$
⟨*proof*⟩

lemma *args-are-strictly-lower*:
assumes *is-compound t*
shows $(lhs\ t,t) \in trm\text{-}ord \wedge (rhs\ t,t) \in trm\text{-}ord$
⟨*proof*⟩

lemma *mset-subst*:
assumes $C' = subst\text{-}cl\ D\ \vartheta$
assumes $\sigma \doteq \vartheta \Diamond \eta$
assumes *finite D*
shows $mset\text{-}cl\ (C',\eta) = mset\text{-}cl\ (D,\sigma) \vee (mset\text{-}cl\ (C',\eta), mset\text{-}cl\ (D,\sigma)) \in (mult\ (mult\ trm\text{-}ord))$
⟨*proof*⟩

lemma *max-lit-exists*:
shows $finite\ C \Longrightarrow C \neq \{\} \longrightarrow ground\text{-}clause\ C \longrightarrow (\exists\, L.\ (L \in C \wedge (maximal\text{-}literal\ L\ C)))$
⟨*proof*⟩

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

lemma *eligible-lit-exists*:
assumes $finite\ (cl\text{-}ecl\ C)$
assumes $(cl\text{-}ecl\ C) \neq \{\}$
assumes $(ground\text{-}clause\ (subst\text{-}cl\ (cl\text{-}ecl\ C)\ \sigma))$
shows $\exists\, L.\ ((eligible\text{-}literal\ L\ C\ \sigma) \wedge (L \in (cl\text{-}ecl\ C)))$
⟨*proof*⟩

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 $(s1,s2) \in trm\text{-}ord \vee (s1,t2) \in trm\text{-}ord$
assumes *orient-lit x1 s1 t1 p1*
assumes *orient-lit x2 s2 t2 p2*
assumes $vars\text{-}of\text{-}lit\ x1 = \{\}$
assumes $vars\text{-}of\text{-}lit\ x2 = \{\}$
shows $(x1,x2) \in lit\text{-}ord$
⟨*proof*⟩

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*⟩

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*⟩

lemma *lit-ord-rhs*:
assumes (*t1*,*t2*) ∈ *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*) ∈ *lit-ord*
⟨*proof*⟩

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*⟩

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*⟩

lemma *subterms-dominated* :
assumes *maximal-literal L C*
assumes *orient-lit L t s p*
assumes *u* ∈ *subterms-of-cl C*
assumes *vars-of-lit L* = {}
assumes *vars-of-cl C* = {}

shows $u = t \vee (u,t) \in$ *trm-ord*
⟨*proof*⟩

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

fun *dominate-eq*:: *'a trm* ⇒ *'a equation* ⇒ *bool*
where (*dominate-eq t* (*Eq u v*)) = ((*t*,*u*) ∉ *trm-ord* ∧ (*t*,*v*) ∉ *trm-ord*)

fun *dominate-lit*:: *'a trm* ⇒ *'a literal* ⇒ *bool*
where (*dominate-lit t* (*Pos e*)) = (*dominate-eq t e*) |
(*dominate-lit t* (*Neg e*)) = (*dominate-eq t e*)

definition *dominate-cl*:: *'a trm* ⇒ *'a clause* ⇒ *bool*
where (*dominate-cl t C*) = (∀ *x* ∈ *C*. (*dominate-lit t x*))

definition *no-disequation-in-cl*:: *'a trm* ⇒ *'a clause* ⇒ *bool*
where (*no-disequation-in-cl t C*) = (∀ *u v*.
(*Neg* (*Eq u v*) ∈ *C* ⟶ (*u* ≠ *t* ∧ *v* ≠ *t*)))

definition *no-taut-eq-in-cl*:: *'a trm* ⇒ *'a clause* ⇒ *bool*
where (*no-taut-eq-in-cl t C*) = (*Pos* (*Eq t t*) ∉ *C*)

definition *eq-occurs-in-cl*
where
(*eq-occurs-in-cl t s C σ*) = (∃ *L t′ s′*. (*L* ∈ *C*) ∧ (*orient-lit-inst L t′ s′ pos σ*)
∧ (*t* = *subst t′ σ*) ∧ (*s* = *subst s′ σ*))

