Formalisation of Ground Resolution and CDCL in Isabelle/HOL

Mathias Fleury and Jasmin Blanchette

January 20, 2020

Contents

theory *Prop-Logic*
imports *Main*
begin

Chapter 1

Normalisation

We define here the normalisation from formula towards conjunctive and disjunctive normal form, including normalisation towards multiset of multisets to represent CNF.

1.1 Logics

In this section we define the syntax of the formula and an abstraction over it to have simpler proofs. After that we define some properties like subformula and rewriting.

1.1.1 Definition and Abstraction

The propositional logic is defined inductively. The type parameter is the type of the variables.

datatype *'v propo* =
  FT | *FF* | *FVar 'v* | *FNot 'v propo* | *FAnd 'v propo 'v propo* | *FOr 'v propo 'v propo*
  | *FImp 'v propo 'v propo* | *FEq 'v propo 'v propo*

We do not define any notation for the formula, to distinguish properly between the formulas and Isabelle's logic.

To ease the proofs, we will write the the formula on a homogeneous manner, namely a connecting argument and a list of arguments.

datatype *'v connective* = *CT* | *CF* | *CVar 'v* | *CNot* | *CAnd* | *COr* | *CImp* | *CEq*

abbreviation *nullary-connective* ≡ {*CF*} ∪ {*CT*} ∪ {*CVar x* | *x*. *True*}
definition *binary-connectives* ≡ {*CAnd*, *COr*, *CImp*, *CEq*}

We define our own induction principal: instead of distinguishing every constructor, we group them by arity.

lemma *propo-induct-arity*[*case-names nullary unary binary*]:
  fixes φ ψ :: *'v propo*
  assumes *nullary*: $\bigwedge\varphi\ x.\ \varphi = FF \vee \varphi = FT \vee \varphi = FVar\ x \Longrightarrow P\ \varphi$
  and *unary*: $\bigwedge\psi.\ P\ \psi \Longrightarrow P\ (FNot\ \psi)$
  and *binary*: $\bigwedge\varphi\ \psi 1\ \psi 2.\ P\ \psi 1 \Longrightarrow P\ \psi 2 \Longrightarrow \varphi = FAnd\ \psi 1\ \psi 2 \vee \varphi = FOr\ \psi 1\ \psi 2 \vee \varphi = FImp\ \psi 1$
$\psi 2$
    $\vee\ \varphi = FEq\ \psi 1\ \psi 2 \Longrightarrow P\ \varphi$
  shows $P\ \psi$
  apply (*induct rule*: *propo.induct*)
  using *assms* **by** *metis+*

The function *conn* is the interpretation of our representation (connective and list of arguments). We define any thing that has no sense to be false

fun *conn* :: $'v$ *connective* $\Rightarrow$ $'v$ *propo list* $\Rightarrow$ $'v$ *propo* **where**
conn CT [] = *FT* |
conn CF [] = *FF* |
conn (*CVar v*) [] = *FVar v* |
conn CNot [φ] = *FNot* φ |
conn CAnd (φ # [ψ]) = *FAnd* φ ψ |
conn COr (φ # [ψ]) = *FOr* φ ψ |
conn CImp (φ # [ψ]) = *FImp* φ ψ |
conn CEq (φ # [ψ]) = *FEq* φ ψ |
conn - - = *FF*

We will often use case distinction, based on the arity of the $'v$ *connective*, thus we define our own splitting principle.

lemma *connective-cases-arity*[*case-names nullary binary unary*]:
  assumes *nullary*: $\bigwedge x.$ $c = CT \vee c = CF \vee c = CVar\ x \Longrightarrow P$
  and *binary*: $c \in$ *binary-connectives* $\Longrightarrow P$
  and *unary*: $c = CNot \Longrightarrow P$
  shows *P*
  using *assms* **by** (*cases c*) (*auto simp*: *binary-connectives-def*)

lemma *connective-cases-arity-2*[*case-names nullary unary binary*]:
  assumes *nullary*: $c \in$ *nullary-connective* $\Longrightarrow P$
  and *unary*: $c = CNot \Longrightarrow P$
  and *binary*: $c \in$ *binary-connectives* $\Longrightarrow P$
  shows *P*
  using *assms* **by** (*cases c*, *auto simp add*: *binary-connectives-def*)

Our previous definition is not necessary correct (connective and list of arguments), so we define an inductive predicate.

inductive *wf-conn* :: $'v$ *connective* $\Rightarrow$ $'v$ *propo list* $\Rightarrow$ *bool* **for** *c* :: $'v$ *connective* **where**
wf-conn-nullary[*simp*]: ($c = CT \vee c = CF \vee c = CVar\ v$) $\Longrightarrow$ *wf-conn c* [] |
wf-conn-unary[*simp*]: $c = CNot \Longrightarrow$ *wf-conn c* [ψ] |
wf-conn-binary[*simp*]: $c \in$ *binary-connectives* $\Longrightarrow$ *wf-conn c* (ψ # ψ' # [])
thm *wf-conn.induct*
lemma *wf-conn-induct*[*consumes 1*, *case-names CT CF CVar CNot COr CAnd CImp CEq*]:
  assumes *wf-conn c x* **and**
    $\bigwedge v.$ $c = CT \Longrightarrow P$ [] **and**
    $\bigwedge v.$ $c = CF \Longrightarrow P$ [] **and**
    $\bigwedge v.$ $c = CVar\ v \Longrightarrow P$ [] **and**
    $\bigwedge \psi.$ $c = CNot \Longrightarrow P$ [ψ] **and**
    $\bigwedge \psi\ \psi'.$ $c = COr \Longrightarrow P$ [ψ, ψ'] **and**
    $\bigwedge \psi\ \psi'.$ $c = CAnd \Longrightarrow P$ [ψ, ψ'] **and**
    $\bigwedge \psi\ \psi'.$ $c = CImp \Longrightarrow P$ [ψ, ψ'] **and**
    $\bigwedge \psi\ \psi'.$ $c = CEq \Longrightarrow P$ [ψ, ψ']
  shows *P x*
  using *assms* **by** *induction* (*auto simp*: *binary-connectives-def*)

1.1.2 Properties of the Abstraction

First we can define simplification rules.

lemma *wf-conn-conn*[*simp*]:

wf-conn CT l $\Longrightarrow$ *conn CT l* = *FT*
wf-conn CF l $\Longrightarrow$ *conn CF l* = *FF*
wf-conn (*CVar x*) *l* $\Longrightarrow$ *conn* (*CVar x*) *l* = *FVar x*
apply (*simp-all add*: *wf-conn.simps*)
unfolding *binary-connectives-def* **by** *simp-all*

lemma *wf-conn-list-decomp*[*simp*]:
wf-conn CT l $\longleftrightarrow$ *l* = []
wf-conn CF l $\longleftrightarrow$ *l* = []
wf-conn (*CVar x*) *l* $\longleftrightarrow$ *l* = []
wf-conn CNot (ξ @ φ # ξ') $\longleftrightarrow$ ξ = [] $\wedge$ ξ' = []
apply (*simp-all add*: *wf-conn.simps*)
unfolding *binary-connectives-def* **apply** *simp-all*
by (*metis append-Nil append-is-Nil-conv list.distinct*(*1*) *list.sel*(*3*) *tl-append2*)

lemma *wf-conn-list*:
wf-conn c l $\Longrightarrow$ *conn c l* = *FT* $\longleftrightarrow$ (*c* = *CT* $\wedge$ *l* = [])
wf-conn c l $\Longrightarrow$ *conn c l* = *FF* $\longleftrightarrow$ (*c* = *CF* $\wedge$ *l* = [])
wf-conn c l $\Longrightarrow$ *conn c l* = *FVar x* $\longleftrightarrow$ (*c* = *CVar x* $\wedge$ *l* = [])
wf-conn c l $\Longrightarrow$ *conn c l* = *FAnd a b* $\longleftrightarrow$ (*c* = *CAnd* $\wedge$ *l* = *a* # *b* # [])
wf-conn c l $\Longrightarrow$ *conn c l* = *FOr a b* $\longleftrightarrow$ (*c* = *COr* $\wedge$ *l* = *a* # *b* # [])
wf-conn c l $\Longrightarrow$ *conn c l* = *FEq a b* $\longleftrightarrow$ (*c* = *CEq* $\wedge$ *l* = *a* # *b* # [])
wf-conn c l $\Longrightarrow$ *conn c l* = *FImp a b* $\longleftrightarrow$ (*c* = *CImp* $\wedge$ *l* = *a* # *b* # [])
wf-conn c l $\Longrightarrow$ *conn c l* = *FNot a* $\longleftrightarrow$ (*c* = *CNot* $\wedge$ *l* = *a* # [])
apply (*induct l rule*: *wf-conn.induct*)
unfolding *binary-connectives-def* **by** *auto*

In the binary connective cases, we will often decompose the list of arguments (of length 2) into two elements.

lemma *list-length2-decomp*: *length l* = *2* $\Longrightarrow$ ($\exists$ *a b*. *l* = *a* # *b* # [])
apply (*induct l*, *auto*)
by (*rename-tac l*, *case-tac l*, *auto*)

wf-conn for binary operators means that there are two arguments.

lemma *wf-conn-bin-list-length*:
fixes *l* :: *'v propo list*
assumes *conn*: *c* $\in$ *binary-connectives*
shows *length l* = *2* $\longleftrightarrow$ *wf-conn c l*
proof
assume *length l* = *2*
then show *wf-conn c l* **using** *wf-conn-binary list-length2-decomp* **using** *conn* **by** *metis*
next
assume *wf-conn c l*
then show *length l* = *2* (**is** *?P l*)
proof (*cases rule*: *wf-conn.induct*)
case *wf-conn-nullary*
then show *?P* [] **using** *conn binary-connectives-def*
using *connective.distinct*(*11*) *connective.distinct*(*13*) *connective.distinct*(*9*) **by** *blast*
next
fix ψ :: *'v propo*
case *wf-conn-unary*
then show *?P* [ψ] **using** *conn binary-connectives-def*
using *connective.distinct* **by** *blast*

next
fix ψ ψ':: $'v$ *propo*
show *?P* $[\psi, \psi']$ **by** *auto*
qed
qed

lemma *wf-conn-not-list-length*[*iff*]:
fixes l :: $'v$ *propo list*
shows *wf-conn CNot* $l \longleftrightarrow$ *length* $l = 1$
apply *auto*
apply (*metis append-Nil connective.distinct*(*5*,*17*,*27*) *length-Cons list.size*(*3*) *wf-conn.simps wf-conn-list-decomp*(*4*))
by (*simp add*: *length-Suc-conv wf-conn.simps*)

Decomposing the Not into an element is moreover very useful.

lemma *wf-conn-Not-decomp*:
fixes l :: $'v$ *propo list* **and** a :: $'v$
assumes *corr*: *wf-conn CNot* l
shows $\exists\ a.\ l = [a]$
by (*metis* (*no-types*, *lifting*) *One-nat-def Suc-length-conv corr length-0-conv wf-conn-not-list-length*)

The *wf-conn* remains correct if the length of list does not change. This lemma is very useful when we do one rewriting step

lemma *wf-conn-no-arity-change*:
length l = *length* $l' \Longrightarrow$ *wf-conn* c $l \longleftrightarrow$ *wf-conn* c l'
proof –
{
fix l l'
have *length* l = *length* $l' \Longrightarrow$ *wf-conn* c $l \Longrightarrow$ *wf-conn* c l'
apply (*cases* c l *rule*: *wf-conn.induct*, *auto*)
by (*metis wf-conn-bin-list-length*)
}
then show *length* l = *length* $l' \Longrightarrow$ *wf-conn* c l = *wf-conn* c l' **by** *metis*
qed

lemma *wf-conn-no-arity-change-helper*:
length $(\xi\ @\ \varphi\ \#\ \xi')$ = *length* $(\xi\ @\ \varphi'\ \#\ \xi')$
by *auto*

The injectivity of *conn* is useful to prove equality of the connectives and the lists.

lemma *conn-inj-not*:
assumes *correct*: *wf-conn* c l
and *conn*: *conn* c l = *FNot* ψ
shows c = *CNot* **and** $l = [\psi]$
apply (*cases* c l *rule*: *wf-conn.cases*)
using *correct conn* **unfolding** *binary-connectives-def* **apply** *auto*
apply (*cases* c l *rule*: *wf-conn.cases*)
using *correct conn* **unfolding** *binary-connectives-def* **by** *auto*

lemma *conn-inj*:
fixes c *ca* :: $'v$ *connective* **and** l ψs :: $'v$ *propo list*
assumes *corr*: *wf-conn ca* l
and *corr'*: *wf-conn* c ψs

and *eq*: $conn\ ca\ l = conn\ c\ \psi s$
shows $ca = c \wedge \psi s = l$
using *corr*
proof (*cases ca l rule*: *wf-conn.cases*)
case (*wf-conn-nullary v*)
then show $ca = c \wedge \psi s = l$ **using** *assms*
by (*metis conn.simps*(*1*) *conn.simps*(*2*) *conn.simps*(*3*) *wf-conn-list*(*1*−*3*))
next
case (*wf-conn-unary* ψ')
then have ∗: *FNot* $\psi' = conn\ c\ \psi s$ **using** *conn-inj-not eq assms* **by** *auto*
then have $c = ca$ **by** (*metis conn-inj-not*(*1*) *corr'* *wf-conn-unary*(*2*))
moreover have $\psi s = l$ **using** ∗ *conn-inj-not*(*2*) *corr'* *wf-conn-unary*(*1*) **by** *force*
ultimately show $ca = c \wedge \psi s = l$ **by** *auto*
next
case (*wf-conn-binary* ψ' ψ'')
then show $ca = c \wedge \psi s = l$
using *eq corr'* **unfolding** *binary-connectives-def* **apply** (*cases ca*, *auto simp add*: *wf-conn-list*)
using *wf-conn-list*(*4*−*7*) *corr'* **by** *metis*+
qed

1.1.3 Subformulas and Properties

A characterization using sub-formulas is interesting for rewriting: we will define our relation on the sub-term level, and then lift the rewriting on the term-level. So the rewriting takes place on a subformula.

inductive *subformula* :: $'v\ propo \Rightarrow\ 'v\ propo \Rightarrow bool$ (**infix** $\preceq$ *45*) **for** φ **where**
subformula-refl[*simp*]: $\varphi \preceq \varphi$ |
subformula-into-subformula: $\psi \in set\ l \Longrightarrow wf\text{-}conn\ c\ l \Longrightarrow \varphi \preceq \psi \Longrightarrow \varphi \preceq conn\ c\ l$

On the *subformula-into-subformula*, we can see why we use our *conn* representation: one case is enough to express the subformulas property instead of listing all the cases.

This is an example of a property related to subformulas.

lemma *subformula-in-subformula-not*:
shows *b*: *FNot* $\varphi \preceq \psi \Longrightarrow \varphi \preceq \psi$
apply (*induct rule*: *subformula.induct*)
using *subformula-into-subformula wf-conn-unary subformula-refl list.set-intros*(*1*) *subformula-refl*
by (*fastforce intro*: *subformula-into-subformula*)+

lemma *subformula-in-binary-conn*:
assumes *conn*: $c \in binary\text{-}connectives$
shows $f \preceq conn\ c\ [f, g]$
and $g \preceq conn\ c\ [f, g]$
proof −
have *a*: *wf-conn c* (f# [g]) **using** *conn wf-conn-binary binary-connectives-def* **by** *auto*
moreover have *b*: $f \preceq f$ **using** *subformula-refl* **by** *auto*
ultimately show $f \preceq conn\ c\ [f, g]$
by (*metis append-Nil in-set-conv-decomp subformula-into-subformula*)
next
have *a*: *wf-conn c* ([f] @ [g]) **using** *conn wf-conn-binary binary-connectives-def* **by** *auto*
moreover have *b*: $g \preceq g$ **using** *subformula-refl* **by** *auto*
ultimately show $g \preceq conn\ c\ [f, g]$ **using** *subformula-into-subformula* **by** *force*
qed

lemma *subformula-trans*:

$\psi \preceq \psi' \Longrightarrow \varphi \preceq \psi \Longrightarrow \varphi \preceq \psi'$
apply (*induct* ψ' *rule*: *subformula.inducts*)
by (*auto simp*: *subformula-into-subformula*)

lemma *subformula-leaf*:
fixes $\varphi\ \psi$ :: *'v propo*
assumes *incl*: $\varphi \preceq \psi$
and *simple*: $\psi = FT \lor \psi = FF \lor \psi = FVar\ x$
shows $\varphi = \psi$
using *incl simple*
by (*induct rule*: *subformula.induct*, *auto simp*: *wf-conn-list*)

lemma *subfurmula-not-incl-eq*:
assumes $\varphi \preceq$ *conn c l*
and *wf-conn c l*
and $\forall \psi.\ \psi \in$ *set l* $\longrightarrow \neg\ \varphi \preceq \psi$
shows $\varphi =$ *conn c l*
using *assms* **apply** (*induction conn c l rule*: *subformula.induct*, *auto*)
using *conn-inj* **by** *blast*

lemma *wf-subformula-conn-cases*:
wf-conn c l $\Longrightarrow \varphi \preceq$ *conn c l* $\longleftrightarrow (\varphi =$ *conn c l* $\lor (\exists \psi.\ \psi \in$ *set l* $\land\ \varphi \preceq \psi))$
apply *standard*
using *subfurmula-not-incl-eq* **apply** *metis*
by (*auto simp add*: *subformula-into-subformula*)

lemma *subformula-decomp-explicit*[*simp*]:
$\varphi \preceq FAnd\ \psi\ \psi' \longleftrightarrow (\varphi = FAnd\ \psi\ \psi' \lor \varphi \preceq \psi \lor \varphi \preceq \psi')$ (**is** *?P FAnd*)
$\varphi \preceq FOr\ \psi\ \psi' \longleftrightarrow (\varphi = FOr\ \psi\ \psi' \lor \varphi \preceq \psi \lor \varphi \preceq \psi')$
$\varphi \preceq FEq\ \psi\ \psi' \longleftrightarrow (\varphi = FEq\ \psi\ \psi' \lor \varphi \preceq \psi \lor \varphi \preceq \psi')$
$\varphi \preceq FImp\ \psi\ \psi' \longleftrightarrow (\varphi = FImp\ \psi\ \psi' \lor \varphi \preceq \psi \lor \varphi \preceq \psi')$
proof −
have *wf-conn CAnd* $[\psi, \psi']$ **by** (*simp add*: *binary-connectives-def*)
then have $\varphi \preceq$ *conn CAnd* $[\psi, \psi'] \longleftrightarrow$
$(\varphi =$ *conn CAnd* $[\psi, \psi'] \lor (\exists \psi''.\ \psi'' \in$ *set* $[\psi, \psi'] \land \varphi \preceq \psi''))$
using *wf-subformula-conn-cases* **by** *metis*
then show *?P FAnd* **by** *auto*
next
have *wf-conn COr* $[\psi, \psi']$ **by** (*simp add*: *binary-connectives-def*)
then have $\varphi \preceq$ *conn COr* $[\psi, \psi'] \longleftrightarrow$
$(\varphi =$ *conn COr* $[\psi, \psi'] \lor (\exists \psi''.\ \psi'' \in$ *set* $[\psi, \psi'] \land \varphi \preceq \psi''))$
using *wf-subformula-conn-cases* **by** *metis*
then show *?P FOr* **by** *auto*
next
have *wf-conn CEq* $[\psi, \psi']$ **by** (*simp add*: *binary-connectives-def*)
then have $\varphi \preceq$ *conn CEq* $[\psi, \psi'] \longleftrightarrow$
$(\varphi =$ *conn CEq* $[\psi, \psi'] \lor (\exists \psi''.\ \psi'' \in$ *set* $[\psi, \psi'] \land \varphi \preceq \psi''))$
using *wf-subformula-conn-cases* **by** *metis*
then show *?P FEq* **by** *auto*
next
have *wf-conn CImp* $[\psi, \psi']$ **by** (*simp add*: *binary-connectives-def*)
then have $\varphi \preceq$ *conn CImp* $[\psi, \psi'] \longleftrightarrow$
$(\varphi =$ *conn CImp* $[\psi, \psi'] \lor (\exists \psi''.\ \psi'' \in$ *set* $[\psi, \psi'] \land \varphi \preceq \psi''))$
using *wf-subformula-conn-cases* **by** *metis*
then show *?P FImp* **by** *auto*
qed

lemma *wf-conn-helper-facts*[*iff*]:
  wf-conn CNot [φ]
  wf-conn CT []
  wf-conn CF []
  wf-conn (*CVar x*) []
  wf-conn CAnd [φ, ψ]
  wf-conn COr [φ, ψ]
  wf-conn CImp [φ, ψ]
  wf-conn CEq [φ, ψ]
  using *wf-conn.intros* **unfolding** *binary-connectives-def* **by** *fastforce+*

lemma *exists-c-conn*: $\exists$ *c l*. φ = *conn c l* $\wedge$ *wf-conn c l*
  by (*cases* φ) *force+*

lemma *subformula-conn-decomp*[*simp*]:
  assumes *wf*: *wf-conn c l*
  shows $\varphi \preceq$ *conn c l* $\longleftrightarrow$ (φ = *conn c l* $\vee$ ($\exists$ $\psi \in$ *set l*. $\varphi \preceq \psi$)) (**is** *?A* $\longleftrightarrow$ *?B*)
proof (*rule iffI*)
  {
    fix ξ
    have $\varphi \preceq \xi \Longrightarrow \xi$ = *conn c l* $\Longrightarrow$ *wf-conn c l* $\Longrightarrow \forall$ *x*::*'a propo*$\in$*set l*. $\neg$ $\varphi \preceq x \Longrightarrow \varphi$ = *conn c l*
      apply (*induct rule*: *subformula.induct*)
        apply *simp*
      using *conn-inj* **by** *blast*
  }
  moreover assume *?A*
  ultimately show *?B* **using** *wf* **by** *metis*
next
  assume *?B*
  then show $\varphi \preceq$ *conn c l* **using** *wf wf-subformula-conn-cases* **by** *blast*
qed

lemma *subformula-leaf-explicit*[*simp*]:
  $\varphi \preceq$ *FT* $\longleftrightarrow \varphi$ = *FT*
  $\varphi \preceq$ *FF* $\longleftrightarrow \varphi$ = *FF*
  $\varphi \preceq$ *FVar x* $\longleftrightarrow \varphi$ = *FVar x*
  apply *auto*
  using *subformula-leaf* **by** *metis* +

The variables inside the formula gives precisely the variables that are needed for the formula.

primrec *vars-of-prop*:: *'v propo* $\Rightarrow$ *'v set* **where**
vars-of-prop FT = {} |
vars-of-prop FF = {} |
vars-of-prop (*FVar x*) = {*x*} |
vars-of-prop (*FNot* φ) = *vars-of-prop* φ |
vars-of-prop (*FAnd* φ ψ) = *vars-of-prop* φ $\cup$ *vars-of-prop* ψ |
vars-of-prop (*FOr* φ ψ) = *vars-of-prop* φ $\cup$ *vars-of-prop* ψ |
vars-of-prop (*FImp* φ ψ) = *vars-of-prop* φ $\cup$ *vars-of-prop* ψ |
vars-of-prop (*FEq* φ ψ) = *vars-of-prop* φ $\cup$ *vars-of-prop* ψ

lemma *vars-of-prop-incl-conn*:
  fixes ξ ξ' :: *'v propo list* **and** ψ :: *'v propo* **and** *c* :: *'v connective*
  assumes *corr*: *wf-conn c l* **and** *incl*: $\psi \in$ *set l*
  shows *vars-of-prop* $\psi \subseteq$ *vars-of-prop* (*conn c l*)
proof (*cases c rule*: *connective-cases-arity-2*)

case *nullary*
then have *False* **using** *corr incl* **by** *auto*
then show *vars-of-prop* $\psi \subseteq$ *vars-of-prop* (*conn c l*) **by** *blast*
next
case *binary* **note** $c = this$
then obtain a b **where** *ab*: $l = [a, b]$
using *wf-conn-bin-list-length list-length2-decomp corr* **by** *metis*
then have $\psi = a \lor \psi = b$ **using** *incl* **by** *auto*
then show *vars-of-prop* $\psi \subseteq$ *vars-of-prop* (*conn c l*)
using *ab c* **unfolding** *binary-connectives-def* **by** *auto*
next
case *unary* **note** $c = this$
fix φ :: *'v propo*
have $l = [\psi]$ **using** *corr c incl split-list* **by** *force*
then show *vars-of-prop* $\psi \subseteq$ *vars-of-prop* (*conn c l*) **using** *c* **by** *auto*
qed

The set of variables is compatible with the subformula order.

lemma *subformula-vars-of-prop*:
$\varphi \preceq \psi \Longrightarrow$ *vars-of-prop* $\varphi \subseteq$ *vars-of-prop* ψ
apply (*induct rule*: *subformula.induct*)
apply *simp*
using *vars-of-prop-incl-conn* **by** *blast*

1.1.4 Positions

Instead of 1 or 2 we use L or R

datatype *sign* = L | R

We use *nil* instead of ε.

fun *pos* :: *'v propo* $\Rightarrow$ *sign list set* **where**
pos FF = $\{[]\}$ |
pos FT = $\{[]\}$ |
pos (*FVar x*) = $\{[]\}$ |
pos (*FAnd* φ ψ) = $\{[]\} \cup \{ L \# p \mid p.\ p \in pos\ \varphi\} \cup \{ R \# p \mid p.\ p \in pos\ \psi\}$ |
pos (*FOr* φ ψ) = $\{[]\} \cup \{ L \# p \mid p.\ p \in pos\ \varphi\} \cup \{ R \# p \mid p.\ p \in pos\ \psi\}$ |
pos (*FEq* φ ψ) = $\{[]\} \cup \{ L \# p \mid p.\ p \in pos\ \varphi\} \cup \{ R \# p \mid p.\ p \in pos\ \psi\}$ |
pos (*FImp* φ ψ) = $\{[]\} \cup \{ L \# p \mid p.\ p \in pos\ \varphi\} \cup \{ R \# p \mid p.\ p \in pos\ \psi\}$ |
pos (*FNot* φ) = $\{[]\} \cup \{ L \# p \mid p.\ p \in pos\ \varphi\}$

lemma *finite-pos*: *finite* (*pos* φ)
by (*induct* φ, *auto*)

lemma *finite-inj-comp-set*:
fixes *s* :: *'v set*
assumes *finite*: *finite s*
and *inj*: *inj f*
shows *card* ($\{f\ p \mid p.\ p \in s\}$) = *card s*
using *finite*
proof (*induct s rule*: *finite-induct*)
show *card* $\{f\ p \mid p.\ p \in \{\}\}$ = *card* $\{\}$ **by** *auto*
next
fix *x* :: *'v* **and** *s*:: *'v set*
assume *f*: *finite s* **and** *notin*: $x \notin s$
and *IH*: *card* $\{f\ p \mid p.\ p \in s\}$ = *card s*

have f': *finite* $\{f\ p\ |p.\ p \in \mathit{insert}\ x\ s\}$ **using** *f* **by** *auto*
have *notin'*: $f\ x \notin \{f\ p\ |p.\ p \in s\}$ **using** *notin inj injD* **by** *fastforce*
have $\{f\ p\ |p.\ p \in \mathit{insert}\ x\ s\} = \mathit{insert}\ (f\ x)\ \{f\ p\ |p.\ p\in s\}$ **by** *auto*
then have $\mathit{card}\ \{f\ p\ |p.\ p \in \mathit{insert}\ x\ s\} = 1 + \mathit{card}\ \{f\ p\ |p.\ p \in s\}$
using *finite card-insert-disjoint* f' *notin'* **by** *auto*
moreover have $\ldots = \mathit{card}\ (\mathit{insert}\ x\ s)$ **using** *notin f IH* **by** *auto*
finally show $\mathit{card}\ \{f\ p\ |p.\ p \in \mathit{insert}\ x\ s\} = \mathit{card}\ (\mathit{insert}\ x\ s)$ **.**
qed

lemma *cons-inject*:
inj ((#) *s*)
by (*meson injI list.inject*)

lemma *finite-insert-nil-cons*:
$\mathit{finite}\ s \Longrightarrow \mathit{card}\ (\mathit{insert}\ []\ \{L \# p\ |p.\ p \in s\}) = 1 + \mathit{card}\ \{L \# p\ |p.\ p \in s\}$
using *card-insert-disjoint* **by** *auto*

lemma *cord-not*[*simp*]:
$\mathit{card}\ (\mathit{pos}\ (\mathit{FNot}\ \varphi)) = 1 + \mathit{card}\ (\mathit{pos}\ \varphi)$
by (*simp add*: *cons-inject finite-inj-comp-set finite-pos*)

lemma *card-seperate*:
assumes *finite s1* **and** *finite s2*
shows $\mathit{card}\ (\{L \# p\ |p.\ p \in s1\} \cup \{R \# p\ |p.\ p \in s2\}) = \mathit{card}\ (\{L \# p\ |p.\ p \in s1\})$
$+\ \mathit{card}(\{R \# p\ |p.\ p \in s2\})$ (**is** $\mathit{card}\ (?L{\cup}?R) = \mathit{card}\ ?L + \mathit{card}\ ?R$)
proof −
have *finite ?L* **using** *assms* **by** *auto*
moreover have *finite ?R* **using** *assms* **by** *auto*
moreover have $?L \cap ?R = \{\}$ **by** *blast*
ultimately show *?thesis* **using** *assms card-Un-disjoint* **by** *blast*
qed

definition *prop-size* **where** $\mathit{prop\text{-}size}\ \varphi = \mathit{card}\ (\mathit{pos}\ \varphi)$

lemma *prop-size-vars-of-prop*:
fixes φ :: *'v propo*
shows $\mathit{card}\ (\mathit{vars\text{-}of\text{-}prop}\ \varphi) \le \mathit{prop\text{-}size}\ \varphi$

unfolding *prop-size-def* **apply** (*induct* φ, *auto simp add*: *cons-inject finite-inj-comp-set finite-pos*)
proof −
fix $\varphi 1\ \varphi 2$ :: *'v propo*
assume *IH1*: $\mathit{card}\ (\mathit{vars\text{-}of\text{-}prop}\ \varphi 1) \le \mathit{card}\ (\mathit{pos}\ \varphi 1)$
and *IH2*: $\mathit{card}\ (\mathit{vars\text{-}of\text{-}prop}\ \varphi 2) \le \mathit{card}\ (\mathit{pos}\ \varphi 2)$
let $?L = \{L \# p\ |p.\ p \in \mathit{pos}\ \varphi 1\}$
let $?R = \{R \# p\ |p.\ p \in \mathit{pos}\ \varphi 2\}$
have $\mathit{card}\ (?L \cup ?R) = \mathit{card}\ ?L + \mathit{card}\ ?R$
using *card-seperate finite-pos* **by** *blast*
moreover have $\ldots = \mathit{card}\ (\mathit{pos}\ \varphi 1) + \mathit{card}\ (\mathit{pos}\ \varphi 2)$
by (*simp add*: *cons-inject finite-inj-comp-set finite-pos*)
moreover have $\ldots \ge \mathit{card}\ (\mathit{vars\text{-}of\text{-}prop}\ \varphi 1) + \mathit{card}\ (\mathit{vars\text{-}of\text{-}prop}\ \varphi 2)$ **using** *IH1 IH2* **by** *arith*
then have $\ldots \ge \mathit{card}\ (\mathit{vars\text{-}of\text{-}prop}\ \varphi 1 \cup \mathit{vars\text{-}of\text{-}prop}\ \varphi 2)$ **using** *card-Un-le le-trans* **by** *blast*
ultimately
show $\mathit{card}\ (\mathit{vars\text{-}of\text{-}prop}\ \varphi 1 \cup \mathit{vars\text{-}of\text{-}prop}\ \varphi 2) \le \mathit{Suc}\ (\mathit{card}\ (?L \cup ?R))$
$\mathit{card}\ (\mathit{vars\text{-}of\text{-}prop}\ \varphi 1 \cup \mathit{vars\text{-}of\text{-}prop}\ \varphi 2) \le \mathit{Suc}\ (\mathit{card}\ (?L \cup ?R))$
$\mathit{card}\ (\mathit{vars\text{-}of\text{-}prop}\ \varphi 1 \cup \mathit{vars\text{-}of\text{-}prop}\ \varphi 2) \le \mathit{Suc}\ (\mathit{card}\ (?L \cup ?R))$

$card\ (vars\text{-}of\text{-}prop\ \varphi 1 \cup vars\text{-}of\text{-}prop\ \varphi 2) \leq Suc\ (card\ (?L \cup ?R))$
by *auto*
qed

value $pos\ (FImp\ (FAnd\ (FVar\ P)\ (FVar\ Q))\ (FOr\ (FVar\ P)\ (FVar\ Q)))$

inductive *path-to* :: $sign\ list \Rightarrow {'v}\ propo \Rightarrow {'v}\ propo \Rightarrow bool$ **where**
path-to-refl[*intro*]: $path\text{-}to\ []\ \varphi\ \varphi$ |
path-to-l: $c{\in}binary\text{-}connectives \lor c = CNot \Longrightarrow wf\text{-}conn\ c\ (\varphi\#l) \Longrightarrow path\text{-}to\ p\ \varphi\ \varphi'{\Longrightarrow}$
$path\text{-}to\ (L\#p)\ (conn\ c\ (\varphi\#l))\ \varphi'$ |
path-to-r: $c{\in}binary\text{-}connectives \Longrightarrow wf\text{-}conn\ c\ (\psi\#\varphi\#[]) \Longrightarrow path\text{-}to\ p\ \varphi\ \varphi' \Longrightarrow$
$path\text{-}to\ (R\#p)\ (conn\ c\ (\psi\#\varphi\#[]))\ \varphi'$

There is a deep link between subformulas and pathes: a (correct) path leads to a subformula and a subformula is associated to a given path.

