Typed Ordered Resolution

Adnan Mohammed Ahmed Balazs Toth

September 3, 2025

Abstract

Ordered Resolution is a proof calculus for reasoning about first-order logic that is implemented in many automatic theorem provers. It works by saturating the given set of clauses and is refutationally complete, meaning that if the set is inconsistent, the saturation will contain a contradiction. In this formalization, we restructured the completeness proof to cleanly separate the ground (i.e., variable-free) and nonground aspects. We also added a type system to the calculus. We relied on the library for first-order clauses and on the saturation framework.

Contents

theory *Ground-Ordered-Resolution*
imports
First-Order-Clause.Selection-Function
First-Order-Clause.Ground-Order
First-Order-Clause.Literal-Functor
begin

1 Resolution Calculus

locale *ground-ordered-resolution-calculus* =

ground-order **where** $less_t = less_t +$
selection-function select
for
$less_t :: 't \Rightarrow 't \Rightarrow bool$ **and**
$select :: 't\ clause \Rightarrow 't\ clause$
begin

1.1 Resolution Calculus

inductive $resolution :: 't\ clause \Rightarrow 't\ clause \Rightarrow 't\ clause \Rightarrow bool$
where
resolutionI:
$C = add\text{-}mset\ L_C\ C' \Longrightarrow$
$D = add\text{-}mset\ L_D\ D' \Longrightarrow$
$L_C = (Neg\ t) \Longrightarrow$
$L_D = (Pos\ t) \Longrightarrow$
$D \prec_c C \Longrightarrow$
$select\ C = \{\#\} \wedge is\text{-}maximal\ L_C\ C \vee is\text{-}maximal\ L_C\ (select\ C) \Longrightarrow$
$select\ D = \{\#\} \Longrightarrow$
$is\text{-}strictly\text{-}maximal\ L_D\ D \Longrightarrow$
$R = (C' + D') \Longrightarrow$
$resolution\ D\ C\ R$

inductive *factoring* ::
$'t\ clause \Rightarrow 't\ clause \Rightarrow bool$
where
factoringI:
$C = add\text{-}mset\ L_1\ (add\text{-}mset\ L_1\ C') \Longrightarrow$
$L_1 = (Pos\ t) \Longrightarrow$
$select\ C = \{\#\} \Longrightarrow$
$is\text{-}maximal\ L_1\ C \Longrightarrow$
$D = add\text{-}mset\ L_1\ C' \Longrightarrow$
$factoring\ C\ D$

1.2 Ground Layer

abbreviation *resolution-inferences* **where**
$resolution\text{-}inferences \equiv \{Infer\ [D,\ C]\ R \mid D\ C\ R.\ resolution\ D\ C\ R\}$

abbreviation *factoring-inferences* **where**
$factoring\text{-}inferences \equiv \{Infer\ [C]\ D \mid C\ D.\ factoring\ C\ D\}$

definition *G-Inf* :: $'t\ clause\ inference\ set$ **where**
G-Inf =
$\{Infer\ [D,\ C]\ R \mid D\ C\ R.\ resolution\ D\ C\ R\} \cup$
$\{Infer\ [P]\ C \mid P\ C.\ factoring\ P\ C\}$

abbreviation *G-Bot* :: $'t\ clause\ set$ **where**
$G\text{-}Bot \equiv \{\{\#\}\}$

definition *G-entails* :: *'t clause set* ⇒ *'t clause set* ⇒ *bool* **where**
G-entails N_1 N_2 ⟷ (∀ *I*. *I* ⊨*s* N_1 ⟶ *I* ⊨*s* N_2)

1.3 Smaller Conclussions

lemma *ground-resolution-smaller-conclusion*:
assumes
step: *resolution D C R*
shows $R \prec_c C$
using *step*
proof (*cases D C R rule*: *resolution.cases*)
case (*resolutionI* L_C C' L_D D' *t*)

have ∀ *k*∈#*D'*. *k* ≺*l* *Pos t*
using ‹*is-strictly-maximal* L_D *D*› ‹*D* = *add-mset* L_D D'›
using *is-strictly-maximal-def local.resolutionI*(*4*)
by *fastforce*

moreover have ⋀*A*. *Pos A* ≺*l* *Neg A*
unfolding *literal.order.multiset-extension-def*
by *auto*

ultimately have ∀ *k*∈#*D'*. *k* ≺*l* *Neg t*
using *literal.order.order.strict-trans*
by *blast*

hence $D' \prec_c$ {#*Neg t*#}
using *one-step-implies-multp*[*of* {#*Neg t*#} D' (≺*l*) {#}]
by (*simp add*: $less_c$-*def*)

hence $D' + C' \prec_c$ *add-mset* (*Neg t*) C'
using *multp-cancel*[*of* (≺*l*) C' D' {#*Neg t*#}]
using $less_c$-*def* **by** *force*

thus *?thesis*
unfolding *resolutionI*
by (*simp only*: *add.commute*)
qed

lemma *ground-factoring-smaller-conclusion*:
assumes *step*: *factoring C D*
shows $D \prec_c C$
using *step*
proof (*cases C D rule*: *factoring.cases*)
case (*factoringI* L_1 C' *t*)

have *D* = *add-mset* L_1 C'
using *factoringI*
by *argo*

then show *?thesis*
by (*metis* (*lifting*) *add.comm-neutral add-mset-add-single add-mset-not-empty clause.order.multiset-extension-def empty-iff local.factoringI*(*1*) *one-step-implies-multp set-mset-empty*)
qed

end

1.4 Redundancy Criterion

sublocale *ground-ordered-resolution-calculus* ⊆ *consequence-relation* **where**
Bot = *G-Bot* **and**
entails = *G-entails*
proof *unfold-locales*

show *G-Bot* ≠ {}
by *simp*
next

show ⋀*B N*. *B* ∈ *G-Bot* ⟹ *G-entails* {*B*} *N*
by (*simp add*: *G-entails-def*)
next

show ⋀*N2 N1*. *N2* ⊆ *N1* ⟹ *G-entails N1 N2*
by (*auto simp*: *G-entails-def elim*!: *true-clss-mono*[*rotated*])
next
fix *N1 N2*
assume *ball-G-entails*: ∀ *C* ∈ *N2*. *G-entails N1* {*C*}

show *G-entails N1 N2*
unfolding *G-entails-def*
proof (*intro allI impI*)
fix *I* :: *'t set*
assume *I* ⊨*s N1*

hence ∀ *C* ∈ *N2*. *I* ⊨*s* {*C*}
using *ball-G-entails*
by (*simp add*: *G-entails-def*)

then show *I* ⊨*s N2*
by (*simp add*: *true-clss-def*)
qed
next

show ⋀*N1 N2 N3*. *G-entails N1 N2* ⟹ *G-entails N2 N3* ⟹ *G-entails N1 N3*
using *G-entails-def*
by *simp*
qed

sublocale *ground-ordered-resolution-calculus* $\subseteq$ *calculus-with-finitary-standard-redundancy*
where
  Inf = *G-Inf* **and**
  Bot = *G-Bot* **and**
  entails = *G-entails* **and**
  less = $(\prec_c)$
  defines *GRed-I* = *Red-I* **and** *GRed-F* = *Red-F*
proof *unfold-locales*

  show *transp* $(\prec_c)$
    by *simp*
next

  show *wfP* $(\prec_c)$
    by *auto*
next

  show $\bigwedge\iota.\ \iota \in$ *G-Inf* $\Longrightarrow$ *prems-of* $\iota \neq []$
    by (*auto simp*: *G-Inf-def*)
next
  fix ι

  have *concl-of* $\iota \prec_c$ *main-prem-of* ι
    if ι*-def*: ι = *Infer* [*D*, *C*] *R* **and**
      infer: *resolution D C R*
    for *D C R*
    unfolding ι*-def*
    using *infer*
    using *ground-resolution-smaller-conclusion*
    by *simp*

  moreover have *concl-of* $\iota \prec_c$ *main-prem-of* ι
    if ι*-def*: ι = *Infer* [*P*] *C* **and**
      infer: *factoring P C*
    for *P C*
    unfolding ι*-def*
    using *infer*
    using *ground-factoring-smaller-conclusion*
    by *simp*

  ultimately show $\iota \in$ *G-Inf* $\Longrightarrow$ *concl-of* $\iota \prec_c$ *main-prem-of* ι
    unfolding *G-Inf-def*
    by *fast*
qed

end
theory *Relation-Extra*
  imports *Main*

begin

lemma *partition-set-around-element*:
assumes *tot*: *totalp-on N R* **and** *x-in*: $x \in N$
shows $N = \{y \in N.\ R\ y\ x\} \cup \{x\} \cup \{y \in N.\ R\ x\ y\}$
proof (*intro Set.equalityI Set.subsetI*)
fix z **assume** $z \in N$
hence $R\ z\ x \vee z = x \vee R\ x\ z$
using *tot*[*THEN totalp-onD*] *x-in* **by** *auto*
thus $z \in \{y \in N.\ R\ y\ x\} \cup \{x\} \cup \{y \in N.\ R\ x\ y\}$
using ‹$z \in N$› **by** *auto*
next
fix z **assume** $z \in \{y \in N.\ R\ y\ x\} \cup \{x\} \cup \{y \in N.\ R\ x\ y\}$
hence $z \in N \vee z = x$
by *auto*
thus $z \in N$
using *x-in* **by** *auto*
qed

end
theory *Ground-Ordered-Resolution-Completeness*
imports
Ground-Ordered-Resolution
Relation-Extra
First-Order-Clause.HOL-Extra
begin

1.5 Mode Construction

context *ground-ordered-resolution-calculus*
begin

context
fixes N :: *'t clause set*
begin

function *epsilon* :: *'t clause* ⇒ *'t set* **where**
epsilon $C = \{A \mid A\ C'.$
$C \in N \wedge$
$C =$ *add-mset* (*Pos A*) $C' \wedge$
select $C = \{\#\} \wedge$
is-strictly-maximal (*Pos A*) $C \wedge$
$\neg\ (\bigcup D \in \{D \in N.\ D \prec_c C\}.\ epsilon\ D) \models C\}$
by *auto*

termination *epsilon*
proof (*relation* $\{(x,\ y).\ x \prec_c y\}$)
show *wf* $\{(x,\ y).\ x \prec_c y\}$
using *wfp-def* **by** *blast*

next
show $\bigwedge C\ D.\ D \in \{D \in N.\ D \prec_c C\} \Longrightarrow (D,\ C) \in \{(x,\ y).\ x \prec_c y\}$
by *simp*
qed

declare *epsilon.simps*[*simp del*]

end

lemma *epsilon-eq-empty-or-singleton*: *epsilon N C* = {} ∨ (∃ *A*. *epsilon N C* = {*A*})
proof −

have $\exists_{\leq 1} A.$ *is-strictly-maximal* (*Pos A*) *C*
by (*metis* (*mono-tags*, *lifting*) *Uniq-def literal.inject*(*1*) *literal.order.Uniq-is-strictly-maximal-in-mset*)

hence $\exists_{\leq 1} A.\ \exists\, C'.$
C = *add-mset* (*Pos A*) *C'* ∧ *is-strictly-maximal* (*Pos A*) *C*
by (*simp add*: *Uniq-def*)

hence *Uniq-epsilon*: $\exists_{\leq 1} A.\ \exists\, C'.$
C ∈ *N* ∧
C = *add-mset* (*Pos A*) *C'* ∧
select C = {#} ∧
is-strictly-maximal (*Pos A*) *C* ∧
¬ (⋃ *D* ∈ {*D* ∈ *N*. *D* ≺*c C*}. *epsilon N D*) ⊨ *C*
using *Uniq-antimono'*
by (*smt* (*verit*) *Uniq-def Uniq-prodI case-prod-conv*)

show *?thesis*
unfolding *epsilon.simps*[*of N C*]
using *Collect-eq-if-Uniq*[*OF Uniq-epsilon*]
by (*smt* (*verit*, *best*) *Collect-cong Collect-empty-eq Uniq-def Uniq-epsilon case-prod-conv insertCI mem-Collect-eq*)
qed

definition *rewrite-sys* **where**
rewrite-sys N C ≡ (⋃ *D* ∈ {*D* ∈ *N*. *D* ≺*c C*}. *epsilon N D*)

lemma *rewrite-sys-subset-if-less-cls*: *C* ≺*c D* ⟶ *rewrite-sys N C* ⊆ *rewrite-sys N D*
unfolding *rewrite-sys-def*
by *fastforce*

lemma *mem-epsilonE*:
assumes *rule-in*: *A* ∈ *epsilon N C*
obtains *C'* **where**

```
    C ∈ N and
    C = add-mset (Pos A) C' and
    select C = {#} and
    is-strictly-maximal (Pos A) C and
    ¬ rewrite-sys N C ⊨ C
  using rule-in
  unfolding epsilon.simps[of N C] mem-Collect-eq Let-def rewrite-sys-def
  by (metis (no-types, lifting))

lemma epsilon-unfold: epsilon N C = {A | A C'.
    C ∈ N ∧
    C = add-mset (Pos A) C' ∧
    select C = {#} ∧
    is-strictly-maximal (Pos A) C ∧
    ¬ rewrite-sys N C ⊨ C}
  by (simp add: epsilon.simps[of N C] rewrite-sys-def)

lemma epsilon-subset-if-less-cls: C ≺c D ⟹ epsilon N C ⊆ rewrite-sys N D
  unfolding rewrite-sys-def
  using epsilon-unfold
  by blast

lemma
  assumes
    D ⪯c C and
    C-prod: A ∈ epsilon N C and
    L-in: L ∈# D
  shows
    lesseq-trm-if-pos: is-pos L ⟹ atm-of L ⪯t A and
    less-trm-if-neg: is-neg L ⟹ atm-of L ≺t A
proof −
  from C-prod obtain C' where
    C-def: C = add-mset (Pos A) C' and
    C-max-lit: is-strictly-maximal (Pos A) C
    by (auto elim: mem-epsilonE)

  have Pos A ≺l L if is-pos L and ¬ atm-of L ⪯t A
  proof −

    from that(2) have A ≺t atm-of L
      by order

    hence multp (≺t) {#A#} {#atm-of L#}
      by auto

    with that(1) show Pos A ≺l L
      by (metis (no-types, lifting) Pos-atm-of-iff less_l-def
          literal-to-mset.simps(1))
  qed
```

```
  moreover have Pos A ≺l L if is-neg L and ¬ atm-of L ≺t A
  proof −

    from that(2) have A ⪯t atm-of L
      by order

    hence multp (≺t) {#A#} {#atm-of L, atm-of L#}
      by auto

    with that(1) show Pos A ≺l L
      by (cases L) (simp-all add: less_l-def)
  qed

  moreover have False if Pos A ≺l L
  proof −

    have C ≺c D
      unfolding less_c-def
    proof (rule multp-if-maximal-of-lhs-is-less)

      show Pos A ∈# C
        by (simp add: C-def)
    next

      show L ∈# D
        using L-in
        by simp
    next

      show is-maximal (Pos A) C
        using C-max-lit
        by auto
    next

      show Pos A ≺l L
        using that
        by simp
    qed simp-all

    with ‹D ⪯c C› show False
      by order
  qed

  ultimately show is-pos L ⟹ atm-of L ⪯t A and is-neg L ⟹ atm-of L ≺t A
    by metis+
qed

lemma less-trm-iff-less-cls-if-mem-epsilon:
```

```
  assumes C-prod: A_C ∈ epsilon N C and D-prod: A_D ∈ epsilon N D
  shows A_C ≺_t A_D ⟷ C ≺_c D
proof −
  from C-prod
  obtain C' where
    C ∈ N and
    C-def: C = add-mset (Pos A_C) C' and
    is-strictly-maximal (Pos A_C) C
    by (auto elim!: mem-epsilonE)

  hence ∀ L ∈# C'. L ≺_l Pos A_C
    unfolding is-strictly-maximal-def
    by auto

  from D-prod
  obtain D' where
    D ∈ N and
    D-def: D = add-mset (Pos A_D) D' and
    is-strictly-maximal (Pos A_D) D
    by (auto elim!: mem-epsilonE)

  hence ∀ L ∈# D'. L≺_l Pos A_D
    unfolding is-strictly-maximal-def
    by auto

  show ?thesis
  proof (rule iffI)
    assume A_C ≺_t A_D

    hence Pos A_C ≺_l Pos A_D
      by (simp add: less_l-def)

    moreover hence ∀ L ∈# C'. L ≺_l Pos A_D
      using ‹∀ L ∈# C'. L ≺_l Pos A_C›
      by fastforce

    ultimately show C ≺_c D
      using one-step-implies-multp[of D C - {#}] less_c-def
      by (simp add: D-def C-def)
  next
    assume C ≺_c D

    hence epsilon N C ⊆ (⋃ (epsilon N ‘ {x ∈ N. x ≺_c D}))
      using ‹C ∈ N›
      by auto

    hence A_C ∈ (⋃ (epsilon N ‘ {x ∈ N. x ≺_c D}))
      using C-prod
      by auto
```

```
    hence A_C ≠ A_D
      by (metis D-prod rewrite-sys-def mem-epsilonE true-cls-add-mset true-lit-iff)

    moreover have ¬ (A_D ≺_t A_C)
    proof (rule notI)
      assume A_D ≺_t A_C

      then have Pos A_D ≺_l Pos A_C
        by (simp add: less_l-def)

      moreover have ∀ L ∈# D'. L ≺_l Pos A_C
        using ‹∀ L ∈# D'. L ≺_l Pos A_D›
        using calculation literal.order.order.strict-trans
        by blast

      ultimately have D ≺_c C
        using one-step-implies-multp[of C D - {#}] less_c-def
        by (simp add: D-def C-def)

      thus False
        using ‹C ≺_c D›
        by order
    qed

    ultimately show A_C ≺_t A_D
      by order
  qed
qed

lemma false-cls-if-productive-epsilon:
  assumes C-prod: A ∈ epsilon N C and D ∈ N and C ≺_c D
  shows ¬ rewrite-sys N D ⊨ C - {#Pos A#}
proof -

  from C-prod obtain C' where
    C-in: C ∈ N and
    C-def: C = add-mset (Pos A) C' and
    select C = {#} and
    Pox-A-max: is-strictly-maximal (Pos A) C and
    ¬ rewrite-sys N C ⊨ C
    by (rule mem-epsilonE) blast

  from ‹D ∈ N› ‹C ≺_c D› have A ∈ rewrite-sys N D
    using C-prod epsilon-subset-if-less-cls
    by auto

  from ‹D ∈ N› have rewrite-sys N D ⊆ (⋃ D ∈ N. epsilon N D)
    by (auto simp: rewrite-sys-def)
```