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*)
⟨*proof*⟩

lemma *subst-preserve-ck-unifier*:
assumes *ck-unifier t s σ k*
shows *ck-unifier t s* (*comp σ η*) *Ground*
⟨*proof*⟩

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*⟩

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*
⟨*proof*⟩

lemma *dom-trm-vars*:
assumes *dom-trm t C*
shows *vars-of t* ⊆ *vars-of-cl C*
⟨*proof*⟩

definition *well-constrained*
where *well-constrained C* = (∀ *y*. (*y* ∈ *trms-ecl C* ⟶ *dom-trm y* (*cl-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 (*all-trms-irreducible E f*) = (∀ *x y*. (*x* ∈ *E* ⟶ *occurs-in y x* ⟶ (*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 eclause ⇒ 'a eclause ⇒ 'a eclause ⇒ 'a subst ⇒ inferences ⇒ 'a clause ⇒ bool
where
(superposition P1 P2 C σ k C′) =
(∃ L t s u v M p Cl-P1 Cl-P2 Cl-C polarity t′ u′ L′ trms-C.
(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)
∧ 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)) k)
∧ (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 k)
∧ (replace-subterm t p v t′)
∧ ((k = FirstOrder) ∨ ((subst-lit M σ),(subst-lit L σ)) ∈ lit-ord)
∧ ((k = FirstOrder) ∨ (strictly-maximal-literal P2 M σ))
∧ (L′ = mk-lit polarity (Eq t′ s))
∧ (Cl-C = (subst-cl C′ σ))
∧ (C′ = (Cl-P1 − { L }) ∪ ((Cl-P2 − { M }) ∪ { 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 ⇒ 'a eclause ⇒ 'a subst ⇒ inferences ⇒ 'a clause ⇒ bool
where
(reflexion P C σ k C′) =
(∃ L1 t s Cl-P Cl-C trms-C.
(eligible-literal L1 P σ)
∧ (L1 ∈ (cl-ecl P)) ∧ (Cl-C = (cl-ecl C)) ∧ (Cl-P = (cl-ecl P))
∧ (orient-lit-inst L1 t s neg σ)
∧ (ck-unifier t s σ k)
∧ (C = (Ecl Cl-C trms-C))
∧ trms-C = (get-trms Cl-C
(dom-trms Cl-C (subst-set ((trms-ecl P) ∪ { t }) σ)) k)
∧ (Cl-C = (subst-cl C′ σ))
∧ (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 ⇒ *'a eclause* ⇒ *'a subst* ⇒ *inferences* ⇒ *'a clause* ⇒ *bool*
where
  (*factorization P C σ k C'*) =
    (∃ *L1 L2 L' t s u v Cl-P Cl-C trms-C*.
      (*eligible-literal L1 P σ*)
      ∧ (*L1* ∈ (*cl-ecl P*)) ∧ (*L2* ∈ (*cl-ecl P*) − { *L1* }) ∧ (*Cl-C* = (*cl-ecl C*)) ∧ (*Cl-P* = (*cl-ecl P*))
      ∧ (*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 k*)
      ∧ (*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 P*) ∪ (*proper-subterms-of t*)) *σ*))) *k*)
      ∧ (*Cl-C* = (*subst-cl C' σ*))
      ∧ (*C'* = ((*Cl-P* − { *L2* }) ∪ { *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* ⇒ *'a eclause set*
  ⇒ *'a eclause set* ⇒ *'a subst* ⇒ *inferences* ⇒ *'a clause* ⇒ *bool*
where
  (*derivable C P S σ k C'*) =
      ((∃ *P1 P2*. (*P1* ∈ *S* ∧ *P2* ∈ *S* ∧ *P* = { *P1*, *P2* } ∧ *superposition P1 P2 C σ k C'*))
    ∨ (∃ *P1*. (*P1* ∈ *S* ∧ *P* = { *P1* } ∧ *factorization P1 C σ k C'*))
    ∨ (∃ *P1*. (*P1* ∈ *S* ∧ *P* = { *P1* } ∧ *reflexion P1 C σ k C'*)))