lemma *path-to-subformula*:
$path\text{-}to\ p\ \varphi\ \varphi' \Longrightarrow \varphi' \preceq \varphi$
apply (*induct rule*: *path-to.induct*)
apply *simp*
apply (*metis list.set-intros*(*1*) *subformula-into-subformula*)
using *subformula-trans subformula-in-binary-conn*(*2*) **by** *metis*

lemma *subformula-path-exists*:
fixes $\varphi\ \varphi'$:: $'v\ propo$
shows $\varphi' \preceq \varphi \Longrightarrow \exists p.\ path\text{-}to\ p\ \varphi\ \varphi'$
proof (*induct rule*: *subformula.induct*)
case *subformula-refl*
have $path\text{-}to\ []\ \varphi'\ \varphi'$ **by** *auto*
then show $\exists p.\ path\text{-}to\ p\ \varphi'\ \varphi'$ **by** *metis*
next
case ($subformula\text{-}into\text{-}subformula\ \psi\ l\ c$)
note $wf = this(2)$ **and** $IH = this(4)$ **and** $\psi = this(1)$
then obtain p **where** p: $path\text{-}to\ p\ \psi\ \varphi'$ **by** *metis*
{
fix x :: $'v$
assume $c = CT \lor c = CF \lor c = CVar\ x$
then have *False* **using** *subformula-into-subformula* **by** *auto*
then have $\exists p.\ path\text{-}to\ p\ (conn\ c\ l)\ \varphi'$ **by** *blast*
}
moreover {
assume c: $c = CNot$
then have $l = [\psi]$ **using** $wf\ \psi$ *wf-conn-Not-decomp* **by** *fastforce*
then have $path\text{-}to\ (L\ \#\ p)\ (conn\ c\ l)\ \varphi'$ **by** (*metis c wf-conn-unary p path-to-l*)
then have $\exists p.\ path\text{-}to\ p\ (conn\ c\ l)\ \varphi'$ **by** *blast*
}
moreover {
assume c: $c{\in}\ binary\text{-}connectives$
obtain $a\ b$ **where** ab: $[a, b] = l$ **using** *subformula-into-subformula c wf-conn-bin-list-length list-length2-decomp* **by** *metis*
then have $a = \psi \lor b = \psi$ **using** ψ **by** *auto*
then have $path\text{-}to\ (L\ \#\ p)\ (conn\ c\ l)\ \varphi' \lor path\text{-}to\ (R\ \#\ p)\ (conn\ c\ l)\ \varphi'$ **using** *c path-to-l path-to-r p ab* **by** (*metis wf-conn-binary*)
then have $\exists p.\ path\text{-}to\ p\ (conn\ c\ l)\ \varphi'$ **by** *blast*
}
ultimately show $\exists p.\ path\text{-}to\ p\ (conn\ c\ l)\ \varphi'$ **using** *connective-cases-arity* **by** *metis*
qed

fun *replace-at* :: *sign list* ⇒ *'v propo* ⇒ *'v propo* ⇒ *'v propo* **where**
$replace\text{-}at$ [] - $\psi = \psi$ |
$replace\text{-}at\ (L \# l)\ (FAnd\ \varphi\ \varphi')\ \psi = FAnd\ (replace\text{-}at\ l\ \varphi\ \psi)\ \varphi'$|
$replace\text{-}at\ (R \# l)\ (FAnd\ \varphi\ \varphi')\ \psi = FAnd\ \varphi\ (replace\text{-}at\ l\ \varphi'\ \psi)$ |
$replace\text{-}at\ (L \# l)\ (FOr\ \varphi\ \varphi')\ \psi = FOr\ (replace\text{-}at\ l\ \varphi\ \psi)\ \varphi'$ |
$replace\text{-}at\ (R \# l)\ (FOr\ \varphi\ \varphi')\ \psi = FOr\ \varphi\ (replace\text{-}at\ l\ \varphi'\ \psi)$ |
$replace\text{-}at\ (L \# l)\ (FEq\ \varphi\ \varphi')\ \psi = FEq\ (replace\text{-}at\ l\ \varphi\ \psi)\ \varphi'$|
$replace\text{-}at\ (R \# l)\ (FEq\ \varphi\ \varphi')\ \psi = FEq\ \varphi\ (replace\text{-}at\ l\ \varphi'\ \psi)$ |
$replace\text{-}at\ (L \# l)\ (FImp\ \varphi\ \varphi')\ \psi = FImp\ (replace\text{-}at\ l\ \varphi\ \psi)\ \varphi'$|
$replace\text{-}at\ (R \# l)\ (FImp\ \varphi\ \varphi')\ \psi = FImp\ \varphi\ (replace\text{-}at\ l\ \varphi'\ \psi)$ |
$replace\text{-}at\ (L \# l)\ (FNot\ \varphi)\ \psi = FNot\ (replace\text{-}at\ l\ \varphi\ \psi)$

1.2 Semantics over the Syntax

Given the syntax defined above, we define a semantics, by defining an evaluation function *eval*. This function is the bridge between the logic as we define it here and the built-in logic of Isabelle.

fun *eval* :: (*'v* ⇒ *bool*) ⇒ *'v propo* ⇒ *bool* (**infix** ⊨ *50*) **where**
$\mathcal{A} \models FT = True$ |
$\mathcal{A} \models FF = False$ |
$\mathcal{A} \models FVar\ v = (\mathcal{A}\ v)$ |
$\mathcal{A} \models FNot\ \varphi = (\neg(\mathcal{A} \models \varphi))$ |
$\mathcal{A} \models FAnd\ \varphi_1\ \varphi_2 = (\mathcal{A} \models \varphi_1 \wedge \mathcal{A} \models \varphi_2)$ |
$\mathcal{A} \models FOr\ \varphi_1\ \varphi_2 = (\mathcal{A} \models \varphi_1 \vee \mathcal{A} \models \varphi_2)$ |
$\mathcal{A} \models FImp\ \varphi_1\ \varphi_2 = (\mathcal{A} \models \varphi_1 \longrightarrow \mathcal{A} \models \varphi_2)$ |
$\mathcal{A} \models FEq\ \varphi_1\ \varphi_2 = (\mathcal{A} \models \varphi_1 \longleftrightarrow \mathcal{A} \models \varphi_2)$

definition *evalf* (**infix** ⊨*f 50*) **where**
$evalf\ \varphi\ \psi = (\forall A.\ A \models \varphi \longrightarrow A \models \psi)$

The deduction rule is in the book. And the proof looks like to the one of the book.

theorem *deduction-theorem*:
  $\varphi \models f\ \psi \longleftrightarrow (\forall A.\ A \models FImp\ \varphi\ \psi)$
proof
  assume *H*: $\varphi \models f\ \psi$
  {
    fix *A*
    have $A \models FImp\ \varphi\ \psi$
      proof (*cases* $A \models \varphi$)
        case *True*
        then have $A \models \psi$ **using** *H* **unfolding** *evalf-def* **by** *metis*
        then show $A \models FImp\ \varphi\ \psi$ **by** *auto*
      next
        case *False*
        then show $A \models FImp\ \varphi\ \psi$ **by** *auto*
      qed
  }
  then show $\forall A.\ A \models FImp\ \varphi\ \psi$ **by** *blast*
next
  assume *A*: $\forall A.\ A \models FImp\ \varphi\ \psi$
  show $\varphi \models f\ \psi$
    proof (*rule ccontr*)
      assume $\neg\ \varphi \models f\ \psi$
      then obtain *A* **where** $A \models \varphi$ **and** $\neg\ A \models \psi$ **using** *evalf-def* **by** *metis*

then have $\neg A \models FImp\ \varphi\ \psi$ **by** *auto*
then show *False* **using** *A* **by** *blast*
qed
qed

A shorter proof:

lemma $\varphi \models f\ \psi \longleftrightarrow (\forall A.\ A \models FImp\ \varphi\ \psi)$
by (*simp add*: *evalf-def*)

definition *same-over-set*:: $('v \Rightarrow bool) \Rightarrow ('v \Rightarrow bool) \Rightarrow 'v\ set \Rightarrow bool$ **where**
$same\text{-}over\text{-}set\ A\ B\ S = (\forall c \in S.\ A\ c = B\ c)$

If two mapping A and B have the same value over the variables, then the same formula are satisfiable.

lemma *same-over-set-eval*:
assumes *same-over-set A B* (*vars-of-prop* φ)
shows $A \models \varphi \longleftrightarrow B \models \varphi$
using *assms* **unfolding** *same-over-set-def* **by** (*induct* φ, *auto*)

end
theory *Prop-Abstract-Transformation*
imports *Prop-Logic Weidenbach-Book-Base.Wellfounded-More*

begin

This file is devoted to abstract properties of the transformations, like consistency preservation and lifting from terms to proposition.

1.3 Rewrite Systems and Properties

1.3.1 Lifting of Rewrite Rules

We can lift a rewrite relation r over a full1 formula: the relation r works on terms, while *propo-rew-step* works on formulas.

inductive *propo-rew-step* :: $('v\ propo \Rightarrow 'v\ propo \Rightarrow bool) \Rightarrow 'v\ propo \Rightarrow 'v\ propo \Rightarrow bool$
for r :: $'v\ propo \Rightarrow 'v\ propo \Rightarrow bool$ **where**
global-rel: $r\ \varphi\ \psi \Longrightarrow propo\text{-}rew\text{-}step\ r\ \varphi\ \psi\ |$
propo-rew-one-step-lift: $propo\text{-}rew\text{-}step\ r\ \varphi\ \varphi' \Longrightarrow wf\text{-}conn\ c\ (\psi s\ @\ \varphi\ \#\ \psi s')$
$\Longrightarrow propo\text{-}rew\text{-}step\ r\ (conn\ c\ (\psi s\ @\ \varphi\ \#\ \psi s'))\ (conn\ c\ (\psi s\ @\ \varphi' \#\ \psi s'))$

Here is a more precise link between the lifting and the subformulas: if a rewriting takes place between φ and φ', then there are two subformulas ψ in φ and ψ' in φ', ψ' is the result of the rewriting of r on ψ.

This lemma is only a health condition:

lemma *propo-rew-step-subformula-imp*:
shows $propo\text{-}rew\text{-}step\ r\ \varphi\ \varphi' \Longrightarrow \exists\ \psi\ \psi'.\ \psi \preceq \varphi \wedge \psi' \preceq \varphi' \wedge r\ \psi\ \psi'$
apply (*induct rule*: *propo-rew-step.induct*)
using *subformula.simps subformula-into-subformula* **apply** *blast*
using *wf-conn-no-arity-change subformula-into-subformula wf-conn-no-arity-change-helper in-set-conv-decomp* **by** *metis*

The converse is moreover true: if there is a ψ and ψ', then every formula φ containing ψ, can be rewritten into a formula φ', such that it contains φ'.

lemma *propo-rew-step-subformula-rec*:
  fixes $\psi\ \psi'\ \varphi$ :: *'v propo*
  shows $\psi \preceq \varphi \Longrightarrow r\ \psi\ \psi' \Longrightarrow (\exists \varphi'.\ \psi' \preceq \varphi' \wedge$ *propo-rew-step* $r\ \varphi\ \varphi')$
proof (*induct* φ *rule*: *subformula.induct*)
  case *subformula-refl*
  then have *propo-rew-step* $r\ \psi\ \psi'$ **using** *propo-rew-step.intros* **by** *auto*
  moreover have $\psi' \preceq \psi'$ **using** *Prop-Logic.subformula-refl* **by** *auto*
  ultimately show $\exists \varphi'.\ \psi' \preceq \varphi' \wedge$ *propo-rew-step* $r\ \psi\ \varphi'$ **by** *fastforce*
next
  case (*subformula-into-subformula* ψ'' l c)
  note $IH = this(4)$ **and** $r = this(5)$ **and** $\psi'' = this(1)$ **and** $wf = this(2)$ **and** $incl = this(3)$
  then obtain φ' **where** $*$: $\psi' \preceq \varphi' \wedge$ *propo-rew-step* $r\ \psi''\ \varphi'$ **by** *metis*
  moreover obtain $\xi\ \xi'$ :: *'v propo list* **where**
    l: $l = \xi\ @\ \psi''\ \#\ \xi'$ **using** *List.split-list* ψ'' **by** *metis*
  ultimately have *propo-rew-step* r (*conn* c l) (*conn* c $(\xi\ @\ \varphi'\ \#\ \xi')$)
    using *propo-rew-step.intros*(2) *wf* **by** *metis*
  moreover have $\psi' \preceq$ *conn* c $(\xi\ @\ \varphi'\ \#\ \xi')$
    using *wf* $*$ *wf-conn-no-arity-change Prop-Logic.subformula-into-subformula*
    by (*metis* (*no-types*) *in-set-conv-decomp* l *wf-conn-no-arity-change-helper*)
  ultimately show $\exists \varphi'.\ \psi' \preceq \varphi' \wedge$ *propo-rew-step* r (*conn* c l) φ' **by** *metis*
qed

lemma *propo-rew-step-subformula*:
  $(\exists \psi\ \psi'.\ \psi \preceq \varphi \wedge r\ \psi\ \psi') \longleftrightarrow (\exists \varphi'.$ *propo-rew-step* $r\ \varphi\ \varphi')$
  using *propo-rew-step-subformula-imp propo-rew-step-subformula-rec* **by** *metis+*

lemma *consistency-decompose-into-list*:
  assumes *wf*: *wf-conn* c l **and** *wf'*: *wf-conn* c l'
  and *same*: $\forall n.\ A \models l\ !\ n \longleftrightarrow (A \models l'\ !\ n)$
  shows $A \models$ *conn* c $l \longleftrightarrow A \models$ *conn* c l'
proof (*cases* c *rule*: *connective-cases-arity-2*)
  case *nullary*
  then show $(A \models$ *conn* c $l) \longleftrightarrow (A \models$ *conn* c $l')$ **using** *wf wf'* **by** *auto*
next
  case *unary* **note** $c = this$
  then obtain a **where** l: $l = [a]$ **using** *wf-conn-Not-decomp wf* **by** *metis*
  obtain a' **where** l': $l' = [a']$ **using** *wf-conn-Not-decomp wf'* c **by** *metis*
  have $A \models a \longleftrightarrow A \models a'$ **using** l l' **by** (*metis nth-Cons-0 same*)
  then show $A \models$ *conn* c $l \longleftrightarrow A \models$ *conn* c l' **using** l l' c **by** *auto*
next
  case *binary* **note** $c = this$
  then obtain a b **where** l: $l = [a, b]$
    using *wf-conn-bin-list-length list-length2-decomp wf* **by** *metis*
  obtain a' b' **where** l': $l' = [a', b']$
    using *wf-conn-bin-list-length list-length2-decomp wf'* c **by** *metis*

  have p: $A \models a \longleftrightarrow A \models a'$ $A \models b \longleftrightarrow A \models b'$
    using l l' *same* **by** (*metis diff-Suc-1 nth-Cons' nat.distinct*(2))+
  show $A \models$ *conn* c $l \longleftrightarrow A \models$ *conn* c l'
    using *wf* c p **unfolding** *binary-connectives-def* l l' **by** *auto*
qed

Relation between *propo-rew-step* and the rewriting we have seen before: *propo-rew-step* $r\ \varphi\ \varphi'$ means that we rewrite ψ inside φ (ie at a path p) into ψ'.

lemma *propo-rew-step-rewrite*:
  fixes $\varphi\ \varphi'$ :: *'v propo* **and** r :: *'v propo* $\Rightarrow$ *'v propo* $\Rightarrow$ *bool*

assumes *propo-rew-step r* φ φ'
shows $\exists \psi\ \psi'\ p.\ r\ \psi\ \psi' \wedge$ *path-to* $p\ \varphi\ \psi \wedge$ *replace-at* $p\ \varphi\ \psi' = \varphi'$
using *assms*
proof (*induct rule*: *propo-rew-step.induct*)
case(*global-rel* φ ψ)
moreover have *path-to* [] φ φ **by** *auto*
moreover have *replace-at* [] φ $\psi = \psi$ **by** *auto*
ultimately show *?case* **by** *metis*
next
case (*propo-rew-one-step-lift* φ φ' c ξ ξ') **note** *rel* = *this*(*1*) **and** *IH0* = *this*(*2*) **and** *corr* = *this*(*3*)
obtain ψ ψ' p **where** *IH*: $r\ \psi\ \psi' \wedge$ *path-to* $p\ \varphi\ \psi \wedge$ *replace-at* $p\ \varphi\ \psi' = \varphi'$ **using** *IH0* **by** *metis*

{
fix $x :: 'v$
assume $c = CT \vee c = CF \vee c = CVar\ x$
then have *False* **using** *corr* **by** *auto*
then have $\exists \psi\ \psi'\ p.\ r\ \psi\ \psi' \wedge$ *path-to* p (*conn* c (ξ@ (φ # ξ'))) ψ
$\wedge$ *replace-at* p (*conn* c (ξ@ (φ # ξ'))) ψ' = *conn* c (ξ@ (φ' # ξ'))
by *fast*
}
moreover {
assume c: c = *CNot*
then have *empty*: ξ =[] ξ'=[] **using** *corr* **by** *auto*
have *path-to* ($L\#p$) (*conn* c (ξ@ (φ # ξ'))) ψ
using *c empty IH wf-conn-unary path-to-l* **by** *fastforce*
moreover have *replace-at* ($L\#p$) (*conn* c (ξ@ (φ # ξ'))) ψ' = *conn* c (ξ@ (φ' # ξ'))
using *c empty IH* **by** *auto*
ultimately have $\exists \psi\ \psi'\ p.\ r\ \psi\ \psi' \wedge$ *path-to* p (*conn* c (ξ@ (φ # ξ'))) ψ
$\wedge$ *replace-at* p (*conn* c (ξ@ (φ # ξ'))) ψ' = *conn* c (ξ@ (φ' # ξ'))
using *IH* **by** *metis*
}
moreover {
assume c: $c \in$ *binary-connectives*
have *length* (ξ@ φ # ξ') = *2* **using** *wf-conn-bin-list-length corr c* **by** *metis*
then have *length* ξ + *length* ξ' = *1* **by** *auto*
then have *ld*: (*length* ξ = *1* $\wedge$ *length* ξ' = *0*) $\vee$ (*length* ξ = *0* $\wedge$ *length* ξ' = *1*) **by** *arith*
obtain a b **where** *ab*: (ξ=[] $\wedge$ ξ'=[b]) $\vee$ (ξ=[a] $\wedge$ ξ'=[])
using *ld* **by** (*case-tac* ξ, *case-tac* ξ', *auto*)
{
assume φ: ξ=[] $\wedge$ ξ'=[b]
have *path-to* ($L\#p$) (*conn* c (ξ@ (φ # ξ'))) ψ
using φ *c IH ab corr* **by** (*simp add*: *path-to-l*)
moreover have *replace-at* ($L\#p$) (*conn* c (ξ@ (φ # ξ'))) ψ' = *conn* c (ξ@ (φ' # ξ'))
using *c IH ab* φ **unfolding** *binary-connectives-def* **by** *auto*
ultimately have $\exists \psi\ \psi'\ p.\ r\ \psi\ \psi' \wedge$ *path-to* p (*conn* c (ξ@ (φ # ξ'))) ψ
$\wedge$ *replace-at* p (*conn* c (ξ@ (φ # ξ'))) ψ' = *conn* c (ξ@ (φ' # ξ'))
using *IH* **by** *metis*
}
moreover {
assume φ: ξ=[a] ξ'=[]
then have *path-to* ($R\#p$) (*conn* c (ξ@ (φ # ξ'))) ψ
using *c IH corr path-to-r corr* φ **by** (*simp add*: *path-to-r*)
moreover have *replace-at* ($R\#p$) (*conn* c (ξ@ (φ # ξ'))) ψ' = *conn* c (ξ@ (φ' # ξ'))
using *c IH ab* φ **unfolding** *binary-connectives-def* **by** *auto*
ultimately have *?case* **using** *IH* **by** *metis*
}

ultimately have *?case* **using** *ab* **by** *blast*
}
ultimately show *?case* **using** *connective-cases-arity* **by** *blast*
qed

1.3.2 Consistency Preservation

We define *preserve-models*: it means that a relation preserves consistency.

definition *preserve-models* **where**
$preserve\text{-}models\ r \longleftrightarrow (\forall \varphi\ \psi.\ r\ \varphi\ \psi \longrightarrow (\forall A.\ A \models \varphi \longleftrightarrow A \models \psi))$

lemma *propo-rew-step-preservers-val-explicit*:
$propo\text{-}rew\text{-}step\ r\ \varphi\ \psi \Longrightarrow preserve\text{-}models\ r \Longrightarrow propo\text{-}rew\text{-}step\ r\ \varphi\ \psi \Longrightarrow (\forall A.\ A \models\varphi \longleftrightarrow A\models\psi)$
unfolding *preserve-models-def*
proof (*induction rule*: *propo-rew-step.induct*)
case *global-rel*
then show *?case* **by** *simp*
next
case $(propo\text{-}rew\text{-}one\text{-}step\text{-}lift\ \varphi\ \varphi'\ c\ \xi\ \xi')$ **note** $rel = this(1)$ **and** $wf = this(2)$
and $IH = this(3)[OF\ this(4)\ this(1)]$ **and** $consistent = this(4)$
{
fix A
from IH **have** $\forall n.\ (A \models (\xi\ @\ \varphi\ \#\ \xi')\ !\ n) = (A \models (\xi\ @\ \varphi'\ \#\ \xi')\ !\ n)$
by (*metis* (*mono-tags*, *hide-lams*) *list-update-length nth-Cons-0 nth-append-length-plus nth-list-update-neq*)
then have $(A \models conn\ c\ (\xi\ @\ \varphi\ \#\ \xi')) = (A \models conn\ c\ (\xi\ @\ \varphi'\ \#\ \xi'))$
by (*meson consistency-decompose-into-list wf wf-conn-no-arity-change-helper wf-conn-no-arity-change*)
}
then show $\forall A.\ A \models conn\ c\ (\xi\ @\ \varphi\ \#\ \xi') \longleftrightarrow A \models conn\ c\ (\xi\ @\ \varphi'\ \#\ \xi')$ **by** *auto*
qed

lemma *propo-rew-step-preservers-val′*:
assumes *preserve-models r*
shows *preserve-models* (*propo-rew-step r*)
using *assms* **by** (*simp add*: *preserve-models-def propo-rew-step-preservers-val-explicit*)

lemma *preserve-models-OO*[*intro*]:
$preserve\text{-}models\ f \Longrightarrow preserve\text{-}models\ g \Longrightarrow preserve\text{-}models\ (f\ OO\ g)$
unfolding *preserve-models-def* **by** *auto*

lemma *star-consistency-preservation-explicit*:
assumes $(propo\text{-}rew\text{-}step\ r)\hat{}{*}{*}\ \varphi\ \psi$ **and** *preserve-models r*
shows $\forall A.\ A \models \varphi \longleftrightarrow A \models \psi$
using *assms* **by** (*induct rule*: *rtranclp-induct*)
(*auto simp add*: *propo-rew-step-preservers-val-explicit*)

lemma *star-consistency-preservation*:
$preserve\text{-}models\ r \Longrightarrow preserve\text{-}models\ (propo\text{-}rew\text{-}step\ r)\hat{}{*}{*}$
by (*simp add*: *star-consistency-preservation-explicit preserve-models-def*)

1.3.3 Full Lifting

In the previous a relation was lifted to a formula, now we define the relation such it is applied as long as possible. The definition is thus simply: it can be derived and nothing more can be derived.

lemma *full-ropo-rew-step-preservers-val*[*simp*]:
preserve-models r $\Longrightarrow$ *preserve-models* (*full* (*propo-rew-step r*))
  by (*metis full-def preserve-models-def star-consistency-preservation*)

lemma *full-propo-rew-step-subformula*:
full (*propo-rew-step r*) φ' φ $\Longrightarrow \neg(\exists\ \psi\ \psi'.\ \psi \preceq \varphi \wedge r\ \psi\ \psi')$
  unfolding *full-def* **using** *propo-rew-step-subformula-rec* **by** *metis*

1.4 Transformation testing

1.4.1 Definition and first Properties

To prove correctness of our transformation, we create a *all-subformula-st* predicate. It tests recursively all subformulas. At each step, the actual formula is tested. The aim of this *test-symb* function is to test locally some properties of the formulas (i.e. at the level of the connective or at first level). This allows a clause description between the rewrite relation and the *test-symb*

definition *all-subformula-st* :: (*'a propo* $\Rightarrow$ *bool*) $\Rightarrow$ *'a propo* $\Rightarrow$ *bool* **where**
all-subformula-st test-symb $\varphi \equiv \forall\psi.\ \psi \preceq \varphi \longrightarrow$ *test-symb* ψ

lemma *test-symb-imp-all-subformula-st*[*simp*]:
  test-symb FT $\Longrightarrow$ *all-subformula-st test-symb FT*
  test-symb FF $\Longrightarrow$ *all-subformula-st test-symb FF*
  test-symb (*FVar x*) $\Longrightarrow$ *all-subformula-st test-symb* (*FVar x*)
  unfolding *all-subformula-st-def* **using** *subformula-leaf* **by** *metis+*

lemma *all-subformula-st-test-symb-true-phi*:
  all-subformula-st test-symb $\varphi \Longrightarrow$ *test-symb* φ
  unfolding *all-subformula-st-def* **by** *auto*

lemma *all-subformula-st-decomp-imp*:
  wf-conn c l $\Longrightarrow$ (*test-symb* (*conn c l*) $\wedge$ ($\forall\varphi\in$ *set l. all-subformula-st test-symb* φ))
  $\Longrightarrow$ *all-subformula-st test-symb* (*conn c l*)
  unfolding *all-subformula-st-def* **by** *auto*

To ease the finding of proofs, we give some explicit theorem about the decomposition.

lemma *all-subformula-st-decomp-rec*:
  all-subformula-st test-symb (*conn c l*) $\Longrightarrow$ *wf-conn c l*
    $\Longrightarrow$ (*test-symb* (*conn c l*) $\wedge$ ($\forall\varphi\in$ *set l. all-subformula-st test-symb* φ))
  unfolding *all-subformula-st-def* **by** *auto*

lemma *all-subformula-st-decomp*:
  fixes *c* :: *'v connective* **and** *l* :: *'v propo list*
  assumes *wf-conn c l*
  shows *all-subformula-st test-symb* (*conn c l*)
    $\longleftrightarrow$ (*test-symb* (*conn c l*) $\wedge$ ($\forall\varphi\in$ *set l. all-subformula-st test-symb* φ))
  using *assms all-subformula-st-decomp-rec all-subformula-st-decomp-imp* **by** *metis*

lemma *helper-fact*: $c \in$ *binary-connectives* $\longleftrightarrow$ (c = *COr* $\vee$ c = *CAnd* $\vee$ c = *CEq* $\vee$ c = *CImp*)
  unfolding *binary-connectives-def* **by** *auto*
lemma *all-subformula-st-decomp-explicit*[*simp*]:
  fixes φ ψ :: *'v propo*
  shows *all-subformula-st test-symb* (*FAnd* φ ψ)
      $\longleftrightarrow$ (*test-symb* (*FAnd* φ ψ) $\wedge$ *all-subformula-st test-symb* φ $\wedge$ *all-subformula-st test-symb* ψ)
  and *all-subformula-st test-symb* (*FOr* φ ψ)
     $\longleftrightarrow$ (*test-symb* (*FOr* φ ψ) $\wedge$ *all-subformula-st test-symb* φ $\wedge$ *all-subformula-st test-symb* ψ)
  and *all-subformula-st test-symb* (*FNot* φ)
     $\longleftrightarrow$ (*test-symb* (*FNot* φ) $\wedge$ *all-subformula-st test-symb* φ)
  and *all-subformula-st test-symb* (*FEq* φ ψ)
     $\longleftrightarrow$ (*test-symb* (*FEq* φ ψ) $\wedge$ *all-subformula-st test-symb* φ $\wedge$ *all-subformula-st test-symb* ψ)
  and *all-subformula-st test-symb* (*FImp* φ ψ)
     $\longleftrightarrow$ (*test-symb* (*FImp* φ ψ) $\wedge$ *all-subformula-st test-symb* φ $\wedge$ *all-subformula-st test-symb* ψ)
proof −
  have *all-subformula-st test-symb* (*FAnd* φ ψ) $\longleftrightarrow$ *all-subformula-st test-symb* (*conn CAnd* [φ, ψ])
    by *auto*
  moreover have ... $\longleftrightarrow$*test-symb* (*conn CAnd* [φ, ψ])$\wedge$($\forall \xi \in$ *set* [φ, ψ]. *all-subformula-st test-symb* ξ)
    using *all-subformula-st-decomp wf-conn-helper-facts*(*5*) **by** *metis*
  finally show *all-subformula-st test-symb* (*FAnd* φ ψ)
    $\longleftrightarrow$ (*test-symb* (*FAnd* φ ψ) $\wedge$ *all-subformula-st test-symb* φ $\wedge$ *all-subformula-st test-symb* ψ)
    by *simp*

  have *all-subformula-st test-symb* (*FOr* φ ψ) $\longleftrightarrow$ *all-subformula-st test-symb* (*conn COr* [φ, ψ])
    by *auto*
  moreover have ...$\longleftrightarrow$
    (*test-symb* (*conn COr* [φ, ψ]) $\wedge$ ($\forall \xi \in$ *set* [φ, ψ]. *all-subformula-st test-symb* ξ))
    using *all-subformula-st-decomp wf-conn-helper-facts*(*6*) **by** *metis*
  finally show *all-subformula-st test-symb* (*FOr* φ ψ)
    $\longleftrightarrow$ (*test-symb* (*FOr* φ ψ) $\wedge$ *all-subformula-st test-symb* φ $\wedge$ *all-subformula-st test-symb* ψ)
    by *simp*

  have *all-subformula-st test-symb* (*FEq* φ ψ) $\longleftrightarrow$ *all-subformula-st test-symb* (*conn CEq* [φ, ψ])
    by *auto*
  moreover have ...
    $\longleftrightarrow$ (*test-symb* (*conn CEq* [φ, ψ]) $\wedge$ ($\forall \xi \in$ *set* [φ, ψ]. *all-subformula-st test-symb* ξ))
    using *all-subformula-st-decomp wf-conn-helper-facts*(*8*) **by** *metis*
  finally show *all-subformula-st test-symb* (*FEq* φ ψ)
    $\longleftrightarrow$ (*test-symb* (*FEq* φ ψ) $\wedge$ *all-subformula-st test-symb* φ $\wedge$ *all-subformula-st test-symb* ψ)
    by *simp*

  have *all-subformula-st test-symb* (*FImp* φ ψ) $\longleftrightarrow$ *all-subformula-st test-symb* (*conn CImp* [φ, ψ])
    by *auto*
  moreover have ...
    $\longleftrightarrow$(*test-symb* (*conn CImp* [φ, ψ]) $\wedge$ ($\forall \xi \in$ *set* [φ, ψ]. *all-subformula-st test-symb* ξ))
    using *all-subformula-st-decomp wf-conn-helper-facts*(*7*) **by** *metis*
  finally show *all-subformula-st test-symb* (*FImp* φ ψ)
    $\longleftrightarrow$ (*test-symb* (*FImp* φ ψ) $\wedge$ *all-subformula-st test-symb* φ $\wedge$ *all-subformula-st test-symb* ψ)
    by *simp*

  have *all-subformula-st test-symb* (*FNot* φ) $\longleftrightarrow$ *all-subformula-st test-symb* (*conn CNot* [φ])
    by *auto*
  moreover have ... = (*test-symb* (*conn CNot* [φ]) $\wedge$ ($\forall \xi \in$ *set* [φ]. *all-subformula-st test-symb* ξ))
    using *all-subformula-st-decomp wf-conn-helper-facts*(*1*) **by** *metis*
  finally show *all-subformula-st test-symb* (*FNot* φ)

$\longleftrightarrow$ (*test-symb* (*FNot* φ) $\wedge$ *all-subformula-st test-symb* φ) **by** *simp*
qed

As *all-subformula-st* tests recursively, the function is true on every subformula.

lemma *subformula-all-subformula-st*:
$\psi \preceq \varphi \Longrightarrow$ *all-subformula-st test-symb* $\varphi \Longrightarrow$ *all-subformula-st test-symb* ψ
by (*induct rule*: *subformula.induct*, *auto simp add*: *all-subformula-st-decomp*)

The following theorem *no-test-symb-step-exists* shows the link between the *test-symb* function and the corresponding rewrite relation r: if we assume that if every time *test-symb* is true, then a r can be applied, finally as long as $\neg$ *all-subformula-st test-symb* φ, then something can be rewritten in φ.

lemma *no-test-symb-step-exists*:
fixes r:: $'v$ *propo* $\Rightarrow$ $'v$ *propo* $\Rightarrow$ *bool* **and** *test-symb*:: $'v$ *propo* $\Rightarrow$ *bool* **and** x :: $'v$
and φ :: $'v$ *propo*
assumes
test-symb-false-nullary: $\forall x.$ *test-symb FF* $\wedge$ *test-symb FT* $\wedge$ *test-symb* (*FVar* x) **and**
$\forall \varphi'. \varphi' \preceq \varphi \longrightarrow (\neg$*test-symb* $\varphi') \longrightarrow (\exists\ \psi.\ r\ \varphi'\ \psi)$ **and**
$\neg$ *all-subformula-st test-symb* φ
shows $\exists \psi\ \psi'.\ \psi \preceq \varphi \wedge r\ \psi\ \psi'$
using *assms*
proof (*induct* φ *rule*: *propo-induct-arity*)
case (*nullary* φ x)
then show $\exists \psi\ \psi'.\ \psi \preceq \varphi \wedge r\ \psi\ \psi'$
using *wf-conn-nullary test-symb-false-nullary* **by** *fastforce*
next
case (*unary* φ) **note** *IH* = *this*(*1*)[*OF this*(*2*)] **and** r = *this*(*2*) **and** *nst* = *this*(*3*) **and** *subf* = *this*(*4*)
from r *IH nst* **have** H: $\neg$ *all-subformula-st test-symb* $\varphi \Longrightarrow \exists \psi.\ \psi \preceq \varphi \wedge (\exists \psi'.\ r\ \psi\ \psi')$
by (*metis subformula-in-subformula-not subformula-refl subformula-trans*)
{
assume n: $\neg$*test-symb* (*FNot* φ)
obtain ψ **where** r (*FNot* φ) ψ **using** *subformula-refl r n nst* **by** *blast*
moreover have *FNot* $\varphi \preceq$ *FNot* φ **using** *subformula-refl* **by** *auto*
ultimately have $\exists \psi\ \psi'.\ \psi \preceq$ *FNot* $\varphi \wedge r\ \psi\ \psi'$ **by** *metis*
}
moreover {
assume n: *test-symb* (*FNot* φ)
then have $\neg$ *all-subformula-st test-symb* φ
using *all-subformula-st-decomp-explicit*(*3*) *nst subf* **by** *blast*
then have $\exists \psi\ \psi'.\ \psi \preceq$ *FNot* $\varphi \wedge r\ \psi\ \psi'$
using *H subformula-in-subformula-not subformula-refl subformula-trans* **by** *blast*
}
ultimately show $\exists \psi\ \psi'.\ \psi \preceq$ *FNot* $\varphi \wedge r\ \psi\ \psi'$ **by** *blast*
next
case (*binary* φ $\varphi 1$ $\varphi 2$)
note *IH*φ*1-0* = *this*(*1*)[*OF this*(*4*)] **and** *IH*φ*2-0* = *this*(*2*)[*OF this*(*4*)] **and** r = *this*(*4*)
and φ = *this*(*3*) **and** *le* = *this*(*5*) **and** *nst* = *this*(*6*)

obtain c :: $'v$ *connective* **where**
c: (c = *CAnd* $\vee$ c = *COr* $\vee$ c = *CImp* $\vee$ c = *CEq*) $\wedge$ *conn* c [$\varphi 1$, $\varphi 2$] = φ
using φ **by** *fastforce*

then have *corr*: *wf-conn* c [$\varphi 1$, $\varphi 2$] **using** *wf-conn.simps* **unfolding** *binary-connectives-def* **by** *auto*
have *inc*: $\varphi 1 \preceq \varphi$ $\varphi 2 \preceq \varphi$ **using** *binary-connectives-def c subformula-in-binary-conn* **by** *blast+*

from *r IHφ1-0* **have** *IHφ1*: $\neg$ *all-subformula-st test-symb φ1* $\Longrightarrow \exists\psi\ \psi'.\ \psi \preceq \varphi 1 \wedge r\ \psi\ \psi'$
  using *inc*(*1*) *subformula-trans le* **by** *blast*
from *r IHφ2-0* **have** *IHφ2*: $\neg$ *all-subformula-st test-symb φ2* $\Longrightarrow \exists\psi.\ \psi \preceq \varphi 2 \wedge (\exists\psi'.\ r\ \psi\ \psi')$
  using *inc*(*2*) *subformula-trans le* **by** *blast*
have *cases*: $\neg$*test-symb* φ $\vee$ $\neg$*all-subformula-st test-symb φ1* $\vee$ $\neg$*all-subformula-st test-symb φ2*
  using *c nst* **by** *auto*
show $\exists\psi\ \psi'.\ \psi \preceq \varphi \wedge r\ \psi\ \psi'$
  using *IHφ1 IHφ2 subformula-trans inc subformula-refl cases le* **by** *blast*
qed

1.4.2 Invariant conservation

If two rewrite relation are independant (or at least independant enough), then the property characterizing the first relation *all-subformula-st test-symb* remains true. The next show the same property, with changes in the assumptions.