have ¬ *rewrite-sys N D* ⟙= *C′*
unfolding *true-cls-def Set.bex-simps*
proof (*intro ballI*)
fix *L*
assume *L-in*: *L* ∈# *C′*

hence *L* ∈# *C*
by (*simp add*: *C-def*)

have *C′* ≺*c* *C*
by (*metis* (*mono-tags*, *lifting*) *C-def add.comm-neutral*
add-mset-add-single add-mset-not-empty
clause.order.multiset-extension-def empty-iff
one-step-implies-multp set-mset-empty)

hence *C′* ⪯*c* *C*
by *order*

show ¬ *rewrite-sys N D* ⟙=*l* *L*
proof (*cases L*)
case (*Pos A*L)

moreover have *A*L ∉ *rewrite-sys N D*
proof −

have ∀ *y*∈#*C′*. *y* ≺*l* *Pos A*
using *Pox-A-max C-def is-strictly-maximal-def*
by *auto*

with *Pos* **have** *A*L ∉ *insert A* (*rewrite-sys N C*)
using *L-in* ‹¬ *rewrite-sys N C* ⟙= *C*› *C-def*
by *blast*

moreover have *A*L ∉ (⋃ *D′* ∈ {*D′* ∈ *N*. *C* ≺*c* *D′* ∧ *D′* ≺*c* *D*}. *epsilon N*
D′)
proof −
have *A*L ⪯*t* *A*
using *Pos lesseq-trm-if-pos*[*OF* ‹*C′* ⪯*c* *C*› *C-prod* ‹*L* ∈# *C′*›]
by *simp*

thus *?thesis*
using *less-trm-iff-less-cls-if-mem-epsilon*
using *C-prod calculation rewrite-sys-def*
by *fastforce*
qed

moreover have *rewrite-sys N D* =
insert A (*rewrite-sys N C*) ∪ (⋃ *D′* ∈ {*D′* ∈ *N*. *C* ≺*c* *D′* ∧ *D′* ≺*c* *D*}.

epsilon N D′)
proof −

have *rewrite-sys N D* = (⋃ *D′* ∈ {*D′* ∈ *N*. *D′* ≺*c D*}. *epsilon N D′*)
by (*simp only*: *rewrite-sys-def*)

also have
. . . = (⋃ *D′* ∈ {*D′* ∈ {*y* ∈ *N*. *y* ≺*c C*} ∪ {*C*} ∪ {*y* ∈ *N*. *C* ≺*c y*}. *D′* ≺*c D*}. *epsilon N D′*)
using *C-in clause.order.antisym-conv3*
by *auto*

also have
. . . = (⋃ *D′* ∈ {*y* ∈ *N*. *y* ≺*c C* ∧ *y* ≺*c D*} ∪ {*C*} ∪ {*y* ∈ *N*. *C* ≺*c y* ∧ *y* ≺*c D*}. *epsilon N D′*)
using ‹*C* ≺*c D*›
by *auto*

also have . . . = (⋃ *D′* ∈ {*y* ∈ *N*. *y* ≺*c C*} ∪ {*C*} ∪ {*y* ∈ *N*. *C* ≺*c y* ∧ *y* ≺*c D*}. *epsilon N D′*)
by (*metis* (*lifting*) *assms*(*3*) *clause.order.order.strict-trans*)

also have
. . . = *rewrite-sys N C* ∪ *epsilon N C* ∪ (⋃ *D′* ∈ {*y* ∈ *N*. *C* ≺*c y* ∧ *y* ≺*c D*}. *epsilon N D′*)
by (*auto simp*: *rewrite-sys-def*)

finally show *?thesis*
using *C-prod*
by (*metis* (*no-types*, *lifting*) *empty-iff insertE insert-is-Un*
epsilon-eq-empty-or-singleton sup-commute)
qed

ultimately show *?thesis*
by *simp*
qed
ultimately show *?thesis*
by *simp*
next
case (*Neg* A_L)

moreover have A_L ∈ *rewrite-sys N D*
using *Neg* ‹*L* ∈# *C*› ‹*C* ≺*c D*› ‹¬ *rewrite-sys N C* ⊨ *C*› *rewrite-sys-subset-if-less-cls*
by *blast*

ultimately show *?thesis*
by *simp*
qed
qed

thus ¬ *rewrite-sys N D* ⊨ *C* − {#*Pos A*#}
by (*simp add*: *C-def*)
qed

lemma *neg-notin-Interp-not-produce*:
Neg A ∈# *C* ⟹ *A* ∉ *rewrite-sys N D* ∪ *epsilon N D* ⟹ *C* ⪯$_c$ *D* ⟹ *A* ∉ *epsilon N D*″
by (*smt* (*verit*, *del-insts*) *Neg-atm-of-iff UN-I Un-iff clause.order.order.strict-trans1 clause.order.not-less less-trm-if-neg literal.sel*(*2*) *mem-Collect-eq mem-epsilonE*

rewrite-sys-def term.order.order.asym)

lemma *lift-interp-entails*:
assumes
D-in: *D* ∈ *N* **and**
D-entailed: *rewrite-sys N D* ⊨ *D* **and**
C-in: *C* ∈ *N* **and**
D-lt-C: *D* ≺$_c$ *C*
shows *rewrite-sys N C* ⊨ *D*
proof −

from *D-entailed* **obtain** *L A* **where**
L-in: *L* ∈# *D* **and**
L-eq-disj-L-eq: *L* = *Pos A* ∧ *A* ∈ *rewrite-sys N D* ∨ *L* = *Neg A* ∧ *A* ∉ *rewrite-sys N D*
unfolding *true-cls-def true-lit-iff*
by *metis*

have *rewrite-sys N D* ⊆ *rewrite-sys N C*
using *D-lt-C rewrite-sys-subset-if-less-cls*
by *auto*

from *L-eq-disj-L-eq*
show *rewrite-sys N C* ⊨ *D*
proof (*elim disjE conjE*)
assume *L* = *Pos A* **and** *A* ∈ *rewrite-sys N D*

thus *rewrite-sys N C* ⊨ *D*
using *L-in* ‹*rewrite-sys N D* ⊆ *rewrite-sys N C*›
by *auto*
next
assume *L*: *L* = *Neg A* **and** *A*: *A* ∉ *rewrite-sys N D*

have *A* ∉ *rewrite-sys N C*
proof (*cases A* ∈ *epsilon N C*)
case *True*

then show *?thesis*

```
    by (meson mem-epsilonE pos-literal-in-imp-true-cls strictly-maximal-in-clause)
  next
    case False

    then have A ∉ epsilon N D
      using D-entailed mem-epsilonE
      by blast

    then show ?thesis
      using neg-notin-Interp-not-produce A L-in
      unfolding rewrite-sys-def L
      by blast
  qed

  thus rewrite-sys N C ⊨ D
    using L-in ‹L = Neg A›
    by blast
 qed
qed

lemma produces-imp-in-interp:
 assumes Neg A ∈# C and D-prod: A ∈ epsilon N D
 shows A ∈ rewrite-sys N C
proof −
 from D-prod have Pos A ∈# D and is-strictly-maximal (Pos A) D
   by (auto elim: mem-epsilonE)

 have D ≺c C
   unfolding less_c-def
 proof (rule multp-if-maximal-of-lhs-is-less)

   show Pos A ∈# D
     using ‹Pos A ∈# D› .
 next

   show Neg A ∈# C
     using ‹Neg A ∈# C› .
 next

   show Pos A ≺l Neg A
     by (simp add: less_l-def)
 next

   show is-maximal (Pos A) D
     using ‹is-strictly-maximal (Pos A) D›
     by auto
 qed simp-all

 hence ¬ (≺c)== C D
```

using *clause.order.not-less*
by *blast*

thus *?thesis*
proof (*rule contrapos-np*)

from *D-prod* **show** $A \notin$ *rewrite-sys N C* $\Longrightarrow (\prec_c)^{==}$ *C D*
using $\langle D \prec_c C \rangle$ *epsilon-subset-if-less-cls*
by *blast*
qed
qed

lemma *split-Union-epsilon*:
assumes *D-in*: $D \in N$
shows $(\bigcup C \in N.$ *epsilon N C*$) =$
rewrite-sys N D $\cup$ *epsilon N D* $\cup$ $(\bigcup C \in \{C \in N.\ D \prec_c C\}.$ *epsilon N C*$)$
proof −

have $N = \{C \in N.\ C \prec_c D\} \cup \{D\} \cup \{C \in N.\ D \prec_c C\}$
proof (*rule partition-set-around-element*)

show *totalp-on N* $(\prec_c)$
using *clause.order.totalp-on-less* **.**
next

show $D \in N$
using *D-in*
by *simp*
qed

hence $(\bigcup C \in N.$ *epsilon N C*$) =$
$(\bigcup C \in \{C \in N.\ C \prec_c D\}.$ *epsilon N C*$) \cup$ *epsilon N D* $\cup$ $(\bigcup C \in \{C \in N.\ D \prec_c C\}.$ *epsilon N C*$)$
by *auto*

thus $(\bigcup C \in N.$ *epsilon N C*$) =$
rewrite-sys N D $\cup$ *epsilon N D* $\cup$ $(\bigcup C \in \{C \in N.\ D \prec_c C\}.$ *epsilon N C*$)$
by (*simp add*: *rewrite-sys-def*)
qed

lemma *split-Union-epsilon′*:
assumes *D-in*: $D \in N$
shows $(\bigcup C \in N.$ *epsilon N C*$) =$ *rewrite-sys N D* $\cup$ $(\bigcup C \in \{C \in N.\ D \preceq_c C\}.$ *epsilon N C*$)$
using *split-Union-epsilon*[*OF D-in*] *D-in*
by *auto*

lemma *lift-entailment-to-Union*:
fixes *N D*

assumes
D-in: $D \in N$ **and**
R_D*-entails-D*: *rewrite-sys* $N\ D \models D$
shows
$(\bigcup C \in N.\ \mathit{epsilon}\ N\ C) \models D$
using *lift-interp-entails*
by (*smt* (*verit*, *best*) *D-in* R_D*-entails-D UN-iff produces-imp-in-interp split-Union-epsilon′ subsetD sup-ge1 true-cls-def true-lit-iff*)

lemma *true-cls-if-productive-epsilon*:
assumes $A \in \mathit{epsilon}\ N\ C\ C \prec_c D$
shows *rewrite-sys* $N\ D \models C$
by (*meson assms epsilon-subset-if-less-cls mem-epsilonE in-mono is-strictly-maximal-def pos-literal-in-imp-true-cls*)

lemma *model-preconstruction*:
fixes
$N :: 't$ *clause set* **and**
$C :: 't$ *clause*
defines
$\mathit{entails} \equiv \lambda E\ C.\ E \models C$
assumes *saturated* N **and** $\{\#\} \notin N$ **and** *C-in*: $C \in N$
shows
$\mathit{epsilon}\ N\ C = \{\} \longleftrightarrow \mathit{entails}\ (\mathit{rewrite\text{-}sys}\ N\ C)\ C$
$(\bigcup D \in N.\ \mathit{epsilon}\ N\ D) \models C$
$D \in N \Longrightarrow C \prec_c D \Longrightarrow \mathit{entails}\ (\mathit{rewrite\text{-}sys}\ N\ D)\ C$
unfolding *atomize-all atomize-conj atomize-imp*
using *clause.order.wfp C-in*
proof (*induction C arbitrary*: *D rule*: *wfp-induct-rule*)
case (*less C*)
note *IH* = *less.IH*

from ‹$\{\#\} \notin N$› ‹$C \in N$› **have** $C \neq \{\#\}$
by *metis*

define $I :: 't$ *set* **where**
I = *rewrite-sys* $N\ C$

have *i*: $(\mathit{epsilon}\ N\ C = \{\}) \longleftrightarrow \mathit{entails}\ (\mathit{rewrite\text{-}sys}\ N\ C)\ C$
proof (*rule iffI*)

show $\mathit{entails}\ (\mathit{rewrite\text{-}sys}\ N\ C)\ C \Longrightarrow \mathit{epsilon}\ N\ C = \{\}$
unfolding *entails-def rewrite-sys-def*
by (*metis* (*no-types*) *empty-iff equalityI mem-epsilonE rewrite-sys-def subsetI*)
next
assume $\mathit{epsilon}\ N\ C = \{\}$

show $\mathit{entails}\ (\mathit{rewrite\text{-}sys}\ N\ C)\ C$
proof (*cases* $\exists A.\ \mathit{Neg}\ A \in\#\ C \wedge (\mathit{Neg}\ A \in\#\ \mathit{select}\ C \vee \mathit{select}\ C = \{\#\} \wedge$

is-maximal (*Neg A*) *C*))
case *ex-neg-lit-sel-or-max*: *True*

then obtain *A* **where**
Neg A ∈# *C* **and**
sel-or-max: *select C* = {#} ∧ *is-maximal* (*Neg A*) *C* ∨ *is-maximal* (*Neg A*)
(*select C*)
by (*metis* (*lifting*) *Neg-atm-of-iff empty-iff*
literal.order.ex-maximal-in-mset maximal-in-clause
mset-subset-eqD select-negative-literals select-subset
set-mset-empty)

then obtain *C′* **where**
C-def: *C* = *add-mset* (*Neg A*) *C′*
by (*metis mset-add*)