lemma *derivable-premisses*:
  assumes *derivable C P S σ k C'*
  shows *P* ⊆ *S*
⟨*proof*⟩

inductive *derivable-ecl* :: *'a eclause* ⇒ *'a eclause set* ⇒ *bool*
where
init [*simp*, *intro*!]: $C \in S \Longrightarrow$ (*derivable-ecl C S*) |
rn [*simp*, *intro*!]: (*derivable-ecl C S*) ⟹ (*renaming-cl C D*) ⟹ (*derivable-ecl D S*) |
deriv [*simp*, *intro*!]: (∀ *x*. (*x* ∈ *P* ⟶ (*derivable-ecl x S*)))
⟹ (*derivable C P S' σ FirstOrder C'*) ⟹ (*derivable-ecl C S*)

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

definition *instances*:: *'a eclause set* ⇒ (*'a eclause* × *'a subst*) *set*
where *instances S* = { *x*. ∃ *C σ*. (*C* ∈ *S* ∧ (*ground-clause* (*subst-cl* (*cl-ecl C*) *σ*))
∧ *x* = (*C*,*σ*))}

definition *clset-instances*:: (*'a eclause* × *'a subst*) *set* ⇒ *'a clause set*
where
clset-instances S = { *C*. ∃ *x*. (*x* ∈ *S* ∧ *C* = (*subst-cl* (*cl-ecl* (*fst x*)) (*snd x*))) }

definition *grounding-set*
where *grounding-set S σ* = (∀ *x*. (*x* ∈ *S* ⟶ (*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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*⟩

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*) $\wedge$ (*finite C'*)
⟨*proof*⟩

lemma *reflexion-preserves-finiteness*:
assumes *finite* (*cl-ecl P1*)
assumes *reflexion P1 C* σ *k C'*
shows *finite* (*cl-ecl C*) $\wedge$ (*finite C'*)
⟨*proof*⟩

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*⟩

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*⟩

lemma *derivable-clauses-lemma*:
assumes *derivable C P S* σ *k C'*
shows ((*cl-ecl C*) = (*subst-cl C'* σ))
⟨*proof*⟩

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*⟩

lemma *substs-preserve-dom-trm*:
assumes *dom-trm t C*
shows *dom-trm* (*subst t* σ) (*subst-cl C* σ)

⟨*proof*⟩

lemma *substs-preserve-well-constrainedness*:
assumes *well-constrained C*
shows *well-constrained* (*subst-ecl C* σ)
⟨*proof*⟩

lemma *ck-trms-sound*:
assumes *T* = *get-trms D* (*dom-trms C E*) *k*
shows *T* ⊆ (*dom-trms C E*)
⟨*proof*⟩

lemma *derivable-clauses-are-well-constrained*:
assumes *derivable C P S* σ *k C′*
shows *well-constrained C*
⟨*proof*⟩

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*⟩

lemma *all-derived-clauses-are-finite*:
shows *derivable-ecl C S* ⟹ ∀ *x* ∈ *S*. (*finite* (*cl-ecl x*)) ⟹ *finite* (*cl-ecl C*)
⟨*proof*⟩

lemma *all-derived-clauses-are-wellconstrained*:
shows *derivable-ecl C S* ⟹ ∀ *x* ∈ *S*. (*well-constrained x*) ⟹ *well-constrained C*
⟨*proof*⟩

lemma *SOUNDNESS*:
shows *derivable-ecl C S* ⟹ ∀ *x* ∈ *S*. (*finite* (*cl-ecl x*))
⟹ *set-entails-clause* (*cl-ecl* ‘ *S*) (*cl-ecl C*)
⟨*proof*⟩

lemma *REFUTABLE-SETS-ARE-UNSAT*:
assumes ∀ *x* ∈ *S*. (*finite* (*cl-ecl x*))
assumes *derivable-ecl C S*
assumes (*cl-ecl C* = {})
shows ¬ (*satisfiable-clause-set* (*cl-ecl* ‘ *S*))
⟨*proof*⟩