The assumption $\forall\varphi'\ \psi.\ \varphi' \preceq \Phi \longrightarrow r\ \varphi'\ \psi \longrightarrow$ *all-subformula-st test-symb* $\varphi' \longrightarrow$ *all-subformula-st test-symb* ψ means that rewriting with r does not mess up the property we want to preserve locally.

The previous assumption is not enough to go from r to *propo-rew-step* r: we have to add the assumption that rewriting inside does not mess up the term: $\forall c\ \xi\ \varphi\ \xi'\ \varphi'.\ \varphi \preceq \Phi \longrightarrow$ *propo-rew-step* $r\ \varphi\ \varphi' \longrightarrow$ *wf-conn* $c\ (\xi\ @\ \varphi\ \#\ \xi') \longrightarrow$ *test-symb* (*conn* $c\ (\xi\ @\ \varphi\ \#\ \xi')) \longrightarrow$ *test-symb* $\varphi' \longrightarrow$ *test-symb* (*conn* $c\ (\xi\ @\ \varphi'\ \#\ \xi')$)

Invariant while lifting of the Rewriting Relation

The condition $\varphi \preceq \Phi$ (that will by used with $\Phi = \varphi$ most of the time) is here to ensure that the recursive conditions on Φ will moreover hold for the subterm we are rewriting. For example if there is no equivalence symbol in Φ, we do not have to care about equivalence symbols in the two previous assumptions.

lemma *propo-rew-step-inv-stay'*:
  fixes *r*:: *'v propo* $\Rightarrow$ *'v propo* $\Rightarrow$ *bool* **and** *test-symb*:: *'v propo* $\Rightarrow$ *bool* **and** *x* :: *'v*
  and $\varphi\ \psi\ \Phi$:: *'v propo*
  assumes *H*: $\forall\varphi'\ \psi.\ \varphi' \preceq \Phi \longrightarrow r\ \varphi'\ \psi \longrightarrow$ *all-subformula-st test-symb* φ'
    $\longrightarrow$ *all-subformula-st test-symb* ψ
  and *H'*: $\forall$(*c*:: *'v connective*) $\xi\ \varphi\ \xi'\ \varphi'.\ \varphi \preceq \Phi \longrightarrow$ *propo-rew-step* $r\ \varphi\ \varphi'$
    $\longrightarrow$ *wf-conn* $c\ (\xi\ @\ \varphi\ \#\ \xi') \longrightarrow$ *test-symb* (*conn* $c\ (\xi\ @\ \varphi\ \#\ \xi')) \longrightarrow$ *test-symb* φ'
    $\longrightarrow$ *test-symb* (*conn* $c\ (\xi\ @\ \varphi'\ \#\ \xi')$) **and**
    propo-rew-step $r\ \varphi\ \psi$ **and**
    $\varphi \preceq \Phi$ **and**
    all-subformula-st test-symb φ
  shows *all-subformula-st test-symb* ψ
  using *assms*(*3*$-$*5*)
proof (*induct rule*: *propo-rew-step.induct*)
  case *global-rel*
  then show *?case* **using** *H* **by** *simp*
next
  case (*propo-rew-one-step-lift* $\varphi\ \varphi'\ c\ \xi\ \xi'$)
  note *rel* = *this*(*1*) **and** φ = *this*(*2*) **and** *corr* = *this*(*3*) **and** Φ = *this*(*4*) **and** *nst* = *this*(*5*)
  have *sq*: $\varphi \preceq \Phi$
    using Φ *corr subformula-into-subformula subformula-refl subformula-trans*
    by (*metis in-set-conv-decomp*)
  from *corr* **have** $\forall\ \psi.\ \psi \in$ *set* $(\xi\ @\ \varphi\ \#\ \xi') \longrightarrow$ *all-subformula-st test-symb* ψ

using *all-subformula-st-decomp nst* **by** *blast*
then have $*$: $\forall \psi.\ \psi \in set\ (\xi\ @\ \varphi'\ \#\ \xi') \longrightarrow$ *all-subformula-st test-symb* ψ **using** φ *sq* **by** *fastforce*
then have *test-symb* φ' **using** *all-subformula-st-test-symb-true-phi* **by** *auto*
moreover from *corr nst* **have** *test-symb* (*conn c* $(\xi\ @\ \varphi\ \#\ \xi')$)
using *all-subformula-st-decomp* **by** *blast*
ultimately have *test-symb*: *test-symb* (*conn c* $(\xi\ @\ \varphi'\ \#\ \xi')$) **using** H' *sq corr rel* **by** *blast*

have *wf-conn c* $(\xi\ @\ \varphi'\ \#\ \xi')$
by (*metis wf-conn-no-arity-change-helper corr wf-conn-no-arity-change*)
then show *all-subformula-st test-symb* (*conn c* $(\xi\ @\ \varphi'\ \#\ \xi')$)
using $*$ *test-symb* **by** (*metis all-subformula-st-decomp*)
qed

The need for $\varphi \preceq \Phi$ is not always necessary, hence we moreover have a version without inclusion.

lemma *propo-rew-step-inv-stay*:
fixes r:: $'v$ *propo* $\Rightarrow$ $'v$ *propo* $\Rightarrow$ *bool* **and** *test-symb*:: $'v$ *propo* $\Rightarrow$ *bool* **and** x :: $'v$
and $\varphi\ \psi$ :: $'v$ *propo*
assumes
H: $\forall \varphi'\ \psi.\ r\ \varphi'\ \psi \longrightarrow$ *all-subformula-st test-symb* $\varphi' \longrightarrow$ *all-subformula-st test-symb* ψ **and**
H': $\forall (c$:: $'v$ *connective*$)\ \xi\ \varphi\ \xi'\ \varphi'.$ *wf-conn c* $(\xi\ @\ \varphi\ \#\ \xi') \longrightarrow$ *test-symb* (*conn c* $(\xi\ @\ \varphi\ \#\ \xi')$)
$\longrightarrow$ *test-symb* $\varphi' \longrightarrow$ *test-symb* (*conn c* $(\xi\ @\ \varphi'\ \#\ \xi')$) **and**
propo-rew-step r $\varphi\ \psi$ **and**
all-subformula-st test-symb φ
shows *all-subformula-st test-symb* ψ
using *propo-rew-step-inv-stay'*[*of* φ *r test-symb* $\varphi\ \psi$] *assms subformula-refl* **by** *metis*

The lemmas can be lifted to *propo-rew-step* $r^{\downarrow}$ instead of *propo-rew-step*

Invariant after all Rewriting

lemma *full-propo-rew-step-inv-stay-with-inc*:
fixes r:: $'v$ *propo* $\Rightarrow$ $'v$ *propo* $\Rightarrow$ *bool* **and** *test-symb*:: $'v$ *propo* $\Rightarrow$ *bool* **and** x :: $'v$
and $\varphi\ \psi$ :: $'v$ *propo*
assumes
H: $\forall\ \ \varphi\ \psi.$ *propo-rew-step r* $\varphi\ \psi \longrightarrow$ *all-subformula-st test-symb* φ
$\longrightarrow$ *all-subformula-st test-symb* ψ **and**
H': $\forall (c$:: $'v$ *connective*$)\ \xi\ \varphi\ \xi'\ \varphi'.\ \varphi \preceq \Phi \longrightarrow$ *propo-rew-step r* $\varphi\ \varphi'$
$\longrightarrow$ *wf-conn c* $(\xi\ @\ \varphi\ \#\ \xi') \longrightarrow$ *test-symb* (*conn c* $(\xi\ @\ \varphi\ \#\ \xi')$) $\longrightarrow$ *test-symb* φ'
$\longrightarrow$ *test-symb* (*conn c* $(\xi\ @\ \varphi'\ \#\ \xi')$) **and**
$\varphi \preceq \Phi$ **and**
full: *full* (*propo-rew-step r*) $\varphi\ \psi$ **and**
init: *all-subformula-st test-symb* φ
shows *all-subformula-st test-symb* ψ
using *assms* **unfolding** *full-def*
proof $-$
have *rel*: (*propo-rew-step r*)** $\varphi\ \psi$
using *full* **unfolding** *full-def* **by** *auto*
then show *all-subformula-st test-symb* ψ
using *init*
proof (*induct rule*: *rtranclp-induct*)
case *base*
then show *all-subformula-st test-symb* φ **by** *blast*
next
case (*step b c*) **note** *star* = *this*(*1*) **and** *IH* = *this*(*3*) **and** *one* = *this*(*2*) **and** *all* = *this*(*4*)
then have *all-subformula-st test-symb b* **by** *metis*
then show *all-subformula-st test-symb c* **using** *propo-rew-step-inv-stay'* $H\ H'$ *rel one* **by** *auto*

qed
qed

lemma *full-propo-rew-step-inv-stay'*:
fixes *r*:: *'v propo* ⇒ *'v propo* ⇒ *bool* **and** *test-symb*:: *'v propo* ⇒ *bool* **and** *x* :: *'v*
and φ ψ :: *'v propo*
assumes
H: ∀ φ ψ. *propo-rew-step r* φ ψ ⟶ *all-subformula-st test-symb* φ
⟶ *all-subformula-st test-symb* ψ **and**
H': ∀ (*c*:: *'v connective*) ξ φ ξ' φ'. *propo-rew-step r* φ φ' ⟶ *wf-conn c* (ξ @ φ # ξ')
⟶ *test-symb* (*conn c* (ξ @ φ # ξ')) ⟶ *test-symb* φ' ⟶ *test-symb* (*conn c* (ξ @ φ' # ξ')) **and**
full: *full* (*propo-rew-step r*) φ ψ **and**
init: *all-subformula-st test-symb* φ
shows *all-subformula-st test-symb* ψ
using *full-propo-rew-step-inv-stay-with-inc*[*of r test-symb* φ] *assms subformula-refl* **by** *metis*

lemma *full-propo-rew-step-inv-stay*:
fixes *r*:: *'v propo* ⇒ *'v propo* ⇒ *bool* **and** *test-symb*:: *'v propo* ⇒ *bool* **and** *x* :: *'v*
and φ ψ :: *'v propo*
assumes
H: ∀ φ ψ. *r* φ ψ ⟶ *all-subformula-st test-symb* φ ⟶ *all-subformula-st test-symb* ψ **and**
H': ∀ (*c*:: *'v connective*) ξ φ ξ' φ'. *wf-conn c* (ξ @ φ # ξ') ⟶ *test-symb* (*conn c* (ξ @ φ # ξ'))
⟶ *test-symb* φ' ⟶ *test-symb* (*conn c* (ξ @ φ' # ξ')) **and**
full: *full* (*propo-rew-step r*) φ ψ **and**
init: *all-subformula-st test-symb* φ
shows *all-subformula-st test-symb* ψ
unfolding *full-def*
proof −
have *rel*: (*propo-rew-step r*)^** φ ψ
using *full* **unfolding** *full-def* **by** *auto*
then show *all-subformula-st test-symb* ψ
using *init*
proof (*induct rule*: *rtranclp-induct*)
case *base*
then show *all-subformula-st test-symb* φ **by** *blast*
next
case (*step b c*)
note *star* = *this*(*1*) **and** *IH* = *this*(*3*) **and** *one* = *this*(*2*) **and** *all* = *this*(*4*)
then have *all-subformula-st test-symb b* **by** *metis*
then show *all-subformula-st test-symb c*
using *propo-rew-step-inv-stay subformula-refl H H' rel one* **by** *auto*
qed
qed

lemma *full-propo-rew-step-inv-stay-conn*:
fixes *r*:: *'v propo* ⇒ *'v propo* ⇒ *bool* **and** *test-symb*:: *'v propo* ⇒ *bool* **and** *x* :: *'v*
and φ ψ :: *'v propo*
assumes
H: ∀ φ ψ. *r* φ ψ ⟶ *all-subformula-st test-symb* φ ⟶ *all-subformula-st test-symb* ψ **and**
H': ∀ (*c*:: *'v connective*) *l l'*. *wf-conn c l* ⟶ *wf-conn c l'*
⟶ (*test-symb* (*conn c l*) ⟷ *test-symb* (*conn c l'*)) **and**
full: *full* (*propo-rew-step r*) φ ψ **and**
init: *all-subformula-st test-symb* φ
shows *all-subformula-st test-symb* ψ
proof −

have $\bigwedge(c{::}\ 'v\ connective)\ \xi\ \varphi\ \xi'\ \varphi'.\ wf\text{-}conn\ c\ (\xi\ @\ \varphi\ \#\ \xi')$
$\Longrightarrow test\text{-}symb\ (conn\ c\ (\xi\ @\ \varphi\ \#\ \xi')) \Longrightarrow test\text{-}symb\ \varphi' \Longrightarrow test\text{-}symb\ (conn\ c\ (\xi\ @\ \varphi'\ \#\ \xi'))$
using *H'* **by** (*metis wf-conn-no-arity-change-helper wf-conn-no-arity-change*)
then show *all-subformula-st test-symb* ψ
using *H full init full-propo-rew-step-inv-stay* **by** *blast*
qed

end
theory *Prop-Normalisation*
imports *Prop-Logic Prop-Abstract-Transformation Nested-Multisets-Ordinals.Multiset-More*
begin

Given the previous definition about abstract rewriting and theorem about them, we now have the detailed rule making the transformation into CNF/DNF.

1.5 Rewrite Rules

The idea of Christoph Weidenbach's book is to remove gradually the operators: first equivalencies, then implication, after that the unused true/false and finally the reorganizing the or/and. We will prove each transformation seperately.

1.5.1 Elimination of the Equivalences

The first transformation consists in removing every equivalence symbol.

inductive *elim-equiv* :: $'v\ propo \Rightarrow\ 'v\ propo \Rightarrow bool$ **where**
elim-equiv[*simp*]: *elim-equiv* $(FEq\ \varphi\ \psi)\ (FAnd\ (FImp\ \varphi\ \psi)\ \ (FImp\ \psi\ \varphi))$

lemma *elim-equiv-transformation-consistent*:
$A \models FEq\ \varphi\ \psi \longleftrightarrow A \models FAnd\ (FImp\ \varphi\ \psi)\ (FImp\ \psi\ \varphi)$
by *auto*

lemma *elim-equiv-explicit*: *elim-equiv* $\varphi\ \psi \Longrightarrow \forall A.\ A \models \varphi \longleftrightarrow A \models \psi$
by (*induct rule*: *elim-equiv.induct*, *auto*)

lemma *elim-equiv-consistent*: *preserve-models elim-equiv*
unfolding *preserve-models-def* **by** (*simp add*: *elim-equiv-explicit*)

lemma *elimEquv-lifted-consistant*:
preserve-models (*full* (*propo-rew-step elim-equiv*))
by (*simp add*: *elim-equiv-consistent*)

This function ensures that there is no equivalencies left in the formula tested by *no-equiv-symb*.

fun *no-equiv-symb* :: $'v\ propo \Rightarrow bool$ **where**
no-equiv-symb (*FEq* - -) = *False* |
no-equiv-symb - = *True*

Given the definition of *no-equiv-symb*, it does not depend on the formula, but only on the connective used.

lemma *no-equiv-symb-conn-characterization*[*simp*]:
fixes c :: $'v\ connective$ **and** l :: $'v\ propo\ list$
assumes *wf*: *wf-conn c l*
shows *no-equiv-symb* $(conn\ c\ l) \longleftrightarrow c \neq CEq$

by (*metis connective.distinct*(*13*,*25*,*35*,*43*) *wf no-equiv-symb.elims*(*3*) *no-equiv-symb.simps*(*1*) *wf-conn.cases wf-conn-list*(*6*))

definition *no-equiv* **where** *no-equiv* = *all-subformula-st no-equiv-symb*

lemma *no-equiv-eq*[*simp*]:
fixes φ ψ :: $'v$ *propo*
shows
$\neg$*no-equiv* (*FEq* φ ψ)
no-equiv FT
no-equiv FF
using *no-equiv-symb.simps*(*1*) *all-subformula-st-test-symb-true-phi* **unfolding** *no-equiv-def* **by** *auto*

The following lemma helps to reconstruct *no-equiv* expressions: this representation is easier to use than the set definition.

lemma *all-subformula-st-decomp-explicit-no-equiv*[*iff*]:
fixes φ ψ :: $'v$ *propo*
shows
no-equiv (*FNot* φ) $\longleftrightarrow$ *no-equiv* φ
no-equiv (*FAnd* φ ψ) $\longleftrightarrow$ (*no-equiv* φ $\wedge$ *no-equiv* ψ)
no-equiv (*FOr* φ ψ) $\longleftrightarrow$ (*no-equiv* φ $\wedge$ *no-equiv* ψ)
no-equiv (*FImp* φ ψ) $\longleftrightarrow$ (*no-equiv* φ $\wedge$ *no-equiv* ψ)
by (*auto simp*: *no-equiv-def*)

A theorem to show the link between the rewrite relation *elim-equiv* and the function *no-equiv-symb*. This theorem is one of the assumption we need to characterize the transformation.

lemma *no-equiv-elim-equiv-step*:
fixes φ :: $'v$ *propo*
assumes *no-equiv*: $\neg$ *no-equiv* φ
shows $\exists\psi\ \psi'.\ \psi \preceq \varphi \wedge$ *elim-equiv* ψ ψ'
proof −
have *test-symb-false-nullary*:
$\forall x$::$'v$. *no-equiv-symb FF* $\wedge$ *no-equiv-symb FT* $\wedge$ *no-equiv-symb* (*FVar* x)
unfolding *no-equiv-def* **by** *auto*
moreover {
fix c:: $'v$ *connective* **and** l :: $'v$ *propo list* **and** ψ :: $'v$ *propo*
assume *a1*: *elim-equiv* (*conn c l*) ψ
have $\bigwedge p$ *pa*. $\neg$ *elim-equiv* (p::$'v$ *propo*) *pa* $\vee$ $\neg$ *no-equiv-symb* p
using *elim-equiv.cases no-equiv-symb.simps*(*1*) **by** *blast*
then have *elim-equiv* (*conn c l*) $\psi \Longrightarrow \neg$*no-equiv-symb* (*conn c l*) **using** *a1* **by** *metis*
}
moreover have H': $\forall\psi$. $\neg$*elim-equiv FT* ψ $\forall\psi$. $\neg$*elim-equiv FF* ψ $\forall\psi$ x. $\neg$*elim-equiv* (*FVar* x) ψ
using *elim-equiv.cases* **by** *auto*
moreover have $\bigwedge\varphi$. $\neg$ *no-equiv-symb* $\varphi \Longrightarrow \exists\psi$. *elim-equiv* φ ψ
by (*case-tac* φ, *auto simp*: *elim-equiv.simps*)
then have $\bigwedge\varphi'.\ \varphi' \preceq \varphi \Longrightarrow \neg$*no-equiv-symb* $\varphi' \Longrightarrow\ \exists\psi$. *elim-equiv* φ' ψ **by** *force*
ultimately show *?thesis*
using *no-test-symb-step-exists no-equiv test-symb-false-nullary* **unfolding** *no-equiv-def* **by** *blast*
qed

Given all the previous theorem and the characterization, once we have rewritten everything, there is no equivalence symbol any more.

lemma *no-equiv-full-propo-rew-step-elim-equiv*:
full (*propo-rew-step elim-equiv*) φ $\psi \Longrightarrow$ *no-equiv* ψ
using *full-propo-rew-step-subformula no-equiv-elim-equiv-step* **by** *blast*

1.5.2 Eliminate Implication

After that, we can eliminate the implication symbols.

inductive *elim-imp* :: *'v propo* ⇒ *'v propo* ⇒ *bool* **where**
[*simp*]: *elim-imp* (*FImp* φ ψ) (*FOr* (*FNot* φ) ψ)

lemma *elim-imp-transformation-consistent*:
  $A \models$ *FImp* $\varphi\ \psi \longleftrightarrow A \models$ *FOr* (*FNot* φ) ψ
  by *auto*

lemma *elim-imp-explicit*: *elim-imp* $\varphi\ \psi \Longrightarrow \forall A.\ A \models \varphi \longleftrightarrow A \models \psi$
  by (*induct* φ ψ *rule*: *elim-imp.induct*, *auto*)

lemma *elim-imp-consistent*: *preserve-models elim-imp*
  unfolding *preserve-models-def* **by** (*simp add*: *elim-imp-explicit*)

lemma *elim-imp-lifted-consistant*:
  preserve-models (*full* (*propo-rew-step elim-imp*))
  by (*simp add*: *elim-imp-consistent*)

fun *no-imp-symb* **where**
no-imp-symb (*FImp* - -) = *False* |
no-imp-symb - = *True*

lemma *no-imp-symb-conn-characterization*:
  wf-conn c l $\Longrightarrow$ *no-imp-symb* (*conn c l*) $\longleftrightarrow$ $c \neq$ *CImp*
  by (*induction rule*: *wf-conn-induct*) *auto*

definition *no-imp* **where** *no-imp* $\equiv$ *all-subformula-st no-imp-symb*
declare *no-imp-def*[*simp*]

lemma *no-imp-Imp*[*simp*]:
  $\neg$*no-imp* (*FImp* φ ψ)
  no-imp FT
  no-imp FF
  unfolding *no-imp-def* **by** *auto*

lemma *all-subformula-st-decomp-explicit-imp*[*simp*]:
  fixes φ ψ :: *'v propo*
  shows
    no-imp (*FNot* φ) $\longleftrightarrow$ *no-imp* φ
    no-imp (*FAnd* φ ψ) $\longleftrightarrow$ (*no-imp* φ $\wedge$ *no-imp* ψ)
    no-imp (*FOr* φ ψ) $\longleftrightarrow$ (*no-imp* φ $\wedge$ *no-imp* ψ)
  by *auto*

Invariant of the *elim-imp* transformation

lemma *elim-imp-no-equiv*:
  elim-imp $\varphi\ \psi \Longrightarrow$ *no-equiv* $\varphi \Longrightarrow$ *no-equiv* ψ
  by (*induct* φ ψ *rule*: *elim-imp.induct*, *auto*)

lemma *elim-imp-inv*:
  fixes φ ψ :: *'v propo*
  assumes *full* (*propo-rew-step elim-imp*) φ ψ **and** *no-equiv* φ
  shows *no-equiv* ψ
  using *full-propo-rew-step-inv-stay-conn*[*of elim-imp no-equiv-symb* φ ψ] *assms elim-imp-no-equiv*

no-equiv-symb-conn-characterization **unfolding** *no-equiv-def* **by** *metis*

lemma *no-no-imp-elim-imp-step-exists*:
fixes φ :: *'v propo*
assumes *no-equiv*: $\neg$ *no-imp* φ
shows $\exists\psi\ \psi'.\ \psi \preceq \varphi \land$ *elim-imp* $\psi\ \psi'$
proof −
have *test-symb-false-nullary*: $\forall x.$ *no-imp-symb FF* $\land$ *no-imp-symb FT* $\land$ *no-imp-symb* (*FVar* (x:: *'v*))
by *auto*
moreover **{**
fix c:: *'v connective* **and** l :: *'v propo list* **and** ψ :: *'v propo*
have *H*: *elim-imp* (*conn c l*) $\psi \Longrightarrow \neg$*no-imp-symb* (*conn c l*)
by (*auto elim*: *elim-imp.cases*)
}
moreover
have *H'*: $\forall\psi.\ \neg$*elim-imp FT* ψ $\forall\psi.\ \neg$*elim-imp FF* ψ $\forall\psi\ x.\ \neg$*elim-imp* (*FVar x*) ψ
by (*auto elim*: *elim-imp.cases*)+
moreover
have $\bigwedge\varphi.\ \neg$ *no-imp-symb* $\varphi \Longrightarrow \exists\psi.$ *elim-imp* $\varphi\ \psi$
by (*case-tac* φ) (*force simp*: *elim-imp.simps*)+
then have $\bigwedge\varphi'.\ \varphi' \preceq \varphi \Longrightarrow \neg$*no-imp-symb* $\varphi' \Longrightarrow\ \exists\ \psi.$ *elim-imp* $\varphi'\ \psi$ **by** *force*
ultimately show *?thesis*
using *no-test-symb-step-exists no-equiv test-symb-false-nullary* **unfolding** *no-imp-def* **by** *blast*
qed

lemma *no-imp-full-propo-rew-step-elim-imp*: *full* (*propo-rew-step elim-imp*) $\varphi\ \psi \Longrightarrow$ *no-imp* ψ
using *full-propo-rew-step-subformula no-no-imp-elim-imp-step-exists* **by** *blast*

1.5.3 Eliminate all the True and False in the formula

Contrary to the book, we have to give the transformation and the "commutative" transformation. The latter is implicit in the book.

inductive *elimTB* **where**
ElimTB1: *elimTB* (*FAnd* φ *FT*) φ |
ElimTB1': *elimTB* (*FAnd FT* φ) φ |

ElimTB2: *elimTB* (*FAnd* φ *FF*) *FF* |
ElimTB2': *elimTB* (*FAnd FF* φ) *FF* |

ElimTB3: *elimTB* (*FOr* φ *FT*) *FT* |
ElimTB3': *elimTB* (*FOr FT* φ) *FT* |

ElimTB4: *elimTB* (*FOr* φ *FF*) φ |
ElimTB4': *elimTB* (*FOr FF* φ) φ |

ElimTB5: *elimTB* (*FNot FT*) *FF* |
ElimTB6: *elimTB* (*FNot FF*) *FT*

lemma *elimTB-consistent*: *preserve-models elimTB*
proof −
{
fix $\varphi\ \psi$:: *'b propo*
have *elimTB* $\varphi\ \psi \Longrightarrow \forall A.\ A \models \varphi \longleftrightarrow A \models \psi$ **by** (*induction rule*: *elimTB.inducts*) *auto*
}

then show *?thesis* **using** *preserve-models-def* **by** *auto*
qed

inductive *no-T-F-symb* :: *'v propo* $\Rightarrow$ *bool* **where**
no-T-F-symb-comp: $c \neq CF \Longrightarrow c \neq CT \Longrightarrow wf\text{-}conn\ c\ l \Longrightarrow (\forall \varphi \in set\ l.\ \varphi \neq FT \wedge \varphi \neq FF)$
$\Longrightarrow no\text{-}T\text{-}F\text{-}symb\ (conn\ c\ l)$

lemma *wf-conn-no-T-F-symb-iff* [*simp*]:
$wf\text{-}conn\ c\ \psi s \Longrightarrow$
$no\text{-}T\text{-}F\text{-}symb\ (conn\ c\ \psi s) \longleftrightarrow (c \neq CF \wedge c \neq CT \wedge (\forall \psi \in set\ \psi s.\ \psi \neq FF \wedge \psi \neq FT))$
unfolding *no-T-F-symb.simps* **apply** (*cases c*)
using *wf-conn-list*(*1*) **apply** *fastforce*
using *wf-conn-list*(*2*) **apply** *fastforce*
using *wf-conn-list*(*3*) **apply** *fastforce*
apply (*metis* (*no-types*, *hide-lams*) *conn-inj connective.distinct*(*5*,*17*))
using *conn-inj* **apply** *blast+*
done

lemma *wf-conn-no-T-F-symb-iff-explicit*[*simp*]:
$no\text{-}T\text{-}F\text{-}symb\ (FAnd\ \varphi\ \psi) \longleftrightarrow (\forall \chi \in set\ [\varphi, \psi].\ \chi \neq FF \wedge \chi \neq FT)$
$no\text{-}T\text{-}F\text{-}symb\ (FOr\ \varphi\ \psi) \longleftrightarrow (\forall \chi \in set\ [\varphi, \psi].\ \chi \neq FF \wedge \chi \neq FT)$
$no\text{-}T\text{-}F\text{-}symb\ (FEq\ \varphi\ \psi) \longleftrightarrow (\forall \chi \in set\ [\varphi, \psi].\ \chi \neq FF \wedge \chi \neq FT)$
$no\text{-}T\text{-}F\text{-}symb\ (FImp\ \varphi\ \psi) \longleftrightarrow (\forall \chi \in set\ [\varphi, \psi].\ \chi \neq FF \wedge \chi \neq FT)$
apply (*metis conn.simps*(*36*) *conn.simps*(*37*) *conn.simps*(*5*) *propo.distinct*(*19*)
wf-conn-helper-facts(*5*) *wf-conn-no-T-F-symb-iff*)
apply (*metis conn.simps*(*36*) *conn.simps*(*37*) *conn.simps*(*6*) *propo.distinct*(*22*)
wf-conn-helper-facts(*6*) *wf-conn-no-T-F-symb-iff*)
using *wf-conn-no-T-F-symb-iff* **apply** *fastforce*
by (*metis conn.simps*(*36*) *conn.simps*(*37*) *conn.simps*(*7*) *propo.distinct*(*23*) *wf-conn-helper-facts*(*7*)
wf-conn-no-T-F-symb-iff)

lemma *no-T-F-symb-false*[*simp*]:
fixes *c* :: *'v connective*
shows
$\neg no\text{-}T\text{-}F\text{-}symb\ (FT :: 'v\ propo)$
$\neg no\text{-}T\text{-}F\text{-}symb\ (FF :: 'v\ propo)$
by (*metis* (*no-types*) *conn.simps*(*1*,*2*) *wf-conn-no-T-F-symb-iff wf-conn-nullary*)+

lemma *no-T-F-symb-bool*[*simp*]:
fixes *x* :: *'v*
shows *no-T-F-symb* (*FVar x*)
using *no-T-F-symb-comp wf-conn-nullary* **by** (*metis connective.distinct*(*3*, *15*) *conn.simps*(*3*)
empty-iff list.set(*1*))

lemma *no-T-F-symb-fnot-imp*:
$\neg no\text{-}T\text{-}F\text{-}symb\ (FNot\ \varphi) \Longrightarrow \varphi = FT \vee \varphi = FF$
proof (*rule ccontr*)
assume *n*: $\neg$ *no-T-F-symb* (*FNot* φ)
assume $\neg (\varphi = FT \vee \varphi = FF)$
then have $\forall \varphi' \in set\ [\varphi].\ \varphi' \neq FT \wedge \varphi' \neq FF$ **by** *auto*
moreover have *wf-conn CNot* [φ] **by** *simp*
ultimately have *no-T-F-symb* (*FNot* φ)
using *no-T-F-symb.intros* **by** (*metis conn.simps*(*4*) *connective.distinct*(*5*,*17*))

then show *False* **using** *n* **by** *blast*
qed

lemma *no-T-F-symb-fnot*[*simp*]:
no-T-F-symb (*FNot* φ) $\longleftrightarrow$ $\neg$(φ = *FT* $\vee$ φ = *FF*)
using *no-T-F-symb.simps no-T-F-symb-fnot-imp* **by** (*metis conn-inj-not*(*2*) *list.set-intros*(*1*))

Actually it is not possible to remover every *FT* and *FF*: if the formula is equal to true or false, we can not remove it.

inductive *no-T-F-symb-except-toplevel* **where**
no-T-F-symb-except-toplevel-true[*simp*]: *no-T-F-symb-except-toplevel FT* |
no-T-F-symb-except-toplevel-false[*simp*]: *no-T-F-symb-except-toplevel FF* |
noTrue-no-T-F-symb-except-toplevel[*simp*]: *no-T-F-symb* $\varphi \Longrightarrow$ *no-T-F-symb-except-toplevel* φ

lemma *no-T-F-symb-except-toplevel-bool*:
fixes *x* :: *'v*
shows *no-T-F-symb-except-toplevel* (*FVar x*)
by *simp*

lemma *no-T-F-symb-except-toplevel-not-decom*:
$\varphi \neq$ *FT* $\Longrightarrow$ $\varphi \neq$ *FF* $\Longrightarrow$ *no-T-F-symb-except-toplevel* (*FNot* φ)
by *simp*

lemma *no-T-F-symb-except-toplevel-bin-decom*:
fixes φ ψ :: *'v propo*
assumes $\varphi \neq$ *FT* **and** $\varphi \neq$ *FF* **and** $\psi \neq$ *FT* **and** $\psi \neq$ *FF*
and *c*: *c*$\in$ *binary-connectives*
shows *no-T-F-symb-except-toplevel* (*conn c* [φ, ψ])
by (*metis* (*no-types*, *lifting*) *assms c conn.simps*(*4*) *list.discI noTrue-no-T-F-symb-except-toplevel wf-conn-no-T-F-symb-iff no-T-F-symb-fnot set-ConsD wf-conn-binary wf-conn-helper-facts*(*1*) *wf-conn-list-decomp*(*1*,*2*))

lemma *no-T-F-symb-except-toplevel-if-is-a-true-false*:
fixes *l* :: *'v propo list* **and** *c* :: *'v connective*
assumes *corr*: *wf-conn c l*
and *FT* $\in$ *set l* $\vee$ *FF* $\in$ *set l*
shows $\neg$*no-T-F-symb-except-toplevel* (*conn c l*)
by (*metis assms empty-iff no-T-F-symb-except-toplevel.simps wf-conn-no-T-F-symb-iff set-empty wf-conn-list*(*1*,*2*))

lemma *no-T-F-symb-except-top-level-false-example*[*simp*]:
fixes φ ψ :: *'v propo*
assumes φ = *FT* $\vee$ ψ = *FT* $\vee$ φ = *FF* $\vee$ ψ = *FF*
shows
$\neg$ *no-T-F-symb-except-toplevel* (*FAnd* φ ψ)
$\neg$ *no-T-F-symb-except-toplevel* (*FOr* φ ψ)
$\neg$ *no-T-F-symb-except-toplevel* (*FImp* φ ψ)
$\neg$ *no-T-F-symb-except-toplevel* (*FEq* φ ψ)
using *assms no-T-F-symb-except-toplevel-if-is-a-true-false* **unfolding** *binary-connectives-def*
by (*metis* (*no-types*) *conn.simps*(*5*−*8*) *insert-iff list.simps*(*14*−*15*) *wf-conn-helper-facts*(*5*−*8*))+

lemma *no-T-F-symb-except-top-level-false-not*[*simp*]:
fixes φ ψ :: *'v propo*
assumes φ = *FT* $\vee$ φ = *FF*
shows