show *?thesis*
proof (*cases A* ∈ *rewrite-sys N C*)
case *True*

then obtain *D* **where**
A ∈ *epsilon N D* **and** *D* ∈ *N* **and** *D* ≺*c C*
unfolding *rewrite-sys-def*
by *auto*

then obtain *D′* **where**
D-def: *D* = *add-mset* (*Pos A*) *D′* **and**
sel-D: *select D* = {#} **and**
max-t-t′: *is-strictly-maximal* (*Pos A*) *D* **and**
¬ *entails* (*rewrite-sys N D*) *D*
by (*metis* (*lifting*) *IH empty-iff mem-epsilonE*)

define ι :: *′t clause inference* **where**
ι = *Infer* [*D*, *C*] (*C′* + *D′*)

have *resolution*: *resolution D C* (*C′* + *D′*)
proof (*rule resolutionI*)

show *C* = *add-mset* (*Neg A*) *C′*
by (*simp add*: *C-def*)
next

show *D* = *add-mset* (*Pos A*) *D′*
by (*simp add*: *D-def*)
next

show *D* ≺*c C*
using ‹*D* ≺*c C*› **.**
next

show *select C* = {#} ∧ *is-maximal* (*Neg A*) *C* ∨ *is-maximal* (*Neg A*) (*select C*)
using *sel-or-max*
by *auto*
next

show *select D* = {#}
using *sel-D* **by** *blast*
next

show *is-strictly-maximal* (*Pos A*) *D*
using *max-t-t'* **.**
qed *simp-all*

hence ι ∈ *G-Inf*
by (*auto simp only*: ι-*def G-Inf-def*)

moreover have ⋀*t*. *t* ∈ *set* (*prems-of* ι) ⟶ *t* ∈ *N*
using ‹*C* ∈ *N*› ‹*D* ∈ *N*›
by (*auto simp*: ι-*def*)

ultimately have ι ∈ *Inf-from N*
by (*auto simp*: *Inf-from-def*)

hence ι ∈ *Red-I N*
using ‹*saturated N*›
by (*auto simp*: *saturated-def*)

then obtain *DD* **where**
DD-subset: *DD* ⊆ *N* **and**
finite DD **and**
DD-entails-CD: *G-entails* (*insert D DD*) {*C'* + *D'*} **and**
ball-DD-lt-C: ∀ *D* ∈ *DD*. *D* ≺$_c$ *C*
unfolding *Red-I-def redundant-infer-def mem-Collect-eq*
by (*auto simp*: ι-*def*)

moreover have ∀ *D* ∈ *insert D DD*. *entails* (*rewrite-sys N C*) *D*
using *IH*[*THEN conjunct2*, *THEN conjunct2*, *rule-format*, *of* - *C*]
using ‹*C* ∈ *N*› ‹*D* ∈ *N*› ‹*D* ≺$_c$ *C* › *DD-subset ball-DD-lt-C*
by *blast*

ultimately have *entails* (*rewrite-sys N C*) (*C'* + *D'*)
using *DD-entails-CD*
unfolding *entails-def G-entails-def*
by (*simp add*: *I-def true-clss-def*)

moreover have ¬ *entails* (*rewrite-sys N D*) *D'*
using ‹¬ *entails* (*rewrite-sys N D*) *D*›

```
      using D-def entails-def
      by fastforce

    moreover have D' ≺c D
      by (metis (lifting) D-def add.comm-neutral add-mset-add-single
          add-mset-not-empty clause.order.multiset-extension-def
          empty-iff one-step-implies-multp set-mset-empty)

    moreover have ¬ entails (rewrite-sys N C) D'
      using D-def ‹A ∈ epsilon N D› ‹D ≺c C› entails-def
       false-cls-if-productive-epsilon less.prems
      by fastforce

    then show entails (rewrite-sys N C) C
      using C-def entails-def
      using calculation(1) by fastforce
  next
    case False

    thus ?thesis
      using ‹Neg A ∈# C›
      by (auto simp add: entails-def true-cls-def)
  qed
 next
  case False

  hence select C = {#}
    using select-subset select-negative-literals
     by (metis (no-types, opaque-lifting) Neg-atm-of-iff mset-subset-eqD multi-
set-nonemptyE)

    from False obtain A where Pos-A-in: Pos A ∈# C and max-Pos-A:
is-maximal (Pos A) C
    by (metis Neg-atm-of-iff ‹C ≠ {#}› ‹select C = {#}› literal.collapse(1)
        literal.order.ex-maximal-in-mset maximal-in-clause)

  then obtain C' where C-def: C = add-mset (Pos A) C'
    by (meson mset-add)

  show ?thesis
  proof (cases entails (rewrite-sys N C) C')
    case True

    then show ?thesis
      using C-def entails-def
      by force
  next
    case False
```

show *?thesis*
proof (*cases is-strictly-maximal* (*Pos A*) *C*)
case *strictly-maximal*: *True*
then show *?thesis*
using ‹*epsilon N C* = {}› ‹*select C* = {#}› *less.prems*
unfolding *epsilon-unfold*[*of N C*] *Collect-empty-eq*
unfolding *C-def entails-def*
by *blast*
next
case *False*

hence *count C* (*Pos A*) ≥ *2*
using *max-Pos-A*
using *C-def is-maximal-def is-strictly-maximal-def*
by *fastforce*

then obtain *C′* **where** *C-def*: *C* = *add-mset* (*Pos A*) (*add-mset* (*Pos A*) *C′*)
by (*metis two-le-countE*)

define ι :: *′t clause inference* **where**
ι = *Infer* [*C*] (*add-mset* (*Pos A*) *C′*)

have *eq-fact*: *factoring C* (*add-mset* (*Pos A*) *C′*)
proof (*rule factoringI*)

show *C* = *add-mset* (*Pos A*) (*add-mset* (*Pos A*) *C′*)
by (*simp add*: *C-def*)
next

show *select C* = {#}
using ‹*select C* = {#}› **.**
next

show *is-maximal* (*Pos A*) *C*
using *max-Pos-A* **.**
qed *simp-all*

hence ι ∈ *G-Inf*
by (*auto simp*: ι*-def G-Inf-def*)

moreover have ⋀*t*. *t* ∈ *set* (*prems-of* ι) ⟶ *t* ∈ *N*
using ‹*C* ∈ *N*›
by (*auto simp add*: ι*-def*)

ultimately have ι ∈ *Inf-from N*
by (*auto simp*: *Inf-from-def*)

hence ι ∈ *Red-I N*

```
      using ‹saturated N›
      by (auto simp: saturated-def)

    then obtain DD where
      DD-subset: DD ⊆ N and
      finite DD and
      DD-entails-concl: G-entails DD {add-mset (Pos A) C'} and
      ball-DD-lt-C: ∀ D ∈ DD. D ≺c C
      unfolding Red-I-def redundant-infer-def
      by (auto simp: ι-def)

    moreover have ∀ D ∈ DD. entails (rewrite-sys N C) D
      using IH[THEN conjunct2, rule-format, of - C]
      using ‹C ∈ N › DD-subset ball-DD-lt-C
      by blast

    ultimately have entails (rewrite-sys N C) (add-mset (Pos A) C')
      using DD-entails-concl
      unfolding G-entails-def entails-def
      by (simp add: I-def true-clss-def)

    then show ?thesis
      using C-def entails-def
      by fastforce
   qed
  qed
 qed
qed

moreover have iia: entails (⋃ (epsilon N ' N)) C
 using epsilon-eq-empty-or-singleton[of N C]
proof (elim disjE exE)
 assume epsilon N C = {}

 hence entails (rewrite-sys N C) C
  by (simp only: i)

 thus ?thesis
  using lift-entailment-to-Union[OF ‹C ∈ N›] entails-def
  by argo
next
 fix A
 assume epsilon N C = {A}

 hence eps: A ∈ epsilon N C
  by simp

 from eps have Pos A ∈# C
  unfolding epsilon.simps[of N C] mem-Collect-eq
```

```
    by force

  moreover from eps have A ∈ ⋃ (epsilon N `N)
    using ‹C ∈ N› UN-upper
    by fast

  ultimately show ?thesis
    using entails-def
    by blast
 qed

 moreover have iib: entails (rewrite-sys N D) C if D ∈ N and C ≺c D
  using epsilon-eq-empty-or-singleton[of N C]
 proof (elim disjE exE)
  assume epsilon N C = {}

  hence entails (rewrite-sys N C) C
    unfolding i
    by simp

  thus ?thesis
    using lift-interp-entails[OF ‹C ∈ N› - that] entails-def
    by argo
 next
  fix A assume epsilon N C = {A}

  thus ?thesis
    by (simp add: entails-def that(1,2) true-cls-if-productive-epsilon)
 qed

 ultimately show ?case
  by (simp add: entails-def)
qed

lemma model-construction:
 fixes
  N :: 't clause set and
  C :: 't clause
 defines entails ≡ λE C. E ⊨ C
 assumes saturated N and {#} ∉ N and C-in: C ∈ N
 shows entails (⋃ D ∈ N. epsilon N D) C
 unfolding atomize-conj atomize-imp
 using epsilon-eq-empty-or-singleton[of N C]
proof (elim disjE exE)
 assume epsilon N C = {}

 hence entails (rewrite-sys N C) C
  using model-preconstruction(1)[OF assms(2,3,4)]
  by (metis entails-def)
```

thus *?thesis*
using *lift-entailment-to-Union*(*1*)[*OF* ‹*C* ∈ *N*›]
by (*simp only*: *entails-def*)
next
fix *A* **assume** *epsilon N C* = {*A*}

thus *?thesis*
using *C-in assms*(*2*,*3*) *entails-def model-preconstruction*(*2*)
by *blast*
qed

1.6 Static Refutational Completeness

lemma *statically-complete*:
fixes *N* :: *'t clause set*
assumes *saturated N* **and** *G-entails N* {{#}}
shows {#} ∈ *N*
using ‹*G-entails N* {{#}}›
proof (*rule contrapos-pp*)
assume {#} ∉ *N*

define *I* :: *'t set* **where**
I = (⋃ *D* ∈ *N*. *epsilon N D*)

show ¬ *G-entails N G-Bot*
unfolding *G-entails-def not-all not-imp*
proof (*intro exI conjI*)

show *I* ⊨*s N*
unfolding *I-def*
using *model-construction*[*OF* ‹*saturated N*› ‹{#} ∉ *N*›]
by (*simp add*: *true-clss-def*)
next

show ¬ *I* ⊨*s G-Bot*
by *simp*
qed
qed

sublocale *statically-complete-calculus* **where**
Bot = *G-Bot* **and**
Inf = *G-Inf* **and**
entails = *G-entails* **and**
Red-I = *Red-I* **and**
Red-F = *Red-F*
using *statically-complete*
by *unfold-locales simp*

```
end

end
theory Ground-Ordered-Resolution-Soundness
  imports Ground-Ordered-Resolution
begin

lemma (in ground-ordered-resolution-calculus) soundness-ground-resolution:
  assumes
    step: resolution D C R
  shows G-entails {D, C} {R}
  using step
proof (cases D C R rule: resolution.cases)
  case (resolutionI L_C C' L_D D')

  show ?thesis
    unfolding G-entails-def true-clss-singleton
    unfolding true-clss-insert
  proof (intro allI impI, elim conjE)
    fix I :: 't set
    assume I ⊨ C and I ⊨ D

    then obtain K1 K2 :: 't literal where
      K1 ∈# C and I ⊨l K1 and K2 ∈# D and I ⊨l K2
      by (auto simp: true-cls-def)

    show I ⊨ R
    proof (cases K1 = L_C)
      case K1-def: True

      hence I ⊨l L_C
        using ‹I ⊨l K1›
        by simp

      show ?thesis
      proof (cases K2 = L_D)
        case K2-def: True

        hence I ⊨l L_D
          using ‹I ⊨l K2›
          by simp

        hence False
          using ‹I ⊨l L_C›
          by (simp add: local.resolutionI(3,4))

        thus ?thesis ..
      next
        case False
```

```
      hence K2 ∈# D'
        using ‹K2 ∈# D› unfolding resolutionI by simp

      hence I ⊨ D'
        using ‹I ⊨l K2› by blast

      thus ?thesis unfolding resolutionI by simp
    qed
  next
    case False

    hence K1 ∈# C'
      using ‹K1 ∈# C› unfolding resolutionI by simp

    hence I ⊨ C'
      using ‹I ⊨l K1› by blast

    thus ?thesis unfolding resolutionI by simp
  qed
 qed
qed

lemma (in ground-ordered-resolution-calculus) soundness-ground-factoring:
 assumes step: factoring C D
 shows G-entails {C} {D}
 using step
proof (cases C D rule: factoring.cases)
 case (factoringI L₁ C')

 show ?thesis
   unfolding G-entails-def true-clss-singleton
 proof (intro allI impI)
   fix I :: 't set
   assume I ⊨ C

   then obtain K :: 't literal where
     K ∈# C and I ⊨l K
     by (auto simp: true-cls-def)
```

```
    show I ⊨ D
    proof (cases K = L₁)
      case True

      hence I ⊨l L₁
        using ‹I ⊨l K›
        by metis

      thus ?thesis
        unfolding factoringI
        by (metis true-cls-add-mset)
    next
      case False

      hence K ∈# C'
        using ‹K ∈# C›
        unfolding factoringI
        by auto

      hence K ∈# D
        unfolding factoringI
        by simp

      thus ?thesis
        using ‹I ⊨l K›
        by blast
    qed
  qed
qed

sublocale ground-ordered-resolution-calculus ⊆ sound-inference-system where
  Inf = G-Inf and
  Bot = G-Bot and
  entails = G-entails
proof unfold-locales
  fix ι
  assume ι ∈ G-Inf

  then show G-entails (set (prems-of ι)) {concl-of ι}
    unfolding G-Inf-def
    using soundness-ground-resolution soundness-ground-factoring
    by auto
qed

end
theory Ordered-Resolution
  imports
    First-Order-Clause.Nonground-Order
```

First-Order-Clause.Nonground-Selection-Function
First-Order-Clause.Nonground-Typing
First-Order-Clause.Typed-Tiebreakers
Saturation-Framework.Lifting-to-Non-Ground-Calculi