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*⟩

lemma *subterms-inclusion-subset*:
  assumes *subterms-inclusion E1 E2*
  assumes *E2* ⊆ *E2'*
  shows *subterms-inclusion E1 E2'*
⟨*proof*⟩

lemma *set-inclusion-preserve-normalization*:
  assumes *all-trms-irreducible E f*
  assumes *E'* ⊆ *E*
  shows *all-trms-irreducible E' f*
⟨*proof*⟩

lemma *subterms-inclusion-preserves-normalization*:
  assumes *all-trms-irreducible E f*
  assumes *subterms-inclusion E' E*
  shows *all-trms-irreducible E' f*
⟨*proof*⟩

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 ⇒ *'a eclause set* ⇒ *'a eclause set* ⇒ *'a subst* ⇒ *bool*
  where *redundant-inference C S P σ* ⟷ (∃ *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'*. ∃ *D'* ∈ *cl-ecl* ‘ *P*. ((*cl-ecl* (*fst x*), *snd x*),(*D'*,*σ*)) ∈ *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 ⇒ *'a eclause set* ⇒ *'a subst* ⇒ *'a clause* ⇒ *bool*
where (*redundant-clause C S σ C'*) =
(∃ *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'*,*σ*))) ∈ (*mult* (*mult*
trm-ord))
∨ (*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*⟩

definition *trms-subsumes*
where *trms-subsumes C D σ*
= ((*subst-cl* (*cl-ecl C*) *σ*) = (*cl-ecl D*)
∧ ((*subst-set* (*trms-ecl C*) *σ*) ⊆ *trms-ecl D*))

definition *inference-closed*
where *inference-closed S* = (∀ *P C' D ϑ*.
(*derivable D P S ϑ FirstOrder C'*) ⟶ (*D* ∈ *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 ⇒ bool*
where (*ground-inference-saturated S*) = (∀ *C P σ C'*. (*derivable C P S σ Ground C'*) ⟶
(*ground-clause* (*cl-ecl C*)) ⟶ (*grounding-set P σ*) ⟶ (*redundant-inference C S P σ*))

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

definition *inference-saturated* :: *'a eclause set ⇒ bool*
where (*inference-saturated S*) = (∀ *C P σ C' D ϑ η*.
(*derivable C P S σ Ground C'*) ⟶ (*ground-clause* (*cl-ecl C*)) ⟶ (*grounding-set P σ*)
⟶ (*derivable D P S ϑ FirstOrder C'*) ⟶ (*trms-subsumes D C η*)
⟶ (*σ* ≐ *ϑ* ◇ *η*)
⟶ (*redundant-inference* (*subst-ecl D η*) *S P σ*))

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

definition *clause-saturated* :: *'a eclause set ⇒ bool*
where (*clause-saturated S*) = (∀ *C P σ C' D ϑ η*.
(*derivable C P S σ Ground C'*) ⟶ (*ground-clause* (*cl-ecl C*))
⟶ (*derivable D P S ϑ FirstOrder C'*) ⟶ (*trms-subsumes D C η*)
⟶ (*σ* ≐ *ϑ* ◇ *η*)
⟶ (*redundant-clause* (*subst-ecl D η*) *S σ 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 σ Ground C'*
assumes ∀ *x* ∈ *S*. (*finite* (*cl-ecl x*))
assumes *grounding-set P σ*
shows ∃ *D*. (*D* ∈ *P* ∧ (((*mset-cl* (*C'*,*σ*)), (*mset-ecl* (*D*,*σ*))) ∈ (*mult* (*mult trm-ord*))))
⟨*proof*⟩

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 ∀ *x* ∈ *S*. (*finite* (*cl-ecl x*))
shows *redundant-inference E S P σ*
⟨*proof*⟩