$\neg$ *no-T-F-symb-except-toplevel* (*FNot* φ)
by (*simp add*: *assms no-T-F-symb-except-toplevel.simps*)

This is the local extension of *no-T-F-symb-except-toplevel*.

definition *no-T-F-except-top-level* **where**
no-T-F-except-top-level $\equiv$ *all-subformula-st no-T-F-symb-except-toplevel*

This is another property we will use. While this version might seem to be the one we want to prove, it is not since *FT* can not be reduced.

definition *no-T-F* **where**
no-T-F $\equiv$ *all-subformula-st no-T-F-symb*

lemma *no-T-F-except-top-level-false*:
fixes *l* :: *'v propo list* **and** *c* :: *'v connective*
assumes *wf-conn c l*
and *FT* $\in$ *set l* $\vee$ *FF* $\in$ *set l*
shows $\neg$*no-T-F-except-top-level* (*conn c l*)
by (*simp add*: *all-subformula-st-decomp assms no-T-F-except-top-level-def no-T-F-symb-except-toplevel-if-is-a-true-false*)

lemma *no-T-F-except-top-level-false-example*[*simp*]:
fixes φ ψ :: *'v propo*
assumes φ = *FT* $\vee$ ψ = *FT* $\vee$ φ = *FF* $\vee$ ψ = *FF*
shows
$\neg$*no-T-F-except-top-level* (*FAnd* φ ψ)
$\neg$*no-T-F-except-top-level* (*FOr* φ ψ)
$\neg$*no-T-F-except-top-level* (*FEq* φ ψ)
$\neg$*no-T-F-except-top-level* (*FImp* φ ψ)
by (*metis all-subformula-st-test-symb-true-phi assms no-T-F-except-top-level-def no-T-F-symb-except-top-level-false-example*)+

lemma *no-T-F-symb-except-toplevel-no-T-F-symb*:
no-T-F-symb-except-toplevel $\varphi \Longrightarrow \varphi \neq$ *FF* $\Longrightarrow \varphi \neq$ *FT* $\Longrightarrow$ *no-T-F-symb* φ
by (*induct rule*: *no-T-F-symb-except-toplevel.induct*, *auto*)

The two following lemmas give the precise link between the two definitions.

lemma *no-T-F-symb-except-toplevel-all-subformula-st-no-T-F-symb*:
no-T-F-except-top-level $\varphi \Longrightarrow \varphi \neq$ *FF* $\Longrightarrow \varphi \neq$ *FT* $\Longrightarrow$ *no-T-F* φ
unfolding *no-T-F-except-top-level-def no-T-F-def* **apply** (*induct* φ)
using *no-T-F-symb-fnot* **by** *fastforce*+

lemma *no-T-F-no-T-F-except-top-level*:
no-T-F $\varphi \Longrightarrow$ *no-T-F-except-top-level* φ
unfolding *no-T-F-except-top-level-def no-T-F-def*
unfolding *all-subformula-st-def* **by** *auto*

lemma *no-T-F-except-top-level-simp*[*simp*]: *no-T-F-except-top-level FF no-T-F-except-top-level FT*
unfolding *no-T-F-except-top-level-def* **by** *auto*

lemma *no-T-F-no-T-F-except-top-level'*[*simp*]:
no-T-F-except-top-level $\varphi \longleftrightarrow$ (φ = *FF* $\vee$ φ = *FT* $\vee$ *no-T-F* φ)
using *no-T-F-symb-except-toplevel-all-subformula-st-no-T-F-symb no-T-F-no-T-F-except-top-level*
by *auto*

lemma *no-T-F-bin-decomp*[*simp*]:
  assumes *c*: $c \in$ *binary-connectives*
  shows *no-T-F* (*conn c* [φ, ψ]) $\longleftrightarrow$ (*no-T-F* $\varphi \wedge$ *no-T-F* ψ)
proof −
  have *wf*: *wf-conn c* [φ, ψ] **using** *c* **by** *auto*
  then have *no-T-F* (*conn c* [φ, ψ]) $\longleftrightarrow$ (*no-T-F-symb* (*conn c* [φ, ψ]) $\wedge$ *no-T-F* $\varphi \wedge$ *no-T-F* ψ)
    by (*simp add*: *all-subformula-st-decomp no-T-F-def*)
  then show *no-T-F* (*conn c* [φ, ψ]) $\longleftrightarrow$ (*no-T-F* $\varphi \wedge$ *no-T-F* ψ)
    using *c wf all-subformula-st-decomp list.discI no-T-F-def no-T-F-symb-except-toplevel-bin-decom*
      no-T-F-symb-except-toplevel-no-T-F-symb no-T-F-symb-false(*1*,*2*) *wf-conn-helper-facts*(*2*,*3*)
      wf-conn-list(*1*,*2*) **by** *metis*
qed

lemma *no-T-F-bin-decomp-expanded*[*simp*]:
  assumes *c*: c = *CAnd* $\vee$ c = *COr* $\vee$ c = *CEq* $\vee$ c = *CImp*
  shows *no-T-F* (*conn c* [φ, ψ]) $\longleftrightarrow$ (*no-T-F* $\varphi \wedge$ *no-T-F* ψ)
  using *no-T-F-bin-decomp assms* **unfolding** *binary-connectives-def* **by** *blast*

lemma *no-T-F-comp-expanded-explicit*[*simp*]:
  fixes φ ψ :: *'v propo*
  shows
    no-T-F (*FAnd* φ ψ) $\longleftrightarrow$ (*no-T-F* $\varphi \wedge$ *no-T-F* ψ)
    no-T-F (*FOr* φ ψ) $\longleftrightarrow$ (*no-T-F* $\varphi \wedge$ *no-T-F* ψ)
    no-T-F (*FEq* φ ψ) $\longleftrightarrow$ (*no-T-F* $\varphi \wedge$ *no-T-F* ψ)
    no-T-F (*FImp* φ ψ) $\longleftrightarrow$ (*no-T-F* $\varphi \wedge$ *no-T-F* ψ)
  using *conn.simps*(*5*−*8*) *no-T-F-bin-decomp-expanded* **by** (*metis* (*no-types*))+

lemma *no-T-F-comp-not*[*simp*]:
  fixes φ ψ :: *'v propo*
  shows *no-T-F* (*FNot* φ) $\longleftrightarrow$ *no-T-F* φ
  by (*metis all-subformula-st-decomp-explicit*(*3*) *all-subformula-st-test-symb-true-phi no-T-F-def*
    no-T-F-symb-false(*1*,*2*) *no-T-F-symb-fnot-imp*)

lemma *no-T-F-decomp*:
  fixes φ ψ :: *'v propo*
  assumes φ: *no-T-F* (*FAnd* φ ψ) $\vee$ *no-T-F* (*FOr* φ ψ) $\vee$ *no-T-F* (*FEq* φ ψ) $\vee$ *no-T-F* (*FImp* φ ψ)
  shows *no-T-F* ψ **and** *no-T-F* φ
  using *assms* **by** *auto*

lemma *no-T-F-decomp-not*:
  fixes φ :: *'v propo*
  assumes φ: *no-T-F* (*FNot* φ)
  shows *no-T-F* φ
  using *assms* **by** *auto*

lemma *no-T-F-symb-except-toplevel-step-exists*:
  fixes φ ψ :: *'v propo*
  assumes *no-equiv* φ **and** *no-imp* φ
  shows $\psi \preceq \varphi \Longrightarrow \neg$ *no-T-F-symb-except-toplevel* $\psi \Longrightarrow \exists \psi'.$ *elimTB* ψ ψ'
proof (*induct* ψ *rule*: *propo-induct-arity*)
  case (*nullary* φ' *x*)
  then have *False* **using** *no-T-F-symb-except-toplevel-true no-T-F-symb-except-toplevel-false* **by** *auto*
  then show *?case* **by** *blast*
next
  case (*unary* ψ)
  then have ψ = *FF* $\vee$ ψ = *FT* **using** *no-T-F-symb-except-toplevel-not-decom* **by** *blast*

  then show *?case* **using** *ElimTB5 ElimTB6* **by** *blast*
next
  case (*binary* φ' $\psi 1$ $\psi 2$)
  note *IH1* = *this*(*1*) **and** *IH2* = *this*(*2*) **and** φ' = *this*(*3*) **and** $F\varphi$ = *this*(*4*) **and** *n* = *this*(*5*)
  {
    assume $\varphi' = FImp\ \psi 1\ \psi 2 \lor \varphi' = FEq\ \psi 1\ \psi 2$
    then have *False* **using** *n* $F\varphi$ *subformula-all-subformula-st assms*
      by (*metis* (*no-types*) *no-equiv-eq*(*1*) *no-equiv-def no-imp-Imp*(*1*) *no-imp-def*)
    then have *?case* **by** *blast*
  }
  moreover {
    assume φ': $\varphi' = FAnd\ \psi 1\ \psi 2 \lor \varphi' = FOr\ \psi 1\ \psi 2$
    then have $\psi 1 = FT \lor \psi 2 = FT \lor \psi 1 = FF \lor \psi 2 = FF$
     using *no-T-F-symb-except-toplevel-bin-decom conn.simps*(*5*,*6*) *n* **unfolding** *binary-connectives-def*
      by *fastforce+*
    then have *?case* **using** *elimTB.intros* φ' **by** *blast*
  }
  ultimately show *?case* **using** φ' **by** *blast*
qed

lemma *no-T-F-except-top-level-rew*:
  fixes φ :: *'v propo*
  assumes *noTB*: $\neg$ *no-T-F-except-top-level* φ **and** *no-equiv*: *no-equiv* φ **and** *no-imp*: *no-imp* φ
  shows $\exists \psi\ \psi'.\ \psi \preceq \varphi \land elimTB\ \psi\ \psi'$
proof −
  have *test-symb-false-nullary*: $\forall x.$ *no-T-F-symb-except-toplevel* (*FF*:: *'v propo*)
    $\land$ *no-T-F-symb-except-toplevel FT* $\land$ *no-T-F-symb-except-toplevel* (*FVar* (*x*:: *'v*)) **by** *auto*
  moreover {
     fix *c*:: *'v connective* **and** *l* :: *'v propo list* **and** ψ :: *'v propo*
     have *H*: *elimTB* (*conn c l*) $\psi \Longrightarrow \neg$*no-T-F-symb-except-toplevel* (*conn c l*)
       by (*cases conn c l rule*: *elimTB.cases*, *auto*)
  }
  moreover {
     fix *x* :: *'v*
     have *H'*: *no-T-F-except-top-level FT no-T-F-except-top-level FF*
       no-T-F-except-top-level (*FVar x*)
       by (*auto simp*: *no-T-F-except-top-level-def test-symb-false-nullary*)
  }
  moreover {
     fix ψ
     have $\psi \preceq \varphi \Longrightarrow \neg$ *no-T-F-symb-except-toplevel* $\psi \Longrightarrow \exists \psi'.$ *elimTB* $\psi\ \psi'$
       using *no-T-F-symb-except-toplevel-step-exists no-equiv no-imp* **by** *auto*
  }
  ultimately show *?thesis*
    using *no-test-symb-step-exists noTB* **unfolding** *no-T-F-except-top-level-def* **by** *blast*
qed

lemma *elimTB-inv*:
  fixes $\varphi\ \psi$ :: *'v propo*
  assumes *full* (*propo-rew-step elimTB*) $\varphi\ \psi$
  and *no-equiv* φ **and** *no-imp* φ
  shows *no-equiv* ψ **and** *no-imp* ψ
proof −
  {
     fix $\varphi\ \psi$ :: *'v propo*
     have *H*: *elimTB* $\varphi\ \psi \Longrightarrow$ *no-equiv* $\varphi \Longrightarrow$ *no-equiv* ψ

by (*induct* φ ψ *rule*: *elimTB.induct*, *auto*)
}
then show *no-equiv* ψ
using *full-propo-rew-step-inv-stay-conn*[*of elimTB no-equiv-symb* φ ψ]
no-equiv-symb-conn-characterization assms **unfolding** *no-equiv-def* **by** *metis*
next
{
fix φ ψ :: *'v propo*
have *H*: *elimTB* φ ψ $\Longrightarrow$ *no-imp* φ $\Longrightarrow$ *no-imp* ψ
by (*induct* φ ψ *rule*: *elimTB.induct*, *auto*)
}
then show *no-imp* ψ
using *full-propo-rew-step-inv-stay-conn*[*of elimTB no-imp-symb* φ ψ] *assms*
no-imp-symb-conn-characterization **unfolding** *no-imp-def* **by** *metis*
qed

lemma *elimTB-full-propo-rew-step*:
fixes φ ψ :: *'v propo*
assumes *no-equiv* φ **and** *no-imp* φ **and** *full* (*propo-rew-step elimTB*) φ ψ
shows *no-T-F-except-top-level* ψ
using *full-propo-rew-step-subformula no-T-F-except-top-level-rew assms elimTB-inv* **by** *fastforce*

1.5.4 PushNeg

Push the negation inside the formula, until the litteral.

inductive *pushNeg* **where**
PushNeg1[*simp*]: *pushNeg* (*FNot* (*FAnd* φ ψ)) (*FOr* (*FNot* φ) (*FNot* ψ)) |
PushNeg2[*simp*]: *pushNeg* (*FNot* (*FOr* φ ψ)) (*FAnd* (*FNot* φ) (*FNot* ψ)) |
PushNeg3[*simp*]: *pushNeg* (*FNot* (*FNot* φ)) φ

lemma *pushNeg-transformation-consistent*:
$A \models$ *FNot* (*FAnd* φ ψ) $\longleftrightarrow$ $A \models$ (*FOr* (*FNot* φ) (*FNot* ψ))
$A \models$ *FNot* (*FOr* φ ψ) $\longleftrightarrow$ $A \models$ (*FAnd* (*FNot* φ) (*FNot* ψ))
$A \models$ *FNot* (*FNot* φ) $\longleftrightarrow$ $A \models \varphi$
by *auto*

lemma *pushNeg-explicit*: *pushNeg* φ ψ $\Longrightarrow \forall A.\ A \models \varphi \longleftrightarrow A \models \psi$
by (*induct* φ ψ *rule*: *pushNeg.induct*, *auto*)

lemma *pushNeg-consistent*: *preserve-models pushNeg*
unfolding *preserve-models-def* **by** (*simp add*: *pushNeg-explicit*)

lemma *pushNeg-lifted-consistant*:
preserve-models (*full* (*propo-rew-step pushNeg*))
by (*simp add*: *pushNeg-consistent*)

fun *simple* **where**
simple FT = *True* |
simple FF = *True* |
simple (*FVar* -) = *True* |
simple - = *False*

lemma *simple-decomp*:
  simple $\varphi \longleftrightarrow (\varphi = FT \lor \varphi = FF \lor (\exists x.\ \varphi = FVar\ x))$
  by (*cases* φ) *auto*

lemma *subformula-conn-decomp-simple*:
  fixes $\varphi\ \psi$ :: *'v propo*
  assumes *s*: *simple* ψ
  shows $\varphi \preceq FNot\ \psi \longleftrightarrow (\varphi = FNot\ \psi \lor \varphi = \psi)$
proof −
  have $\varphi \preceq conn\ CNot\ [\psi] \longleftrightarrow (\varphi = conn\ CNot\ [\psi] \lor (\exists\ \psi \in set\ [\psi].\ \varphi \preceq \psi))$
    using *subformula-conn-decomp wf-conn-helper-facts(1)* **by** *metis*
  then show $\varphi \preceq FNot\ \psi \longleftrightarrow (\varphi = FNot\ \psi \lor \varphi = \psi)$ **using** *s* **by** (*auto simp*: *simple-decomp*)
qed

lemma *subformula-conn-decomp-explicit*[*simp*]:
  fixes φ :: *'v propo* **and** *x* :: *'v*
  shows
    $\varphi \preceq FNot\ FT \longleftrightarrow (\varphi = FNot\ FT \lor \varphi = FT)$
    $\varphi \preceq FNot\ FF \longleftrightarrow (\varphi = FNot\ FF \lor \varphi = FF)$
    $\varphi \preceq FNot\ (FVar\ x) \longleftrightarrow (\varphi = FNot\ (FVar\ x) \lor \varphi = FVar\ x)$
  by (*auto simp*: *subformula-conn-decomp-simple*)

fun *simple-not-symb* **where**
simple-not-symb (*FNot* φ) = (*simple* φ) |
simple-not-symb - = *True*

definition *simple-not* **where**
simple-not = *all-subformula-st simple-not-symb*
declare *simple-not-def*[*simp*]

lemma *simple-not-Not*[*simp*]:
  ¬ *simple-not* (*FNot* (*FAnd* $\varphi\ \psi$))
  ¬ *simple-not* (*FNot* (*FOr* $\varphi\ \psi$))
  by *auto*

lemma *simple-not-step-exists*:
  fixes $\varphi\ \psi$ :: *'v propo*
  assumes *no-equiv* φ **and** *no-imp* φ
  shows $\psi \preceq \varphi \Longrightarrow \neg$ *simple-not-symb* $\psi \Longrightarrow \exists \psi'.$ *pushNeg* $\psi\ \psi'$
  apply (*induct* ψ, *auto*)
  apply (*rename-tac* ψ, *case-tac* ψ, *auto intro*: *pushNeg.intros*)
  by (*metis assms(1,2) no-imp-Imp(1) no-equiv-eq(1) no-imp-def no-equiv-def*
    subformula-in-subformula-not subformula-all-subformula-st)+

lemma *simple-not-rew*:
  fixes φ :: *'v propo*
  assumes *noTB*: ¬ *simple-not* φ **and** *no-equiv*: *no-equiv* φ **and** *no-imp*: *no-imp* φ
  shows $\exists \psi\ \psi'.\ \psi \preceq \varphi \land$ *pushNeg* $\psi\ \psi'$
proof −
  have $\forall x.$ *simple-not-symb* (*FF*:: *'v propo*) ∧ *simple-not-symb FT* ∧ *simple-not-symb* (*FVar* (*x*:: *'v*))
    by *auto*
  moreover {
    fix *c*:: *'v connective* **and** *l* :: *'v propo list* **and** ψ :: *'v propo*
    have *H*: *pushNeg* (*conn c l*) $\psi \Longrightarrow \neg$*simple-not-symb* (*conn c l*)
      by (*cases conn c l rule*: *pushNeg.cases*) *auto*

```
  }
  moreover {
    fix x :: 'v
    have H': simple-not FT simple-not FF simple-not (FVar x)
      by simp-all
  }
  moreover {
    fix ψ :: 'v propo
    have ψ ≼ φ ⟹ ¬ simple-not-symb ψ ⟹ ∃ψ'. pushNeg ψ ψ'
      using simple-not-step-exists no-equiv no-imp by blast
  }
  ultimately show ?thesis using no-test-symb-step-exists noTB unfolding simple-not-def by blast
qed

lemma no-T-F-except-top-level-pushNeg1:
  no-T-F-except-top-level (FNot (FAnd φ ψ)) ⟹ no-T-F-except-top-level (FOr (FNot φ) (FNot ψ))
  using no-T-F-symb-except-toplevel-all-subformula-st-no-T-F-symb no-T-F-comp-not no-T-F-decomp(1)
    no-T-F-decomp(2) no-T-F-no-T-F-except-top-level by (metis no-T-F-comp-expanded-explicit(2)
      propo.distinct(5,17))

lemma no-T-F-except-top-level-pushNeg2:
  no-T-F-except-top-level (FNot (FOr φ ψ)) ⟹ no-T-F-except-top-level (FAnd (FNot φ) (FNot ψ))
  by auto

lemma no-T-F-symb-pushNeg:
  no-T-F-symb (FOr (FNot φ') (FNot ψ'))
  no-T-F-symb (FAnd (FNot φ') (FNot ψ'))
  no-T-F-symb (FNot (FNot φ'))
  by auto

lemma propo-rew-step-pushNeg-no-T-F-symb:
  propo-rew-step pushNeg φ ψ ⟹ no-T-F-except-top-level φ ⟹ no-T-F-symb φ ⟹ no-T-F-symb ψ
  apply (induct rule: propo-rew-step.induct)
  apply (cases rule: pushNeg.cases)
  apply simp-all
  apply (metis no-T-F-symb-pushNeg(1))
  apply (metis no-T-F-symb-pushNeg(2))
  apply (simp, metis all-subformula-st-test-symb-true-phi no-T-F-def)
proof -
  fix φ φ':: 'a propo and c:: 'a connective and ξ ξ':: 'a propo list
  assume rel: propo-rew-step pushNeg φ φ'
  and IH: no-T-F φ ⟹ no-T-F-symb φ ⟹ no-T-F-symb φ'
  and wf: wf-conn c (ξ @ φ # ξ')
  and n: conn c (ξ @ φ # ξ') = FF ∨ conn c (ξ @ φ # ξ') = FT ∨ no-T-F (conn c (ξ @ φ # ξ'))
  and x: c ≠ CF ∧ c ≠ CT ∧ φ ≠ FF ∧ φ ≠ FT ∧ (∀ψ ∈ set ξ ∪ set ξ'. ψ ≠ FF ∧ ψ ≠ FT)
  then have c ≠ CF ∧ c ≠ CF ∧ wf-conn c (ξ @ φ' # ξ')
    using wf-conn-no-arity-change-helper wf-conn-no-arity-change by metis
  moreover have n': no-T-F (conn c (ξ @ φ # ξ')) using n by (simp add: wf wf-conn-list(1,2))
  moreover
  {
    have no-T-F φ
      by (metis Un-iff all-subformula-st-decomp list.set-intros(1) n' wf no-T-F-def set-append)
    moreover then have no-T-F-symb φ
      by (simp add: all-subformula-st-test-symb-true-phi no-T-F-def)
    ultimately have φ' ≠ FF ∧ φ' ≠ FT
      using IH no-T-F-symb-false(1) no-T-F-symb-false(2) by blast
```

```
    then have ∀ψ∈ set (ξ @ φ' # ξ'). ψ ≠ FF ∧ ψ ≠ FT using x by auto
  }
  ultimately show no-T-F-symb (conn c (ξ @ φ' # ξ')) by (simp add: x)
qed

lemma propo-rew-step-pushNeg-no-T-F:
  propo-rew-step pushNeg φ ψ ⟹ no-T-F φ ⟹ no-T-F ψ
proof (induct rule: propo-rew-step.induct)
  case global-rel
  then show ?case
    by (metis (no-types, lifting) no-T-F-symb-except-toplevel-all-subformula-st-no-T-F-symb
      no-T-F-def no-T-F-except-top-level-pushNeg1 no-T-F-except-top-level-pushNeg2
      no-T-F-no-T-F-except-top-level all-subformula-st-decomp-explicit(3) pushNeg.simps
      simple.simps(1,2,5,6))
next
  case (propo-rew-one-step-lift φ φ' c ξ ξ')
  note rel = this(1) and IH = this(2) and wf = this(3) and no-T-F = this(4)
  moreover have wf': wf-conn c (ξ @ φ' # ξ')
    using wf-conn-no-arity-change wf-conn-no-arity-change-helper wf by metis
  ultimately show no-T-F (conn c (ξ @ φ' # ξ'))
    using all-subformula-st-test-symb-true-phi
    by (fastforce simp: no-T-F-def all-subformula-st-decomp wf wf')
qed


lemma pushNeg-inv:
  fixes φ ψ :: 'v propo
  assumes full (propo-rew-step pushNeg) φ ψ
  and no-equiv φ and no-imp φ and no-T-F-except-top-level φ
  shows no-equiv ψ and no-imp ψ and no-T-F-except-top-level ψ
proof –
  {
    fix φ ψ :: 'v propo
    assume rel: propo-rew-step pushNeg φ ψ
    and no: no-T-F-except-top-level φ
    then have  no-T-F-except-top-level ψ
      proof –
        {
          assume φ = FT ∨ φ = FF
          from rel this have False
            apply (induct rule: propo-rew-step.induct)
              using pushNeg.cases apply blast
            using wf-conn-list(1) wf-conn-list(2) by auto
          then have no-T-F-except-top-level ψ by blast
        }
        moreover {
          assume φ ≠ FT ∧ φ ≠ FF
          then have no-T-F φ
            by (metis no no-T-F-symb-except-toplevel-all-subformula-st-no-T-F-symb)
          then have no-T-F ψ
            using propo-rew-step-pushNeg-no-T-F rel by auto
          then have no-T-F-except-top-level ψ by (simp add: no-T-F-no-T-F-except-top-level)
        }
        ultimately show no-T-F-except-top-level ψ by metis
      qed
  }
```

moreover {
fix *c* :: *'v connective* **and** ξ ξ' :: *'v propo list* **and** ζ ζ' :: *'v propo*
assume *rel*: *propo-rew-step pushNeg* ζ ζ'
and *incl*: $\zeta \preceq \varphi$
and *corr*: *wf-conn c* (ξ @ ζ # ξ')
and *no-T-F*: *no-T-F-symb-except-toplevel* (*conn c* (ξ @ ζ # ξ'))
and *n*: *no-T-F-symb-except-toplevel* ζ'
have *no-T-F-symb-except-toplevel* (*conn c* (ξ @ ζ' # ξ'))
proof
have *p*: *no-T-F-symb* (*conn c* (ξ @ ζ # ξ'))
using *corr wf-conn-list(1) wf-conn-list(2) no-T-F-symb-except-toplevel-no-T-F-symb no-T-F*
by *blast*
have *l*: $\forall\varphi\in$*set* (ξ @ ζ # ξ'). $\varphi \neq FT \wedge \varphi \neq FF$
using *corr wf-conn-no-T-F-symb-iff p* **by** *blast*
from *rel incl* **have** $\zeta'\neq FT \wedge\zeta'\neq FF$
apply (*induction* ζ ζ' *rule*: *propo-rew-step.induct*)
apply (*cases rule*: *pushNeg.cases*, *auto*)
by (*metis assms(4) no-T-F-symb-except-top-level-false-not no-T-F-except-top-level-def*
all-subformula-st-test-symb-true-phi subformula-in-subformula-not
subformula-all-subformula-st append-is-Nil-conv list.distinct(1)
wf-conn-no-arity-change-helper wf-conn-list(1,2) wf-conn-no-arity-change)+
then have $\forall\varphi \in$ *set* (ξ @ ζ' # ξ'). $\varphi \neq FT \wedge \varphi \neq FF$ **using** *l* **by** *auto*
moreover have $c \neq CT \wedge c \neq CF$ **using** *corr* **by** *auto*
ultimately show *no-T-F-symb* (*conn c* (ξ @ ζ' # ξ'))
by (*metis corr no-T-F-symb-comp wf-conn-no-arity-change wf-conn-no-arity-change-helper*)
qed
}
ultimately show *no-T-F-except-top-level* ψ
using *full-propo-rew-step-inv-stay-with-inc[of pushNeg no-T-F-symb-except-toplevel* φ*] assms*
subformula-refl **unfolding** *no-T-F-except-top-level-def full-unfold* **by** *metis*
next
{
fix φ ψ :: *'v propo*
have *H*: *pushNeg* φ $\psi \Longrightarrow$ *no-equiv* $\varphi \Longrightarrow$ *no-equiv* ψ
by (*induct* φ ψ *rule*: *pushNeg.induct*, *auto*)
}
then show *no-equiv* ψ
using *full-propo-rew-step-inv-stay-conn[of pushNeg no-equiv-symb* φ ψ*]*
no-equiv-symb-conn-characterization assms **unfolding** *no-equiv-def full-unfold* **by** *metis*
next
{
fix φ ψ :: *'v propo*
have *H*: *pushNeg* φ $\psi \Longrightarrow$ *no-imp* $\varphi \Longrightarrow$ *no-imp* ψ
by (*induct* φ ψ *rule*: *pushNeg.induct*, *auto*)
}
then show *no-imp* ψ
using *full-propo-rew-step-inv-stay-conn[of pushNeg no-imp-symb* φ ψ*] assms*
no-imp-symb-conn-characterization **unfolding** *no-imp-def full-unfold* **by** *metis*
qed

lemma *pushNeg-full-propo-rew-step*:
fixes φ ψ :: *'v propo*
assumes
no-equiv φ **and**
no-imp φ **and**

full (*propo-rew-step pushNeg*) φ ψ **and**
no-T-F-except-top-level φ
shows *simple-not* ψ
using *assms full-propo-rew-step-subformula pushNeg-inv*(*1*,*2*) *simple-not-rew* **by** *blast*

1.5.5 Push Inside

inductive *push-conn-inside* :: *'v connective* ⇒ *'v connective* ⇒ *'v propo* ⇒ *'v propo* ⇒ *bool*
for *c c'*:: *'v connective* **where**
push-conn-inside-l[*simp*]: $c = CAnd \lor c = COr \Longrightarrow c' = CAnd \lor c' = COr$
$\Longrightarrow$ *push-conn-inside c c'* (*conn c* [*conn c'* [$\varphi 1$, $\varphi 2$], ψ])
(*conn c'* [*conn c* [$\varphi 1$, ψ], *conn c* [$\varphi 2$, ψ]]) |
push-conn-inside-r[*simp*]: $c = CAnd \lor c = COr \Longrightarrow c' = CAnd \lor c' = COr$
$\Longrightarrow$ *push-conn-inside c c'* (*conn c* [ψ, *conn c'* [$\varphi 1$, $\varphi 2$]])
(*conn c'* [*conn c* [ψ, $\varphi 1$], *conn c* [ψ, $\varphi 2$]])

lemma *push-conn-inside-explicit*: *push-conn-inside c c'* φ $\psi \Longrightarrow \forall A.\ A \models \varphi \longleftrightarrow A \models \psi$
by (*induct* φ ψ *rule*: *push-conn-inside.induct*, *auto*)

lemma *push-conn-inside-consistent*: *preserve-models* (*push-conn-inside c c'*)
unfolding *preserve-models-def* **by** (*simp add*: *push-conn-inside-explicit*)

lemma *propo-rew-step-push-conn-inside*[*simp*]:
¬*propo-rew-step* (*push-conn-inside c c'*) *FT* ψ ¬*propo-rew-step* (*push-conn-inside c c'*) *FF* ψ
proof −
{
{
fix φ ψ
have *push-conn-inside c c'* φ $\psi \Longrightarrow \varphi = FT \lor \varphi = FF \Longrightarrow False$
by (*induct rule*: *push-conn-inside.induct*, *auto*)
} **note** $H = this$
fix φ
have *propo-rew-step* (*push-conn-inside c c'*) φ $\psi \Longrightarrow \varphi = FT \lor \varphi = FF \Longrightarrow False$
apply (*induct rule*: *propo-rew-step.induct*, *auto simp*: *wf-conn-list*(*1*) *wf-conn-list*(*2*))
using *H* **by** *blast+*
}
then show
¬*propo-rew-step* (*push-conn-inside c c'*) *FT* ψ
¬*propo-rew-step* (*push-conn-inside c c'*) *FF* ψ **by** *blast+*
qed

inductive *not-c-in-c'-symb*:: *'v connective* ⇒ *'v connective* ⇒ *'v propo* ⇒ *bool* **for** *c c'* **where**
not-c-in-c'-symb-l[*simp*]: *wf-conn c* [*conn c'* [φ, φ'], ψ] $\Longrightarrow$ *wf-conn c'* [φ, φ']
$\Longrightarrow$ *not-c-in-c'-symb c c'* (*conn c* [*conn c'* [φ, φ'], ψ]) |
not-c-in-c'-symb-r[*simp*]: *wf-conn c* [ψ, *conn c'* [φ, φ']] $\Longrightarrow$ *wf-conn c'* [φ, φ']
$\Longrightarrow$ *not-c-in-c'-symb c c'* (*conn c* [ψ, *conn c'* [φ, φ']])

abbreviation *c-in-c'-symb c c'* φ ≡ ¬*not-c-in-c'-symb c c'* φ

lemma *c-in-c'-symb-simp*:
not-c-in-c'-symb c c' $\xi \Longrightarrow \xi = FF \lor \xi = FT \lor \xi = FVar\ x \lor \xi = FNot\ FF \lor \xi = FNot\ FT$
$\lor\ \xi = FNot\ (FVar\ x) \Longrightarrow False$
apply (*induct rule*: *not-c-in-c'-symb.induct*, *auto simp*: *wf-conn.simps wf-conn-list*(*1*−*3*))

using *conn-inj-not*(*2*) *wf-conn-binary* **unfolding** *binary-connectives-def* **by** *fastforce+*

lemma *c-in-c′-symb-simp′*[*simp*]:
$\neg$*not-c-in-c′-symb c c′ FF*
$\neg$*not-c-in-c′-symb c c′ FT*
$\neg$*not-c-in-c′-symb c c′* (*FVar x*)
$\neg$*not-c-in-c′-symb c c′* (*FNot FF*)
$\neg$*not-c-in-c′-symb c c′* (*FNot FT*)
$\neg$*not-c-in-c′-symb c c′* (*FNot* (*FVar x*))
using *c-in-c′-symb-simp* **by** *metis+*

definition *c-in-c′-only* **where**
c-in-c′-only c c′ $\equiv$ *all-subformula-st* (*c-in-c′-symb c c′*)

lemma *c-in-c′-only-simp*[*simp*]:
c-in-c′-only c c′ FF
c-in-c′-only c c′ FT
c-in-c′-only c c′ (*FVar x*)
c-in-c′-only c c′ (*FNot FF*)
c-in-c′-only c c′ (*FNot FT*)
c-in-c′-only c c′ (*FNot* (*FVar x*))
unfolding *c-in-c′-only-def* **by** *auto*

lemma *not-c-in-c′-symb-commute*:
not-c-in-c′-symb c c′ $\xi \Longrightarrow$ *wf-conn c* $[\varphi, \psi] \Longrightarrow \xi =$ *conn c* $[\varphi, \psi]$
$\Longrightarrow$ *not-c-in-c′-symb c c′* (*conn c* $[\psi, \varphi]$)
proof (*induct rule*: *not-c-in-c′-symb.induct*)
case (*not-c-in-c′-symb-r* φ' φ'' ψ') **note** $H = this$
then have ψ: $\psi =$ *conn c′* $[\varphi'', \psi']$ **using** *conn-inj* **by** *auto*
have *wf-conn c* [*conn c′* $[\varphi'', \psi']$, φ]
using *H*(*1*) *wf-conn-no-arity-change length-Cons* **by** *metis*
then show *not-c-in-c′-symb c c′* (*conn c* $[\psi, \varphi]$)
unfolding ψ **using** *not-c-in-c′-symb.intros*(*1*) *H* **by** *auto*
next
case (*not-c-in-c′-symb-l* φ' φ'' ψ') **note** $H = this$
then have $\varphi =$ *conn c′* $[\varphi', \varphi'']$ **using** *conn-inj* **by** *auto*
moreover have *wf-conn c* [ψ', *conn c′* $[\varphi', \varphi'']$]
using *H*(*1*) *wf-conn-no-arity-change length-Cons* **by** *metis*
ultimately show *not-c-in-c′-symb c c′* (*conn c* $[\psi, \varphi]$)
using *not-c-in-c′-symb.intros*(*2*) *conn-inj not-c-in-c′-symb-l.hyps*
not-c-in-c′-symb-l.prems(*1*,*2*) **by** *blast*
qed

lemma *not-c-in-c′-symb-commute′*:
wf-conn c $[\varphi, \psi] \Longrightarrow$ *c-in-c′-symb c c′* (*conn c* $[\varphi, \psi]$) $\longleftrightarrow$ *c-in-c′-symb c c′* (*conn c* $[\psi, \varphi]$)
using *not-c-in-c′-symb-commute wf-conn-no-arity-change* **by** (*metis length-Cons*)

lemma *not-c-in-c′-comm*:
assumes *wf*: *wf-conn c* $[\varphi, \psi]$
shows *c-in-c′-only c c′* (*conn c* $[\varphi, \psi]$) $\longleftrightarrow$ *c-in-c′-only c c′* (*conn c* $[\psi, \varphi]$) (**is** *?A* $\longleftrightarrow$ *?B*)
proof −
have *?A* $\longleftrightarrow$ (*c-in-c′-symb c c′* (*conn c* $[\varphi, \psi]$)
$\wedge$ ($\forall \xi \in$ *set* $[\varphi, \psi]$. *all-subformula-st* (*c-in-c′-symb c c′*) ξ))
using *all-subformula-st-decomp wf* **unfolding** *c-in-c′-only-def* **by** *fastforce*
also have . . . $\longleftrightarrow$ (*c-in-c′-symb c c′* (*conn c* $[\psi, \varphi]$)