Ground-Ordered-Resolution
begin

locale *ordered-resolution-calculus* =
witnessed-nonground-typing **where**
welltyped = *welltyped* **and** *term-to-ground* = *term-to-ground* :: *'t* ⇒ *'t*$_G$ +
nonground-order **where** *less*$_t$ = *less*$_t$ +
nonground-selection-function **where**
select = *select* **and** *atom-subst* = (·*t*) **and** *atom-vars* = *term.vars* **and**
atom-from-ground = *term.from-ground* **and** *atom-to-ground* = *term.to-ground* +
tiebreakers tiebreakers
for
select :: *'t select* **and**
less$_t$ :: *'t* ⇒ *'t* ⇒ *bool* **and**
tiebreakers :: (*'t*$_G$, *'t*) *tiebreakers* **and**
welltyped :: (*'v* :: *infinite*, *'ty*) *var-types* ⇒ *'t* ⇒ *'ty* ⇒ *bool*
begin

inductive *factoring* :: (*'t*, *'v*, *'ty*) *typed-clause* ⇒ (*'t*, *'v*, *'ty*) *typed-clause* ⇒ *bool*
where
factoringI:
C = *add-mset L*$_1$ (*add-mset L*$_2$ *C'*) ⟹
L$_1$ = (*Pos t*$_1$) ⟹
L$_2$ = (*Pos t*$_2$) ⟹
D = (*add-mset L*$_1$ *C'*) · *μ* ⟹
factoring (*𝒱*, *C*) (*𝒱*, *D*)
if
select C = {#}
is-maximal (*L*$_1$ ·*l μ*) (*C* · *μ*)
type-preserving-on (*clause.vars C*) *𝒱 μ*
term.is-imgu μ {{*t*$_1$, *t*$_2$}}

inductive *resolution* ::
(*'t*, *'v*, *'ty*) *typed-clause* ⇒ (*'t*, *'v*, *'ty*) *typed-clause* ⇒ (*'t*, *'v*, *'ty*) *typed-clause* ⇒ *bool*
where
resolutionI:
C = *add-mset L*$_1$ *C'* ⟹
D = *add-mset L*$_2$ *D'* ⟹
L$_1$ = (*Neg t*$_1$) ⟹
L$_2$ = (*Pos t*$_2$) ⟹
R = (*C'* · *ϱ*$_1$ + *D'* · *ϱ*$_2$) · *μ* ⟹
resolution (*𝒱*$_2$, *D*) (*𝒱*$_1$, *C*) (*𝒱*$_3$, *R*)

if
infinite-variables-per-type $\mathcal{V}_1$
infinite-variables-per-type $\mathcal{V}_2$
term.is-renaming ϱ_1
term.is-renaming ϱ_2
clause.vars $(C \cdot \varrho_1) \cap$ *clause.vars* $(D \cdot \varrho_2) = \{\}$
type-preserving-on (*clause.vars* $(C \cdot \varrho_1) \cup$ *clause.vars* $(D \cdot \varrho_2))$ $\mathcal{V}_3$ μ
term.is-imgu μ $\{\{t_1 \cdot t \varrho_1, t_2 \cdot t \varrho_2\}\}$
$\neg$ $(C \cdot \varrho_1 \odot \mu \preceq_c D \cdot \varrho_2 \odot \mu)$
select $C = \{\#\} \Longrightarrow$ *is-maximal* $(L_1 \cdot l \varrho_1 \odot \mu)$ $(C \cdot \varrho_1 \odot \mu)$
select $C \neq \{\#\} \Longrightarrow$ *is-maximal* $(L_1 \cdot l \varrho_1 \odot \mu)$ $((\textit{select } C) \cdot \varrho_1 \odot \mu)$
select $D = \{\#\}$
is-strictly-maximal $(L_2 \cdot l \varrho_2 \odot \mu)$ $(D \cdot \varrho_2 \odot \mu)$
$\forall x \in$ *clause.vars* C. $\mathcal{V}_1$ $x = \mathcal{V}_3$ (*term.rename* ϱ_1 x)
$\forall x \in$ *clause.vars* D. $\mathcal{V}_2$ $x = \mathcal{V}_3$ (*term.rename* ϱ_2 x)
type-preserving-on (*clause.vars* C) $\mathcal{V}_1$ ϱ_1
type-preserving-on (*clause.vars* D) $\mathcal{V}_2$ ϱ_2

abbreviation *factoring-inferences* **where**
factoring-inferences $\equiv$ { *Infer* $[C]$ D | C D. *factoring* C D}

abbreviation *resolution-inferences* **where**
resolution-inferences $\equiv$ { *Infer* $[D, C]$ R | D C R. *resolution* D C R}

definition *inferences* :: ('*t*, '*v*, '*ty*) *typed-clause inference set* **where**
inferences $\equiv$ *resolution-inferences* $\cup$ *factoring-inferences*

abbreviation *bottom*$_F$ :: ('*t*, '*v*, '*ty*) *typed-clause set* ($\bot_F$) **where**
bottom$_F$ $\equiv$ {($\mathcal{V}$, {#}) | $\mathcal{V}$. *infinite-variables-per-type* $\mathcal{V}$ }

end

end
theory *Grounded-Ordered-Resolution*
imports
Ordered-Resolution
Ground-Ordered-Resolution

First-Order-Clause.Grounded-Selection-Function
First-Order-Clause.Nonground-Inference
Saturation-Framework.Lifting-to-Non-Ground-Calculi

Polynomial-Factorization.Missing-List
begin

locale *grounded-ordered-resolution-calculus* =
ordered-resolution-calculus **where**
select = *select* **and** *welltyped* = *welltyped* **and** *term-from-ground* = *term-from-ground*
:: 't_G $\Rightarrow$ 't +

grounded-selection-function **where**
select = *select* **and**
atom-subst = $(\cdot t)$ **and**
atom-vars = *term.vars* **and**
atom-from-ground = *term.from-ground* **and**
atom-to-ground = *term.to-ground* **and**
is-ground-instance = *is-ground-instance*
for
select :: $'t$ *select* **and**
welltyped :: $('v :: infinite, 'ty)$ *var-types* $\Rightarrow$ $'t \Rightarrow 'ty \Rightarrow$ *bool*
begin

sublocale *ground*: *ground-ordered-resolution-calculus* **where**
$less_t = (\prec_{tG})$ **and** *select* = $select_G$
rewrites
multiset-extension.multiset-extension $(\prec_{tG})$ *ground.literal-to-mset* = $(\prec_{lG})$ **and**
multiset-extension.multiset-extension $(\prec_{lG})$ $(\lambda x.\ x) = (\prec_{cG})$ **and**
$\bigwedge l_G\ C_G$. *ground.is-maximal* $l_G\ C_G \longleftrightarrow$ *ground-is-maximal* $l_G\ C_G$ **and**
$\bigwedge l_G\ C_G$. *ground.is-strictly-maximal* $l_G\ C_G \longleftrightarrow$ *ground-is-strictly-maximal* $l_G\ C_G$
unfolding *is-maximal-rewrite*[*symmetric*] *is-strictly-maximal-rewrite*[*symmetric*]
by *unfold-locales simp-all*

abbreviation *is-inference-ground-instance-one-premise* **where**
is-inference-ground-instance-one-premise $D\ C\ \iota_G\ \gamma \equiv$
case $(D,\ C)$ *of* $((\mathcal{V}',\ D),\ (\mathcal{V},\ C)) \Rightarrow$
inference.is-ground $(Infer\ [D]\ C \cdot\iota\ \gamma) \wedge$
ι_G = *inference.to-ground* $(Infer\ [D]\ C \cdot\iota\ \gamma) \wedge$
type-preserving-on $(clause.vars\ C)\ \mathcal{V}\ \gamma \wedge$
$\mathcal{V} = \mathcal{V}' \wedge$
infinite-variables-per-type $\mathcal{V}$

abbreviation *is-inference-ground-instance-two-premises* **where**
is-inference-ground-instance-two-premises $D\ E\ C\ \iota_G\ \gamma\ \varrho_1\ \varrho_2 \equiv$
case $(D,\ E,\ C)$ *of* $((\mathcal{V}_2,\ D),\ (\mathcal{V}_1,\ E),\ (\mathcal{V}_3,\ C)) \Rightarrow$
term.is-renaming $\varrho_1 \wedge$
term.is-renaming $\varrho_2 \wedge$
$clause.vars\ (E \cdot \varrho_1) \cap clause.vars\ (D \cdot \varrho_2) = \{\} \wedge$
inference.is-ground $(Infer\ [D \cdot \varrho_2,\ E \cdot \varrho_1]\ C \cdot\iota\ \gamma) \wedge$
ι_G = *inference.to-ground* $(Infer\ [D \cdot \varrho_2,\ E \cdot \varrho_1]\ C \cdot\iota\ \gamma) \wedge$
type-preserving-on $(clause.vars\ C)\ \mathcal{V}_3\ \gamma \wedge$
infinite-variables-per-type $\mathcal{V}_1 \wedge$
infinite-variables-per-type $\mathcal{V}_2 \wedge$
infinite-variables-per-type $\mathcal{V}_3$

abbreviation *is-inference-ground-instance* **where**
is-inference-ground-instance $\iota\ \iota_G\ \gamma \equiv$
(*case* ι *of*
$Infer\ [D]\ C \Rightarrow$ *is-inference-ground-instance-one-premise* $D\ C\ \iota_G\ \gamma$

```
    | Infer [D, E] C ⇒ ∃ ϱ1 ϱ2. is-inference-ground-instance-two-premises D E C
ιG γ ϱ1 ϱ2
    | - ⇒ False)
  ∧ ιG ∈ ground.G-Inf

definition inference-ground-instances where
  inference-ground-instances ι = { ιG | ιG γ. is-inference-ground-instance ι ιG γ }

lemma is-inference-ground-instance:
  is-inference-ground-instance ι ιG γ ⟹ ιG ∈ inference-ground-instances ι
  unfolding inference-ground-instances-def
  by blast

lemma is-inference-ground-instance-one-premise:
  assumes is-inference-ground-instance-one-premise D C ιG γ ιG ∈ ground.G-Inf
  shows ιG ∈ inference-ground-instances (Infer [D] C)
  using assms
  unfolding inference-ground-instances-def
  by auto

lemma is-inference-ground-instance-two-premises:
   assumes is-inference-ground-instance-two-premises D E C ιG γ ϱ1 ϱ2 ιG ∈
ground.G-Inf
  shows ιG ∈ inference-ground-instances (Infer [D, E] C)
  using assms
  unfolding inference-ground-instances-def
  by auto

lemma ground-inference_concl-in-ground-instances:
  assumes ιG ∈ inference-ground-instances ι
  shows concl-of ιG ∈ uncurried-ground-instances (concl-of ι)
proof −
  obtain premises C V where
    ι: ι = Infer premises (C, V)
    using Calculus.inference.exhaust
    by (metis prod.collapse)

  show ?thesis
    using assms
    unfolding ι inference-ground-instances-def ground-instances-def
    by (cases premises rule: list-4-cases) auto
qed

lemma ground-inference-red-in-ground-instances-of-concl:
  assumes ιG ∈ inference-ground-instances ι
  shows ιG ∈ ground.Red-I (uncurried-ground-instances (concl-of ι))
proof −
  from assms have ιG ∈ ground.G-Inf
    unfolding inference-ground-instances-def
```

by *blast*

moreover have *concl-of* $\iota_G \in$ *uncurried-ground-instances* (*concl-of* ι)
using *assms ground-inference_concl-in-ground-instances*
by *auto*

ultimately show $\iota_G \in$ *ground.Red-I* (*uncurried-ground-instances* (*concl-of* ι))
using *ground.Red-I-of-Inf-to-N*
by *blast*
qed

sublocale *lifting*:
tiebreaker-lifting
$\bot_F$
inferences
ground.G-Bot
ground.G-entails
ground.G-Inf
ground.GRed-I
ground.GRed-F
uncurried-ground-instances
Some $\circ$ *inference-ground-instances*
typed-tiebreakers
proof (*unfold-locales*; (*intro impI typed-tiebreakers.wfp typed-tiebreakers.transp*)*?*)

show $\bot_F \neq \{\}$
using *obtain-infinite-variables-per-type-on''*[*of* {}]
by *auto*
next
fix *bottom*
assume *bottom* $\in \bot_F$

then show *uncurried-ground-instances bottom* $\neq \{\}$
unfolding *ground-instances-def*
by *fastforce*
next
fix *bottom*
assume *bottom* $\in \bot_F$

then show *uncurried-ground-instances bottom* $\subseteq$ *ground.G-Bot*
unfolding *ground-instances-def*
by *auto*
next
fix C :: ('*t*, '*v*, '*ty*) *typed-clause*

assume *uncurried-ground-instances* $C \cap$ *ground.G-Bot* $\neq \{\}$

moreover then have *snd* $C = \{\#\}$
unfolding *ground-instances-def*

by *simp*

then have $C = (fst\ C, \{\#\})$
by (*metis split-pairs*)

ultimately show $C \in \bot_F$
unfolding *ground-instances-def*
by *blast*
next
fix $\iota :: ('t, 'v, 'ty)$ *typed-clause inference*

show *the* ((*Some* ∘ *inference-ground-instances*) ι) ⊆
ground.GRed-I (*uncurried-ground-instances* (*concl-of* ι))
using *ground-inference-red-in-ground-instances-of-concl*
by *auto*
qed
end

context *ordered-resolution-calculus*
begin

abbreviation *grounded-inference-ground-instances* **where**
grounded-inference-ground-instances $select_G \equiv$
grounded-ordered-resolution-calculus.inference-ground-instances
(⊙) *Var* (·*t*) *term.vars term.to-ground* ($\prec_t$) *term.from-ground* $select_G$ *welltyped*

sublocale
lifting-intersection
inferences
{{#}}
$select_{Gs}$
ground-ordered-resolution-calculus.G-Inf ($\prec_{tG}$)
λ-. *ground-ordered-resolution-calculus.G-entails*
ground-ordered-resolution-calculus.GRed-I ($\prec_{tG}$)
λ-. *ground-ordered-resolution-calculus.GRed-F* ($\prec_{tG}$)
$\bot_F$
λ-. *uncurried-ground-instances*
λ$select_G$. *Some* ∘ *grounded-inference-ground-instances* $select_G$
typed-tiebreakers
proof (*unfold-locales*; (*intro ballI*)*?*)

show $select_{Gs} \neq \{\}$
using $select_G$*-simple*
unfolding $select_{Gs}$*-def*
by *blast*
next
fix $select_G$
assume $select_G \in select_{Gs}$

then interpret *grounded-ordered-resolution-calculus*
where $select_G = select_G$
by *unfold-locales* (*simp add*: $select_{Gs}$-*def*)

show *consequence-relation ground.G-Bot ground.G-entails*
using *ground.consequence-relation-axioms* **.**

show *tiebreaker-lifting*
$\bot_F$
inferences
ground.G-Bot
ground.G-entails
ground.G-Inf
ground.GRed-I
ground.GRed-F
uncurried-ground-instances
(*Some* ∘ *inference-ground-instances*)
typed-tiebreakers
by *unfold-locales*
qed

end

end
theory *Ordered-Resolution-Soundness*
imports *Grounded-Ordered-Resolution*
begin

1.7 Soundness

context *grounded-ordered-resolution-calculus*
begin

notation *lifting.entails-*$\mathcal{G}$ (**infix** $\models_F$ *50*)

lemma *factoring-sound*:
assumes *factoring*: *factoring C D*
shows $\{C\} \models_F \{D\}$
using *factoring*
proof (*cases C D rule*: *factoring.cases*)
case (*factoringI C* L_1 μ $\mathcal{V}$ t_1 t_2 L_2 C' *D*)

{
fix $I :: 't_G\ set$ **and** $\gamma :: 'v \Rightarrow 't$

assume
entails-ground-instances: $\forall\, C_G \in$ *ground-instances* $\mathcal{V}$ *C*. $I \models C_G$ **and**
D-is-ground: *clause.is-ground* $(D \cdot \gamma)$ **and**
type-preserving-γ: *type-preserving-on* (*clause.vars D*) $\mathcal{V}$ γ **and**

$\mathcal{V}$: *infinite-variables-per-type* $\mathcal{V}$

obtain γ' **where**
γ'*-is-ground-subst*: *term.is-ground-subst* γ' **and**
type-preserving-γ': *type-preserving* $\mathcal{V}$ γ' **and**
γ'-γ: $\forall x \in$ *clause.vars* D. $\gamma\ x = \gamma'\ x$
using *clause.type-preserving-ground-subst-extension*[*OF D-is-ground type-preserving-*γ]
.