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*⟩

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 $(\mathit{same\text{-}values}\ f) =$
$(\lambda x\ y.\ (f\ x) = (f\ y))$

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

lemma *same-values-fo-int*:
assumes *value-is-compatible-with-structure f*
shows *fo-interpretation* (*same-values f*)
⟨*proof*⟩

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-trms E*) = ({ *x*. (∃ *pair*. (*pair* ∈ *E*
∧ (∀ *pair′* ∈ *E*. (*snd pair′*,*snd pair*) ∉ *trm-ord*)) ∧ *x* = *fst pair*) })

lemma *min-trms-not-empty*:
assumes *E* ≠ {}
shows *min-trms E* ≠ {}
⟨*proof*⟩

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*))
⟶ ((*x*,*t*) ∈ *trm-ord* ⟶ (*trm-rep x S*) = *x*))
∧ (*C′* = (*cl-ecl CC*)) ∧ (*s*,*t*) ∈ *trm-ord* ∧ ((*s*,*t*) ∈ *trm-ord* ⟶ (*z* = *trm-rep*
s S))
∧ (*orient-lit-inst L′ t′ s′ pos σ*) ∧ (*sel C′*) = {} ∧ (*L′* ∈ *C′*)
∧ (*maximal-literal L C*) ∧ (*L* = (*subst-lit L′ σ*)) ∧ (*C* = (*subst-cl C′ σ*))
∧ (*ground-clause C*) ∧ (*t* = (*subst t′ σ*)) ∧ (*s* = (*subst s′ σ*)) ∧ (*finite C′*)

$\wedge$
$(\forall\ L\ u\ v.$
$(L \in C \longrightarrow \mathit{orient\text{-}lit}\ L\ u\ v\ \mathit{pos}$
$\longrightarrow (u,t) \in \mathit{trm\text{-}ord} \longrightarrow\ (v,t) \in \mathit{trm\text{-}ord}$
$\longrightarrow (\mathit{trm\text{-}rep}\ u\ S) \neq\ (\mathit{trm\text{-}rep}\ v\ S))$
$\wedge$
$(L \in C \longrightarrow \mathit{orient\text{-}lit}\ L\ u\ v\ \mathit{neg} \longrightarrow (u,t) \in \mathit{trm\text{-}ord} \longrightarrow\ (v,t) \in \mathit{trm\text{-}ord}$
$\longrightarrow (\mathit{trm\text{-}rep}\ u\ S) = (\mathit{trm\text{-}rep}\ v\ S)))$
$\wedge\ (\forall\ s''.\ ($
$(\mathit{eq\text{-}occurs\text{-}in\text{-}cl}\ t\ s''\ (C' - \{\ L'\ \})\ \sigma) \longrightarrow (s'',t) \in \mathit{trm\text{-}ord} \longrightarrow\ (s,t) \in \mathit{trm\text{-}ord}$
$\longrightarrow (\mathit{trm\text{-}rep}\ s\ S) \neq (\mathit{trm\text{-}rep}\ s''\ S))) \}))))$
$\langle proof \rangle$
termination $\langle proof \rangle$

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''* $(C' - \{\ L'\ \})\ \sigma) \longrightarrow (s'',t) \in \mathit{trm\text{-}ord} \longrightarrow\ (s,t) \in \mathit{trm\text{-}ord}$
$\longrightarrow (\mathit{trm\text{-}rep}\ s\ S) \neq (\mathit{trm\text{-}rep}\ s''\ S)))$

definition *smaller-lits-are-false*
where (*smaller-lits-are-false t C S*) =
$(\forall\ L\ u\ v.$
$(L \in C \longrightarrow \mathit{orient\text{-}lit}\ L\ u\ v\ \mathit{pos}$
$\longrightarrow (u,t) \in \mathit{trm\text{-}ord} \longrightarrow\ (v,t) \in \mathit{trm\text{-}ord}$
$\longrightarrow (\mathit{trm\text{-}rep}\ u\ S) \neq\ (\mathit{trm\text{-}rep}\ v\ S))$
$\wedge$
$(L \in C \longrightarrow \mathit{orient\text{-}lit}\ L\ u\ v\ \mathit{neg} \longrightarrow (u,t) \in \mathit{trm\text{-}ord} \longrightarrow\ (v,t) \in \mathit{trm\text{-}ord}$
$\longrightarrow (\mathit{trm\text{-}rep}\ u\ S) = (\mathit{trm\text{-}rep}\ v\ S)))$