$\wedge$ ($\forall\xi \in$ *set* [ψ, φ]. *all-subformula-st* (*c-in-c'-symb c c'*) ξ))
using *not-c-in-c'-symb-commute' wf* **by** *auto*
also
have *wf-conn c* [ψ, φ] **using** *wf-conn-no-arity-change wf* **by** (*metis length-Cons*)
then have (*c-in-c'-symb c c'* (*conn c* [ψ, φ])
$\wedge$ ($\forall\xi \in$ *set* [ψ, φ]. *all-subformula-st* (*c-in-c'-symb c c'*) ξ))
$\longleftrightarrow$ *?B*
using *all-subformula-st-decomp* **unfolding** *c-in-c'-only-def* **by** *fastforce*
finally show *?thesis* **.**
qed

lemma *not-c-in-c'-simp*[*simp*]:
fixes $\varphi 1$ $\varphi 2$ ψ :: *'v propo* **and** *x* :: *'v*
shows
c-in-c'-symb c c' FT
c-in-c'-symb c c' FF
c-in-c'-symb c c' (*FVar x*)
wf-conn c [*conn c'* [$\varphi 1$, $\varphi 2$], ψ] $\Longrightarrow$ *wf-conn c'* [$\varphi 1$, $\varphi 2$]
$\Longrightarrow$ $\neg$ *c-in-c'-only c c'* (*conn c* [*conn c'* [$\varphi 1$, $\varphi 2$], ψ])
apply (*simp-all add*: *c-in-c'-only-def*)
using *all-subformula-st-test-symb-true-phi not-c-in-c'-symb-l* **by** *blast*

lemma *c-in-c'-symb-not*[*simp*]:
fixes *c c'* :: *'v connective* **and** ψ :: *'v propo*
shows *c-in-c'-symb c c'* (*FNot* ψ)
proof −
{
fix ξ :: *'v propo*
have *not-c-in-c'-symb c c'* (*FNot* ψ) $\Longrightarrow$ *False*
apply (*induct FNot* ψ *rule*: *not-c-in-c'-symb.induct*)
using *conn-inj-not*(*2*) **by** *blast+*
}
then show *?thesis* **by** *auto*
qed

lemma *c-in-c'-symb-step-exists*:
fixes φ :: *'v propo*
assumes *c*: *c* = *CAnd* $\vee$ *c* = *COr* **and** *c'*: *c'* = *CAnd* $\vee$ *c'* = *COr*
shows $\psi \preceq \varphi \Longrightarrow \neg$ *c-in-c'-symb c c'* $\psi \Longrightarrow \exists\psi'$. *push-conn-inside c c'* ψ ψ'
apply (*induct* ψ *rule*: *propo-induct-arity*)
apply *auto*[*2*]
proof −
fix $\psi 1$ $\psi 2$ φ':: *'v propo*
assume *IH*$\psi 1$: $\psi 1 \preceq \varphi \Longrightarrow \neg$ *c-in-c'-symb c c'* $\psi 1$ $\Longrightarrow$ *Ex* (*push-conn-inside c c'* $\psi 1$)
and *IH*$\psi 2$: $\psi 1 \preceq \varphi \Longrightarrow \neg$ *c-in-c'-symb c c'* $\psi 1$ $\Longrightarrow$ *Ex* (*push-conn-inside c c'* $\psi 1$)
and φ': φ' = *FAnd* $\psi 1$ $\psi 2$ $\vee$ φ' = *FOr* $\psi 1$ $\psi 2$ $\vee$ φ' = *FImp* $\psi 1$ $\psi 2$ $\vee$ φ' = *FEq* $\psi 1$ $\psi 2$
and *in*φ: $\varphi' \preceq \varphi$ **and** *n0*: $\neg$*c-in-c'-symb c c'* φ'
then have *n*: *not-c-in-c'-symb c c'* φ' **by** *auto*
{
assume φ': φ' = *conn c* [$\psi 1$, $\psi 2$]
obtain *a b* **where** $\psi 1$ = *conn c'* [*a*, *b*] $\vee$ $\psi 2$ = *conn c'* [*a*, *b*]
using *n* φ' **apply** (*induct rule*: *not-c-in-c'-symb.induct*)
using *c* **by** *force+*
then have *Ex* (*push-conn-inside c c'* φ')
unfolding φ' **apply** *auto*
using *push-conn-inside.intros*(*1*) *c c'* **apply** *blast*

using *push-conn-inside.intros*(*2*) *c c'* **by** *blast*
}
moreover {
assume φ': $\varphi' \neq$ *conn c* [$\psi 1$, $\psi 2$]
have $\forall \varphi$ *c ca*. $\exists \varphi 1\ \psi 1\ \psi 2\ \psi 1'\ \psi 2'\ \varphi 2'$. *conn* (*c*::*'v connective*) [$\varphi 1$, *conn ca* [$\psi 1$, $\psi 2$]] $= \varphi$ $\vee$ *conn c* [*conn ca* [$\psi 1'$, $\psi 2'$], $\varphi 2'$] $= \varphi \vee$ *c-in-c'-symb c ca* φ
by (*metis not-c-in-c'-symb.cases*)
then have *Ex* (*push-conn-inside c c'* φ')
by (*metis* (*no-types*) *c c' n push-conn-inside-l push-conn-inside-r*)
}
ultimately show *Ex* (*push-conn-inside c c'* φ') **by** *blast*
qed

lemma *c-in-c'-symb-rew*:
fixes φ :: *'v propo*
assumes *noTB*: $\neg$*c-in-c'-only c c'* φ
and *c*: $c = CAnd \vee c = COr$ **and** *c'*: $c' = CAnd \vee c' = COr$
shows $\exists \psi\ \psi'$. $\psi \preceq \varphi \wedge$ *push-conn-inside c c'* $\psi\ \psi'$
proof −
have *test-symb-false-nullary*:
$\forall x$. *c-in-c'-symb c c'* (*FF*:: *'v propo*) $\wedge$ *c-in-c'-symb c c' FT* $\wedge$ *c-in-c'-symb c c'* (*FVar* (*x*:: *'v*))
by *auto*
moreover {
fix *x* :: *'v*
have *H'*: *c-in-c'-symb c c' FT c-in-c'-symb c c' FF c-in-c'-symb c c'* (*FVar x*)
by *simp+*
}
moreover {
fix ψ :: *'v propo*
have $\psi \preceq \varphi \Longrightarrow \neg$ *c-in-c'-symb c c'* $\psi \Longrightarrow \exists \psi'$. *push-conn-inside c c'* $\psi\ \psi'$
by (*auto simp*: *assms*(*2*) *c' c-in-c'-symb-step-exists*)
}
ultimately show *?thesis* **using** *noTB no-test-symb-step-exists*[*of c-in-c'-symb c c'*]
unfolding *c-in-c'-only-def* **by** *metis*
qed

lemma *push-conn-insidec-in-c'-symb-no-T-F*:
fixes $\varphi\ \psi$ :: *'v propo*
shows *propo-rew-step* (*push-conn-inside c c'*) $\varphi\ \psi \Longrightarrow$ *no-T-F* $\varphi \Longrightarrow$ *no-T-F* ψ
proof (*induct rule*: *propo-rew-step.induct*)
case (*global-rel* $\varphi\ \psi$)
then show *no-T-F* ψ
by (*cases rule*: *push-conn-inside.cases*, *auto*)
next
case (*propo-rew-one-step-lift* $\varphi\ \varphi'$ *c* $\xi\ \xi'$)
note *rel* = *this*(*1*) **and** *IH* = *this*(*2*) **and** *wf* = *this*(*3*) **and** *no-T-F* = *this*(*4*)
have *no-T-F* φ
using *wf no-T-F no-T-F-def subformula-into-subformula subformula-all-subformula-st subformula-refl* **by** (*metis* (*no-types*) *in-set-conv-decomp*)
then have φ': *no-T-F* φ' **using** *IH* **by** *blast*

have $\forall \zeta \in$ *set* (ξ @ φ # ξ'). *no-T-F* ζ **by** (*metis wf no-T-F no-T-F-def all-subformula-st-decomp*)
then have *n*: $\forall \zeta \in$ *set* (ξ @ φ' # ξ'). *no-T-F* ζ **using** φ' **by** *auto*
then have *n'*: $\forall \zeta \in$ *set* (ξ @ φ' # ξ'). $\zeta \neq FF \wedge \zeta \neq FT$

using φ' **by** (*metis no-T-F-symb-false*(*1*) *no-T-F-symb-false*(*2*) *no-T-F-def* *all-subformula-st-test-symb-true-phi*)

have *wf'*: *wf-conn c* $(\xi$ @ φ' # $\xi')$
using *wf wf-conn-no-arity-change* **by** (*metis wf-conn-no-arity-change-helper*)
{
fix x :: *'v*
assume $c = CT \lor c = CF \lor c = CVar\ x$
then have *False* **using** *wf* **by** *auto*
then have *no-T-F* (*conn c* $(\xi$ @ φ' # $\xi')$) **by** *blast*
}
moreover {
assume *c*: $c = CNot$
then have $\xi = []$ $\xi' = []$ **using** *wf* **by** *auto*
then have *no-T-F* (*conn c* $(\xi$ @ φ' # $\xi')$)
using *c* **by** (*metis* φ' *conn.simps*(*4*) *no-T-F-symb-false*(*1*,*2*) *no-T-F-symb-fnot no-T-F-def* *all-subformula-st-decomp-explicit*(*3*) *all-subformula-st-test-symb-true-phi self-append-conv2*)
}
moreover {
assume *c*: $c \in$ *binary-connectives*
then have *no-T-F-symb* (*conn c* $(\xi$ @ φ' # $\xi')$) **using** *wf' n' no-T-F-symb.simps* **by** *fastforce*
then have *no-T-F* (*conn c* $(\xi$ @ φ' # $\xi')$)
by (*metis all-subformula-st-decomp-imp wf' n no-T-F-def*)
}
ultimately show *no-T-F* (*conn c* $(\xi$ @ φ' # $\xi')$) **using** *connective-cases-arity* **by** *auto*
qed

lemma *simple-propo-rew-step-push-conn-inside-inv*:
propo-rew-step (*push-conn-inside c c'*) $\varphi\ \psi \Longrightarrow$ *simple* $\varphi \Longrightarrow$ *simple* ψ
apply (*induct rule*: *propo-rew-step.induct*)
apply (*rename-tac* φ, *case-tac* φ, *auto simp*: *push-conn-inside.simps*)[]
by (*metis append-is-Nil-conv list.distinct*(*1*) *simple.elims*(*2*) *wf-conn-list*(*1*−*3*))

lemma *simple-propo-rew-step-inv-push-conn-inside-simple-not*:
fixes *c c'* :: *'v connective* **and** $\varphi\ \psi$ :: *'v propo*
shows *propo-rew-step* (*push-conn-inside c c'*) $\varphi\ \psi \Longrightarrow$ *simple-not* $\varphi \Longrightarrow$ *simple-not* ψ
proof (*induct rule*: *propo-rew-step.induct*)
case (*global-rel* $\varphi\ \psi$)
then show *?case* **by** (*cases* φ, *auto simp*: *push-conn-inside.simps*)
next
case (*propo-rew-one-step-lift* $\varphi\ \varphi'$ *ca* $\xi\ \xi'$) **note** *rew* = *this*(*1*) **and** *IH* = *this*(*2*) **and** *wf* = *this*(*3*) **and** *simple* = *this*(*4*)
show *?case*
proof (*cases ca rule*: *connective-cases-arity*)
case *nullary*
then show *?thesis* **using** *propo-rew-one-step-lift* **by** *auto*
next
case *binary* **note** *ca* = *this*
obtain *a b* **where** *ab*: ξ @ φ' # $\xi' = [a,\ b]$
using *wf ca list-length2-decomp wf-conn-bin-list-length*
by (*metis* (*no-types*) *wf-conn-no-arity-change-helper*)
have $\forall \zeta \in$ *set* $(\xi$ @ φ # $\xi')$. *simple-not* ζ
by (*metis wf all-subformula-st-decomp simple simple-not-def*)
then have $\forall \zeta \in$ *set* $(\xi$ @ φ' # $\xi')$. *simple-not* ζ **using** *IH* **by** *simp*

moreover have *simple-not-symb* (*conn ca* (ξ @ φ' # ξ')) **using** *ca*
by (*metis ab conn.simps*(*5−8*) *helper-fact simple-not-symb.simps*(*5*) *simple-not-symb.simps*(*6*)
simple-not-symb.simps(*7*) *simple-not-symb.simps*(*8*))
ultimately show *?thesis*
by (*simp add*: *ab all-subformula-st-decomp ca*)
next
case *unary*
then show *?thesis*
using *rew simple-propo-rew-step-push-conn-inside-inv*[*OF rew*] *IH local.wf simple* **by** *auto*
qed
qed

lemma *propo-rew-step-push-conn-inside-simple-not*:
fixes φ φ' :: *'v propo* **and** ξ ξ' :: *'v propo list* **and** *c* :: *'v connective*
assumes
propo-rew-step (*push-conn-inside c c'*) φ φ' **and**
wf-conn c (ξ @ φ # ξ') **and**
simple-not-symb (*conn c* (ξ @ φ # ξ')) **and**
simple-not-symb φ'
shows *simple-not-symb* (*conn c* (ξ @ φ' # ξ'))
using *assms*
proof (*induction rule*: *propo-rew-step.induct*)
print-cases
case (*global-rel*)
then show *?case*
by (*metis conn.simps*(*12*,*17*) *list.discI push-conn-inside.cases simple-not-symb.elims*(*3*)
wf-conn-helper-facts(*5*) *wf-conn-list*(*2*) *wf-conn-list*(*8*) *wf-conn-no-arity-change*
wf-conn-no-arity-change-helper)
next
case (*propo-rew-one-step-lift* φ φ' *c'* χs $\chi s'$) **note** *tel* = *this*(*1*) **and** *wf* = *this*(*2*) **and**
IH = *this*(*3*) **and** *wf'* = *this*(*4*) **and** *simple'* = *this*(*5*) **and** *simple* = *this*(*6*)
then show *?case*
proof (*cases c' rule*: *connective-cases-arity*)
case *nullary*
then show *?thesis* **using** *wf simple simple'* **by** *auto*
next
case *binary* **note** *c* = *this*(*1*)
have *corr'*: *wf-conn c* (ξ @ *conn c'* (χs @ φ' # $\chi s'$) # ξ')
using *wf wf-conn-no-arity-change*
by (*metis wf' wf-conn-no-arity-change-helper*)
then show *?thesis*
using *c propo-rew-one-step-lift wf*
by (*metis conn.simps*(*17*) *connective.distinct*(*37*) *propo-rew-step-subformula-imp*
push-conn-inside.cases simple-not-symb.elims(*3*) *wf-conn.simps wf-conn-list*(*2*,*8*))
next
case *unary*
then have *empty*: χs = [] $\chi s'$ = [] **using** *wf* **by** *auto*
then show *?thesis* **using** *simple unary simple' wf wf'*
by (*metis connective.distinct*(*37*) *connective.distinct*(*39*) *propo-rew-step-subformula-imp*
push-conn-inside.cases simple-not-symb.elims(*3*) *tel wf-conn-list*(*8*)
wf-conn-no-arity-change wf-conn-no-arity-change-helper)
qed
qed

lemma *push-conn-inside-not-true-false*:
push-conn-inside c c' φ ψ $\Longrightarrow$ $\psi \neq FT \wedge \psi \neq FF$

by (*induct rule*: *push-conn-inside.induct*, *auto*)

lemma *push-conn-inside-inv*:
fixes φ ψ :: *'v propo*
assumes *full* (*propo-rew-step* (*push-conn-inside c c'*)) φ ψ
and *no-equiv* φ **and** *no-imp* φ **and** *no-T-F-except-top-level* φ **and** *simple-not* φ
shows *no-equiv* ψ **and** *no-imp* ψ **and** *no-T-F-except-top-level* ψ **and** *simple-not* ψ
proof –
{
{
fix φ ψ :: *'v propo*
have *H*: *push-conn-inside c c'* φ ψ $\Longrightarrow$ *all-subformula-st simple-not-symb* φ
$\Longrightarrow$ *all-subformula-st simple-not-symb* ψ
by (*induct* φ ψ *rule*: *push-conn-inside.induct*, *auto*)
} **note** *H* = *this*

fix φ ψ :: *'v propo*
have *H*: *propo-rew-step* (*push-conn-inside c c'*) φ ψ $\Longrightarrow$ *all-subformula-st simple-not-symb* φ
$\Longrightarrow$ *all-subformula-st simple-not-symb* ψ
apply (*induct* φ ψ *rule*: *propo-rew-step.induct*)
using *H* **apply** *simp*
proof (*rename-tac* φ φ' *ca* ψs $\psi s'$, *case-tac ca rule*: *connective-cases-arity*)
fix φ φ' :: *'v propo* **and** *c*:: *'v connective* **and** ξ ξ':: *'v propo list*
and *x*:: *'v*
assume *wf-conn c* (ξ @ φ # ξ')
and $c = CT \lor c = CF \lor c = CVar\ x$
then have ξ @ φ # ξ' = [] **by** *auto*
then have *False* **by** *auto*
then show *all-subformula-st simple-not-symb* (*conn c* (ξ @ φ' # ξ')) **by** *blast*
next
fix φ φ' :: *'v propo* **and** *ca*:: *'v connective* **and** ξ ξ':: *'v propo list*
and *x* :: *'v*
assume *rel*: *propo-rew-step* (*push-conn-inside c c'*) φ φ'
and φ-φ': *all-subformula-st simple-not-symb* φ $\Longrightarrow$ *all-subformula-st simple-not-symb* φ'
and *corr*: *wf-conn ca* (ξ @ φ # ξ')
and *n*: *all-subformula-st simple-not-symb* (*conn ca* (ξ @ φ # ξ'))
and *c*: *ca* = *CNot*

have *empty*: ξ = [] ξ' = [] **using** *c corr* **by** *auto*
then have *simple-not*:*all-subformula-st simple-not-symb* (*FNot* φ) **using** *corr c n* **by** *auto*
then have *simple* φ
using *all-subformula-st-test-symb-true-phi simple-not-symb.simps*(*1*) **by** *blast*
then have *simple* φ'
using *rel simple-propo-rew-step-push-conn-inside-inv* **by** *blast*
then show *all-subformula-st simple-not-symb* (*conn ca* (ξ @ φ' # ξ')) **using** *c empty*
by (*metis simple-not* φ-φ' *append-Nil conn.simps*(*4*) *all-subformula-st-decomp-explicit*(*3*)
simple-not-symb.simps(*1*))
next
fix φ φ' :: *'v propo* **and** *ca* :: *'v connective* **and** ξ ξ' :: *'v propo list*
and *x* :: *'v*
assume *rel*: *propo-rew-step* (*push-conn-inside c c'*) φ φ'
and $n\varphi$: *all-subformula-st simple-not-symb* φ $\Longrightarrow$ *all-subformula-st simple-not-symb* φ'
and *corr*: *wf-conn ca* (ξ @ φ # ξ')
and *n*: *all-subformula-st simple-not-symb* (*conn ca* (ξ @ φ # ξ'))
and *c*: *ca* $\in$ *binary-connectives*

have *all-subformula-st simple-not-symb* φ
using *n c corr all-subformula-st-decomp* **by** *fastforce*
then have φ': *all-subformula-st simple-not-symb* φ' **using** *n*φ **by** *blast*
obtain *a b* **where** *ab*: $[a, b] = (\xi \mathbin{@} \varphi \mathbin{\#} \xi')$
using *corr c list-length2-decomp wf-conn-bin-list-length* **by** *metis*
then have $\xi \mathbin{@} \varphi' \mathbin{\#} \xi' = [a, \varphi'] \lor (\xi \mathbin{@} \varphi' \mathbin{\#} \xi') = [\varphi', b]$
using *ab* **by** (*metis* (*no-types*, *hide-lams*) *append-Cons append-Nil append-Nil2 append-is-Nil-conv butlast.simps*(*2*) *butlast-append list.sel*(*3*) *tl-append2*)
moreover
{
fix χ :: *'v propo*
have *wf'*: *wf-conn ca* $[a, b]$
using *ab corr* **by** *presburger*
have *all-subformula-st simple-not-symb* (*conn ca* $[a, b]$)
using *ab n* **by** *presburger*
then have *all-subformula-st simple-not-symb* $\chi \lor \chi \notin$ *set* $(\xi \mathbin{@} \varphi' \mathbin{\#} \xi')$
using *wf'* **by** (*metis* (*no-types*) φ' *all-subformula-st-decomp calculation insert-iff list.set*(*2*))
}
then have $\forall \varphi.\ \varphi \in$ *set* $(\xi \mathbin{@} \varphi' \mathbin{\#} \xi') \longrightarrow$ *all-subformula-st simple-not-symb* φ
by (*metis* (*no-types*))

moreover have *simple-not-symb* (*conn ca* $(\xi \mathbin{@} \varphi' \mathbin{\#} \xi')$)
using *ab conn-inj-not*(*1*) *corr wf-conn-list-decomp*(*4*) *wf-conn-no-arity-change not-Cons-self2 self-append-conv2 simple-not-symb.elims*(*3*) **by** (*metis* (*no-types*) *c calculation*(*1*) *wf-conn-binary*)
moreover have *wf-conn ca* $(\xi \mathbin{@} \varphi' \mathbin{\#} \xi')$ **using** *c calculation*(*1*) **by** *auto*
ultimately show *all-subformula-st simple-not-symb* (*conn ca* $(\xi \mathbin{@} \varphi' \mathbin{\#} \xi')$)
by (*metis all-subformula-st-decomp-imp*)
qed
}
moreover {
fix *ca* :: *'v connective* **and** $\xi\ \xi'$ :: *'v propo list* **and** $\varphi\ \varphi'$ :: *'v propo*
have *propo-rew-step* (*push-conn-inside c c'*) $\varphi\ \varphi' \Longrightarrow$ *wf-conn ca* $(\xi \mathbin{@} \varphi \mathbin{\#} \xi')$
$\Longrightarrow$ *simple-not-symb* (*conn ca* $(\xi \mathbin{@} \varphi \mathbin{\#} \xi')$) $\Longrightarrow$ *simple-not-symb* φ'
$\Longrightarrow$ *simple-not-symb* (*conn ca* $(\xi \mathbin{@} \varphi' \mathbin{\#} \xi')$)
by (*metis append-self-conv2 conn.simps*(*4*) *conn-inj-not*(*1*) *simple-not-symb.elims*(*3*) *simple-not-symb.simps*(*1*) *simple-propo-rew-step-push-conn-inside-inv wf-conn-no-arity-change-helper wf-conn-list-decomp*(*4*) *wf-conn-no-arity-change*)
}
ultimately show *simple-not* ψ
using *full-propo-rew-step-inv-stay'*[*of push-conn-inside c c' simple-not-symb*] *assms*
unfolding *no-T-F-except-top-level-def simple-not-def full-unfold* **by** *metis*
next
{
fix $\varphi\ \psi$ :: *'v propo*
have *H*: *propo-rew-step* (*push-conn-inside c c'*) $\varphi\ \psi \Longrightarrow$ *no-T-F-except-top-level* φ
$\Longrightarrow$ *no-T-F-except-top-level* ψ
proof −
assume *rel*: *propo-rew-step* (*push-conn-inside c c'*) $\varphi\ \psi$
and *no-T-F-except-top-level* φ
then have *no-T-F* $\varphi \lor \varphi = FF \lor \varphi = FT$
by (*metis no-T-F-symb-except-toplevel-all-subformula-st-no-T-F-symb*)
moreover {
assume $\varphi = FF \lor \varphi = FT$
then have *False* **using** *rel propo-rew-step-push-conn-inside* **by** *blast*

then have *no-T-F-except-top-level* ψ **by** *blast*
}
moreover {
assume *no-T-F* $\varphi \wedge \varphi \neq FF \wedge \varphi \neq FT$
then have *no-T-F* ψ **using** *rel push-conn-insidec-in-c'-symb-no-T-F* **by** *blast*
then have *no-T-F-except-top-level* ψ **using** *no-T-F-no-T-F-except-top-level* **by** *blast*
}
ultimately show *no-T-F-except-top-level* ψ **by** *blast*
qed
}
moreover {
fix *ca* :: *'v connective* **and** ξ ξ' :: *'v propo list* **and** φ φ' :: *'v propo*
assume *rel*: *propo-rew-step* (*push-conn-inside c c'*) φ φ'
assume *corr*: *wf-conn ca* (ξ @ φ # ξ')
then have *c*: $ca \neq CT \wedge ca \neq CF$ **by** *auto*
assume *no-T-F*: *no-T-F-symb-except-toplevel* (*conn ca* (ξ @ φ # ξ'))
have *no-T-F-symb-except-toplevel* (*conn ca* (ξ @ φ' # ξ'))
proof
have *c*: $ca \neq CT \wedge ca \neq CF$ **using** *corr* **by** *auto*
have ζ: $\forall \zeta \in$ *set* (ξ @ φ # ξ'). $\zeta \neq FT \wedge \zeta \neq FF$
using *corr no-T-F no-T-F-symb-except-toplevel-if-is-a-true-false* **by** *blast*
then have $\varphi \neq FT \wedge \varphi \neq FF$ **by** *auto*
from *rel this* **have** $\varphi' \neq FT \wedge \varphi' \neq FF$
apply (*induct rule*: *propo-rew-step.induct*)
by (*metis append-is-Nil-conv conn.simps*(*2*) *conn-inj list.distinct*(*1*)
wf-conn-helper-facts(*3*) *wf-conn-list*(*1*) *wf-conn-no-arity-change*
wf-conn-no-arity-change-helper push-conn-inside-not-true-false)+
then have $\forall \zeta \in$ *set* (ξ @ φ' # ξ'). $\zeta \neq FT \wedge \zeta \neq FF$ **using** ζ **by** *auto*
moreover have *wf-conn ca* (ξ @ φ' # ξ')
using *corr wf-conn-no-arity-change* **by** (*metis wf-conn-no-arity-change-helper*)
ultimately show *no-T-F-symb* (*conn ca* (ξ @ φ' # ξ')) **using** *no-T-F-symb.intros c* **by** *metis*
qed
}
ultimately show *no-T-F-except-top-level* ψ
using *full-propo-rew-step-inv-stay'*[*of push-conn-inside c c' no-T-F-symb-except-toplevel*]
assms **unfolding** *no-T-F-except-top-level-def full-unfold* **by** *metis*

next
{
fix φ ψ :: *'v propo*
have *H*: *push-conn-inside c c'* φ ψ $\Longrightarrow$ *no-equiv* φ $\Longrightarrow$ *no-equiv* ψ
by (*induct* φ ψ *rule*: *push-conn-inside.induct*, *auto*)
}
then show *no-equiv* ψ
using *full-propo-rew-step-inv-stay-conn*[*of push-conn-inside c c' no-equiv-symb*] *assms*
no-equiv-symb-conn-characterization **unfolding** *no-equiv-def* **by** *metis*

next
{
fix φ ψ :: *'v propo*
have *H*: *push-conn-inside c c'* φ ψ $\Longrightarrow$ *no-imp* φ $\Longrightarrow$ *no-imp* ψ
by (*induct* φ ψ *rule*: *push-conn-inside.induct*, *auto*)
}
then show *no-imp* ψ
using *full-propo-rew-step-inv-stay-conn*[*of push-conn-inside c c' no-imp-symb*] *assms*
no-imp-symb-conn-characterization **unfolding** *no-imp-def* **by** *metis*

qed

lemma *push-conn-inside-full-propo-rew-step*:
fixes φ ψ :: *'v propo*
assumes
no-equiv φ **and**
no-imp φ **and**
full (*propo-rew-step* (*push-conn-inside* *c* *c'*)) φ ψ **and**
no-T-F-except-top-level φ **and**
simple-not φ **and**
$c = CAnd \lor c = COr$ **and**
$c' = CAnd \lor c' = COr$
shows *c-in-c'-only* *c* *c'* ψ
using *c-in-c'-symb-rew assms full-propo-rew-step-subformula* **by** *blast*

Only one type of connective in the formula (+ not)

inductive *only-c-inside-symb* :: *'v connective* $\Rightarrow$ *'v propo* $\Rightarrow$ *bool* **for** *c* :: *'v connective* **where**
simple-only-c-inside[*simp*]: *simple* $\varphi \Longrightarrow$ *only-c-inside-symb* *c* φ |
simple-cnot-only-c-inside[*simp*]: *simple* $\varphi \Longrightarrow$ *only-c-inside-symb* *c* (*FNot* φ) |
only-c-inside-into-only-c-inside: *wf-conn* *c* *l* $\Longrightarrow$ *only-c-inside-symb* *c* (*conn* *c* *l*)

lemma *only-c-inside-symb-simp*[*simp*]:
only-c-inside-symb *c* *FF* *only-c-inside-symb* *c* *FT* *only-c-inside-symb* *c* (*FVar* *x*) **by** *auto*

definition *only-c-inside* **where** *only-c-inside* *c* = *all-subformula-st* (*only-c-inside-symb* *c*)

lemma *only-c-inside-symb-decomp*:
only-c-inside-symb *c* $\psi \longleftrightarrow$ (*simple* ψ
$\lor$ ($\exists$ φ'. ψ = *FNot* φ' $\land$ *simple* φ')
$\lor$ ($\exists$ *l*. ψ = *conn* *c* *l* $\land$ *wf-conn* *c* *l*))
by (*auto simp*: *only-c-inside-symb.intros*(*3*)) (*induct rule*: *only-c-inside-symb.induct*, *auto*)

lemma *only-c-inside-symb-decomp-not*[*simp*]:
fixes *c* :: *'v connective*
assumes *c*: $c \neq CNot$
shows *only-c-inside-symb* *c* (*FNot* ψ) $\longleftrightarrow$ *simple* ψ
apply (*auto simp*: *only-c-inside-symb.intros*(*3*))
by (*induct FNot* ψ *rule*: *only-c-inside-symb.induct*, *auto simp*: *wf-conn-list*(*8*) *c*)

lemma *only-c-inside-decomp-not*[*simp*]:
assumes *c*: $c \neq CNot$
shows *only-c-inside* *c* (*FNot* ψ) $\longleftrightarrow$ *simple* ψ
by (*metis* (*no-types*, *hide-lams*) *all-subformula-st-def all-subformula-st-test-symb-true-phi c only-c-inside-def only-c-inside-symb-decomp-not simple-only-c-inside subformula-conn-decomp-simple*)

lemma *only-c-inside-decomp*:
only-c-inside *c* $\varphi \longleftrightarrow$
($\forall \psi$. $\psi \preceq \varphi \longrightarrow$ (*simple* ψ $\lor$ ($\exists$ φ'. ψ = *FNot* φ' $\land$ *simple* φ')
$\lor$ ($\exists$ *l*. ψ = *conn* *c* *l* $\land$ *wf-conn* *c* *l*)))
unfolding *only-c-inside-def* **by** (*auto simp*: *all-subformula-st-def only-c-inside-symb-decomp*)

lemma *only-c-inside-c-c′-false*:
  fixes $c\ c'$ :: *′v connective* **and** l :: *′v propo list* **and** φ :: *′v propo*
  assumes *cc′*: $c \neq c'$ **and** *c*: $c = CAnd \vee c = COr$ **and** *c′*: $c' = CAnd \vee c' = COr$
  and *only*: *only-c-inside* $c\ \varphi$ **and** *incl*: *conn* $c'\ l \preceq \varphi$ **and** *wf*: *wf-conn* $c'\ l$
  shows *False*
proof −
  let $?\psi$ = *conn* $c'\ l$
  have *simple* $?\psi \vee (\exists\ \varphi'.\ ?\psi = FNot\ \varphi' \wedge simple\ \varphi') \vee (\exists l.\ ?\psi = conn\ c\ l \wedge wf\text{-}conn\ c\ l)$
    using *only-c-inside-decomp only incl* **by** *blast*
  moreover have $\neg$ *simple* $?\psi$
    using *wf simple-decomp* **by** (*metis* c' *connective.distinct*(*19*) *connective.distinct*(*7*,*9*,*21*,*29*,*31*)
      wf-conn-list(*1*−*3*))
  moreover
    {
      fix φ'
      have $?\psi \neq FNot\ \varphi'$ **using** c' *conn-inj-not*(*1*) *wf* **by** *blast*
    }
  ultimately obtain l :: *′v propo list* **where** $?\psi = conn\ c\ l \wedge wf\text{-}conn\ c\ l$ **by** *metis*
  then have $c = c'$ **using** *conn-inj wf* **by** *metis*
  then show *False* **using** *cc′* **by** *auto*
qed

lemma *only-c-inside-implies-c-in-c′-symb*:
  assumes δ: $c \neq c'$ **and** *c*: $c = CAnd \vee c = COr$ **and** *c′*: $c' = CAnd \vee c' = COr$
  shows *only-c-inside* $c\ \varphi \Longrightarrow$ *c-in-c′-symb* $c\ c'\ \varphi$
  apply (*rule ccontr*)
  apply (*cases rule*: *not-c-in-c′-symb.cases*, *auto*)
  by (*metis* δ *c c′ connective.distinct*(*37*,*39*) *list.distinct*(*1*) *only-c-inside-c-c′-false*
    subformula-in-binary-conn(*1*,*2*) *wf-conn.simps*)+

lemma *c-in-c′-symb-decomp-level1*:
  fixes l :: *′v propo list* **and** $c\ c'\ ca$ :: *′v connective*
  shows *wf-conn* $ca\ l \Longrightarrow ca \neq c \Longrightarrow$ *c-in-c′-symb* $c\ c'$ (*conn* $ca\ l$)
proof −
  have *not-c-in-c′-symb* $c\ c'$ (*conn* $ca\ l$) $\Longrightarrow$ *wf-conn* $ca\ l \Longrightarrow ca = c$
    by (*induct conn ca l rule*: *not-c-in-c′-symb.induct*, *auto simp*: *conn-inj*)
  then show *wf-conn* $ca\ l \Longrightarrow ca \neq c \Longrightarrow$ *c-in-c′-symb* $c\ c'$ (*conn* $ca\ l$) **by** *blast*
qed