let *?*C_G = *clause.to-ground* $(C \cdot \mu \cdot \gamma')$
let *?*$C_G{}'$ = *clause.to-ground* $(C' \cdot \mu \cdot \gamma')$
let *?*l_{G1} = *literal.to-ground* $(L_1 \cdot l\ \mu \cdot l\ \gamma')$
let *?*l_{G2} = *literal.to-ground* $(L_2 \cdot l\ \mu \cdot l\ \gamma')$
let *?*t_{G1} = *term.to-ground* $(t_1 \cdot t\ \mu \cdot t\ \gamma')$
let *?*t_{G2} = *term.to-ground* $(t_2 \cdot t\ \mu \cdot t\ \gamma')$
let *?*D_G = *clause.to-ground* $(D \cdot \gamma')$

have *type-preserving-*μ: *type-preserving-on* (*clause.vars* C) $\mathcal{V}$ μ
using *factoringI*(*5*)
by *blast*

have [*simp*]: *?*t_{G2} = *?*t_{G1}
using *factoringI*(*6*) *term.is-imgu-unifies-pair*
by *metis*

have [*simp*]: $t_1 \cdot t\ \mu \cdot t\ \gamma' = t_1 \cdot t\ \mu \cdot t\ \gamma$
using γ'-γ *term.subst-eq*
unfolding *factoringI*
by *fastforce*

have *?*$C_G \in$ *ground-instances* $\mathcal{V}$ C
proof(*unfold ground-instances-def mem-Collect-eq fst-conv snd-conv*,
intro exI, *intro conjI* $\mathcal{V}$)

show *clause.to-ground* $(C \cdot \mu \cdot \gamma')$ = *clause.to-ground* $(C \cdot \mu \odot \gamma')$
by *simp*
next

show *clause.is-ground* $(C \cdot \mu \odot \gamma')$
using γ'*-is-ground-subst clause.is-ground-subst-is-ground*
by *auto*
next

show *type-preserving-on* (*clause.vars* C) $\mathcal{V}$ $(\mu \odot \gamma')$
using
type-preserving-μ
type-preserving-γ'
γ'*-is-ground-subst*
term.type-preserving-ground-compose-ground-subst

```
      by presburger
    qed

    then have I ⊨ ?C_G
      using entails-ground-instances
      by blast

    then have I ⊨ clause.to-ground (D · γ)
      using clause.subst-eq[OF γ'-γ[rule-format]]
      unfolding factoringI
      by auto
  }

  then show ?thesis
    unfolding
      factoringI(1, 2)
      ground.G-entails-def
      true-clss-def
      ground-instances-def
    by auto
qed

lemma resolution-sound:
  assumes resolution: resolution D C R
  shows {C, D} ⊨_F {R}
  using resolution
proof (cases D C R rule: resolution.cases)
  case (resolutionI V₁ V₂ ϱ₁ ϱ₂ C D V₃ μ t₁ t₂ L₁ L₂ C' D' R)

  {
    fix I :: 't_G set and γ :: 'v ⇒ 't

    assume
      C-entails-ground-instances: ∀ C_G ∈ ground-instances V₁ C. I ⊨ C_G and
      D-entails-ground-instances: ∀ D_G ∈ ground-instances V₂ D. I ⊨ D_G and
      R-is-ground: clause.is-ground (R · γ) and
      type-preserving-γ: type-preserving-on (clause.vars R) V₃ γ

    obtain γ' where
      γ'-is-ground-subst: term.is-ground-subst γ' and
      type-preserving-γ': type-preserving V₃ γ' and
      γ'-γ: ∀ x ∈ clause.vars R. γ x = γ' x
    using clause.type-preserving-ground-subst-extension[OF R-is-ground type-preserving-γ]
.

    let ?C_G = clause.to-ground (C · ϱ₁ · μ · γ')
    let ?D_G = clause.to-ground (D · ϱ₂ · μ · γ')

    let ?l_G1 = literal.to-ground (L₁ ·l ϱ₁ ·l μ ·l γ')
```

let *?l*$_{G2}$ = *literal.to-ground* (*L*$_2$ ·*l* ϱ_2 ·*l* μ ·*l* γ')

let *?C*$_G$' = *clause.to-ground* (*C'* · ϱ_1 · μ · γ')
let *?D*$_G$' = *clause.to-ground* (*D'* · ϱ_2 · μ · γ')

let *?t*$_{G1}$ = *term.to-ground* (*t*$_1$ ·*t* ϱ_1 ·*t* μ ·*t* γ')
let *?t*$_{G2}$ = *term.to-ground* (*t*$_2$ ·*t* ϱ_2 ·*t* μ ·*t* γ')

let *?R*$_G$ = *clause.to-ground* (*R* · γ')

have μ-γ'-*is-ground-subst*: *term.is-ground-subst* (μ ⊙ γ')
using *term.is-ground-subst-comp-right*[*OF* γ'-*is-ground-subst*] **.**

have *type-preserving-*μ: *type-preserving-on* (*clause.vars* (*C* · ϱ_1) ∪ *clause.vars* (*D* · ϱ_2)) $\mathcal{V}_3$ μ
using *resolutionI*(*9*)
by *blast*

have *type-preserving-*μ-γ:
type-preserving-on (*clause.vars* (*C* · ϱ_1) ∪ *clause.vars* (*D* · ϱ_2)) $\mathcal{V}_3$ (μ ⊙ γ')
using
type-preserving-γ'
type-preserving-μ
γ'-*is-ground-subst*
term.type-preserving-ground-compose-ground-subst
by *presburger*

note *type-preserving-*ϱ-μ-γ = *term.renaming-ground-subst*[*OF* - μ-γ'-*is-ground-subst*]

have *?C*$_G$ ∈ *ground-instances* $\mathcal{V}_1$ *C*
proof(
unfold ground-instances-def mem-Collect-eq fst-conv snd-conv,
intro exI, *intro conjI resolutionI*)

show *clause.to-ground* (*C* · ϱ_1 · μ · γ') = *clause.to-ground* (*C* · ϱ_1 ⊙ μ ⊙ γ')
by *simp*
next

show *clause.is-ground* (*C* · ϱ_1 ⊙ μ ⊙ γ')
using γ'-*is-ground-subst clause.is-ground-subst-is-ground*
by *auto*
next

show *type-preserving-on* (*clause.vars C*) $\mathcal{V}_1$ (ϱ_1 ⊙ μ ⊙ γ')
using
type-preserving-μ-γ
type-preserving-ϱ-μ-γ[*OF resolutionI*(*6*, *18*) - *resolutionI*(*16*)]
by (*simp add*: *term.assoc clause.vars-subst*)
qed

```
then have entails-C_G: I ⊨ ?C_G
  using C-entails-ground-instances
  by blast

have ?D_G ∈ ground-instances V_2 D
proof(
    unfold ground-instances-def mem-Collect-eq fst-conv snd-conv,
    intro exI, intro conjI resolutionI)

  show clause.to-ground (D · ϱ_2 · μ · γ') = clause.to-ground (D · ϱ_2 ⊙ μ ⊙ γ')
    by simp
next

  show clause.is-ground (D · ϱ_2 ⊙ μ ⊙ γ')
    using γ'-is-ground-subst clause.is-ground-subst-is-ground
    by auto
next

  show type-preserving-on (clause.vars D) V_2 (ϱ_2 ⊙ μ ⊙ γ')
    using
      type-preserving-μ-γ
      type-preserving-ϱ-μ-γ[OF resolutionI(7, 19) - resolutionI(17)]
    by (simp add: term.assoc clause.vars-subst)
qed

then have entails-D_G: I ⊨ ?D_G
  using D-entails-ground-instances
  by blast

have I ⊨ clause.to-ground (R · γ')
proof −
  have [simp]: ?t_G1 = ?t_G2
    using resolutionI(10) term.is-imgu-unifies-pair
    by metis

  have [simp]: ?l_G1 = Neg ?t_G1
    unfolding resolutionI
    by simp

  have [simp]: ?l_G2 = Pos ?t_G2
    unfolding resolutionI
    by simp

  have [simp]: ?C_G = add-mset ?l_G1 ?C_G'
    unfolding resolutionI
    by simp

  have [simp]: ?D_G = add-mset ?l_G2 ?D_G'
```

```
      unfolding resolutionI
      by simp

    have ¬ I ⊨l ?l_G1 ∨ ¬ I ⊨l ?l_G2
      by simp

    then have I ⊨ ?C_G' ∨ I ⊨ ?D_G'
      using entails-C_G entails-D_G
      by force

    moreover have ?R_G = ?C_G' + ?D_G'
      unfolding resolutionI
      by simp

    ultimately show ?thesis
      by auto
  qed

  then have I ⊨ clause.to-ground (R · γ)
    by (metis γ'-γ clause.subst-eq)
}

then show ?thesis
 unfolding ground.G-entails-def ground-instances-def true-clss-def resolutionI(1–3)
  by auto
qed

end

sublocale grounded-ordered-resolution-calculus ⊆ sound-inference-system inferences
⊥_F (⊨_F)
proof unfold-locales
fix ι

  assume ι ∈ inferences

  then show set (prems-of ι) ⊨_F {concl-of ι}
    using
      factoring-sound
      resolution-sound
    unfolding inferences-def ground.G-entails-def
    by auto
qed

sublocale ordered-resolution-calculus ⊆ sound-inference-system inferences ⊥_F en-
tails-𝒢
proof unfold-locales
  obtain select_G where select_G: select_G ∈ select_Gs
    using Q-nonempty by blast
```

then interpret *grounded-ordered-resolution-calculus*
where $select_G = select_G$
by *unfold-locales* (*simp add*: $select_{Gs}$-*def*)

fix ι
assume $\iota \in inferences$

then show *entails-*$\mathcal{G}$ (*set* (*prems-of* ι)) {*concl-of* ι}
unfolding *entails-def*
using *sound*
by *blast*
qed

end
theory *Ordered-Resolution-Completeness*
imports
Grounded-Ordered-Resolution
Ground-Ordered-Resolution-Completeness
begin

2 Completeness

context *grounded-ordered-resolution-calculus*
begin

2.1 Liftings

lemma *factoring-lifting*:
fixes
D_G C_G :: $'t_G$ *clause* **and**
D C :: $'t$ *clause* **and**
γ :: $'v \Rightarrow 't$
defines
[*simp*]: $D_G \equiv$ *clause.to-ground* $(D \cdot \gamma)$ **and**
[*simp*]: $C_G \equiv$ *clause.to-ground* $(C \cdot \gamma)$
assumes
ground-factoring: *ground.factoring* D_G C_G **and**
D-grounding: *clause.is-ground* $(D \cdot \gamma)$ **and**
C-grounding: *clause.is-ground* $(C \cdot \gamma)$ **and**
select: *clause.from-ground* $(select_G\ D_G) = (select\ D) \cdot \gamma$ **and**
type-preserving-γ: *type-preserving-on* (*clause.vars* D) $\mathcal{V}$ γ **and**
$\mathcal{V}$: *infinite-variables-per-type* $\mathcal{V}$
obtains C'
where
factoring $(\mathcal{V}, D)$ $(\mathcal{V}, C')$
Infer $[D_G]$ $C_G \in$ *inference-ground-instances* (*Infer* $[(\mathcal{V}, D)]$ $(\mathcal{V}, C')$)
$C' \cdot \gamma = C \cdot \gamma$
using *ground-factoring*

proof(*cases* D_G C_G *rule*: *ground.factoring.cases*)
case *ground-factoringI*: (*factoringI* L_G $D_G{}'$ t_G)

have $D \neq \{\#\}$
using *ground-factoringI*(*1*)
by *auto*

then obtain l_1 **where**
l_1-*is-maximal*: *is-maximal* l_1 D **and**
l_1-γ-*is-maximal*: *is-maximal* (l_1 $\cdot l$ γ) ($D \cdot \gamma$)
using *that obtain-maximal-literal D-grounding*
by *blast*

obtain t_1 **where**
l_1: l_1 = (*Pos* t_1) **and**
l_1-γ: l_1 $\cdot l$ γ = (*Pos* (*term.from-ground* t_G)) **and**
t_1-γ: t_1 $\cdot t$ γ = *term.from-ground* t_G
proof−
have *is-maximal* (*literal.from-ground* L_G) ($D \cdot \gamma$)
using *D-grounding ground-factoringI*(*4*)
by *auto*

then have l_1 $\cdot l$ γ = (*Pos* (*term.from-ground* t_G))
unfolding *ground-factoringI*(*2*)
using *unique-maximal-in-ground-clause*[*OF D-grounding* l_1-γ-*is-maximal*]
by *simp*

then show *?thesis*
using *that*
unfolding *ground-factoringI*(*2*)
by (*metis Neg-atm-of-iff clause-safe-unfolds*(*9*) *literal.collapse*(*1*) *literal.sel*(*1*)
subst-polarity-stable(*2*))
qed

obtain l_2 D' **where**
l_2-γ: l_2 $\cdot l$ γ = *Pos* (*term.from-ground* t_G) **and**
D: D = *add-mset* l_1 (*add-mset* l_2 D')
proof−

obtain D'' **where** D: D = *add-mset* l_1 D''
using *maximal-in-clause*[*OF* l_1-*is-maximal*]
by (*meson multi-member-split*)

moreover have $D \cdot \gamma$ = *clause.from-ground* (*add-mset* L_G (*add-mset* L_G $D_G{}'$))
using *ground-factoringI*(*1*) C_G-*def*
by (*metis* D_G-*def D-grounding clause.to-ground-inverse*)

ultimately have $D'' \cdot \gamma$ = *add-mset* (*literal.from-ground* L_G) (*clause.from-ground* $D_G{}'$)

```
    using l₁-γ
    by (simp add: ground-factoringI(2))

  then obtain l₂ where l₂ ·l γ = Pos (term.from-ground t_G) l₂ ∈# D''
    unfolding clause.subst-def ground-factoringI
    using msed-map-invR
    by force

  then show ?thesis
    using that
    unfolding D
    by (metis mset-add)
qed

then obtain t₂ where
  l₂: l₂ = (Pos t₂) and
  t₂-γ: t₂ ·t γ = term.from-ground t_G
  unfolding ground-factoringI(2)
 by (metis clause-safe-unfolds(9) is-pos-def literal.sel(1) subst-polarity-stable(2))

have D'-γ: D' · γ = clause.from-ground D_G'
  using D D-grounding ground-factoringI(1,2,3) l₁-γ l₂-γ
  by force

obtain μ σ where
  γ: γ = μ ⊙ σ and
  type-preserving-μ: type-preserving-on (clause.vars D) 𝒱 μ and
  imgu: term.is-imgu μ {{t₁, t₂}}
proof (rule obtain-type-preserving-on-imgu[OF - that], intro conjI)

  show t₁ ·t γ = t₂ ·t γ
    using t₁-γ t₂-γ
    by argo
next

  show type-preserving-on (term.vars t₁ ∪ term.vars t₂) 𝒱 γ
    using type-preserving-γ
    unfolding D l₁ l₂
    by auto
qed

let ?C'' = add-mset l₁ D'
let ?C' = ?C'' · μ

show ?thesis
proof(rule that)

  show factoring: factoring (𝒱, D) (𝒱, ?C')
  proof (rule factoringI; (rule D imgu type-preserving-μ refl 𝒱)?)
```

```
  show select D = {#}
    using ground-factoringI(3) select
    by simp
next

  have l₁ ·l μ ∈# D · μ
    using l₁-is-maximal clause.subst-in-to-set-subst maximal-in-clause
    by blast

  then show is-maximal (l₁ ·l μ) (D · μ)
    using is-maximal-if-grounding-is-maximal D-grounding l₁-γ-is-maximal
    unfolding γ
    by auto
next

  show D = add-mset l₁ (add-mset (Pos t₂) D')
    unfolding D l₂ ..
next
  show l₁ = Pos t₁
    using l₁ .
qed

show C'-γ: ?C' · γ = C · γ
proof−
  have term.is-idem μ
    using imgu
    unfolding term.is-imgu-iff-is-idem-and-is-mgu
    by blast

  then have μ-γ: μ ⊙ γ = γ
    unfolding γ term.is-idem-def term.assoc[symmetric]
    by argo

  have C · γ = clause.from-ground (add-mset (Pos t_G) D_G')
  using ground-factoringI(5) clause.to-ground-eq[OF C-grounding clause.ground-is-ground]
    unfolding C_G-def
    by (metis clause.from-ground-inverse ground-factoringI(2))

  also have ... = ?C'' · γ
    using t₁-γ D'-γ l₁-γ
    by auto

  also have ... = ?C' · γ
    unfolding clause.subst-comp-subst[symmetric] μ-γ ..

  finally show ?thesis ..
qed
```

```
    show Infer [D_G] C_G ∈ inference-ground-instances (Infer [(𝒱, D)] (𝒱, ?C'))
    proof (rule is-inference-ground-instance-one-premise)
      show is-inference-ground-instance-one-premise (𝒱, D) (𝒱, ?C') (Infer [D_G]
C_G) γ
      proof(unfold split, intro conjI; (rule refl 𝒱)?)

        show inference.is-ground (Infer [D] ?C' ·ι γ)
          using C-grounding D-grounding C'-γ
          by auto
      next

        show Infer [D_G] C_G = inference.to-ground (Infer [D] ?C' ·ι γ)
          using C'-γ
          by simp
      next

        have clause.vars ?C' ⊆ clause.vars D
          using clause.variables-in-base-imgu[OF imgu, of ?C'']
          unfolding D l_1 l_2
          by auto

        then show type-preserving-on (clause.vars ?C') 𝒱 γ
          using type-preserving-γ
          by blast
      qed

      show Infer [D_G] C_G ∈ ground.G-Inf
        unfolding ground.G-Inf-def
        using ground-factoring
        by blast
    qed
  qed
qed

lemma resolution-lifting:
  fixes
    C_G D_G R_G :: 't_G clause and
    C D R :: 't clause and
    γ ϱ_1 ϱ_2 :: 'v ⇒ 't and
    𝒱_1 𝒱_2 :: ('v, 'ty) var-types
  defines
    [simp]: C_G ≡ clause.to-ground (C · ϱ_1 ⊙ γ) and
    [simp]: D_G ≡ clause.to-ground (D · ϱ_2 ⊙ γ) and
    [simp]: R_G ≡ clause.to-ground (R · γ) and
    [simp]: N_G ≡ ground-instances 𝒱_1 C ∪ ground-instances 𝒱_2 D and
    [simp]: ι_G ≡ Infer [D_G, C_G] R_G
  assumes
    ground-resolution: ground.resolution D_G C_G R_G and
    ϱ_1: term.is-renaming ϱ_1 and
```