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

lemma *smaller-lits-are-false-if-cl-not-valid*:
assumes $\neg$(*validate-ground-clause* (*int-clset S*) *C*)
shows *smaller-lits-are-false t C S*
⟨*proof*⟩

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

lemma *trms-irreducible-lemma*:
assumes *all-trms-irreducible* (*subst-set* (*trms-ecl C*) *σ*) (*λt. trm-rep t S*)
shows *trms-irreducible C σ S t*
⟨*proof*⟩

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

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

definition *subterm-reduction-applicable-aux*:: *'a eclause set ⇒ 'a trm ⇒ bool*
where *subterm-reduction-applicable-aux S t* =
$(is\text{-}compound\ t \wedge (lhs\ t,t) \in trm\text{-}ord\ \wedge (rhs\ t,t) \in trm\text{-}ord$
$\wedge\ (\ ((lhs\ t,t) \in trm\text{-}ord \longrightarrow (trm\text{-}rep\ (lhs\ t)\ S) \neq (lhs\ t))$
$\vee\ ((rhs\ t,t) \in trm\text{-}ord \longrightarrow (trm\text{-}rep\ (rhs\ t)\ S) \neq (rhs\ t))))$

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*⟩

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*))))
⟨*proof*⟩

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*⟩

lemma *trm-rep-is-lower*:
  shows (*t* ≠ (*trm-rep t S*)) ⟶ (((*trm-rep t S*),*t*) ∈ *trm-ord*) (**is** *?P t*)
⟨*proof*⟩

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

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*⟩

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*)))
⟨*proof*⟩

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*⟩

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*⟩

lemma *subterms-of-irred-trms-are-irred*:
(*trm-rep t S*) ≠ *t* ⟶ *st-red S t* (**is** (*?P t*))
⟨*proof*⟩

lemma *trm-rep-compatible-with-structure*:
shows *value-is-compatible-with-structure* (λ*x*. *trm-rep x S*)
⟨*proof*⟩

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-redex p t C S t'*
= (∀ *q s*. ((*q*,*p*) ∈ (*pos-ord C t'*) ⟶ (*subterm t q s*) ⟶ (*trm-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-eq-exists t s C I σ L1* = (∃ *L*∈ *C* − { *L1* }. ∃ *u v*.
(*orient-lit-inst L u v pos σ*) ∧ ((*subst t σ*) = (*subst u σ*))

$\wedge$ (I (*subst* s σ) (*subst* v σ)))

lemma *maximal-literal-is-unique-lemma*:
assumes $\neg$*equivalent-eq-exists* t s C (*int-clset* S) σ *L1*
shows *maximal-literal-is-unique* (*subst* t σ) (*subst* s σ) C *L1* S σ
⟨*proof*⟩

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

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* (λt. (*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* (λt. (*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* (λt. (*trm-rep* t S))) (*subst-cl* (C − { *L1* }) σ))
⟨*proof*⟩

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) =
(($D \in S$) $\wedge$ (*eligible-literal L1* C σ') $\wedge$ (*eligible-literal L2* D σ')
$\wedge$ *ground-clause* (*subst-cl* (*cl-ecl* D) σ')
$\wedge$ (*minimal-redex* p (*subst* t σ) C S t)
$\wedge$ (*coincide-on* σ σ' (*vars-of-cl* (*cl-ecl* C)))
$\wedge$ (*allowed-redex* u' C σ)
$\wedge$ ($\neg$ *is-a-variable* u')
$\wedge$ ($L1 \in$ (*cl-ecl* C)) $\wedge$ ($L2 \in$ (*cl-ecl* D))
$\wedge$ (*orient-lit-inst L1* t s *polarity* σ')
$\wedge$ (*orient-lit-inst L2* u v *pos* σ')