lemma *only-c-inside-implies-c-in-c′-only*:
  assumes δ: $c \neq c'$ **and** *c*: $c = CAnd \vee c = COr$ **and** *c′*: $c' = CAnd \vee c' = COr$
  shows *only-c-inside* $c\ \varphi \Longrightarrow$ *c-in-c′-only* $c\ c'\ \varphi$
  unfolding *c-in-c′-only-def all-subformula-st-def*
  using *only-c-inside-implies-c-in-c′-symb*
    by (*metis all-subformula-st-def assms*(*1*) *c c′ only-c-inside-def subformula-trans*)

lemma *c-in-c′-symb-c-implies-only-c-inside*:
  assumes δ: $c = CAnd \vee c = COr$ $c' = CAnd \vee c' = COr$ $c \neq c'$ **and** *wf*: *wf-conn* c $[\varphi, \psi]$
  and *inv*: *no-equiv* (*conn* $c\ l$) *no-imp* (*conn* $c\ l$) *simple-not* (*conn* $c\ l$)
  shows *wf-conn* $c\ l \Longrightarrow$ *c-in-c′-only* $c\ c'$ (*conn* $c\ l$) $\Longrightarrow (\forall \psi \in set\ l.\ only\text{-}c\text{-}inside\ c\ \psi)$
using *inv*
proof (*induct conn c l arbitrary*: *l rule*: *propo-induct-arity*)
  case (*nullary x*)

then show *?case* **by** (*auto simp*: *wf-conn-list assms*)
next
  case (*unary* φ *la*)
  then have $c = CNot \wedge la = [\varphi]$ **by** (*metis* (*no-types*) *wf-conn-list*(*8*))
  then show *?case* **using** *assms*(*2*) *assms*(*1*) **by** *blast*
next
  case (*binary* $\varphi 1$ $\varphi 2$)
  note $IH\varphi 1 = this(1)$ **and** $IH\varphi 2 = this(2)$ **and** $\varphi = this(3)$ **and** *only* = *this*(*5*) **and** *wf* = *this*(*4*)
    and *no-equiv* = *this*(*6*) **and** *no-imp* = *this*(*7*) **and** *simple-not* = *this*(*8*)
  then have *l*: $l = [\varphi 1, \varphi 2]$ **by** (*meson wf-conn-list*(*4*−*7*))
  let $?\varphi = conn\ c\ l$

  obtain *c1 l1 c2 l2* **where** $\varphi 1$: $\varphi 1 = conn\ c1\ l1$ **and** $wf\varphi 1$: *wf-conn c1 l1*
    and $\varphi 2$: $\varphi 2 = conn\ c2\ l2$ **and** $wf\varphi 2$: *wf-conn c2 l2* **using** *exists-c-conn* **by** *metis*
  then have *c-in-only*$\varphi 1$: *c-in-c′-only c c′* (*conn c1 l1*) **and** *c-in-c′-only c c′* (*conn c2 l2*)
    using *only l* **unfolding** *c-in-c′-only-def* **using** *assms*(*1*) **by** *auto*
  have *inc*$\varphi 1$: $\varphi 1 \preceq ?\varphi$ **and** *inc*$\varphi 2$: $\varphi 2 \preceq ?\varphi$
    using $\varphi 1$ $\varphi 2$ φ *local.wf* **by** (*metis conn.simps*(*5*−*8*) *helper-fact subformula-in-binary-conn*(*1*,*2*))+

  have *c1-eq*: $c1 \neq CEq$ **and** *c2-eq*: $c2 \neq CEq$
    unfolding *no-equiv-def* **using** *inc*$\varphi 1$ *inc*$\varphi 2$ **by** (*metis* $\varphi 1$ $\varphi 2$ $wf\varphi 1$ $wf\varphi 2$ *assms*(*1*) *no-equiv*
      no-equiv-eq(*1*) *no-equiv-symb.elims*(*3*) *no-equiv-symb-conn-characterization wf-conn-list*(*4*,*5*)
      no-equiv-def subformula-all-subformula-st)+
  have *c1-imp*: $c1 \neq CImp$ **and** *c2-imp*: $c2 \neq CImp$
    using *no-imp* **by** (*metis* $\varphi 1$ $\varphi 2$ *all-subformula-st-decomp-explicit-imp*(*2*,*3*) *assms*(*1*)
      conn.simps(*5*,*6*) *l no-imp-Imp*(*1*) *no-imp-symb.elims*(*3*) *no-imp-symb-conn-characterization*
      $wf\varphi 1$ $wf\varphi 2$ *all-subformula-st-decomp no-imp-symb-conn-characterization*)+
  have *c1c*: $c1 \neq c'$
    proof
      assume *c1c*: $c1 = c'$
      then obtain $\xi 1$ $\xi 2$ **where** *l1*: $l1 = [\xi 1, \xi 2]$
        by (*metis assms*(*2*) *connective.distinct*(*37*,*39*) *helper-fact* $wf\varphi 1$ *wf-conn.simps*
          wf-conn-list-decomp(*1*−*3*))
      have *c-in-c′-only c c′* (*conn c* [*conn c′ l1*, $\varphi 2$]) **using** *c1c l only* $\varphi 1$ **by** *auto*
      moreover have *not-c-in-c′-symb c c′* (*conn c* [*conn c′ l1*, $\varphi 2$])
        using *l1* $\varphi 1$ *c1c l local.wf not-c-in-c′-symb-l* $wf\varphi 1$ **by** *blast*
      ultimately show *False* **using** $\varphi 1$ *c1c l l1 local.wf not-c-in-c′-simp*(*4*) $wf\varphi 1$ **by** *blast*
   qed
  then have $(\varphi 1 = conn\ c\ l1 \wedge wf\text{-}conn\ c\ l1) \vee (\exists \psi 1.\ \varphi 1 = FNot\ \psi 1) \vee simple\ \varphi 1$
    by (*metis* $\varphi 1$ *assms*(*1*−*3*) *c1-eq c1-imp simple.elims*(*3*) $wf\varphi 1$ *wf-conn-list*(*4*) *wf-conn-list*(*5*−*7*))
  moreover {
    assume $\varphi 1 = conn\ c\ l1 \wedge wf\text{-}conn\ c\ l1$
    then have *only-c-inside c* $\varphi 1$
      by (*metis* $IH\varphi 1$ $\varphi 1$ *all-subformula-st-decomp-imp inc*$\varphi 1$ *no-equiv no-equiv-def no-imp no-imp-def*
        c-in-only$\varphi 1$ *only-c-inside-def only-c-inside-into-only-c-inside simple-not simple-not-def*
        subformula-all-subformula-st)
  }
  moreover {
    assume $\exists \psi 1.\ \varphi 1 = FNot\ \psi 1$
    then obtain $\psi 1$ **where** $\varphi 1 = FNot\ \psi 1$ **by** *metis*
    then have *only-c-inside c* $\varphi 1$
      by (*metis all-subformula-st-def assms*(*1*) *connective.distinct*(*37*,*39*) *inc*$\varphi 1$
        only-c-inside-decomp-not simple-not simple-not-def simple-not-symb.simps(*1*))
  }
  moreover {
    assume *simple* $\varphi 1$

```
    then have only-c-inside c φ1
      by (metis all-subformula-st-decomp-explicit(3) assms(1) connective.distinct(37,39)
        only-c-inside-decomp-not only-c-inside-def)
  }
  ultimately have only-c-insideφ1: only-c-inside c φ1 by metis

  have c-in-onlyφ2: c-in-c′-only c c′ (conn c2 l2)
    using only l φ2 wfφ2 assms unfolding c-in-c′-only-def by auto
  have c2c: c2 ≠ c′
    proof
      assume c2c: c2 = c′
      then obtain ξ1 ξ2 where l2: l2 = [ξ1, ξ2]
       by (metis assms(2) wfφ2 wf-conn.simps connective.distinct(7,9,19,21,29,31,37,39))
      then have c-in-c′-symb c c′ (conn c [φ1, conn c′ l2])
        using c2c l only φ2 all-subformula-st-test-symb-true-phi unfolding c-in-c′-only-def by auto
      moreover have not-c-in-c′-symb c c′ (conn c [φ1, conn c′ l2])
        using assms(1) c2c l2 not-c-in-c′-symb-r wfφ2 wf-conn-helper-facts(5,6) by metis
      ultimately show False by auto
    qed
  then have (φ2 = conn c l2 ∧ wf-conn c l2) ∨ (∃ψ2. φ2 = FNot ψ2) ∨ simple φ2
    using c2-eq by (metis φ2 assms(1−3) c2-eq c2-imp simple.elims(3) wfφ2 wf-conn-list(4−7))
  moreover {
    assume φ2 = conn c l2 ∧ wf-conn c l2
    then have only-c-inside c φ2
      by (metis IHφ2 φ2 all-subformula-st-decomp incφ2 no-equiv no-equiv-def no-imp no-imp-def
        c-in-onlyφ2 only-c-inside-def only-c-inside-into-only-c-inside simple-not simple-not-def
        subformula-all-subformula-st)
  }
  moreover {
    assume ∃ψ2. φ2 = FNot ψ2
    then obtain ψ2 where φ2 = FNot ψ2 by metis
    then have only-c-inside c φ2
      by (metis all-subformula-st-def assms(1−3) connective.distinct(38,40) incφ2
        only-c-inside-decomp-not simple-not simple-not-def simple-not-symb.simps(1))
  }
  moreover {
    assume simple φ2
    then have only-c-inside c φ2
      by (metis all-subformula-st-decomp-explicit(3) assms(1) connective.distinct(37,39)
        only-c-inside-decomp-not only-c-inside-def)
  }
  ultimately have only-c-insideφ2: only-c-inside c φ2 by metis
  show ?case using l only-c-insideφ1 only-c-insideφ2 by auto
qed
```

Push Conjunction

```
definition pushConj where pushConj = push-conn-inside CAnd COr

lemma pushConj-consistent: preserve-models pushConj
  unfolding pushConj-def by (simp add: push-conn-inside-consistent)

definition and-in-or-symb where and-in-or-symb = c-in-c′-symb CAnd COr

definition and-in-or-only where
and-in-or-only = all-subformula-st (c-in-c′-symb CAnd COr)
```

lemma *pushConj-inv*:
fixes φ ψ :: *'v propo*
assumes *full* (*propo-rew-step pushConj*) φ ψ
and *no-equiv* φ **and** *no-imp* φ **and** *no-T-F-except-top-level* φ **and** *simple-not* φ
shows *no-equiv* ψ **and** *no-imp* ψ **and** *no-T-F-except-top-level* ψ **and** *simple-not* ψ
using *push-conn-inside-inv assms* **unfolding** *pushConj-def* **by** *metis+*

lemma *pushConj-full-propo-rew-step*:
fixes φ ψ :: *'v propo*
assumes
no-equiv φ **and**
no-imp φ **and**
full (*propo-rew-step pushConj*) φ ψ **and**
no-T-F-except-top-level φ **and**
simple-not φ
shows *and-in-or-only* ψ
using *assms push-conn-inside-full-propo-rew-step*
unfolding *pushConj-def and-in-or-only-def c-in-c'-only-def* **by** (*metis* (*no-types*))

Push Disjunction

definition *pushDisj* **where** *pushDisj* = *push-conn-inside COr CAnd*

lemma *pushDisj-consistent*: *preserve-models pushDisj*
unfolding *pushDisj-def* **by** (*simp add*: *push-conn-inside-consistent*)

definition *or-in-and-symb* **where** *or-in-and-symb* = *c-in-c'-symb COr CAnd*

definition *or-in-and-only* **where**
or-in-and-only = *all-subformula-st* (*c-in-c'-symb COr CAnd*)

lemma *not-or-in-and-only-or-and*[*simp*]:
~*or-in-and-only* (*FOr* (*FAnd* $\psi 1$ $\psi 2$) φ')
unfolding *or-in-and-only-def*
by (*metis all-subformula-st-test-symb-true-phi conn.simps*(*5*−*6*) *not-c-in-c'-symb-l*
wf-conn-helper-facts(*5*) *wf-conn-helper-facts*(*6*))

lemma *pushDisj-inv*:
fixes φ ψ :: *'v propo*
assumes *full* (*propo-rew-step pushDisj*) φ ψ
and *no-equiv* φ **and** *no-imp* φ **and** *no-T-F-except-top-level* φ **and** *simple-not* φ
shows *no-equiv* ψ **and** *no-imp* ψ **and** *no-T-F-except-top-level* ψ **and** *simple-not* ψ
using *push-conn-inside-inv assms* **unfolding** *pushDisj-def* **by** *metis+*

lemma *pushDisj-full-propo-rew-step*:
fixes φ ψ :: *'v propo*
assumes
no-equiv φ **and**
no-imp φ **and**
full (*propo-rew-step pushDisj*) φ ψ **and**
no-T-F-except-top-level φ **and**
simple-not φ
shows *or-in-and-only* ψ

using *assms push-conn-inside-full-propo-rew-step*
unfolding *pushDisj-def or-in-and-only-def c-in-c′-only-def* **by** (*metis* (*no-types*))

1.6 The Full Transformations

1.6.1 Abstract Definition

The normal form is a super group of groups

inductive *grouped-by* :: *′a connective* ⇒ *′a propo* ⇒ *bool* **for** *c* **where**
simple-is-grouped[*simp*]: *simple* $\varphi \Longrightarrow$ *grouped-by c* φ |
simple-not-is-grouped[*simp*]: *simple* $\varphi \Longrightarrow$ *grouped-by c* (*FNot* φ) |
connected-is-group[*simp*]: *grouped-by c* $\varphi \Longrightarrow$ *grouped-by c* $\psi \Longrightarrow$ *wf-conn c* [φ, ψ]
$\Longrightarrow$ *grouped-by c* (*conn c* [φ, ψ])

lemma *simple-clause*[*simp*]:
grouped-by c FT
grouped-by c FF
grouped-by c (*FVar x*)
grouped-by c (*FNot FT*)
grouped-by c (*FNot FF*)
grouped-by c (*FNot* (*FVar x*))
by *simp+*

lemma *only-c-inside-symb-c-eq-c′*:
only-c-inside-symb c (*conn c′* [$\varphi 1$, $\varphi 2$]) $\Longrightarrow$ $c' = CAnd \lor c' = COr$ $\Longrightarrow$ *wf-conn c′* [$\varphi 1$, $\varphi 2$]
$\Longrightarrow$ $c' = c$
by (*induct conn c′* [$\varphi 1$, $\varphi 2$] *rule*: *only-c-inside-symb.induct*, *auto simp*: *conn-inj*)

lemma *only-c-inside-c-eq-c′*:
only-c-inside c (*conn c′* [$\varphi 1$, $\varphi 2$]) $\Longrightarrow$ $c' = CAnd \lor c' = COr$ $\Longrightarrow$ *wf-conn c′* [$\varphi 1$, $\varphi 2$] $\Longrightarrow$ $c = c'$
unfolding *only-c-inside-def all-subformula-st-def* **using** *only-c-inside-symb-c-eq-c′ subformula-refl*
by *blast*

lemma *only-c-inside-imp-grouped-by*:
assumes *c*: $c \neq CNot$ **and** *c′*: $c' = CAnd \lor c' = COr$
shows *only-c-inside c* $\varphi \Longrightarrow$ *grouped-by c* φ (**is** *?O* $\varphi \Longrightarrow$ *?G* φ)
proof (*induct* φ *rule*: *propo-induct-arity*)
case (*nullary* φ *x*)
then show *?G* φ **by** *auto*
next
case (*unary* ψ)
then show *?G* (*FNot* ψ) **by** (*auto simp*: *c*)
next
case (*binary* φ $\varphi 1$ $\varphi 2$)
note *IH*$\varphi 1$ = *this*(*1*) **and** *IH*$\varphi 2$ = *this*(*2*) **and** φ = *this*(*3*) **and** *only* = *this*(*4*)
have φ*-conn*: φ = *conn c* [$\varphi 1$, $\varphi 2$] **and** *wf*: *wf-conn c* [$\varphi 1$, $\varphi 2$]
proof −
obtain c'' l'' **where** φ*-c″*: φ = *conn* c'' l'' **and** *wf*: *wf-conn* c'' l''
using *exists-c-conn* **by** *metis*
then have l'': $l'' = [\varphi 1, \varphi 2]$ **using** φ **by** (*metis wf-conn-list*(*4*−*7*))
have *only-c-inside-symb c* (*conn* c'' [$\varphi 1$, $\varphi 2$])
using *only all-subformula-st-test-symb-true-phi*
unfolding *only-c-inside-def* φ*-c″* l'' **by** *metis*
then have $c = c''$

```
      by (metis φ φ-c'' conn-inj conn-inj-not(2) l'' list.distinct(1) list.inject wf
        only-c-inside-symb.cases simple.simps(5−8))
    then show φ = conn c [φ1, φ2] and wf-conn c [φ1, φ2] using φ-c'' wf l'' by auto
  qed
  have grouped-by c φ1 using wf IHφ1 IHφ2 φ-conn only φ unfolding only-c-inside-def by auto
  moreover have grouped-by c φ2
    using wf φ IHφ1 IHφ2 φ-conn only unfolding only-c-inside-def by auto
  ultimately show ?G φ using φ-conn connected-is-group local.wf by blast
qed


lemma grouped-by-false:
  grouped-by c (conn c' [φ, ψ]) ⟹ c ≠ c' ⟹ wf-conn c' [φ, ψ] ⟹ False
  apply (induct conn c' [φ, ψ] rule: grouped-by.induct)
  apply (auto simp: simple-decomp wf-conn-list, auto simp: conn-inj)
  by (metis list.distinct(1) list.sel(3) wf-conn-list(8))+
```

Then the CNF form is a conjunction of clauses: every clause is in CNF form and two formulas in CNF form can be related by an and.

```
inductive super-grouped-by:: 'a connective ⇒ 'a connective ⇒ 'a propo ⇒ bool for c c' where
grouped-is-super-grouped[simp]: grouped-by c φ ⟹ super-grouped-by c c' φ |
connected-is-super-group: super-grouped-by c c' φ ⟹ super-grouped-by c c' ψ ⟹ wf-conn c [φ, ψ]
  ⟹ super-grouped-by c c' (conn c' [φ, ψ])

lemma simple-cnf[simp]:
  super-grouped-by c c' FT
  super-grouped-by c c' FF
  super-grouped-by c c' (FVar x)
  super-grouped-by c c' (FNot FT)
  super-grouped-by c c' (FNot FF)
  super-grouped-by c c' (FNot (FVar x))
  by auto

lemma c-in-c'-only-super-grouped-by:
  assumes c: c = CAnd ∨ c = COr and c': c' = CAnd ∨ c' = COr and cc': c ≠ c'
  shows no-equiv φ ⟹ no-imp φ ⟹ simple-not φ ⟹ c-in-c'-only c c' φ
    ⟹ super-grouped-by c c' φ
    (is ?NE φ ⟹ ?NI φ ⟹ ?SN φ ⟹ ?C φ ⟹ ?S φ)
proof (induct φ rule: propo-induct-arity)
  case (nullary φ x)
  then show ?S φ by auto
next
  case (unary φ)
  then have simple-not-symb (FNot φ)
    using all-subformula-st-test-symb-true-phi unfolding simple-not-def by blast
  then have φ = FT ∨ φ = FF ∨ (∃ x. φ = FVar x) by (cases φ, auto)
  then show ?S (FNot φ) by auto
next
  case (binary φ φ1 φ2)
  note IHφ1 = this(1) and IHφ2 = this(2) and no-equiv = this(4) and no-imp = this(5)
    and simpleN = this(6) and c-in-c'-only = this(7) and φ' = this(3)
  {
    assume φ = FImp φ1 φ2 ∨ φ = FEq φ1 φ2
    then have False using no-equiv no-imp by auto
    then have ?S φ by auto
  }
```

moreover {
  assume *φ*: *φ* = *conn c′* [*φ1*, *φ2*] ∧ *wf-conn c′* [*φ1*, *φ2*]
  have *c-in-c′-only*: *c-in-c′-only c c′ φ1* ∧ *c-in-c′-only c c′ φ2* ∧ *c-in-c′-symb c c′ φ*
    using *c-in-c′-only φ′* **unfolding** *c-in-c′-only-def* **by** *auto*
  have *super-grouped-by c c′ φ1* **using** *φ c′ no-equiv no-imp simpleN IHφ1 c-in-c′-only* **by** *auto*
  moreover have *super-grouped-by c c′ φ2*
    using *φ c′ no-equiv no-imp simpleN IHφ2 c-in-c′-only* **by** *auto*
  ultimately have *?S φ*
    using *super-grouped-by.intros*(*2*) *φ* **by** (*metis c wf-conn-helper-facts*(*5*,*6*))
}
moreover {
  assume *φ*: *φ* = *conn c* [*φ1*, *φ2*] ∧ *wf-conn c* [*φ1*, *φ2*]
  then have *only-c-inside c φ1* ∧ *only-c-inside c φ2*
    using *c-in-c′-symb-c-implies-only-c-inside c c′ c-in-c′-only list.set-intros*(*1*)
      wf-conn-helper-facts(*5*,*6*) *no-equiv no-imp simpleN last-ConsL last-ConsR last-in-set*
      list.distinct(*1*) **by** (*metis* (*no-types*, *hide-lams*) *cc′*)
  then have *only-c-inside c* (*conn c* [*φ1*, *φ2*])
    unfolding *only-c-inside-def* **using** *φ*
    by (*simp add*: *only-c-inside-into-only-c-inside all-subformula-st-decomp*)
  then have *grouped-by c φ* **using** *φ only-c-inside-imp-grouped-by c* **by** *blast*
  then have *?S φ* **using** *super-grouped-by.intros*(*1*) **by** *metis*
}
ultimately show *?S φ* **by** (*metis φ′ c c′ cc′ conn.simps*(*5*,*6*) *wf-conn-helper-facts*(*5*,*6*))
qed

1.6.2 Conjunctive Normal Form

Definition

definition *is-conj-with-TF* **where** *is-conj-with-TF* == *super-grouped-by COr CAnd*

lemma *or-in-and-only-conjunction-in-disj*:
  shows *no-equiv φ* ⟹ *no-imp φ* ⟹ *simple-not φ* ⟹ *or-in-and-only φ* ⟹ *is-conj-with-TF φ*
  using *c-in-c′-only-super-grouped-by*
  unfolding *is-conj-with-TF-def or-in-and-only-def c-in-c′-only-def*
  by (*simp add*: *c-in-c′-only-def c-in-c′-only-super-grouped-by*)

definition *is-cnf* **where**
is-cnf φ ≡ *is-conj-with-TF φ* ∧ *no-T-F-except-top-level φ*

Full CNF transformation

The full1 CNF transformation consists simply in chaining all the transformation defined before.

definition *cnf-rew* **where** *cnf-rew* =
  (*full* (*propo-rew-step elim-equiv*)) *OO*
  (*full* (*propo-rew-step elim-imp*)) *OO*
  (*full* (*propo-rew-step elimTB*)) *OO*
  (*full* (*propo-rew-step pushNeg*)) *OO*
  (*full* (*propo-rew-step pushDisj*))

lemma *cnf-rew-equivalent*: *preserve-models cnf-rew*
  by (*simp add*: *cnf-rew-def elimEquv-lifted-consistant elim-imp-lifted-consistant elimTB-consistent*
    preserve-models-OO pushDisj-consistent pushNeg-lifted-consistant)

lemma *cnf-rew-is-cnf*: *cnf-rew φ φ′* ⟹ *is-cnf φ′*

apply (*unfold cnf-rew-def OO-def*)
  apply *auto*
proof −
  fix φ φEq φImp φTB φNeg $\varphi Disj$ :: *'v propo*
  assume *Eq*: *full* (*propo-rew-step elim-equiv*) φ φEq
  then have *no-equiv*: *no-equiv* φEq **using** *no-equiv-full-propo-rew-step-elim-equiv* **by** *blast*

  assume *Imp*: *full* (*propo-rew-step elim-imp*) φEq φImp
  then have *no-imp*: *no-imp* φImp **using** *no-imp-full-propo-rew-step-elim-imp* **by** *blast*
  have *no-imp-inv*: *no-equiv* φImp **using** *no-equiv Imp elim-imp-inv* **by** *blast*

  assume *TB*: *full* (*propo-rew-step elimTB*) φImp φTB
  then have *noTB*: *no-T-F-except-top-level* φTB
    using *no-imp-inv no-imp elimTB-full-propo-rew-step* **by** *blast*
  have *noTB-inv*: *no-equiv* φTB *no-imp* φTB **using** *elimTB-inv TB no-imp no-imp-inv* **by** *blast+*

  assume *Neg*: *full* (*propo-rew-step pushNeg*) φTB φNeg
  then have *noNeg*: *simple-not* φNeg
    using *noTB-inv noTB pushNeg-full-propo-rew-step* **by** *blast*
  have *noNeg-inv*: *no-equiv* φNeg *no-imp* φNeg *no-T-F-except-top-level* φNeg
    using *pushNeg-inv Neg noTB noTB-inv* **by** *blast+*

  assume *Disj*: *full* (*propo-rew-step pushDisj*) φNeg $\varphi Disj$
  then have *no-Disj*: *or-in-and-only* $\varphi Disj$
    using *noNeg-inv noNeg pushDisj-full-propo-rew-step* **by** *blast*
  have *noDisj-inv*: *no-equiv* $\varphi Disj$ *no-imp* $\varphi Disj$ *no-T-F-except-top-level* $\varphi Disj$
    simple-not $\varphi Disj$
  using *pushDisj-inv Disj noNeg noNeg-inv* **by** *blast+*

  moreover have *is-conj-with-TF* $\varphi Disj$
    using *or-in-and-only-conjunction-in-disj noDisj-inv no-Disj* **by** *blast*
  ultimately show *is-cnf* $\varphi Disj$ **unfolding** *is-cnf-def* **by** *blast*
qed

1.6.3 Disjunctive Normal Form

Definition

definition *is-disj-with-TF* **where** *is-disj-with-TF* $\equiv$ *super-grouped-by CAnd COr*

lemma *and-in-or-only-conjunction-in-disj*:
  shows *no-equiv* $\varphi \Longrightarrow$ *no-imp* $\varphi \Longrightarrow$ *simple-not* $\varphi \Longrightarrow$ *and-in-or-only* $\varphi \Longrightarrow$ *is-disj-with-TF* φ
  using *c-in-c'-only-super-grouped-by*
  unfolding *is-disj-with-TF-def and-in-or-only-def c-in-c'-only-def*
  by (*simp add*: *c-in-c'-only-def c-in-c'-only-super-grouped-by*)

definition *is-dnf* :: *'a propo* $\Rightarrow$ *bool* **where**
is-dnf $\varphi \longleftrightarrow$ *is-disj-with-TF* $\varphi \wedge$ *no-T-F-except-top-level* φ

Full DNF transform

The full1 DNF transformation consists simply in chaining all the transformation defined before.

definition *dnf-rew* **where** *dnf-rew* $\equiv$
  (*full* (*propo-rew-step elim-equiv*)) *OO*
  (*full* (*propo-rew-step elim-imp*)) *OO*
  (*full* (*propo-rew-step elimTB*)) *OO*

(*full* (*propo-rew-step pushNeg*)) *OO*
(*full* (*propo-rew-step pushConj*))

lemma *dnf-rew-consistent*: *preserve-models dnf-rew*
by (*simp add*: *dnf-rew-def elimEquv-lifted-consistant elim-imp-lifted-consistant elimTB-consistent preserve-models-OO pushConj-consistent pushNeg-lifted-consistant*)

theorem *dnf-transformation-correction*:
$dnf\text{-}rew\ \varphi\ \varphi' \Longrightarrow is\text{-}dnf\ \varphi'$
apply (*unfold dnf-rew-def OO-def*)
by (*meson and-in-or-only-conjunction-in-disj elimTB-full-propo-rew-step elimTB-inv*(*1*,*2*) *elim-imp-inv is-dnf-def no-equiv-full-propo-rew-step-elim-equiv no-imp-full-propo-rew-step-elim-imp pushConj-full-propo-rew-step pushConj-inv*(*1*−*4*) *pushNeg-full-propo-rew-step pushNeg-inv*(*1*−*3*))

1.7 More aggressive simplifications: Removing true and false at the beginning

1.7.1 Transformation

We should remove *FT* and *FF* at the beginning and not in the middle of the algorithm. To do this, we have to use more rules (one for each connective):

inductive *elimTBFull* **where**
ElimTBFull1[*simp*]: *elimTBFull* (*FAnd* φ *FT*) φ |
ElimTBFull1'[*simp*]: *elimTBFull* (*FAnd FT* φ) φ |

ElimTBFull2[*simp*]: *elimTBFull* (*FAnd* φ *FF*) *FF* |
ElimTBFull2'[*simp*]: *elimTBFull* (*FAnd FF* φ) *FF* |

ElimTBFull3[*simp*]: *elimTBFull* (*FOr* φ *FT*) *FT* |
ElimTBFull3'[*simp*]: *elimTBFull* (*FOr FT* φ) *FT* |

ElimTBFull4[*simp*]: *elimTBFull* (*FOr* φ *FF*) φ |
ElimTBFull4'[*simp*]: *elimTBFull* (*FOr FF* φ) φ |

ElimTBFull5[*simp*]: *elimTBFull* (*FNot FT*) *FF* |
ElimTBFull5'[*simp*]: *elimTBFull* (*FNot FF*) *FT* |

ElimTBFull6-l[*simp*]: *elimTBFull* (*FImp FT* φ) φ |
ElimTBFull6-l'[*simp*]: *elimTBFull* (*FImp FF* φ) *FT* |
ElimTBFull6-r[*simp*]: *elimTBFull* (*FImp* φ *FT*) *FT* |
ElimTBFull6-r'[*simp*]: *elimTBFull* (*FImp* φ *FF*) (*FNot* φ) |

ElimTBFull7-l[*simp*]: *elimTBFull* (*FEq FT* φ) φ |
ElimTBFull7-l'[*simp*]: *elimTBFull* (*FEq FF* φ) (*FNot* φ) |
ElimTBFull7-r[*simp*]: *elimTBFull* (*FEq* φ *FT*) φ |
ElimTBFull7-r'[*simp*]: *elimTBFull* (*FEq* φ *FF*) (*FNot* φ)

The transformation is still consistent.

lemma *elimTBFull-consistent*: *preserve-models elimTBFull*
proof −
{
fix φ ψ:: *'b propo*
have *elimTBFull* $\varphi\ \psi \Longrightarrow \forall A.\ A \models \varphi \longleftrightarrow A \models \psi$

by (*induct-tac rule*: *elimTBFull.inducts*, *auto*)
}
then show *?thesis* **using** *preserve-models-def* **by** *auto*
qed

Contrary to the theorem *no-T-F-symb-except-toplevel-step-exists*, we do not need the assumption *no-equiv* φ and *no-imp* φ, since our transformation is more general.

lemma *no-T-F-symb-except-toplevel-step-exists'*:
fixes φ :: *'v propo*
shows $\psi \preceq \varphi \Longrightarrow \neg$ *no-T-F-symb-except-toplevel* $\psi \Longrightarrow \exists \psi'$. *elimTBFull* ψ ψ'
proof (*induct* ψ *rule*: *propo-induct-arity*)
case (*nullary* φ')
then have *False* **using** *no-T-F-symb-except-toplevel-true no-T-F-symb-except-toplevel-false* **by** *auto*
then show *Ex* (*elimTBFull* φ') **by** *blast*
next
case (*unary* ψ)
then have $\psi = FF \lor \psi = FT$ **using** *no-T-F-symb-except-toplevel-not-decom* **by** *blast*
then show *Ex* (*elimTBFull* (*FNot* ψ)) **using** *ElimTBFull5 ElimTBFull5'* **by** *blast*
next
case (*binary* φ' $\psi 1$ $\psi 2$)
then have $\psi 1 = FT \lor \psi 2 = FT \lor \psi 1 = FF \lor \psi 2 = FF$
by (*metis binary-connectives-def conn.simps*(*5−8*) *insertI1 insert-commute no-T-F-symb-except-toplevel-bin-decom binary.hyps*(*3*))
then show *Ex* (*elimTBFull* φ') **using** *elimTBFull.intros binary.hyps*(*3*) **by** *blast*
qed

The same applies here. We do not need the assumption, but the deep link between $\neg$ *no-T-F-except-top-level* φ and the existence of a rewriting step, still exists.

lemma *no-T-F-except-top-level-rew'*:
fixes φ :: *'v propo*
assumes *noTB*: $\neg$ *no-T-F-except-top-level* φ
shows $\exists \psi\ \psi'$. $\psi \preceq \varphi \land$ *elimTBFull* ψ ψ'
proof −
have *test-symb-false-nullary*:
$\forall x$. *no-T-F-symb-except-toplevel* (*FF*:: *'v propo*) $\land$ *no-T-F-symb-except-toplevel FT* $\land$ *no-T-F-symb-except-toplevel* (*FVar* (*x*:: *'v*))
by *auto*
moreover {
fix *c*:: *'v connective* **and** *l* :: *'v propo list* **and** ψ :: *'v propo*
have *H*: *elimTBFull* (*conn c l*) $\psi \Longrightarrow \neg$*no-T-F-symb-except-toplevel* (*conn c l*)
by (*cases conn c l rule*: *elimTBFull.cases*) *auto*
}
ultimately show *?thesis*
using *no-test-symb-step-exists*[*of no-T-F-symb-except-toplevel* φ *elimTBFull*] *noTB no-T-F-symb-except-toplevel-step-exists'* **unfolding** *no-T-F-except-top-level-def* **by** *metis*
qed

lemma *elimTBFull-full-propo-rew-step*:
fixes φ ψ :: *'v propo*
assumes *full* (*propo-rew-step elimTBFull*) φ ψ
shows *no-T-F-except-top-level* ψ
using *full-propo-rew-step-subformula no-T-F-except-top-level-rew' assms* **by** *fastforce*