ϱ_2: *term.is-renaming* ϱ_2 **and**
rename-apart: *clause.vars* $(C \cdot \varrho_1) \cap$ *clause.vars* $(D \cdot \varrho_2) = \{\}$ **and**
C-grounding: *clause.is-ground* $(C \cdot \varrho_1 \odot \gamma)$ **and**
D-grounding: *clause.is-ground* $(D \cdot \varrho_2 \odot \gamma)$ **and**
R-grounding: *clause.is-ground* $(R \cdot \gamma)$ **and**
select-from-C: *clause.from-ground* $(select_G\ C_G) = (select\ C) \cdot \varrho_1 \odot \gamma$ **and**
select-from-D: *clause.from-ground* $(select_G\ D_G) = (select\ D) \cdot \varrho_2 \odot \gamma$ **and**
type-preserving-ϱ_1*-*γ: *type-preserving-on* (*clause.vars C*) $\mathcal{V}_1$ $(\varrho_1 \odot \gamma)$ **and**
type-preserving-ϱ_2*-*γ: *type-preserving-on* (*clause.vars D*) $\mathcal{V}_2$ $(\varrho_2 \odot \gamma)$ **and**
type-preserving-ϱ_1: *type-preserving-on* (*clause.vars C*) $\mathcal{V}_1$ ϱ_1 **and**
type-preserving-ϱ_2: *type-preserving-on* (*clause.vars D*) $\mathcal{V}_2$ ϱ_2 **and**
$\mathcal{V}_1$: *infinite-variables-per-type* $\mathcal{V}_1$ **and**
$\mathcal{V}_2$: *infinite-variables-per-type* $\mathcal{V}_2$
obtains R' $\mathcal{V}_3$
where
resolution $(\mathcal{V}_2,\ D)$ $(\mathcal{V}_1,\ C)$ $(\mathcal{V}_3,\ R')$
$\iota_G \in$ *inference-ground-instances* (*Infer* $[(\mathcal{V}_2,\ D),\ (\mathcal{V}_1,\ C)]$ $(\mathcal{V}_3,\ R')$)
$R' \cdot \gamma = R \cdot \gamma$
using *ground-resolution*
proof(*cases* D_G C_G R_G *rule*: *ground.resolution.cases*)
case *ground-resolutionI*: (*resolutionI* L_{GC} $C_G{}'$ L_{GD} $D_G{}'$ t_G)

have *C-*γ: $C \cdot \varrho_1 \odot \gamma =$ *clause.from-ground* (*add-mset* L_{GC} $C_G{}'$)
using *ground-resolutionI*(*1*)
unfolding C_G*-def*
by (*metis C-grounding clause.to-ground-inverse*)

have *D-*γ: $D \cdot \varrho_2 \odot \gamma =$ *clause.from-ground* (*add-mset* L_{GD} $D_G{}'$)
using *ground-resolutionI*(*2*) D_G*-def*
by (*metis D-grounding clause.to-ground-inverse*)

let *?*$select_G$*-empty* $= select_G$ (*clause.to-ground* $(C \cdot \varrho_1 \odot \gamma)) = \{\#\}$
let *?*$select_G$*-not-empty* $= select_G$ (*clause.to-ground* $(C \cdot \varrho_1 \odot \gamma)) \neq \{\#\}$

obtain l_C **where**
l_C*-*γ: $l_C \cdot l\ \varrho_1 \odot \gamma =$ *literal.from-ground* L_{GC} **and**
l_C*-is-maximal*: *?*$select_G$*-empty* $\Longrightarrow$ *is-maximal* l_C C **and**
l_C*-*γ*-is-maximal*: *?*$select_G$*-empty* $\Longrightarrow$ *is-maximal* $(l_C \cdot l\ \varrho_1 \odot \gamma)$ $(C \cdot \varrho_1 \odot \gamma)$
and
l_C*-selected*: *?*$select_G$*-not-empty* $\Longrightarrow$ *is-maximal* l_C (*select C*) **and**
l_C*-*γ*-selected*: *?*$select_G$*-not-empty* $\Longrightarrow$*is-maximal* $(l_C \cdot l\ \varrho_1 \odot \gamma)$ $((select\ C) \cdot \varrho_1 \odot \gamma)$ **and**
l_C*-in-C*: $l_C \in\#\ C$
proof −
have *C-not-empty*: $C \neq \{\#\}$
using *ground-resolutionI*(*1*)
by *auto*

then obtain *max-l* **where**

is-maximal max-l C **and**
is-max-in-C-γ: *is-maximal* (*max-l* ·*l* $\varrho_1 \odot \gamma$) ($C \cdot \varrho_1 \odot \gamma$)
using *that C-grounding obtain-maximal-literal C-not-empty*
by *blast*

moreover then have *max-l* ∈# *C*
unfolding *is-maximal-def*
by *blast*

moreover have *max-l* ·*l* $\varrho_1 \odot \gamma$ = *literal.from-ground* L_{GC} **if** *?select*$_G$*-empty*
proof −
have *ground-is-maximal* L_{GC} C_G
using *ground-resolutionI*(*6*) *that*
unfolding *is-maximal-def*
by *simp*

then show *?thesis*
using *unique-maximal-in-ground-clause*[*OF C-grounding is-max-in-C-γ*]
C-grounding
unfolding *ground-resolutionI*(*3*)
by *simp*
qed

moreover then obtain *selected-l* **where**
is-maximal selected-l (*select C*)
is-maximal (*selected-l* ·*l* $\varrho_1 \odot \gamma$) ((*select C*) · $\varrho_1 \odot \gamma$)
selected-l ·*l* $\varrho_1 \odot \gamma$ = *literal.from-ground* L_{GC}
if *?select*$_G$*-not-empty*
proof −

have *is-maximal* (*literal.from-ground* L_{GC}) ((*select C*) · $\varrho_1 \odot \gamma$)
if *?select*$_G$*-not-empty*
using *ground-resolutionI*(*6*) *that*
unfolding *ground-resolutionI*(*3*)
by (*metis* C_G*-def select-from-C*)

then show *?thesis*
using
that
obtain-maximal-literal[*OF - select-ground-subst*[*OF C-grounding*]]
unique-maximal-in-ground-clause[*OF select-ground-subst*[*OF C-grounding*]]
by (*metis* (*no-types*, *lifting*) *clause.magma-subst-empty*(*1*) *is-maximal-not-empty*)
qed

moreover then have *selected-l* ∈# *C* **if** *?select*$_G$*-not-empty*
using *that maximal-in-clause mset-subset-eqD select-subset*
by *meson*

ultimately show *?thesis*

using *that ground-resolutionI*
by *blast*
qed

obtain C' **where** C: $C = \mathit{add\text{-}mset}\ l_C\ C'$
by (*meson* l_C*-in-C multi-member-split*)

then have C'-γ: $C' \cdot \varrho_1 \odot \gamma = \mathit{clause.from\text{-}ground}\ {C_G}'$
using l_C-γ C-γ
by *auto*

obtain t_C **where**
l_C: $l_C = \mathit{Neg}\ t_C$ **and**
t_C-γ: $t_C \cdot t\ \varrho_1 \odot \gamma = \mathit{term.from\text{-}ground}\ t_G$
using l_C-γ
by (*metis Neg-atm-of-iff literal-from-ground-atom-from-ground*(*1*) *clause-safe-unfolds*(*9*)
ground-resolutionI(*3*) *literal.sel*(*2*) *subst-polarity-stable*(*2*))

obtain l_D **where**
l_D-γ: $l_D \cdot l\ \varrho_2 \odot \gamma = \mathit{literal.from\text{-}ground}\ L_{G\,D}$ **and**
l_D*-is-strictly-maximal*: *is-strictly-maximal* l_D D
proof−
have *is-strictly-maximal* (*literal.from-ground* $L_{G\,D}$) ($D \cdot \varrho_2 \odot \gamma$)
using *ground-resolutionI*(*8*) *D-grounding*
by *simp*

then show *?thesis*
using *obtain-strictly-maximal-literal*[*OF D-grounding*] *that*
by *force*
qed

then have l_D*-in-D*: $l_D \in\# D$
using *strictly-maximal-in-clause*
by *blast*

from l_D-γ **have** l_D-γ: $l_D \cdot l\ \varrho_2 \odot \gamma = (\mathit{Pos}\ (\mathit{term.from\text{-}ground}\ t_G))$
unfolding *ground-resolutionI*
by *simp*

then obtain t_D **where**
l_D: $l_D = \mathit{Pos}\ t_D$ **and**
t_D-γ: $t_D \cdot t\ \varrho_2 \odot \gamma = \mathit{term.from\text{-}ground}\ t_G$
by (*metis clause-safe-unfolds*(*9*) *literal.collapse*(*1*) *literal.disc*(*1*) *literal.sel*(*1*)
subst-polarity-stable(*2*))

obtain D' **where** D: $D = \mathit{add\text{-}mset}\ l_D\ D'$
by (*meson* l_D*-in-D multi-member-split*)

then have D'-γ: $D' \cdot \varrho_2 \odot \gamma = \mathit{clause.from\text{-}ground}\ {D_G}'$

using *D*-γ l_D-γ
unfolding *ground-resolutionI*
by *auto*

obtain $\mathcal{V}_3$ **where**
$\mathcal{V}_3$: *infinite-variables-per-type* $\mathcal{V}_3$ **and**
$\mathcal{V}_1$-$\mathcal{V}_3$: $\forall\, x{\in}$*clause.vars* C. $\mathcal{V}_1\ x = \mathcal{V}_3$ (*term.rename* $\varrho_1\ x$) **and**
$\mathcal{V}_2$-$\mathcal{V}_3$: $\forall\, x{\in}$*clause.vars* D. $\mathcal{V}_2\ x = \mathcal{V}_3$ (*term.rename* $\varrho_2\ x$)
using *clause.obtain-merged-*$\mathcal{V}$[*OF* ϱ_1 ϱ_2 *rename-apart clause.finite-vars clause.finite-vars*

infinite-UNIV] **.**

have *type-preserving-*γ: *type-preserving-on* (*clause.vars* ($C \cdot \varrho_1$) $\cup$ *clause.vars* ($D \cdot \varrho_2$)) $\mathcal{V}_3\ \gamma$
proof(*unfold Set.ball-Un*, *intro conjI*)

show *type-preserving-on* (*clause.vars* ($C \cdot \varrho_1$)) $\mathcal{V}_3\ \gamma$
using *clause.renaming-grounding*[*OF* ϱ_1 *type-preserving-*ϱ_1-γ *C-grounding* $\mathcal{V}_1$-$\mathcal{V}_3$] **.**
next

show *type-preserving-on* (*clause.vars* ($D \cdot \varrho_2$)) $\mathcal{V}_3\ \gamma$
using *clause.renaming-grounding*[*OF* ϱ_2 *type-preserving-*ϱ_2-γ *D-grounding* $\mathcal{V}_2$-$\mathcal{V}_3$] **.**
qed

obtain $\mu\ \sigma$ **where**
γ: $\gamma = \mu \odot \sigma$ **and**
type-preserving-μ: *type-preserving-on* (*clause.vars* ($C \cdot \varrho_1$) $\cup$ *clause.vars* ($D \cdot \varrho_2$)) $\mathcal{V}_3\ \ \mu$ **and**
imgu: *term.is-imgu* μ {{$t_C \cdot_t \varrho_1$, $t_D \cdot_t \varrho_2$}}
proof (*rule obtain-type-preserving-on-imgu*[*OF - that*], *intro conjI*)

show $t_C \cdot_t \varrho_1 \cdot_t \gamma = t_D \cdot_t \varrho_2 \cdot_t \gamma$
using t_C-γ t_D-γ
by *simp*
next

show *type-preserving-on* (*term.vars* ($t_C \cdot_t \varrho_1$) $\cup$ *term.vars* ($t_D \cdot_t \varrho_2$)) $\mathcal{V}_3\ \gamma$
using *type-preserving-*γ
unfolding $C\ D\ l_C\ l_D$
by *auto*
qed

define R' **where**
R': $R' = (C' \cdot \varrho_1 + D' \cdot \varrho_2) \cdot \mu$

show *?thesis*
proof(*rule that*)

show *resolution*: *resolution* $(\mathcal{V}_2,\ D)$ $(\mathcal{V}_1,\ C)$ $(\mathcal{V}_3,\ R')$
proof (*rule resolutionI*; ((*rule* ϱ_1 ϱ_2 C D l_C l_D *imgu type-preserving-*μ *rename-apart*
type-preserving-ϱ_1 *type-preserving-*ϱ_2 $\mathcal{V}_1$ $\mathcal{V}_2$ R' $\mathcal{V}_1$-$\mathcal{V}_3$ $\mathcal{V}_2$-$\mathcal{V}_3$)+)*?*)

show $\neg\ C \cdot \varrho_1 \odot \mu \preceq_c D \cdot \varrho_2 \odot \mu$
proof(*rule clause.order.ground-less-not-less-eq*)

show *clause.vars* $(D \cdot \varrho_2 \odot \mu \cdot \sigma) = \{\}$
using *D-grounding*
unfolding γ
by *simp*

show *clause.vars* $(C \cdot \varrho_1 \odot \mu \cdot \sigma) = \{\}$
using *C-grounding*
unfolding γ
by *simp*

show $D \cdot \varrho_2 \odot \mu \cdot \sigma \prec_c C \cdot \varrho_1 \odot \mu \cdot \sigma$
using *ground-resolutionI*(*5*) *D-grounding C-grounding*
unfolding C_G*-def* D_G*-def clause.order.less*$_G$*-def* γ
by *simp*
qed
next
assume *select* $C = \{\#\}$

moreover then have *?select*$_G$*-empty*
using *is-maximal-not-empty* l_C*-selected*
by *blast*

moreover have $l_C \cdot l\ \varrho_1 \odot \mu \in\# C \cdot \varrho_1 \odot \mu$
using l_C*-in-C*
by *blast*

ultimately show *is-maximal* $(l_C \cdot l\ \varrho_1 \odot \mu)$ $(C \cdot \varrho_1 \odot \mu)$
using l_C-γ*-is-maximal is-maximal-if-grounding-is-maximal C-grounding*
unfolding γ
by *force*
next
assume *select* $C \neq \{\#\}$

then have $\neg$*?select*$_G$*-empty*
using *is-maximal-not-empty* l_C*-selected select-from-C*
by *auto*