$\wedge$ (*subst u* σ') $\neq$ (*subst v* σ')
$\wedge$ (*subst u'* σ') = (*subst u* σ')
$\wedge$ ($\neg$ *validate-ground-clause I* (*subst-cl* ((*cl-ecl D*) − { *L2* }) σ'))
$\wedge$ ((*subst-lit L2* σ'),(*subst-lit L1* σ')) $\in$ *lit-ord*
$\wedge$ ($\forall$ *x* $\in$ (*cl-ecl D*) − { *L2* }. ((*subst-lit x* σ'),(*subst-lit L2* σ'))
$\in$ *lit-ord*)
$\wedge$ (*all-trms-irreducible* (*subst-set* (*trms-ecl D*) σ')
(λ*t*. (*trm-rep t S*)))
$\wedge$ (*I* (*subst u* σ') (*subst v* σ'))
$\wedge$ (*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 $\exists$ *CC-bis C'-bis L'-bis* σ*-bis t'-bis s'-bis t-bis.*
(*candidate-values z CC-bis C'-bis C s L L'-bis* σ*-bis t'-bis s'-bis t S*)
$\wedge$ (*vars-of-cl* (*cl-ecl D*)) $\cap$ *vars-of-cl* (*cl-ecl CC-bis*) = {}

⟨*proof*⟩

lemma *pos-ord-acyclic*:
shows *acyclic* (*pos-ord C t*)
⟨*proof*⟩

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*) ≠ (*subst t σ*)
assumes *L1* ∈ (*cl-ecl C*)
assumes (*orient-lit-inst L1 t s polarity σ*)
assumes ∀ *x* ∈ *S*. *finite* (*cl-ecl x*)
assumes *ground-clause* (*subst-cl* (*cl-ecl C*) *σ*)
assumes (*fo-interpretation* (*same-values* (λ*t*. (*trm-rep t S*))))
assumes *C* ∈ *S*
assumes *Ball S well-constrained*
assumes *all-trms-irreducible* (*subst-set* (*trms-ecl C*) *σ*) (λ*t*. *trm-rep t S*)
assumes ¬ *validate-ground-clause* (*int-clset S*) (*subst-cl* (*cl-ecl C*) *σ*)
assumes *closed-under-renaming S*

shows ∃ *σ′ u u′ p v D L2*.
((*reduction L1 C σ′ t s polarity L2 u u′ p v D* (*same-values* (λ*t*. (*trm-rep t S*)))
S σ)
∧ (*variable-disjoint C D*))

⟨*proof*⟩

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*⟩

lemma *allowed-redex-coincide*:
assumes *allowed-redex t C σ*
assumes *t* ∈ *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*⟩

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* = (λ*t*. (*trm-rep t S*))
assumes *η* = (*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 ∀ *x* ∈ *S*. ∀ *y*. (*y* ∈ *trms-ecl x* ⟶ *dom-trm y* (*cl-ecl x*))
assumes *C* ∈ *S*
assumes *fo-interpretation I*
shows *all-trms-irreducible* (*subst-set E η*) *f*
⟨*proof*⟩

lemma *no-valid-literal*:
assumes $L \in C$
assumes *orient-lit-inst L t s pos* σ
assumes $\neg$(*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*⟩

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*⟩

lemma *lift-maximal-literal*:
assumes *maximal-literal* (*subst-lit L* σ) (*subst-cl C* σ)
shows *maximal-literal L C*
⟨*proof*⟩

lemma *lift-eligible-literal*:
assumes *eligible-literal L C* σ
assumes $\sigma \doteq \vartheta \Diamond \eta$
shows *eligible-literal L C* ϑ
⟨*proof*⟩

lemma *lift-allowed-redex*:
assumes $\sigma \doteq \vartheta \Diamond \eta$
assumes (*allowed-redex u C* σ)
shows (*allowed-redex u C* ϑ)
⟨*proof*⟩