1.7.2 More invariants

As the aim is to use the transformation as the first transformation, we have to show some more invariants for *elim-equiv* and *elim-imp*. For the other transformation, we have already proven it.

```
lemma propo-rew-step-ElimEquiv-no-T-F: propo-rew-step elim-equiv φ ψ ⟹ no-T-F φ ⟹ no-T-F ψ
proof (induct rule: propo-rew-step.induct)
  fix φ' :: 'v propo and ψ' :: 'v propo
  assume a1: no-T-F φ'
  assume a2: elim-equiv φ' ψ'
  have ∀x0 x1. (¬ elim-equiv (x1 :: 'v propo) x0 ∨ (∃v2 v3 v4 v5 v6 v7. x1 = FEq v2 v3
    ∧ x0 = FAnd (FImp v4 v5) (FImp v6 v7) ∧ v2 = v4 ∧ v4 = v7 ∧ v3 = v5 ∧ v3 = v6))
 = (¬ elim-equiv x1 x0 ∨ (∃v2 v3 v4 v5 v6 v7. x1 = FEq v2 v3
    ∧ x0 = FAnd (FImp v4 v5) (FImp v6 v7) ∧ v2 = v4 ∧ v4 = v7 ∧ v3 = v5 ∧ v3 = v6))
    by meson
  then have ∀p pa. ¬ elim-equiv (p :: 'v propo) pa ∨ (∃pb pc pd pe pf pg. p = FEq pb pc
    ∧ pa = FAnd (FImp pd pe) (FImp pf pg) ∧ pb = pd ∧ pd = pg ∧ pc = pe ∧ pc = pf)
    using elim-equiv.cases by force
  then show no-T-F ψ' using a1 a2 by fastforce
next
  fix φ φ' :: 'v propo and ξ ξ' :: 'v propo list and c :: 'v connective
  assume rel: propo-rew-step elim-equiv φ φ'
  and IH: no-T-F φ ⟹ no-T-F φ'
  and corr: wf-conn c (ξ @ φ # ξ')
  and no-T-F: no-T-F (conn c (ξ @ φ # ξ'))
  {
    assume c: c = CNot
    then have empty: ξ = [] ξ' = [] using corr by auto
    then have no-T-F φ using no-T-F c no-T-F-decomp-not by auto
    then have no-T-F (conn c (ξ @ φ' # ξ')) using c empty no-T-F-comp-not IH by auto
  }
  moreover {
    assume c: c ∈ binary-connectives
    obtain a b where ab: ξ @ φ # ξ' = [a, b]
      using corr c list-length2-decomp wf-conn-bin-list-length by metis
    then have φ: φ = a ∨ φ = b
      by (metis append.simps(1) append-is-Nil-conv list.distinct(1) list.sel(3) nth-Cons-0
        tl-append2)
    have ζ: ∀ζ∈ set (ξ @ φ # ξ'). no-T-F ζ
      using no-T-F unfolding no-T-F-def using corr all-subformula-st-decomp by blast

    then have φ': no-T-F φ' using ab IH φ by auto
    have l': ξ @ φ' # ξ' = [φ', b] ∨ ξ @ φ' # ξ' = [a, φ']
      by (metis (no-types, hide-lams) ab append-Cons append-Nil append-Nil2 butlast.simps(2)
        butlast-append list.distinct(1) list.sel(3))
    then have ∀ζ ∈ set (ξ @ φ' # ξ'). no-T-F ζ using ζ φ' ab by fastforce
    moreover
      have ∀ζ ∈ set (ξ @ φ # ξ'). ζ ≠ FT ∧ ζ ≠ FF
        using ζ corr no-T-F no-T-F-except-top-level-false no-T-F-no-T-F-except-top-level by blast
      then have no-T-F-symb (conn c (ξ @ φ' # ξ'))
        by (metis φ' l' ab all-subformula-st-test-symb-true-phi c list.distinct(1)
          list.set-intros(1,2) no-T-F-symb-except-toplevel-bin-decom
          no-T-F-symb-except-toplevel-no-T-F-symb no-T-F-symb-false(1,2) no-T-F-def wf-conn-binary
          wf-conn-list(1,2))
    ultimately have no-T-F (conn c (ξ @ φ' # ξ'))
```

by (*metis* l' *all-subformula-st-decomp-imp c no-T-F-def wf-conn-binary*)
}
moreover {
fix x
assume $c = CVar\ x \lor c = CF \lor c = CT$
then have *False* **using** *corr* **by** *auto*
then have *no-T-F* (*conn c* (ξ @ φ' # ξ')) **by** *auto*
}
ultimately show *no-T-F* (*conn c* (ξ @ φ' # ξ')) **using** *corr wf-conn.cases* **by** *metis*
qed

lemma *elim-equiv-inv'*:
fixes φ ψ :: *'v propo*
assumes *full* (*propo-rew-step elim-equiv*) φ ψ **and** *no-T-F-except-top-level* φ
shows *no-T-F-except-top-level* ψ
proof −
{
fix φ ψ :: *'v propo*
have *propo-rew-step elim-equiv* φ ψ $\Longrightarrow$ *no-T-F-except-top-level* φ
$\Longrightarrow$ *no-T-F-except-top-level* ψ
proof −
assume *rel*: *propo-rew-step elim-equiv* φ ψ
and *no*: *no-T-F-except-top-level* φ
{
assume $\varphi = FT \lor \varphi = FF$
from *rel this* **have** *False*
apply (*induct rule*: *propo-rew-step.induct*, *auto simp*: *wf-conn-list*(*1*,*2*))
using *elim-equiv.simps* **by** *blast+*
then have *no-T-F-except-top-level* ψ **by** *blast*
}
moreover {
assume $\varphi \neq FT \land \varphi \neq FF$
then have *no-T-F* φ
by (*metis no no-T-F-symb-except-toplevel-all-subformula-st-no-T-F-symb*)
then have *no-T-F* ψ **using** *propo-rew-step-ElimEquiv-no-T-F rel* **by** *blast*
then have *no-T-F-except-top-level* ψ **by** (*simp add*: *no-T-F-no-T-F-except-top-level*)
}
ultimately show *no-T-F-except-top-level* ψ **by** *metis*
qed
}
moreover {
fix *c* :: *'v connective* **and** ξ ξ' :: *'v propo list* **and** ζ ζ' :: *'v propo*
assume *rel*: *propo-rew-step elim-equiv* ζ ζ'
and *incl*: $\zeta \preceq \varphi$
and *corr*: *wf-conn c* (ξ @ ζ # ξ')
and *no-T-F*: *no-T-F-symb-except-toplevel* (*conn c* (ξ @ ζ # ξ'))
and *n*: *no-T-F-symb-except-toplevel* ζ'
have *no-T-F-symb-except-toplevel* (*conn c* (ξ @ ζ' # ξ'))
proof
have *p*: *no-T-F-symb* (*conn c* (ξ @ ζ # ξ'))
using *corr wf-conn-list*(*1*) *wf-conn-list*(*2*) *no-T-F-symb-except-toplevel-no-T-F-symb no-T-F*
by *blast*
have *l*: $\forall \varphi \in set$ (ξ @ ζ # ξ'). $\varphi \neq FT \land \varphi \neq FF$
using *corr wf-conn-no-T-F-symb-iff p* **by** *blast*
from *rel incl* **have** $\zeta' \neq FT \land \zeta' \neq FF$
apply (*induction* ζ ζ' *rule*: *propo-rew-step.induct*)

apply (*cases rule*: *elim-equiv.cases*, *auto simp*: *elim-equiv.simps*)
by (*metis append-is-Nil-conv list.distinct wf-conn-list*(*1*,*2*) *wf-conn-no-arity-change wf-conn-no-arity-change-helper*)+
then have $\forall\varphi \in$ *set* $(\xi$ @ ζ' # $\xi')$. $\varphi \neq FT \wedge \varphi \neq FF$ **using** *l* **by** *auto*
moreover have $c \neq CT \wedge c \neq CF$ **using** *corr* **by** *auto*
ultimately show *no-T-F-symb* (*conn c* $(\xi$ @ ζ' # $\xi'))$
by (*metis corr wf-conn-no-arity-change wf-conn-no-arity-change-helper no-T-F-symb-comp*)
qed
}
ultimately show *no-T-F-except-top-level* ψ
using *full-propo-rew-step-inv-stay-with-inc*[*of elim-equiv no-T-F-symb-except-toplevel* φ]
assms subformula-refl **unfolding** *no-T-F-except-top-level-def* **by** *metis*
qed

lemma *propo-rew-step-ElimImp-no-T-F*: *propo-rew-step elim-imp* φ $\psi \Longrightarrow$ *no-T-F* $\varphi \Longrightarrow$ *no-T-F* ψ
proof (*induct rule*: *propo-rew-step.induct*)
case (*global-rel* φ' ψ')
then show *no-T-F* ψ'
using *elim-imp.cases no-T-F-comp-not no-T-F-decomp*(*1*,*2*)
by (*metis no-T-F-comp-expanded-explicit*(*2*))
next
case (*propo-rew-one-step-lift* φ φ' *c* ξ ξ')
note *rel* = *this*(*1*) **and** *IH* = *this*(*2*) **and** *corr* = *this*(*3*) **and** *no-T-F* = *this*(*4*)
{
assume *c*: *c* = *CNot*
then have *empty*: ξ = [] ξ' = [] **using** *corr* **by** *auto*
then have *no-T-F* φ **using** *no-T-F c no-T-F-decomp-not* **by** *auto*
then have *no-T-F* (*conn c* $(\xi$ @ φ' # $\xi'))$ **using** *c empty no-T-F-comp-not IH* **by** *auto*
}
moreover {
assume *c*: $c \in$ *binary-connectives*
then obtain *a b* **where** *ab*: ξ @ φ # ξ' = [*a*, *b*]
using *corr list-length2-decomp wf-conn-bin-list-length* **by** *metis*
then have φ: $\varphi = a \vee \varphi = b$
by (*metis append-self-conv2 wf-conn-list-decomp*(*4*) *wf-conn-unary list.discI list.sel*(*3*)
nth-Cons-0 tl-append2)
have ζ: $\forall\zeta \in$ *set* $(\xi$ @ φ # $\xi')$. *no-T-F* ζ **using** *ab c propo-rew-one-step-lift.prems* **by** *auto*

then have φ': *no-T-F* φ'
using *ab IH* φ *corr no-T-F no-T-F-def all-subformula-st-decomp-explicit* **by** *auto*
have χ: ξ @ φ' # ξ' = [φ', *b*] $\vee$ ξ @ φ' # ξ' = [*a*, φ']
by (*metis* (*no-types*, *hide-lams*) *ab append-Cons append-Nil append-Nil2 butlast.simps*(*2*)
butlast-append list.distinct(*1*) *list.sel*(*3*))
then have $\forall\zeta\in$ *set* $(\xi$ @ φ' # $\xi')$. *no-T-F* ζ **using** ζ φ' *ab* **by** *fastforce*
moreover
have *no-T-F* (*last* $(\xi$ @ φ' # $\xi'))$ **by** (*simp add*: *calculation*)
then have *no-T-F-symb* (*conn c* $(\xi$ @ φ' # $\xi'))$
by (*metis* χ φ' ζ *ab all-subformula-st-test-symb-true-phi c last.simps list.distinct*(*1*)
list.set-intros(*1*) *no-T-F-bin-decomp no-T-F-def*)
ultimately have *no-T-F* (*conn c* $(\xi$ @ φ' # $\xi'))$ **using** *c* χ **by** *fastforce*
}
moreover {
fix *x*
assume *c* = *CVar x* $\vee$ *c* = *CF* $\vee$ *c* = *CT*
then have *False* **using** *corr* **by** *auto*

then have *no-T-F* (*conn c* (ξ @ φ' # ξ')) **by** *auto*
}
ultimately show *no-T-F* (*conn c* (ξ @ φ' # ξ')) **using** *corr wf-conn.cases* **by** *blast*
qed

lemma *elim-imp-inv'*:
fixes φ ψ :: *'v propo*
assumes *full* (*propo-rew-step elim-imp*) φ ψ **and** *no-T-F-except-top-level* φ
shows*no-T-F-except-top-level* ψ
proof −
{
{
fix φ ψ :: *'v propo*
have *H*: *elim-imp* φ ψ $\Longrightarrow$ *no-T-F-except-top-level* φ $\Longrightarrow$ *no-T-F-except-top-level* ψ
by (*induct* φ ψ *rule*: *elim-imp.induct*, *auto*)
} note *H* = *this*
fix φ ψ :: *'v propo*
have *propo-rew-step elim-imp* φ ψ $\Longrightarrow$ *no-T-F-except-top-level* φ $\Longrightarrow$ *no-T-F-except-top-level* ψ
proof −
assume *rel*: *propo-rew-step elim-imp* φ ψ
and *no*: *no-T-F-except-top-level* φ
{
assume $\varphi = FT \vee \varphi = FF$
from *rel this* **have** *False*
apply (*induct rule*: *propo-rew-step.induct*)
by (*cases rule*: *elim-imp.cases*, *auto simp*: *wf-conn-list*(*1*,*2*))
then have *no-T-F-except-top-level* ψ **by** *blast*
}
moreover {
assume $\varphi \neq FT \wedge \varphi \neq FF$
then have *no-T-F* φ
by (*metis no no-T-F-symb-except-toplevel-all-subformula-st-no-T-F-symb*)
then have *no-T-F* ψ
using *rel propo-rew-step-ElimImp-no-T-F* **by** *blast*
then have *no-T-F-except-top-level* ψ **by** (*simp add*: *no-T-F-no-T-F-except-top-level*)
}
ultimately show *no-T-F-except-top-level* ψ **by** *metis*
qed
}
moreover {
fix *c* :: *'v connective* **and** ξ ξ' :: *'v propo list* **and** ζ ζ' :: *'v propo*
assume *rel*: *propo-rew-step elim-imp* ζ ζ'
and *incl*: $\zeta \preceq \varphi$
and *corr*: *wf-conn c* (ξ @ ζ # ξ')
and *no-T-F*: *no-T-F-symb-except-toplevel* (*conn c* (ξ @ ζ # ξ'))
and *n*: *no-T-F-symb-except-toplevel* ζ'
have *no-T-F-symb-except-toplevel* (*conn c* (ξ @ ζ' # ξ'))
proof
have *p*: *no-T-F-symb* (*conn c* (ξ @ ζ # ξ'))
by (*simp add*: *corr no-T-F no-T-F-symb-except-toplevel-no-T-F-symb wf-conn-list*(*1*,*2*))

have *l*: $\forall\varphi\in$*set* (ξ @ ζ # ξ'). $\varphi \neq FT \wedge \varphi \neq FF$
using *corr wf-conn-no-T-F-symb-iff p* **by** *blast*
from *rel incl* **have** $\zeta' \neq FT \wedge \zeta' \neq FF$
apply (*induction* ζ ζ' *rule*: *propo-rew-step.induct*)

apply (*cases rule*: *elim-imp.cases*, *auto*)
using *wf-conn-list*(*1*,*2*) *wf-conn-no-arity-change wf-conn-no-arity-change-helper*
by (*metis append-is-Nil-conv list.distinct*(*1*))+
then have $\forall \varphi \in set\ (\xi\ @\ \zeta'\ \#\ \xi').\ \varphi \neq FT \wedge \varphi \neq FF$ **using** *l* **by** *auto*
moreover have $c \neq CT \wedge c \neq CF$ **using** *corr* **by** *auto*
ultimately show *no-T-F-symb* (*conn c* (ξ @ ζ' # ξ'))
using *corr wf-conn-no-arity-change no-T-F-symb-comp*
by (*metis wf-conn-no-arity-change-helper*)
qed
}
ultimately show *no-T-F-except-top-level* ψ
using *full-propo-rew-step-inv-stay-with-inc*[*of elim-imp no-T-F-symb-except-toplevel* φ]
assms subformula-refl **unfolding** *no-T-F-except-top-level-def* **by** *metis*
qed

1.7.3 The new CNF and DNF transformation

The transformation is the same as before, but the order is not the same.

definition *dnf-rew'* :: *'a propo* ⇒ *'a propo* ⇒ *bool* **where**
dnf-rew' =
(*full* (*propo-rew-step elimTBFull*)) *OO*
(*full* (*propo-rew-step elim-equiv*)) *OO*
(*full* (*propo-rew-step elim-imp*)) *OO*
(*full* (*propo-rew-step pushNeg*)) *OO*
(*full* (*propo-rew-step pushConj*))

lemma *dnf-rew'-consistent*: *preserve-models dnf-rew'*
by (*simp add*: *dnf-rew'-def elimEquv-lifted-consistant elim-imp-lifted-consistant*
elimTBFull-consistent preserve-models-OO pushConj-consistent pushNeg-lifted-consistant)

theorem *cnf-transformation-correction*:
dnf-rew' φ φ' ⟹ *is-dnf* φ'
unfolding *dnf-rew'-def OO-def*
by (*meson and-in-or-only-conjunction-in-disj elimTBFull-full-propo-rew-step elim-equiv-inv'*
elim-imp-inv elim-imp-inv' is-dnf-def no-equiv-full-propo-rew-step-elim-equiv
no-imp-full-propo-rew-step-elim-imp pushConj-full-propo-rew-step pushConj-inv(*1*−*4*)
pushNeg-full-propo-rew-step pushNeg-inv(*1*−*3*))

Given all the lemmas before the CNF transformation is easy to prove:

definition *cnf-rew'* :: *'a propo* ⇒ *'a propo* ⇒ *bool* **where**
cnf-rew' =
(*full* (*propo-rew-step elimTBFull*)) *OO*
(*full* (*propo-rew-step elim-equiv*)) *OO*
(*full* (*propo-rew-step elim-imp*)) *OO*
(*full* (*propo-rew-step pushNeg*)) *OO*
(*full* (*propo-rew-step pushDisj*))

lemma *cnf-rew'-consistent*: *preserve-models cnf-rew'*
by (*simp add*: *cnf-rew'-def elimEquv-lifted-consistant elim-imp-lifted-consistant*
elimTBFull-consistent preserve-models-OO pushDisj-consistent pushNeg-lifted-consistant)

theorem *cnf'-transformation-correction*:
cnf-rew' φ φ' ⟹ *is-cnf* φ'
unfolding *cnf-rew'-def OO-def*
by (*meson elimTBFull-full-propo-rew-step elim-equiv-inv' elim-imp-inv elim-imp-inv' is-cnf-def*

no-equiv-full-propo-rew-step-elim-equiv no-imp-full-propo-rew-step-elim-imp or-in-and-only-conjunction-in-disj pushDisj-full-propo-rew-step pushDisj-inv(*1*−*4*) *pushNeg-full-propo-rew-step pushNeg-inv*(*1*) *pushNeg-inv*(*2*) *pushNeg-inv*(*3*))

end
theory *Prop-Logic-Multiset*
imports *Nested-Multisets-Ordinals.Multiset-More Prop-Normalisation Entailment-Definition.Partial-Herbrand-Interpretation*
begin

1.8 Link with Multiset Version

1.8.1 Transformation to Multiset

fun *mset-of-conj* :: *'a propo* ⇒ *'a literal multiset* **where**
mset-of-conj (*FOr* φ ψ) = *mset-of-conj* φ + *mset-of-conj* ψ |
mset-of-conj (*FVar v*) = {# *Pos v* #} |
mset-of-conj (*FNot* (*FVar v*)) = {# *Neg v* #} |
mset-of-conj FF = {#}

fun *mset-of-formula* :: *'a propo* ⇒ *'a literal multiset set* **where**
mset-of-formula (*FAnd* φ ψ) = *mset-of-formula* φ ∪ *mset-of-formula* ψ |
mset-of-formula (*FOr* φ ψ) = {*mset-of-conj* (*FOr* φ ψ)} |
mset-of-formula (*FVar* ψ) = {*mset-of-conj* (*FVar* ψ)} |
mset-of-formula (*FNot* ψ) = {*mset-of-conj* (*FNot* ψ)} |
mset-of-formula FF = {{#}} |
mset-of-formula FT = {}

1.8.2 Equisatisfiability of the two Versions

lemma *is-conj-with-TF-FNot*:
  is-conj-with-TF (*FNot* φ) ⟷ (∃ *v*. φ = *FVar v* ∨ φ = *FF* ∨ φ = *FT*)
  unfolding *is-conj-with-TF-def* **apply** (*rule iffI*)
  apply (*induction FNot* φ *rule*: *super-grouped-by.induct*)
  apply (*induction FNot* φ *rule*: *grouped-by.induct*)
      apply *simp*
    apply (*cases* φ; *simp*)
  apply *auto*
  done

lemma *grouped-by-COr-FNot*:
  grouped-by COr (*FNot* φ) ⟷ (∃ *v*. φ = *FVar v* ∨ φ = *FF* ∨ φ = *FT*)
  unfolding *is-conj-with-TF-def* **apply** (*rule iffI*)
  apply (*induction FNot* φ *rule*: *grouped-by.induct*)
      apply *simp*
    apply (*cases* φ; *simp*)
  apply *auto*
  done

lemma
  shows *no-T-F-FF*[*simp*]: ¬*no-T-F FF* **and**
    no-T-F-FT[*simp*]: ¬*no-T-F FT*
  unfolding *no-T-F-def all-subformula-st-def* **by** *auto*

lemma *grouped-by-CAnd-FAnd*:
  grouped-by CAnd (*FAnd* $\varphi 1$ $\varphi 2$) ⟷ *grouped-by CAnd* $\varphi 1$ ∧ *grouped-by CAnd* $\varphi 2$

apply (*rule iffI*)
apply (*induction FAnd φ1 φ2 rule*: *grouped-by.induct*)
using *connected-is-group*[*of CAnd φ1 φ2*] **by** *auto*

lemma *grouped-by-COr-FOr*:
grouped-by COr (*FOr φ1 φ2*) ⟷ *grouped-by COr φ1* ∧ *grouped-by COr φ2*
apply (*rule iffI*)
apply (*induction FOr φ1 φ2 rule*: *grouped-by.induct*)
using *connected-is-group*[*of COr φ1 φ2*] **by** *auto*

lemma *grouped-by-COr-FAnd*[*simp*]: ¬ *grouped-by COr* (*FAnd φ1 φ2*)
apply *clarify*
apply (*induction FAnd φ1 φ2 rule*: *grouped-by.induct*)
apply *auto*
done

lemma *grouped-by-COr-FEq*[*simp*]: ¬ *grouped-by COr* (*FEq φ1 φ2*)
apply *clarify*
apply (*induction FEq φ1 φ2 rule*: *grouped-by.induct*)
apply *auto*
done

lemma [*simp*]: ¬*grouped-by COr* (*FImp φ ψ*)
apply *clarify*
by (*induction FImp φ ψ rule*: *grouped-by.induct*) *simp-all*

lemma [*simp*]: ¬ *is-conj-with-TF* (*FImp φ ψ*)
unfolding *is-conj-with-TF-def* **apply** *clarify*
by (*induction FImp φ ψ rule*: *super-grouped-by.induct*) *simp-all*

lemma [*simp*]: ¬ *is-conj-with-TF* (*FEq φ ψ*)
unfolding *is-conj-with-TF-def* **apply** *clarify*
by (*induction FEq φ ψ rule*: *super-grouped-by.induct*) *simp-all*

lemma *is-conj-with-TF-Fand*:
is-conj-with-TF (*FAnd φ1 φ2*) ⟹ *is-conj-with-TF φ1* ∧ *is-conj-with-TF φ2*
unfolding *is-conj-with-TF-def*
apply (*induction FAnd φ1 φ2 rule*: *super-grouped-by.induct*)
apply (*auto simp*: *grouped-by-CAnd-FAnd intro*: *grouped-is-super-grouped*)[]
apply *auto*[]
done

lemma *is-conj-with-TF-FOr*:
is-conj-with-TF (*FOr φ1 φ2*) ⟹ *grouped-by COr φ1* ∧ *grouped-by COr φ2*
unfolding *is-conj-with-TF-def*
apply (*induction FOr φ1 φ2 rule*: *super-grouped-by.induct*)
apply (*auto simp*: *grouped-by-COr-FOr*)[]
apply *auto*[]
done

lemma *grouped-by-COr-mset-of-formula*:
grouped-by COr φ ⟹ *mset-of-formula φ* = (*if φ* = *FT then* {} *else* {*mset-of-conj φ*})
by (*induction φ*) (*auto simp add*: *grouped-by-COr-FNot*)

When a formula is in CNF form, then there is equisatisfiability between the multiset version and the CNF form. Remark that the definition for the entailment are slightly different: ($\models$) uses a function assigning *True* or *False*, while ($\models s$) uses a set where being in the list means entailment of a literal.

theorem *cnf-eval-true-clss*:
  fixes φ :: *'v propo*
  assumes *is-cnf* φ
  shows *eval A* $\varphi \longleftrightarrow$ *Partial-Herbrand-Interpretation.true-clss* $(\{Pos\ v|v.\ A\ v\} \cup \{Neg\ v|v.\ \neg A\ v\})$
    (*mset-of-formula* φ)
  using *assms*
proof (*induction* φ)
  case *FF*
  then show *?case* **by** *auto*
next
  case *FT*
  then show *?case* **by** *auto*
next
  case (*FVar v*)
  then show *?case* **by** *auto*
next
  case (*FAnd* φ ψ)
  then show *?case*
    unfolding *is-cnf-def* **by** (*auto simp*: *is-conj-with-TF-FNot dest*: *is-conj-with-TF-Fand*
    dest!: *is-conj-with-TF-FOr*)
next
  case (*FOr* φ ψ)
  then have [*simp*]: *mset-of-formula* φ = {*mset-of-conj* φ} *mset-of-formula* ψ = {*mset-of-conj* ψ}
    unfolding *is-cnf-def* **by** (*auto dest*!:*is-conj-with-TF-FOr simp*: *grouped-by-COr-mset-of-formula*
      split: *if-splits*)
  have *is-conj-with-TF* φ *is-conj-with-TF* ψ
    using *FOr*(*3*) **unfolding** *is-cnf-def no-T-F-def*
    by (*metis grouped-is-super-grouped is-conj-with-TF-FOr is-conj-with-TF-def*)+
  then show *?case* **using** *FOr*
    unfolding *is-cnf-def* **by** *simp*
next
  case (*FImp* φ ψ)
  then show *?case*
    unfolding *is-cnf-def* **by** *auto*
next
  case (*FEq* φ ψ)
  then show *?case*
    unfolding *is-cnf-def* **by** *auto*
next
  case (*FNot* φ)
  then show *?case*
    unfolding *is-cnf-def* **by** (*auto simp*: *is-conj-with-TF-FNot*)
qed

function *formula-of-mset* :: *'a clause* $\Rightarrow$ *'a propo* **where**
  ‹*formula-of-mset* φ =
    (*if* φ = {#} *then FF*
     else
       let v = (*SOME v. v* ∈# φ);
           v' = (*if is-pos v then FVar* (*atm-of v*) *else FNot* (*FVar* (*atm-of v*))) *in*
       if remove1-mset v φ = {#} *then v'*
       else FOr v' (*formula-of-mset* (*remove1-mset v* φ)))›

  by *auto*
termination
  apply (*relation* ‹*measure size*›)
   apply (*auto simp*: *size-mset-remove1-mset-le-iff*)
  by (*meson multiset-nonemptyE someI-ex*)

lemma *formula-of-mset-empty*[*simp*]: ‹*formula-of-mset* {#} = *FF*›
  by (*auto simp*: *Let-def*)

lemma *formula-of-mset-empty-iff*[*iff*]: ‹*formula-of-mset* φ = *FF* $\longleftrightarrow$ φ = {#}›
  by (*induction* φ) (*auto simp*: *Let-def*)

declare *formula-of-mset.simps*[*simp del*]

function *formula-of-msets* :: *'a literal multiset set* $\Rightarrow$ *'a propo* **where**
  ‹*formula-of-msets* φs =
     (*if* φs = {} $\vee$ *infinite* φs *then FT*
      else
         let v = (*SOME v. v* $\in$ φs);
             v' = *formula-of-mset v in*
         if φs − {*v*} = {} *then v'*
         else FAnd v' (*formula-of-msets* (φs − {*v*})))›
  by *auto*
termination
  apply (*relation* ‹*measure card*›)
   apply (*auto simp*: *some-in-eq*)
  by (*metis all-not-in-conv card-gt-0-iff diff-less lessI*)

declare *formula-of-msets.simps*[*simp del*]

lemma *remove1-mset-empty-iff*:
  ‹*remove1-mset v* φ = {#} $\longleftrightarrow$ (φ = {#} $\vee$ φ = {#*v*#})›
  using *remove1-mset-eqE* **by** *force*

definition *fun-of-set* **where**
  ‹*fun-of-set A x* = (*if Pos x* $\in$ *A then True else if Neg x* $\in$ *A then False else undefined*)›

lemma *grouped-by-COr-formula-of-mset*: ‹*grouped-by COr* (*formula-of-mset* φ)›
proof (*induction* ‹*size* φ› *arbitrary*: φ)
  case *0*
  then show *?case* **by** (*subst formula-of-mset.simps*) (*auto simp*: *Let-def*)
next
  case (*Suc n*) **note** *IH* = *this*(*1*) **and** *s* = *this*(*2*)
  then have ‹*n* = *size* (*remove1-mset* (*SOME v. v* $\in\#$ φ) φ)› **if** ‹$\varphi \neq$ {#}›
    using *that* **by** (*auto simp*: *size-Diff-singleton-if some-in-eq*)
  then show *?case*
    using *IH*[*of* ‹*remove1-mset* (*SOME v. v* $\in\#$ φ) φ›]
    by(*subst formula-of-mset.simps*) (*auto simp*: *Let-def grouped-by-COr-FOr*)
qed
lemma *no-T-F-formula-of-mset*: ‹*no-T-F* (*formula-of-mset* φ)› **if** ‹*formula-of-mset* $\varphi \neq$ *FF*› **for** φ
  using *that*
proof (*induction* ‹*size* φ› *arbitrary*: φ)
  case *0*
  then show *?case* **by** (*subst formula-of-mset.simps*) (*auto simp*: *Let-def no-T-F-def*
        all-subformula-st-def)
next

case (*Suc n*) **note** *IH* = *this*(*1*) **and** *s* = *this*(*2*) **and** *FF* = *this*(*3*)
then have ‹*n* = *size* (*remove1-mset* (*SOME v. v* ∈# φ) φ)› **if** ‹φ ≠ {#}›
using *that* **by** (*auto simp*: *size-Diff-singleton-if some-in-eq*)
moreover have ‹*no-T-F* (*FVar* (*atm-of* (*SOME v. v* ∈# φ)))›
by (*auto simp*: *no-T-F-def*)
ultimately show *?case*
using *IH*[*of* ‹*remove1-mset* (*SOME v. v* ∈# φ) φ›] *FF*
by(*subst formula-of-mset.simps*) (*auto simp*: *Let-def grouped-by-COr-FOr*)
qed

lemma *mset-of-conj-formula-of-mset*[*simp*]: ‹*mset-of-conj*(*formula-of-mset* φ) = φ› **for** φ
proof (*induction* ‹*size* φ› *arbitrary*: φ)
case *0*
then show *?case* **by** (*subst formula-of-mset.simps*) (*auto simp*: *Let-def no-T-F-def all-subformula-st-def*)
next
case (*Suc n*) **note** *IH* = *this*(*1*) **and** *s* = *this*(*2*)
then have ‹*n* = *size* (*remove1-mset* (*SOME v. v* ∈# φ) φ)› **if** ‹φ ≠ {#}›
using *that* **by** (*auto simp*: *size-Diff-singleton-if some-in-eq*)
moreover have ‹*no-T-F* (*FVar* (*atm-of* (*SOME v. v* ∈# φ)))›
by (*auto simp*: *no-T-F-def*)
ultimately show *?case*
using *IH*[*of* ‹*remove1-mset* (*SOME v. v* ∈# φ) φ›]
by(*subst formula-of-mset.simps*) (*auto simp*: *some-in-eq Let-def grouped-by-COr-FOr remove1-mset-empty-iff*)
qed

lemma *mset-of-formula-formula-of-mset* [*simp*]: ‹*mset-of-formula* (*formula-of-mset* φ) = {φ}› **for** φ
proof (*induction* ‹*size* φ› *arbitrary*: φ)
case *0*
then show *?case* **by** (*subst formula-of-mset.simps*) (*auto simp*: *Let-def no-T-F-def all-subformula-st-def*)
next
case (*Suc n*) **note** *IH* = *this*(*1*) **and** *s* = *this*(*2*)
then have ‹*n* = *size* (*remove1-mset* (*SOME v. v* ∈# φ) φ)› **if** ‹φ ≠ {#}›
using *that* **by** (*auto simp*: *size-Diff-singleton-if some-in-eq*)
moreover have ‹*no-T-F* (*FVar* (*atm-of* (*SOME v. v* ∈# φ)))›
by (*auto simp*: *no-T-F-def*)
ultimately show *?case*
using *IH*[*of* ‹*remove1-mset* (*SOME v. v* ∈# φ) φ›]
by(*subst formula-of-mset.simps*) (*auto simp*: *some-in-eq Let-def grouped-by-COr-FOr remove1-mset-empty-iff*)
qed

lemma *formula-of-mset-is-cnf*: ‹*is-cnf* (*formula-of-mset* φ)›
by (*auto simp*: *is-cnf-def is-conj-with-TF-def grouped-by-COr-formula-of-mset no-T-F-formula-of-mset intro*!: *grouped-is-super-grouped*)

lemma *eval-clss-iff*:
assumes ‹*consistent-interp A*› **and** ‹*total-over-set A UNIV*›
shows ‹*eval* (*fun-of-set A*) (*formula-of-mset* φ) ⟷ *Partial-Herbrand-Interpretation.true-clss A* {φ}›
apply (*subst cnf-eval-true-clss*[*OF formula-of-mset-is-cnf*])
using *assms*
apply (*auto simp add*: *true-cls-def fun-of-set-def consistent-interp-def total-over-set-def*)
apply (*case-tac L*)
by (*fastforce simp add*: *true-cls-def fun-of-set-def consistent-interp-def total-over-set-def*)+

lemma *is-conj-with-TF-Fand-iff*:

is-conj-with-TF (*FAnd* *φ1* *φ2*) ⟷ *is-conj-with-TF* *φ1* ∧ *is-conj-with-TF* *φ2*
unfolding *is-conj-with-TF-def* **by** (*subst super-grouped-by.simps*) *auto*

lemma *is-CNF-Fand*:
‹*is-cnf* (*FAnd* *φ* *ψ*) ⟷ (*is-cnf* *φ* ∧ *no-T-F* *φ*) ∧ *is-cnf* *ψ* ∧ *no-T-F* *ψ*›
by (*auto simp*: *is-cnf-def is-conj-with-TF-Fand-iff*)

lemma *no-T-F-formula-of-mset-iff*: ‹*no-T-F* (*formula-of-mset* *φ*) ⟷ *φ* ≠ {#}›
proof (*induction* ‹*size* *φ*› *arbitrary*: *φ*)
case *0*
then show *?case* **by** (*subst formula-of-mset.simps*) (*auto simp*: *Let-def no-T-F-def*
all-subformula-st-def)
next
case (*Suc n*) **note** *IH* = *this*(*1*) **and** *s* = *this*(*2*)
then have ‹*n* = *size* (*remove1-mset* (*SOME v. v* ∈# *φ*) *φ*)› **if** ‹*φ* ≠ {#}›
using *that* **by** (*auto simp*: *size-Diff-singleton-if some-in-eq*)
moreover have ‹*no-T-F* (*FVar* (*atm-of* (*SOME v. v* ∈# *φ*)))›
by (*auto simp*: *no-T-F-def*)
ultimately show *?case*
using *IH*[*of* ‹*remove1-mset* (*SOME v. v* ∈# *φ*) *φ*›]
by(*subst formula-of-mset.simps*) (*auto simp*: *some-in-eq Let-def grouped-by-COr-FOr remove1-mset-empty-iff*)
qed

lemma *no-T-F-formula-of-msets*:
assumes ‹*finite* *φ*› **and** ‹{#} ∉ *φ*› **and** ‹*φ* ≠ {}›
shows ‹*no-T-F* (*formula-of-msets* (*φ*))›
using *assms* **apply** (*induction* ‹*card* *φ*› *arbitrary*: *φ*)
subgoal by (*subst formula-of-msets.simps*) (*auto simp*: *no-T-F-def all-subformula-st-def*)[]
subgoal
apply (*subst formula-of-msets.simps*)
apply (*auto split*: *simp*: *Let-def formula-of-mset-is-cnf is-CNF-Fand*
no-T-F-formula-of-mset-iff some-in-eq)
apply (*metis* (*mono-tags*, *lifting*) *some-eq-ex*)
done
done

lemma *is-cnf-formula-of-msets*:
assumes ‹*finite* *φ*› **and** ‹{#} ∉ *φ*›
shows ‹*is-cnf* (*formula-of-msets* *φ*)›
using *assms* **apply** (*induction* ‹*card* *φ*› *arbitrary*: *φ*)
subgoal by (*subst formula-of-msets.simps*) (*auto simp*: *is-cnf-def is-conj-with-TF-def*)[]
subgoal
apply (*subst formula-of-msets.simps*)
apply (*auto split*: *simp*: *Let-def formula-of-mset-is-cnf is-CNF-Fand*
no-T-F-formula-of-mset-iff some-in-eq intro: *no-T-F-formula-of-msets*)
apply (*metis* (*mono-tags*, *lifting*) *some-eq-ex*)
done
done

lemma *mset-of-formula-formula-of-msets*:
assumes ‹*finite* *φ*›
shows ‹*mset-of-formula* (*formula-of-msets* *φ*) = *φ*›
using *assms* **apply** (*induction* ‹*card* *φ*› *arbitrary*: *φ*)
subgoal by (*subst formula-of-msets.simps*) (*auto simp*: *is-cnf-def is-conj-with-TF-def*)[]
subgoal
apply (*subst formula-of-msets.simps*)

apply (*auto split*: *simp*: *Let-def formula-of-mset-is-cnf is-CNF-Fand no-T-F-formula-of-mset-iff some-in-eq intro*: *no-T-F-formula-of-msets*)
done
done

lemma
assumes ‹*consistent-interp A*› **and** ‹*total-over-set A UNIV*› **and** ‹*finite* φ› **and** ‹$\{\#\} \notin \varphi$›
shows ‹*eval* (*fun-of-set A*) (*formula-of-msets* φ) $\longleftrightarrow$ *Partial-Herbrand-Interpretation.true-clss A* φ›
apply (*subst cnf-eval-true-clss*[*OF is-cnf-formula-of-msets*[*OF assms*(*3*−*4*)]])
using *assms*(*3*) **unfolding** *mset-of-formula-formula-of-msets*[*OF assms*(*3*)]
by (*induction* φ)
(*use eval-clss-iff* [*OF assms*(*1*,*2*)] **in** ‹*simp-all add*: *cnf-eval-true-clss formula-of-mset-is-cnf*›)

end
theory *Prop-Resolution*
imports *Entailment-Definition.Partial-Herbrand-Interpretation*
Weidenbach-Book-Base.WB-List-More
Weidenbach-Book-Base.Wellfounded-More

begin

Chapter 2

Resolution-based techniques

This chapter contains the formalisation of resolution and superposition.