moreover have $l_C \cdot l\ \varrho_1 \odot \mu \in\#$ *select* $C \cdot \varrho_1 \odot \mu$
using l_C*-selected maximal-in-clause calculation*
by *blast*

```
    ultimately show is-maximal (l_C ·l ϱ1 ⊙ μ) (select C · ϱ1 ⊙ μ)
    using select-ground-subst[OF C-grounding] is-maximal-if-grounding-is-maximal
l_C-γ-selected
      unfolding γ
      by fastforce
  next

    show select D = {#}
      using ground-resolutionI(7) select-from-D
      by fastforce
  next

    show is-strictly-maximal (l_D ·l ϱ2 ⊙ μ) (D · ϱ2 ⊙ μ)
    proof(rule is-strictly-maximal-if-grounding-is-strictly-maximal)

      show l_D ·l ϱ2 ⊙ μ ∈# D · ϱ2 ⊙ μ
        using l_D-in-D
        by blast

      show clause.is-ground (D · ϱ2 ⊙ μ · σ)
        using D-grounding[unfolded γ]
        by simp

      show is-strictly-maximal (l_D ·l ϱ2 ⊙ μ ·l σ) (D · ϱ2 ⊙ μ · σ)
        using l_D-γ D-γ ground-resolutionI(8)
        unfolding γ ground-resolutionI
        by fastforce
    qed
  qed

  show R'-γ: R' · γ = R · γ
  proof−

    have term.is-idem μ
      using imgu term.is-imgu-iff-is-idem-and-is-mgu
      by blast

    then have μ-γ: μ ⊙ γ = γ
      unfolding γ term.is-idem-def
      by (metis term.assoc)

    have R · γ = (clause.from-ground C_G' + clause.from-ground D_G')
      using ground-resolutionI(8, 9) clause.to-ground-inverse[OF R-grounding]
      by auto

    then show ?thesis
      unfolding
        R'
```

```
      C'-γ[symmetric]
      D'-γ[symmetric]
      clause.subst-comp-subst[symmetric]
      μ-γ
    by simp
  qed

  show ι_G ∈ inference-ground-instances (Infer [(V_2, D), (V_1, C)] (V_3, R'))
  proof (rule is-inference-ground-instance-two-premises)

    show is-inference-ground-instance-two-premises (V_2, D) (V_1, C) (V_3, R') ι_G
γ ϱ_1 ϱ_2
    proof(unfold split, intro conjI;
        (rule ϱ_1 ϱ_2 rename-apart refl V_1 V_2 V_3)?)

      show inference.is-ground (Infer [D · ϱ_2, C · ϱ_1] R' ·ι γ)
        using  D-grounding C-grounding R-grounding R'-γ
        by auto
    next

      show ι_G = inference.to-ground (Infer [D · ϱ_2, C · ϱ_1] R' ·ι γ)
        using R'-γ
        by simp
    next

      show type-preserving-on (clause.vars R') V_3 γ
      proof(rule type-preserving-on-subset[OF type-preserving-γ])

        show clause.vars R' ⊆ clause.vars (C · ϱ_1) ∪ clause.vars (D · ϱ_2)
        proof (unfold subset-eq, intro ballI)
          fix x

          have is-imgu: term.is-imgu μ {{t_C ·t ϱ_1, t_D ·t ϱ_2}}
            using imgu
            by blast

          assume x ∈ clause.vars R'

          then consider
            (C') x ∈ clause.vars (C' · ϱ_1 ⊙ μ) |
            (D') x ∈ clause.vars (D' · ϱ_2 ⊙ μ)
            unfolding R'
            by auto

          then show x ∈ clause.vars (C · ϱ_1) ∪ clause.vars (D · ϱ_2)
          proof cases
            case C'

            then show ?thesis
```

```
            using clause.variables-in-base-imgu[OF is-imgu]
            unfolding C l_C D l_D
            by auto
          next
            case D'

            then show ?thesis
            using clause.variables-in-base-imgu[OF is-imgu]
            unfolding C l_C D l_D
            by auto
          qed
        qed
      qed
    qed
    show ι_G ∈ ground.G-Inf
      unfolding ground.G-Inf-def
      using ground-resolution
      by simp
  qed
 qed
qed
```

2.2 Ground instances

```
context
 fixes ι_G N
 assumes
  subst-stability: subst-stability-on N and
  ι_G-Inf-from: ι_G ∈ ground.Inf-from-q select_G (⋃(uncurried-ground-instances ‘ N))
begin

lemma factoring-ground-instance:
 assumes ground-factoring: ι_G ∈ ground.factoring-inferences
 obtains ι where
  ι ∈ Inf-from N
  ι_G ∈ inference-ground-instances ι
proof −

 obtain D_G C_G where
  ι_G: ι_G = Infer [D_G] C_G and
  ground-inference: ground.factoring D_G C_G
  using ground-factoring
  by blast

 have D_G-in-groundings: D_G ∈ ⋃(uncurried-ground-instances ‘ N)
  using ι_G-Inf-from
  unfolding ι_G ground.Inf-from-q-def ground.Inf-from-def
  by simp
```

```
  obtain D γ 𝒱 where
    D-grounding: clause.is-ground (D · γ) and
    type-preserving-γ: type-preserving-on (clause.vars D) 𝒱 γ and
    𝒱: infinite-variables-per-type 𝒱 and
    D-in-N: (𝒱, D) ∈ N and
    select_G D_G = clause.to-ground (select D · γ)
    D · γ = clause.from-ground D_G
    using subst-stability[rule-format, OF D_G-in-groundings]
    by blast

  then have
    D_G: D_G = clause.to-ground (D · γ) and
    select: clause.from-ground (select_G D_G) = select D · γ
    by (simp-all add: select-ground-subst)

  obtain C where
    C_G: C_G = clause.to-ground (C · γ) and
    C-grounding: clause.is-ground (C · γ)
    by (metis clause.all-subst-ident-iff-ground clause.from-ground-inverse
        clause.ground-is-ground)

  obtain C' where
    factoring: factoring (𝒱, D) (𝒱, C') and
    inference-ground-instances: ι_G ∈ inference-ground-instances (Infer [(𝒱, D)] (𝒱,
C')) and
    C'-C: C' · γ = C · γ
    using
      factoring-lifting[OF
        ground-inference[unfolded D_G C_G]
        D-grounding
        C-grounding
        select[unfolded D_G]
        type-preserving-γ
        𝒱]
    unfolding D_G C_G ι_G .

  let ?ι = Infer [(𝒱, D)] (𝒱, C')

  show ?thesis
  proof(rule that[OF - inference-ground-instances])

    show ?ι ∈ Inf-from N
      using D-in-N factoring
      unfolding Inf-from-def inferences-def inference-system.Inf-from-def
      by auto
  qed
qed
```

lemma *resolution-ground-instance*:
assumes *ground-resolution*: $\iota_G \in$ *ground.resolution-inferences*
obtains ι **where**
$\iota \in$ *Inf-from* N
$\iota_G \in$ *inference-ground-instances* ι
proof−
obtain C_G D_G R_G **where**
$\iota_G : \iota_G =$ *Infer* $[D_G, C_G]$ R_G **and**
ground-resolution: *ground.resolution* D_G C_G R_G
using *assms*(*1*)
by *blast*

have
C_G*-in-groundings*: $C_G \in \bigcup$ (*uncurried-ground-instances* ‘ N) **and**
D_G*-in-groundings*: $D_G \in \bigcup$ (*uncurried-ground-instances* ‘ N)
using ι_G*-Inf-from*
unfolding ι_G *ground.Inf-from-q-def ground.Inf-from-def*
by *simp-all*

obtain C $\mathcal{V}_1$ γ_1 **where**
C-grounding: *clause.is-ground* $(C \cdot \gamma_1)$ **and**
type-preserving-γ_1: *type-preserving-on* (*clause.vars* C) $\mathcal{V}_1$ γ_1 **and**
$\mathcal{V}_1$: *infinite-variables-per-type* $\mathcal{V}_1$ **and**
C-in-N: $(\mathcal{V}_1, C) \in N$ **and**
$select_G$ $C_G =$ *clause.to-ground* (*select* $C \cdot \gamma_1$)
$C \cdot \gamma_1 =$ *clause.from-ground* C_G
using *subst-stability*[*rule-format*, *OF* C_G*-in-groundings*]
by *blast*

then have
C_G: $C_G =$ *clause.to-ground* $(C \cdot \gamma_1)$ **and**
select-from-C: *clause.from-ground* $(select_G\ C_G) =$ *select* $C \cdot \gamma_1$
by (*simp-all add*: *select-ground-subst*)

obtain D $\mathcal{V}_2$ γ_2 **where**
D-grounding: *clause.is-ground* $(D \cdot \gamma_2)$ **and**
type-preserving-γ_2: *type-preserving-on* (*clause.vars* D) $\mathcal{V}_2$ γ_2 **and**
$\mathcal{V}_2$: *infinite-variables-per-type* $\mathcal{V}_2$ **and**
D-in-N: $(\mathcal{V}_2, D) \in N$ **and**
$select_G$ $D_G =$ *clause.to-ground* (*select* $D \cdot \gamma_2$)
$D \cdot \gamma_2 =$ *clause.from-ground* D_G
using *subst-stability*[*rule-format*, *OF* D_G*-in-groundings*]
by *blast*

then have
D_G: $D_G =$ *clause.to-ground* $(D \cdot \gamma_2)$ **and**
select-from-D: *clause.from-ground* $(select_G\ D_G) =$ *select* $D \cdot \gamma_2$
by (*simp-all add*: *select-ground-subst*)

```
  obtain ϱ1 ϱ2 γ :: 'v ⇒ 't where
    ϱ1: term.is-renaming ϱ1 and
    ϱ2: term.is-renaming ϱ2 and
    rename-apart: clause.vars (C · ϱ1) ∩ clause.vars (D · ϱ2) = {} and
    type-preserving-ϱ1: type-preserving-on (clause.vars C) V1 ϱ1 and
    type-preserving-ϱ2: type-preserving-on (clause.vars D) V2 ϱ2 and
    γ1-γ: ∀ x ∈ clause.vars C. γ1 x = (ϱ1 ⊙ γ) x and
    γ2-γ: ∀ x ∈ clause.vars D. γ2 x = (ϱ2 ⊙ γ) x
    using clause.obtain-merged-grounding[OF
      type-preserving-γ1 type-preserving-γ2 C-grounding D-grounding V2 clause.finite-vars]
.

  have C-grounding: clause.is-ground (C · ϱ1 ⊙ γ)
    using clause.subst-eq γ1-γ C-grounding
    by fastforce

  have CG: CG = clause.to-ground (C · ϱ1 ⊙ γ)
    using clause.subst-eq γ1-γ CG
    by fastforce

  have D-grounding: clause.is-ground (D · ϱ2 ⊙ γ)
    using clause.subst-eq γ2-γ D-grounding
    by fastforce

  have DG: DG = clause.to-ground (D · ϱ2 ⊙ γ)
    using clause.subst-eq γ2-γ DG
    by fastforce

  have type-preserving-ϱ1-γ: type-preserving-on (clause.vars C) V1 (ϱ1 ⊙ γ)
    using type-preserving-γ1 γ1-γ
    by fastforce

  have type-preserving-ϱ2-γ: type-preserving-on (clause.vars D) V2 (ϱ2 ⊙ γ)
    using type-preserving-γ2 γ2-γ
    by fastforce

  have select-from-C:
    clause.from-ground (selectG (clause.to-ground (C · ϱ1 ⊙ γ))) = select C · ϱ1 ⊙
γ
  proof−
    have C · γ1 = C · ϱ1 ⊙ γ
      using γ1-γ clause.subst-eq
      by fast

    moreover have select C · γ1 = select C · ϱ1 · γ
      using clause.subst-eq γ1-γ select-vars-subset
     by (metis (no-types, lifting) clause.comp-subst.left.monoid-action-compatibility
in-mono)
```

ultimately show *?thesis*
using *select-from-C*
unfolding C_G
by *simp*
qed

have *select-from-D*:
clause.from-ground (*select*$_G$ (*clause.to-ground* ($D \cdot \varrho_2 \odot \gamma$))) = *select* $D \cdot \varrho_2 \odot \gamma$
proof−
have $D \cdot \gamma_2 = D \cdot \varrho_2 \odot \gamma$
using γ_2-γ *clause.subst-eq*
by *fast*

moreover have *select* $D \cdot \gamma_2$ = *select* $D \cdot \varrho_2 \cdot \gamma$
using *clause.subst-eq* γ_2-γ *select-vars-subset*[*of D*]
by (*metis* (*no-types*, *lifting*) *clause.comp-subst.left.monoid-action-compatibility in-mono*)

ultimately show *?thesis*
using *select-from-D*
unfolding D_G
by *simp*
qed

obtain R **where**
R-grounding: *clause.is-ground* ($R \cdot \gamma$) **and**
R_G: R_G = *clause.to-ground* ($R \cdot \gamma$)
by (*metis clause.all-subst-ident-if-ground clause.from-ground-inverse clause.ground-is-ground*)

have *ground-instances* $\mathcal{V}_1$ C ∪ *ground-instances* $\mathcal{V}_2$ $D \subseteq \bigcup$ (*uncurried-ground-instances* ‘ N)
using *C-in-N D-in-N*
by *force*

obtain R' $\mathcal{V}_3$ **where**
resolution: *resolution* ($\mathcal{V}_2$, D) ($\mathcal{V}_1$, C) ($\mathcal{V}_3$, R') **and**
inference-groundings: $\iota_G \in$ *inference-ground-instances* (*Infer* [($\mathcal{V}_2$, D), ($\mathcal{V}_1$, C)] ($\mathcal{V}_3$, R')) **and**
R'-γ-R-γ: $R' \cdot \gamma = R \cdot \gamma$
using *resolution-lifting*[*OF ground-resolution*[*unfolded* C_G D_G R_G]
ϱ_1 ϱ_2
rename-apart
C-grounding D-grounding R-grounding
select-from-C select-from-D
type-preserving-ϱ_1-γ *type-preserving-*ϱ_2-γ
type-preserving-ϱ_1 *type-preserving-*ϱ_2
$\mathcal{V}_1$ $\mathcal{V}_2$]

```
    unfolding ι_G R_G C_G D_G .

  let ?ι = Infer [(V_2, D), (V_1, C)] (V_3, R')

  show ?thesis
  proof(rule that[OF - inference-groundings])

    show ?ι ∈ Inf-from N
      using C-in-N D-in-N resolution
      unfolding Inf-from-def inferences-def inference-system.Inf-from-def
      by auto
  qed
qed

lemma ground-instances:
  obtains ι where
    ι ∈ Inf-from N
    ι_G ∈ inference-ground-instances ι
proof −

  consider
    (resolution) ι_G ∈ ground.resolution-inferences |
    (factoring) ι_G ∈ ground.factoring-inferences
    using ι_G-Inf-from
    unfolding
      ground.Inf-from-q-def
      ground.G-Inf-def
      inference-system.Inf-from-def
    by fastforce

  then show ?thesis
  proof cases
    case resolution

    then show ?thesis
      using that resolution-ground-instance
      by blast
  next
    case factoring

    then show ?thesis
      using that factoring-ground-instance
      by blast
  qed
qed

end

end
```

context *ordered-resolution-calculus*
begin

lemma *overapproximation*:
  obtains $select_G$ **where**
    ground-Inf-overapproximated $select_G$ *premises*
    is-grounding $select_G$
proof−
  obtain $select_G$ **where**
    subst-stability: *select-subst-stability-on select* $select_G$ *premises* **and**
    is-grounding $select_G$
    using *obtain-subst-stable-on-select-grounding*
    by *blast*

  then interpret *grounded-ordered-resolution-calculus*
    where $select_G = select_G$
    by *unfold-locales*

  show *thesis*
  proof(*rule that*[*OF* - $select_G$])

    show *ground-Inf-overapproximated* $select_G$ *premises*
      using *ground-instances*[*OF subst-stability*]
      by *auto*
  qed
qed