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*⟩

lemma *lift-dom-trm*:
assumes *dom-trm* (*subst t* ϑ) (*subst-cl C* ϑ)
assumes $\sigma \doteq \vartheta \Diamond \eta$
shows *dom-trm* (*subst t* σ) (*subst-cl C* σ)
⟨*proof*⟩

lemma *lift-irreducible-terms*:
assumes *T* = *get-trms C* (*dom-trms* (*subst-cl D* σ) (*subst-set E* σ)) *Ground*

assumes $\sigma \doteq \vartheta \Diamond \eta$
shows $\exists T'.$ ($(subst\text{-}set\ T'\ \eta) \subseteq T \wedge T' = get\text{-}trms\ C'$
$(dom\text{-}trms\ (subst\text{-}cl\ D\ \vartheta)\ (subst\text{-}set\ E\ \vartheta))\ FirstOrder$)
⟨*proof*⟩

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 $\exists D\ \vartheta.$ *superposition P1 P2 D ϑ FirstOrder C′* $\wedge\ \sigma \doteq \vartheta \Diamond \sigma \wedge$ *trms-subsumes D C σ*
⟨*proof*⟩

lemma *lifting-lemma-factorization*:
assumes *factorization P1 C σ Ground C′*
shows $\exists D\ \vartheta.$ *factorization P1 D ϑ FirstOrder C′* $\wedge\ \sigma \doteq \vartheta \Diamond \sigma \wedge$ *trms-subsumes D C σ*
⟨*proof*⟩

lemma *lifting-lemma-reflexion*:
assumes *reflexion P1 C σ Ground C′*
shows $\exists D\ \vartheta.$ *reflexion P1 D ϑ FirstOrder C′* $\wedge\ \sigma \doteq \vartheta \Diamond \sigma \wedge$ *trms-subsumes D C σ*
⟨*proof*⟩

lemma *lifting-lemma*:
assumes *deriv*: *derivable C P S σ Ground C′*
shows $\exists\ D\ \vartheta.$ ((*derivable D P S ϑ FirstOrder C′*) $\wedge$ ($\sigma \doteq \vartheta \Diamond \sigma$) $\wedge$ (*trms-subsumes D C σ*))
⟨*proof*⟩

lemma *trms-subsumes-and-red-inf*:
assumes *trms-subsumes D C η*
assumes *redundant-inference* (*subst-ecl D η*) *S P σ*
assumes $\sigma \doteq \vartheta \Diamond \eta$
shows *redundant-inference C S P σ*
⟨*proof*⟩

lemma *lift-inference*:
assumes *inference-saturated S*
shows *ground-inference-saturated S*
⟨*proof*⟩

lemma *lift-redundant-cl* :
assumes $C' =$ *subst-cl D ϑ*
assumes *redundant-clause C S η C′*
assumes $\sigma \doteq \vartheta \Diamond \eta$

assumes *finite D*
shows *redundant-clause C S σ D*
⟨*proof*⟩

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 ∀ *x* ∈ *S*. (*finite* (*cl-ecl x*))
shows *clause-saturated S*
⟨*proof*⟩

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*: ∀ *x* ∈ *S*. (*finite* (*cl-ecl x*))
assumes *Ball S well-constrained*
assumes *all-non-empty*: ∀ *x* ∈ *S*. (*cl-ecl x*) ≠ {}
assumes *closed-under-renaming S*
shows ∀ *C σ*. (*fst pair* = *C*) ⟶ *σ* = (*snd pair*) ⟶ *C* ∈ *S* ⟶
ground-clause (*subst-cl* (*cl-ecl C*) *σ*)
⟶ (*all-trms-irreducible* (*subst-set* (*trms-ecl C*) *σ*) (λ*t*. (*trm-rep t S*)))
⟶ *validate-ground-clause* (*same-values* (λ*t*. (*trm-rep t S*))) (*subst-cl* (*cl-ecl C*) *σ*)
(**is** *?P pair*)
⟨*proof*⟩

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*⟩

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.