sublocale *statically-complete-calculus* $\bot_F$ *inferences entails-*$\mathcal{G}$ *Red-I-*$\mathcal{G}$ *Red-F-*$\mathcal{G}$
proof (*unfold static-empty-ord-inter-equiv-static-inter*,
    rule stat-ref-comp-to-non-ground-fam-inter,
    rule ballI)
  fix $select_G$
  assume $select_G \in select_{Gs}$

  then interpret *grounded-ordered-resolution-calculus*
    where $select_G = select_G$
    by *unfold-locales* (*simp add*: $select_{Gs}$*-def*)

  show *statically-complete-calculus*
        ground.G-Bot
        ground.G-Inf
        ground.G-entails
        ground.Red-I
        ground.Red-F
    by *unfold-locales*
next

  show $\bigwedge N.\ \exists\, select_G \in select_{Gs}.$ *ground-Inf-overapproximated* $select_G$ N

```
    using overapproximation
    unfolding select_Gs-def
    by (smt (verit, best) mem-Collect-eq)
qed

end

end
theory Ordered-Resolution-Welltypedness-Preservation
  imports Grounded-Ordered-Resolution
begin

context ordered-resolution-calculus
begin

lemma factoring-preserves-typing:
  assumes factoring: factoring (𝒱, C) (𝒱, D)
  shows clause.is-welltyped 𝒱 C ⟷ clause.is-welltyped 𝒱 D
  using assms
proof (cases (𝒱, C) (𝒱, D) rule: factoring.cases)
  case (factoringI L₁ μ t₁ t₂ L₂ C')

  show ?thesis
  proof (rule iffI)
    assume clause.is-welltyped 𝒱 C
    then show clause.is-welltyped 𝒱 D
      using factoringI
      by simp
  next
    assume D-is-welltyped: clause.is-welltyped 𝒱 D

    note imgu = factoringI(3, 4)

    have clause.is-welltyped 𝒱 (add-mset L₁ C')
      using D-is-welltyped imgu
      unfolding factoringI
      by simp

    moreover have literal.is-welltyped 𝒱 L₂
      using  D-is-welltyped term.imgu-same-type'[OF imgu] imgu
      unfolding factoringI
      by force

    ultimately show clause.is-welltyped 𝒱 C
      unfolding factoringI
      by simp
  qed
qed
```

lemma *resolution-preserves-typing*:
assumes
resolution: *resolution* $(\mathcal{V}_2,\ D)\ (\mathcal{V}_1,\ C)\ (\mathcal{V}_3,\ R)$ **and**
D-is-welltyped: *clause.is-welltyped* $\mathcal{V}_2\ D$ **and**
C-is-welltyped: *clause.is-welltyped* $\mathcal{V}_1\ C$
shows *clause.is-welltyped* $\mathcal{V}_3\ R$
using *resolution*
proof (*cases* $(\mathcal{V}_2,\ D)\ (\mathcal{V}_1,\ C)\ (\mathcal{V}_3,\ R)$ *rule*: *resolution.cases*)
case (*resolutionI* $\varrho_1\ \varrho_2\ \mu\ t_1\ t_2\ L_1\ L_2\ C'\ D'$)

note μ-*type-preserving* = *resolutionI*(*6*)

have *clause.is-welltyped* $\mathcal{V}_3\ (C \cdot \varrho_1)$
using *C-is-welltyped clause.welltyped-renaming*[*OF resolutionI*(*3*, *13*)]
by *blast*

then have $C\mu$-*is-welltyped*: *clause.is-welltyped* $\mathcal{V}_3\ (C \cdot \varrho_1 \odot \mu)$
using μ-*type-preserving*
by *simp*

moreover have *clause.is-welltyped* $\mathcal{V}_3\ (D \cdot \varrho_2)$
using *D-is-welltyped clause.welltyped-renaming*[*OF resolutionI*(*4*, *14*)]
by *blast*

then have $D\mu$-*is-welltyped*: *clause.is-welltyped* $\mathcal{V}_3\ (D \cdot \varrho_2 \odot \mu)$
using μ-*type-preserving*
by *simp*

ultimately show *?thesis*
unfolding *resolutionI*
by *auto*
qed

end

end
theory *Untyped-Ordered-Resolution*
imports
First-Order-Clause.Nonground-Order
First-Order-Clause.Nonground-Selection-Function
First-Order-Clause.Tiebreakers

Fresh-Identifiers.Fresh
begin

locale *untyped-ordered-resolution-calculus* =
nonground-order **where**
$less_t = less_t$ **and** *Var* = *Var* **and** *term-from-ground* = *term-from-ground* :: $'t_G$
$\Rightarrow$ $'t$ +

nonground-selection-function **where**
select = *select* **and** *atom-subst* = $(\cdot t)$ **and** *atom-vars* = *term.vars* **and**
atom-from-ground = *term.from-ground* **and** *atom-to-ground* = *term.to-ground*
and *Var* = *Var* +

tiebreakers tiebreakers +

term: *exists-imgu* **where** *vars* = *term-vars* **and** *subst* = $(\cdot t)$ **and** *id-subst* = *Var*
for
select :: $'t$ *select* **and**
$less_t$ :: $'t \Rightarrow 't \Rightarrow$ *bool* **and**
tiebreakers :: $('t_G, 't)$ *tiebreakers* **and**
Var :: $'v$ :: *infinite* $\Rightarrow 't$
begin

inductive *factoring* :: $'t$ *clause* $\Rightarrow$ $'t$ *clause* $\Rightarrow$ *bool*
where
factoringI:
C = *add-mset* L_1 (*add-mset* L_2 C') $\Longrightarrow$
L_1 = (*Pos* t_1) $\Longrightarrow$
L_2 = (*Pos* t_2) $\Longrightarrow$
D = (*add-mset* L_1 C') $\cdot$ μ $\Longrightarrow$
factoring C D
if
select C = $\{\#\}$
is-maximal $(L_1 \cdot l \ \mu)$ $(C \cdot \mu)$
term.is-imgu μ $\{\{t_1, t_2\}\}$

inductive *resolution* :: $'t$ *clause* $\Rightarrow$ $'t$ *clause* $\Rightarrow$ $'t$ *clause* $\Rightarrow$ *bool*
where
resolutionI:
C = *add-mset* L_1 C' $\Longrightarrow$
D = *add-mset* L_2 D' $\Longrightarrow$
L_1 = (*Neg* t_1) $\Longrightarrow$
L_2 = (*Pos* t_2) $\Longrightarrow$
$R = (C' \cdot \varrho_1 + D' \cdot \varrho_2) \cdot \mu \Longrightarrow$
resolution D C R
if
term.is-renaming ϱ_1
term.is-renaming ϱ_2
clause.vars $(C \cdot \varrho_1) \cap$ *clause.vars* $(D \cdot \varrho_2) = \{\}$
term.is-imgu μ $\{\{t_1 \cdot t \ \varrho_1, t_2 \cdot t \ \varrho_2\}\}$
$\neg$ $(C \cdot \varrho_1 \odot \mu \preceq_c D \cdot \varrho_2 \odot \mu)$
select C = $\{\#\} \Longrightarrow$ *is-maximal* $(L_1 \cdot l \ \varrho_1 \odot \mu)$ $(C \cdot \varrho_1 \odot \mu)$
select $C \neq \{\#\} \Longrightarrow$ *is-maximal* $(L_1 \cdot l \ \varrho_1 \odot \mu)$ $((\textit{select}\ C) \cdot \varrho_1 \odot \mu)$
select D = $\{\#\}$
is-strictly-maximal $(L_2 \cdot l \ \varrho_2 \odot \mu)$ $(D \cdot \varrho_2 \odot \mu)$

abbreviation *factoring-inferences* **where**
factoring-inferences ≡ { *Infer* [*C*] *D* | *C D*. *factoring C D*}

abbreviation *resolution-inferences* **where**
resolution-inferences ≡ { *Infer* [*D*, *C*] *R* | *D C R*. *resolution D C R*}

definition *inferences* :: *'t clause inference set* **where**
inferences ≡ *resolution-inferences* ∪ *factoring-inferences*

abbreviation *bottom* :: *'t clause set* **where**
bottom ≡ {{#}}

end

end
theory *Untyped-Ordered-Resolution-Inference-System*
imports
Untyped-Ordered-Resolution
First-Order-Clause.Untyped-Calculus
Grounded-Ordered-Resolution
begin

context *untyped-ordered-resolution-calculus*
begin

sublocale *typed*: *ordered-resolution-calculus* **where**
welltyped = λ- - (). *True*
by
unfold-locales
(*auto intro*: *term.ground-exists simp*: *term.exists-imgu right-unique-def split*: *unit.splits*)

declare
typed.term.welltyped-renaming [*simp del*]
typed.term.welltyped-subst-stability [*simp del*]
typed.term.welltyped-subst-stability' [*simp del*]

abbreviation *entails* **where**
entails N N' ≡ *typed.entails-𝒢* (*empty-typed ‘ N*) (*empty-typed ‘ N'*)

sublocale *untyped-consequence-relation* **where**
typed-bottom = $\bot_F$ **and** *typed-entails* = *typed.entails-𝒢* **and**
bottom = *bottom* **and** *entails* = *entails*
proof *unfold-locales*

have *bottom* = *snd ‘* $\bot_F$
using *image-iff typed.bot-not-empty*

```
    by fastforce

  then show bottom ≡ snd ‘ ⊥F
    by argo
qed

sublocale untyped-inference-system where
  inferences = inferences and typed-inferences = typed.inferences
proof unfold-locales

  {
    fix 𝒱 D 𝒱' C
    have typed.factoring (𝒱, D) (𝒱', C) ⟷ factoring D C
      unfolding 𝒱-all-same[of 𝒱] 𝒱-all-same[of 𝒱']
    proof (rule iffI)
      assume typed.factoring (empty-typed D) (empty-typed C)

      then show factoring D C
      proof (cases rule: typed.factoring.cases)
        case typed-factoringI: factoringI

        then show ?thesis
          by (intro factoringI; (rule typed-factoringI)?)
      qed
    next
      assume factoring D C

      then show typed.factoring (empty-typed D) (empty-typed C)
      proof (cases rule: factoring.cases)
        case factoringI

        then show ?thesis
          by (intro typed.factoringI) (auto simp: case-unit-Unity)
      qed
    qed
  }

  then have remove-types ‘ typed.factoring-inferences = factoring-inferences
    by auto

  moreover {
    fix 𝒱1 D 𝒱2 E 𝒱3 C
    have typed.resolution (𝒱1, D) (𝒱2, E) (𝒱3, C) ⟷ resolution D E C
      unfolding 𝒱-all-same[of 𝒱1] 𝒱-all-same[of 𝒱2] 𝒱-all-same[of 𝒱3]
    proof (rule iffI)
      assume typed.resolution (empty-typed D) (empty-typed E) (empty-typed C)

      then show resolution D E C
      proof (cases rule: typed.resolution.cases)
```

```
      case typed-resolutionI: resolutionI

      then show ?thesis
        by (intro resolutionI; (rule typed-resolutionI)?)
    qed
  next
    assume resolution D E C

     then show typed.resolution (empty-typed D) (empty-typed E) (empty-typed
C)
    proof (cases rule: resolution.cases)
      case resolutionI

      show ?thesis
        by
          (intro typed.resolutionI; (rule resolutionI)?)
          (auto simp: case-unit-Unity)
    qed
  qed
 }

 then have remove-types ‘ typed.resolution-inferences = resolution-inferences
   by auto

 ultimately show inferences ≡ remove-types ‘ typed.inferences
   unfolding inferences-def typed.inferences-def image-Un
   by auto
qed

end

end
theory Untyped-Ordered-Resolution-Completeness
 imports
   Untyped-Ordered-Resolution-Inference-System
   Ordered-Resolution-Completeness
begin

context untyped-ordered-resolution-calculus
begin


abbreviation Red-F where
 Red-F N ≡ snd ‘ typed.Red-F-G (empty-typed ‘ N)

abbreviation Red-I where
 Red-I N ≡ remove-types ‘ typed.Red-I-G (empty-typed ‘ N)

sublocale untyped-complete-calculus where
```

typed-bottom = $\bot_F$ **and** *typed-entails* = *typed.entails-*$\mathcal{G}$ **and**
typed-inferences = *typed.inferences* **and** *typed-Red-I* = *typed.Red-I-*$\mathcal{G}$ **and**
typed-Red-F = *typed.Red-F-*$\mathcal{G}$ **and** *bottom* = *bottom* **and** *inferences* = *inferences* **and** *Red-F* = *Red-F* **and**
Red-I = *Red-I* **and** *entails* = *entails*
by *unfold-locales*

end

end
theory *Untyped-Ordered-Resolution-Soundness*
imports
Untyped-Ordered-Resolution-Inference-System
Ordered-Resolution-Soundness
begin

context *untyped-ordered-resolution-calculus*
begin

sublocale *untyped-sound-inference-system* **where**
typed-bottom = $\bot_F$ **and** *typed-entails* = *typed.entails-*$\mathcal{G}$ **and**
typed-inferences = *typed.inferences* **and** *bottom* = *bottom* **and** *inferences* = *inferences* **and**
entails = *entails*
by *unfold-locales*

end

end
theory *Monomorphic-Ordered-Resolution*
imports
Ordered-Resolution

First-Order-Clause.IsaFoR-Nonground-Clause
First-Order-Clause.Monomorphic-Typing
begin

locale *monomorphic-ordered-resolution-calculus* =
monomorphic-term-typing +

ordered-resolution-calculus **where**
comp-subst = ($\circ_s$) **and** *Var* = *Var* **and** *term-subst* = ($\cdot$) **and** *term-vars* = *term.vars* **and**
term-from-ground = *term.from-ground* **and** *term-to-ground* = *term.to-ground* **and** *welltyped* = *welltyped*

end
theory *Ordered-Resolution-Example*
imports

Monomorphic-Ordered-Resolution

First-Order-Clause.IsaFoR-KBO
begin

hide-type *Uprod-Literal-Functor.clause*

abbreviation *trivial-tiebreakers* ::
'f gterm clause ⇒ (*'f*,*'v*) *term clause* ⇒ (*'f*,*'v*) *term clause* ⇒ *bool* **where**
trivial-tiebreakers ≡ ⊥

abbreviation *trivial-select* :: *'a clause* ⇒ *'a clause* **where**
trivial-select - ≡ {#}

abbreviation *unit-typing* **where**
unit-typing - - ≡ *Some* ([], ())

interpretation *unit-types*: *witnessed-monomorphic-term-typing* **where** $\mathcal{F}$ = *unit-typing*
by *unfold-locales auto*

interpretation *example1*: *monomorphic-ordered-resolution-calculus* **where**
select = *trivial-select* :: ((*'f* :: *weighted* , *'v* :: *infinite*) *term*) *select* **and**
$less_t$ = *less-kbo* **and**
$\mathcal{F}$ = *unit-typing* **and**
tiebreakers = *trivial-tiebreakers*
by *unfold-locales auto*

instantiation *nat* :: *infinite*
begin

instance
by *intro-classes simp*

end

datatype *type* = *A* | *B*

abbreviation *types* :: *nat* ⇒ *nat* ⇒ (*type list* × *type*) *option* **where**
types f n ≡
let type = *if even f then A else B*
in Some (*replicate n type*, *type*)

interpretation *example-types*: *witnessed-monomorphic-term-typing* **where** $\mathcal{F}$ = *types*
proof *unfold-locales*
fix *τ*
show ∃*f*. *types f 0* = *Some* ([], *τ*)
proof (*cases τ*)

```
    case A
    show ?thesis
      unfolding A
      by (rule exI[of - 0]) auto
  next
    case B
    show ?thesis
      unfolding B
      by (rule exI[of - 1]) auto
  qed
qed


interpretation example2: monomorphic-ordered-resolution-calculus where
    select = trivial-select :: (nat, nat) term select and
    less_t = less-kbo and
    F = types and
    tiebreakers = trivial-tiebreakers
  by unfold-locales

end
```