2.1 Resolution

2.1.1 Simplification Rules

inductive *simplify* :: *'v clause-set* ⇒ *'v clause-set* ⇒ *bool* **for** *N* :: *'v clause set* **where**
tautology-deletion:
  add-mset (*Pos P*) (*add-mset* (*Neg P*) *A*) ∈ *N* ⟹ *simplify N* (*N* − {*add-mset* (*Pos P*) (*add-mset* (*Neg P*) *A*)})|
condensation:
  add-mset L (*add-mset L A*) ∈ *N* ⟹ *simplify N* (*N* − {*add-mset L* (*add-mset L A*)} ∪ {*add-mset L A*}) |
subsumption:
  A ∈ *N* ⟹ *A* ⊂# *B* ⟹ *B* ∈ *N* ⟹ *simplify N* (*N* − {*B*})

lemma *simplify-preserve-models'*:
  fixes *N N'* :: *'v clause-set*
  assumes *simplify N N'*
  and *total-over-m I N*
  shows *I* ⊨*s N'* ⟶ *I* ⊨*s N*
  using *assms*
proof (*induct rule*: *simplify.induct*)
  case (*tautology-deletion P A*)
  then have *I* ⊨ *add-mset* (*Pos P*) (*add-mset* (*Neg P*) *A*)
    by (*fastforce dest*: *mk-disjoint-insert*)
  then show *?case* **by** (*metis Un-Diff-cancel2 true-clss-singleton true-clss-union*)
next
  case (*condensation A P*)
  then show *?case*
    by (*fastforce dest*: *mk-disjoint-insert*)
next
  case (*subsumption A B*)
  have *A* ≠ *B* **using** *subsumption.hyps*(*2*) **by** *auto*
  then have *I* ⊨*s N* − {*B*} ⟹ *I* ⊨ *A* **using** ⟨*A* ∈ *N*⟩ **by** (*simp add*: *true-clss-def*)
  moreover have *I* ⊨ *A* ⟹ *I* ⊨ *B* **using** ⟨*A* <# *B*⟩ **by** *auto*
  ultimately show *?case* **by** (*metis insert-Diff-single true-clss-insert*)
qed

lemma *simplify-preserve-models*:

fixes N N' :: $'v$ *clause-set*
assumes *simplify* N N'
and *total-over-m* I N
shows $I \models s\ N \longrightarrow I \models s\ N'$
using *assms* **apply** (*induct rule*: *simplify.induct*)
using *true-clss-def* **by** *fastforce*+

lemma *simplify-preserve-models''*:
fixes N N' :: $'v$ *clause-set*
assumes *simplify* N N'
and *total-over-m* I N'
shows $I \models s\ N \longrightarrow I \models s\ N'$
using *assms* **apply** (*induct rule*: *simplify.induct*)
using *true-clss-def* **by** *fastforce*+

lemma *simplify-preserve-models-eq*:
fixes N N' :: $'v$ *clause-set*
assumes *simplify* N N'
and *total-over-m* I N
shows $I \models s\ N \longleftrightarrow I \models s\ N'$
using *simplify-preserve-models simplify-preserve-models' assms* **by** *blast*

lemma *simplify-preserves-finite*:
assumes *simplify* ψ ψ'
shows *finite* $\psi \longleftrightarrow$ *finite* ψ'
using *assms* **by** (*induct rule*: *simplify.induct*, *auto simp add*: *remove-def*)

lemma *rtranclp-simplify-preserves-finite*:
assumes *rtranclp simplify* ψ ψ'
shows *finite* $\psi \longleftrightarrow$ *finite* ψ'
using *assms* **by** (*induct rule*: *rtranclp-induct*) (*auto simp add*: *simplify-preserves-finite*)

lemma *simplify-atms-of-ms*:
assumes *simplify* ψ ψ'
shows *atms-of-ms* $\psi' \subseteq$ *atms-of-ms* ψ
using *assms* **unfolding** *atms-of-ms-def*
proof (*induct rule*: *simplify.induct*)
case (*tautology-deletion* A P)
then show *?case* **by** *auto*
next
case (*condensation* P A)
moreover have $A + \{\#P\#\} + \{\#P\#\} \in \psi \Longrightarrow \exists x\in\psi.$ *atm-of* $P \in$ *atm-of* ' *set-mset* x
by (*metis Un-iff atms-of-def atms-of-plus atms-of-singleton insert-iff*)
ultimately show *?case* **by** (*auto simp add*: *atms-of-def*)
next
case (*subsumption* A P)
then show *?case* **by** *auto*
qed

lemma *rtranclp-simplify-atms-of-ms*:
assumes *rtranclp simplify* ψ ψ'
shows *atms-of-ms* $\psi' \subseteq$ *atms-of-ms* ψ
using *assms* **apply** (*induct rule*: *rtranclp-induct*)
apply (*fastforce intro*: *simplify-atms-of-ms*)
using *simplify-atms-of-ms* **by** *blast*

lemma *factoring-imp-simplify*:
  assumes $\{\#L, L\#\} + C \in N$
  shows $\exists N'.$ *simplify* $N\ N'$
proof −
  have *add-mset* L (*add-mset* $L\ C$) $\in N$ **using** *assms* **by** (*simp add*: *add.commute union-lcomm*)
  from *condensation*[*OF this*] **show** *?thesis* **by** *blast*
qed

2.1.2 Unconstrained Resolution

type-synonym *'v uncon-state* = *'v clause-set*

inductive *uncon-res* :: *'v uncon-state* ⇒ *'v uncon-state* ⇒ *bool* **where**
resolution:
  $\{\#Pos\ p\#\} + C \in N \Longrightarrow \{\#Neg\ p\#\} + D \in N \Longrightarrow$ (*add-mset* (*Pos p*) *C*, *add-mset* (*Neg P*) *D*) $\notin$ *already-used*
    $\Longrightarrow$ *uncon-res* N ($N \cup \{C + D\}$) |
factoring: $\{\#L\#\} + \{\#L\#\} + C \in N \Longrightarrow$ *uncon-res* N (*insert* (*add-mset* $L\ C$) N)

lemma *uncon-res-increasing*:
  assumes *uncon-res* $S\ S'$ **and** $\psi \in S$
  shows $\psi \in S'$
  using *assms* **by** (*induct rule*: *uncon-res.induct*) *auto*

lemma *rtranclp-uncon-inference-increasing*:
  assumes *rtranclp uncon-res* $S\ S'$ **and** $\psi \in S$
  shows $\psi \in S'$
  using *assms* **by** (*induct rule*: *rtranclp-induct*) (*auto simp add*: *uncon-res-increasing*)

Subsumption

definition *subsumes* :: *'a literal multiset* ⇒ *'a literal multiset* ⇒ *bool* **where**
subsumes $\chi\ \chi' \longleftrightarrow$
  ($\forall I.$ *total-over-m* $I\ \{\chi'\} \longrightarrow$ *total-over-m* $I\ \{\chi\}$)
  $\wedge$ ($\forall I.$ *total-over-m* $I\ \{\chi\} \longrightarrow I \models \chi \longrightarrow I \models \chi'$)

lemma *subsumes-refl*[*simp*]:
  subsumes $\chi\ \chi$
  unfolding *subsumes-def* **by** *auto*

lemma *subsumes-subsumption*:
  assumes *subsumes* $D\ \chi$
  and $C \subset\# D$ **and** ¬*tautology* χ
  shows *subsumes* $C\ \chi$ **unfolding** *subsumes-def*
  using *assms subsumption-total-over-m subsumption-chained* **unfolding** *subsumes-def*
  by (*blast intro*!: *subset-mset.less-imp-le*)

lemma *subsumes-tautology*:
  assumes *subsumes* (*add-mset* (*Pos P*) (*add-mset* (*Neg P*) *C*)) χ
  shows *tautology* χ
  using *assms* **unfolding** *subsumes-def* **by** (*auto simp add*: *tautology-def*)

2.1.3 Inference Rule

type-synonym *'v state* = *'v clause-set* × (*'v clause* × *'v clause*) *set*

inductive *inference-clause* :: $'v$ *state* $\Rightarrow$ $'v$ *clause* $\times$ ($'v$ *clause* $\times$ $'v$ *clause*) *set* $\Rightarrow$ *bool*
(**infix** $\Rightarrow_{\text{Res}}$ *100*) **where**
resolution:
$\{\#Pos\ p\#\} + C \in N \Longrightarrow \{\#Neg\ p\#\} + D \in N \Longrightarrow (\{\#Pos\ p\#\} + C, \{\#Neg\ p\#\} + D) \notin$ *already-used*
$\Longrightarrow$ *inference-clause* (N, *already-used*) ($C + D$, *already-used* $\cup$ $\{(\{\#Pos\ p\#\} + C, \{\#Neg\ p\#\} + D)\}$) |
factoring: $\{\#L, L\#\} + C \in N \Longrightarrow$ *inference-clause* (N, *already-used*) ($C + \{\#L\#\}$, *already-used*)

inductive *inference* :: $'v$ *state* $\Rightarrow$ $'v$ *state* $\Rightarrow$ *bool* **where**
inference-step: *inference-clause* S (*clause*, *already-used*)
$\Longrightarrow$ *inference* S (*fst* $S \cup \{$*clause*$\}$, *already-used*)

abbreviation *already-used-inv*
:: $'a$ *literal multiset set* $\times$ ($'a$ *literal multiset* $\times$ $'a$ *literal multiset*) *set* $\Rightarrow$ *bool* **where**
already-used-inv state $\equiv$
($\forall (A, B) \in$ *snd state*. $\exists p$. *Pos* $p \in\#$ $A \wedge$ *Neg* $p \in\#$ $B \wedge$
(($\exists \chi \in$ *fst state*. *subsumes* χ $((A - \{\#Pos\ p\#\}) + (B - \{\#Neg\ p\#\}))$)
$\vee$ *tautology* $((A - \{\#Pos\ p\#\}) + (B - \{\#Neg\ p\#\}))$))

lemma *inference-clause-preserves-already-used-inv*:
assumes *inference-clause* S S'
and *already-used-inv* S
shows *already-used-inv* (*fst* $S \cup \{$*fst* $S'\}$, *snd* S')
using *assms* **apply** (*induct rule*: *inference-clause.induct*)
by *fastforce+*

lemma *inference-preserves-already-used-inv*:
assumes *inference* S S'
and *already-used-inv* S
shows *already-used-inv* S'
using *assms*
proof (*induct rule*: *inference.induct*)
case (*inference-step* S *clause already-used*)
then show *?case*
using *inference-clause-preserves-already-used-inv*[*of* S (*clause*, *already-used*)] **by** *simp*
qed

lemma *rtranclp-inference-preserves-already-used-inv*:
assumes *rtranclp inference* S S'
and *already-used-inv* S
shows *already-used-inv* S'
using *assms* **apply** (*induct rule*: *rtranclp-induct*, *simp*)
using *inference-preserves-already-used-inv* **unfolding** *tautology-def* **by** *fast*

lemma *subsumes-condensation*:
assumes *subsumes* $(C + \{\#L\#\} + \{\#L\#\})$ D
shows *subsumes* $(C + \{\#L\#\})$ D
using *assms* **unfolding** *subsumes-def* **by** *simp*

lemma *simplify-preserves-already-used-inv*:
assumes *simplify* N N'
and *already-used-inv* (N, *already-used*)
shows *already-used-inv* (N', *already-used*)

```
  using assms
proof (induct rule: simplify.induct)
  case (condensation C L)
  then show ?case
    using subsumes-condensation by simp fast
next
  {
     fix a:: 'a and A :: 'a set and P
     have (∃x ∈ Set.remove a A. P x) ⟷ (∃x ∈ A. x ≠ a ∧ P x) by auto
  } note ex-member-remove = this
  {
    fix a a0 :: 'v clause and A :: 'v clause-set and y
    assume a ∈ A and a0 ⊂# a
    then have (∃x ∈ A. subsumes x y) ⟷ (subsumes a y ∨ (∃x ∈ A. x ≠ a ∧ subsumes x y))
      by auto
  } note tt2 = this
  case (subsumption A B) note A = this(1) and AB = this(2) and B = this(3) and inv = this(4)
  show ?case
    proof (standard, standard)
      fix x a b
      assume x: x ∈ snd (N − {B}, already-used) and [simp]: x = (a, b)
      obtain p where p: Pos p ∈# a ∧ Neg p ∈# b and
        q: (∃χ∈N. subsumes χ (a − {#Pos p#} + (b − {#Neg p#})))
          ∨ tautology (a − {#Pos p#} + (b − {#Neg p#}))
        using inv x by fastforce
      consider (taut) tautology (a − {#Pos p#} + (b − {#Neg p#})) |
        (χ) χ where χ ∈ N subsumes χ (a − {#Pos p#} + (b − {#Neg p#}))
          ¬tautology (a − {#Pos p#} + (b − {#Neg p#}))
        using q by auto
      then show
        ∃p. Pos p ∈# a ∧ Neg p ∈# b
            ∧ ((∃χ∈fst (N − {B}, already-used). subsumes χ (a − {#Pos p#} + (b − {#Neg p#})))
                ∨ tautology (a − {#Pos p#} + (b − {#Neg p#})))
        proof cases
          case taut
          then show ?thesis using p by auto
        next
          case χ note H = this
          show ?thesis using p A AB B subsumes-subsumption[OF - AB H(3)] H(1,2) by fastforce
        qed
    qed
next
  case (tautology-deletion P C)
  then show ?case
  proof clarify
    fix a b
    assume add-mset (Pos P) (add-mset (Neg P) C) ∈ N
    assume already-used-inv (N, already-used)
    and (a, b) ∈ snd (N − {add-mset (Pos P) (add-mset (Neg P) C)}, already-used)
    then obtain p where
      Pos p ∈# a ∧ Neg p ∈# b ∧
        ((∃χ∈fst (N ∪ {add-mset (Pos P) (add-mset (Neg P) C)}, already-used).
              subsumes χ (a − {#Pos p#} + (b − {#Neg p#})))
          ∨ tautology (a − {#Pos p#} + (b − {#Neg p#})))
      by fastforce
    moreover have tautology (add-mset (Pos P) (add-mset (Neg P) C)) by auto
```

ultimately show
$\exists p.$ *Pos p* $\in\#$ *a* $\wedge$ *Neg p* $\in\#$ *b* $\wedge$
$((\exists \chi\in$*fst* (*N* $-$ {*add-mset* (*Pos P*) (*add-mset* (*Neg P*) *C*)}, *already-used*).
subsumes χ (*remove1-mset* (*Pos p*) *a* + *remove1-mset* (*Neg p*) *b*)) $\vee$
tautology (*remove1-mset* (*Pos p*) *a* + *remove1-mset* (*Neg p*) *b*))
by (*metis* (*no-types*) *Diff-iff Un-insert-right empty-iff fst-conv insertE subsumes-tautology sup-bot.right-neutral*)
qed
qed

lemma
factoring-satisfiable: *I* $\models$ *add-mset L* (*add-mset L C*) $\longleftrightarrow$ *I* $\models$ *add-mset L C* **and**
resolution-satisfiable:
consistent-interp I $\Longrightarrow$ *I* $\models$ *add-mset* (*Pos p*) *C* $\Longrightarrow$ *I* $\models$ *add-mset* (*Neg p*) *D* $\Longrightarrow$ *I* $\models$ *C* + *D* **and**
factoring-same-vars: *atms-of* (*add-mset L* (*add-mset L C*)) = *atms-of* (*add-mset L C*)
unfolding *true-cls-def consistent-interp-def* **by** (*fastforce split*: *if-split-asm*)+

lemma *inference-increasing*:
assumes *inference S S'* **and** $\psi \in$ *fst S*
shows $\psi \in$ *fst S'*
using *assms* **by** (*induct rule*: *inference.induct*, *auto*)

lemma *rtranclp-inference-increasing*:
assumes *rtranclp inference S S'* **and** $\psi \in$ *fst S*
shows $\psi \in$ *fst S'*
using *assms* **by** (*induct rule*: *rtranclp-induct*, *auto simp add*: *inference-increasing*)

lemma *inference-clause-already-used-increasing*:
assumes *inference-clause S S'*
shows *snd S* $\subseteq$ *snd S'*
using *assms* **by** (*induct rule*:*inference-clause.induct*, *auto*)

lemma *inference-already-used-increasing*:
assumes *inference S S'*
shows *snd S* $\subseteq$ *snd S'*
using *assms* **apply** (*induct rule*:*inference.induct*)
using *inference-clause-already-used-increasing* **by** *fastforce*

lemma *inference-clause-preserve-models*:
fixes *N N'* :: *'v clause-set*
assumes *inference-clause T T'*
and *total-over-m I* (*fst T*)
and *consistent*: *consistent-interp I*
shows *I* $\models$*s fst T* $\longleftrightarrow$ *I* $\models$*s fst T* $\cup$ {*fst T'*}
using *assms* **apply** (*induct rule*: *inference-clause.induct*)
unfolding *consistent-interp-def true-clss-def* **by** *auto force*+

lemma *inference-preserve-models*:
fixes *N N'* :: *'v clause-set*
assumes *inference T T'*
and *total-over-m I* (*fst T*)
and *consistent*: *consistent-interp I*
shows *I* $\models$*s fst T* $\longleftrightarrow$ *I* $\models$*s fst T'*

using *assms* **apply** (*induct rule*: *inference.induct*)
using *inference-clause-preserve-models* **by** *fastforce*

lemma *inference-clause-preserves-atms-of-ms*:
assumes *inference-clause* S S'
shows *atms-of-ms* (*fst* (*fst* S $\cup$ {*fst* S'}, *snd* S')) $\subseteq$ *atms-of-ms* (*fst* S)
using *assms* **by** (*induct rule*: *inference-clause.induct*) (*auto dest*!: *atms-of-atms-of-ms-mono*)

lemma *inference-preserves-atms-of-ms*:
fixes N N' :: *'v clause-set*
assumes *inference* T T'
shows *atms-of-ms* (*fst* T') $\subseteq$ *atms-of-ms* (*fst* T)
using *assms* **apply** (*induct rule*: *inference.induct*)
using *inference-clause-preserves-atms-of-ms* **by** *fastforce*

lemma *inference-preserves-total*:
fixes N N' :: *'v clause-set*
assumes *inference* (N, *already-used*) (N', *already-used'*)
shows *total-over-m* I N $\Longrightarrow$ *total-over-m* I N'
using *assms inference-preserves-atms-of-ms* **unfolding** *total-over-m-def total-over-set-def*
by *fastforce*

lemma *rtranclp-inference-preserves-total*:
assumes *rtranclp inference* T T'
shows *total-over-m* I (*fst* T) $\Longrightarrow$ *total-over-m* I (*fst* T')
using *assms* **by** (*induct rule*: *rtranclp-induct*, *auto simp add*: *inference-preserves-total*)

lemma *rtranclp-inference-preserve-models*:
assumes *rtranclp inference* N N'
and *total-over-m* I (*fst* N)
and *consistent*: *consistent-interp* I
shows $I \models s$ *fst* $N \longleftrightarrow I \models s$ *fst* N'
using *assms* **apply** (*induct rule*: *rtranclp-induct*)
apply (*simp add*: *inference-preserve-models*)
using *inference-preserve-models rtranclp-inference-preserves-total* **by** *blast*

lemma *inference-preserves-finite*:
assumes *inference* ψ ψ' **and** *finite* (*fst* ψ)
shows *finite* (*fst* ψ')
using *assms* **by** (*induct rule*: *inference.induct*, *auto simp add*: *simplify-preserves-finite*)

lemma *inference-clause-preserves-finite-snd*:
assumes *inference-clause* ψ ψ' **and** *finite* (*snd* ψ)
shows *finite* (*snd* ψ')
using *assms* **by** (*induct rule*: *inference-clause.induct*, *auto*)

lemma *inference-preserves-finite-snd*:
assumes *inference* ψ ψ' **and** *finite* (*snd* ψ)
shows *finite* (*snd* ψ')
using *assms inference-clause-preserves-finite-snd* **by** (*induct rule*: *inference.induct*, *fastforce*)

lemma *rtranclp-inference-preserves-finite*:

assumes *rtranclp inference* ψ ψ' **and** *finite* (*fst* ψ)
shows *finite* (*fst* ψ')
using *assms* **by** (*induct rule*: *rtranclp-induct*)
(*auto simp add*: *simplify-preserves-finite inference-preserves-finite*)

lemma *consistent-interp-insert*:
assumes *consistent-interp I*
and *atm-of P* $\notin$ *atm-of* ' *I*
shows *consistent-interp* (*insert P I*)
proof –
have *P*: *insert P I* = *I* $\cup$ {*P*} **by** *auto*
show *?thesis* **unfolding** *P*
apply (*rule consistent-interp-disjoint*)
using *assms* **by** (*auto simp*: *image-iff*)
qed

lemma *simplify-clause-preserves-sat*:
assumes *simp*: *simplify* ψ ψ'
and *satisfiable* ψ'
shows *satisfiable* ψ
using *assms*
proof *induction*
case (*tautology-deletion P A*) **note** *AP* = *this*(*1*) **and** *sat* = *this*(*2*)
let *?A'* = *add-mset* (*Pos P*) (*add-mset* (*Neg P*) *A*)
let *?*ψ' = ψ – {*?A'*}
obtain *I* **where**
I: *I* $\models$*s ?*ψ' **and**
cons: *consistent-interp I* **and**
tot: *total-over-m I ?*ψ'
using *sat* **unfolding** *satisfiable-def* **by** *auto*
{ assume *Pos P* $\in$ *I* $\vee$ *Neg P* $\in$ *I*
then have *I* $\models$ *?A'* **by** *auto*
then have *I* $\models$*s* ψ **using** *I* **by** (*metis insert-Diff tautology-deletion.hyps true-clss-insert*)
then have *?case* **using** *cons tot* **by** *auto*
}
moreover {
assume *Pos*: *Pos P* $\notin$ *I* **and** *Neg*: *Neg P* $\notin$ *I*
then have *consistent-interp* (*I* $\cup$ {*Pos P*}) **using** *cons* **by** *simp*
moreover have *I'A*: *I* $\cup$ {*Pos P*} $\models$ *?A'* **by** *auto*
have {*Pos P*} $\cup$ *I* $\models$*s* ψ – {*?A'*}
using ‹*I* $\models$*s* ψ – {*?A'*}› *true-clss-union-increase'* **by** *blast*
then have *I* $\cup$ {*Pos P*} $\models$*s* ψ
by (*metis* (*no-types*) *Un-empty-right Un-insert-left Un-insert-right I'A insert-Diff sup-bot.left-neutral tautology-deletion.hyps true-clss-insert*)
ultimately have *?case* **using** *satisfiable-carac'* **by** *blast*
}
ultimately show *?case* **by** *blast*
next
case (*condensation L A*) **note** *AL* = *this*(*1*) **and** *sat* = *this*(*2*)
let *?A'* = *add-mset L A*
let *?A* = *add-mset L* (*add-mset L A*)
have *f3*: *simplify* ψ (ψ – {*?A*} $\cup$ {*?A'*})
using *AL simplify.condensation* **by** *blast*
obtain *LL* :: *'a literal set* **where**
f4: *LL* $\models$*s* ψ – {*?A*} $\cup$ {*?A'*}
$\wedge$ *consistent-interp LL*

$\wedge$ *total-over-m LL* $(\psi - \{?A\} \cup \{?A'\})$
using *sat* **by** (*meson satisfiable-def*)
have *f5*: *insert* $(A + \{\#L\#\} + \{\#L\#\})$ $(\psi - \{A + \{\#L\#\} + \{\#L\#\}\}) = \psi$
using *AL* **by** *fastforce*
have *atms-of* $(?A')$ = *atms-of* $(?A)$
by *simp*
then show *?case*
using *f5 f4 f3* **by** (*metis Un-insert-right add-mset-add-single atms-of-ms-insert satisfiable-carac simplify-preserve-models' sup-bot.right-neutral total-over-m-def*)
next
case (*subsumption A B*) **note** A = *this*(*1*) **and** *AB* = *this*(*2*) **and** B = *this*(*3*) **and** *sat* = *this*(*4*)
let $?\psi' = \psi - \{B\}$
obtain I **where** I: $I \models s\ ?\psi'$ **and** *cons*: *consistent-interp I* **and** *tot*: *total-over-m I* $?\psi'$
using *sat* **unfolding** *satisfiable-def* **by** *auto*
have $I \models A$ **using** A I **by** (*metis AB Diff-iff subset-mset.less-irrefl singletonD true-clss-def*)
then have $I \models B$ **using** *AB subset-mset.less-imp-le true-cls-mono-leD* **by** *blast*
then have $I \models s\ \psi$ **using** I **by** (*metis insert-Diff-single true-clss-insert*)
then show *?case* **using** *cons satisfiable-carac'* **by** *blast*
qed

lemma *simplify-preserves-unsat*:
assumes *inference* ψ ψ'
shows *satisfiable* (*fst* ψ') $\longrightarrow$ *satisfiable* (*fst* ψ)
using *assms* **apply** (*induct rule*: *inference.induct*)
using *satisfiable-decreasing* **by** (*metis fst-conv*)+

lemma *inference-preserves-unsat*:
assumes *inference*** S S'
shows *satisfiable* (*fst* S') $\longrightarrow$ *satisfiable* (*fst* S)
using *assms* **apply** (*induct rule*: *rtranclp-induct*)
apply *simp-all*
using *simplify-preserves-unsat* **by** *blast*

datatype *'v sem-tree* = *Node 'v 'v sem-tree 'v sem-tree* | *Leaf*

fun *sem-tree-size* :: *'v sem-tree* $\Rightarrow$ *nat* **where**
sem-tree-size Leaf = *0* |
sem-tree-size (*Node - ag ad*) = *1* + *sem-tree-size ag* + *sem-tree-size ad*

lemma *sem-tree-size*[*case-names bigger*]:
$(\bigwedge xs$:: *'v sem-tree*. $(\bigwedge ys$:: *'v sem-tree*. *sem-tree-size ys* < *sem-tree-size xs* $\Longrightarrow$ *P ys*) $\Longrightarrow$ *P xs*)
$\Longrightarrow$ *P xs*
by (*fact Nat.measure-induct-rule*)

fun *partial-interps* :: *'v sem-tree* $\Rightarrow$ *'v partial-interp* $\Rightarrow$ *'v clause-set* $\Rightarrow$ *bool* **where**
partial-interps Leaf I $\psi = (\exists \chi.\ \neg\ I \models \chi \wedge \chi \in \psi \wedge$ *total-over-m I* $\{\chi\})$ |
partial-interps (*Node v ag ad*) *I* $\psi \longleftrightarrow$
(*partial-interps ag* $(I \cup \{Pos\ v\})$ ψ $\wedge$ *partial-interps ad* $(I\cup \{Neg\ v\})$ ψ)

lemma *simplify-preserve-partial-leaf*:
simplify N N' $\Longrightarrow$ *partial-interps Leaf I N* $\Longrightarrow$ *partial-interps Leaf I N'*
apply (*induct rule*: *simplify.induct*)
using *union-lcomm* **apply** *auto*[*1*]
apply (*simp*)

apply (*metis atms-of-remdups-mset remdups-mset-singleton-sum true-cls-add-mset union-single-eq-member*)
apply *auto*
by (*metis atms-of-ms-emtpy-set subsumption-total-over-m total-over-m-def total-over-m-insert total-over-set-empty true-cls-mono-leD*)

lemma *simplify-preserve-partial-tree*:
assumes *simplify N N′*
and *partial-interps t I N*
shows *partial-interps t I N′*
using *assms* **apply** (*induct t arbitrary*: *I*, *simp*)
using *simplify-preserve-partial-leaf* **by** *metis*

lemma *inference-preserve-partial-tree*:
assumes *inference S S′*
and *partial-interps t I* (*fst S*)
shows *partial-interps t I* (*fst S′*)
using *assms* **apply** (*induct t arbitrary*: *I*, *simp-all*)
by (*meson inference-increasing*)

lemma *rtranclp-inference-preserve-partial-tree*:
assumes *rtranclp inference N N′*
and *partial-interps t I* (*fst N*)
shows *partial-interps t I* (*fst N′*)
using *assms* **apply** (*induct rule*: *rtranclp-induct*, *auto*)
using *inference-preserve-partial-tree* **by** *force*

function *build-sem-tree* :: *′v* :: *linorder set* ⇒ *′v clause-set* ⇒ *′v sem-tree* **where**
build-sem-tree atms ψ =
(*if atms* = {} ∨ ¬ *finite atms*
then Leaf
else Node (*Min atms*) (*build-sem-tree* (*Set.remove* (*Min atms*) *atms*) ψ)
(*build-sem-tree* (*Set.remove* (*Min atms*) *atms*) ψ))
by *auto*
termination
apply (*relation measure* (λ(*A*, -). *card A*), *simp-all*)
apply (*metis Min-in card-Diff1-less remove-def*)+
done
declare *build-sem-tree.induct*[*case-names tree*]

lemma *unsatisfiable-empty*[*simp*]:
¬*unsatisfiable* {}
unfolding *satisfiable-def* **apply** *auto*
using *consistent-interp-def* **unfolding** *total-over-m-def total-over-set-def atms-of-ms-def* **by** *blast*

lemma *partial-interps-build-sem-tree-atms-general*:
fixes ψ :: *′v* :: *linorder clause-set* **and** *p* :: *′v literal list*
assumes *unsat*: *unsatisfiable* ψ **and** *finite* ψ **and** *consistent-interp I*
and *finite atms*
and *atms-of-ms* ψ = *atms* ∪ *atms-of-s I* **and** *atms* ∩ *atms-of-s I* = {}
shows *partial-interps* (*build-sem-tree atms* ψ) *I* ψ
using *assms*
proof (*induct arbitrary*: *I rule*: *build-sem-tree.induct*)
case (*1 atms* ψ *Ia*) **note** *IH1* = *this*(*1*) **and** *IH2* = *this*(*2*) **and** *unsat* = *this*(*3*) **and** *finite* = *this*(*4*)

and *cons* = *this*(*5*) **and** *f* = *this*(*6*) **and** *un* = *this*(*7*) **and** *disj* = *this*(*8*)
{
assume *atms*: *atms* = {}
then have *atmsIa*: *atms-of-ms* ψ = *atms-of-s Ia* **using** *un* **by** *auto*
then have *total-over-m Ia* ψ **unfolding** *total-over-m-def atmsIa* **by** *auto*
then have χ: $\exists\chi \in \psi.\ \neg\ Ia \models \chi$
using *unsat cons* **unfolding** *true-clss-def satisfiable-def* **by** *auto*
then have *build-sem-tree atms* ψ = *Leaf* **using** *atms* **by** *auto*
moreover
have *tot*: $\bigwedge\chi.\ \chi \in \psi \Longrightarrow$ *total-over-m Ia* $\{\chi\}$
unfolding *total-over-m-def total-over-set-def atms-of-ms-def atms-of-s-def*
using *atmsIa atms-of-ms-def* **by** *fastforce*
have *partial-interps Leaf Ia* ψ
using χ *tot* **by** (*auto simp add*: *total-over-m-def total-over-set-def atms-of-ms-def*)

ultimately have *?case* **by** *metis*
}
moreover {
assume *atms*: *atms* $\neq$ {}
have *build-sem-tree atms* ψ = *Node* (*Min atms*) (*build-sem-tree* (*Set.remove* (*Min atms*) *atms*) ψ)
(*build-sem-tree* (*Set.remove* (*Min atms*) *atms*) ψ)
using *build-sem-tree.simps*[*of atms* ψ] *f atms* **by** *metis*

have *consistent-interp* (*Ia* $\cup$ {*Pos* (*Min atms*)}) **unfolding** *consistent-interp-def*
by (*metis Int-iff Min-in Un-iff atm-of-uminus atms cons consistent-interp-def disj empty-iff f in-atms-of-s-decomp insert-iff literal.distinct*(*1*) *literal.exhaust-sel literal.sel*(*2*) *uminus-Neg uminus-Pos*)
moreover have *atms-of-ms* ψ = *Set.remove* (*Min atms*) *atms* $\cup$ *atms-of-s* (*Ia* $\cup$ {*Pos* (*Min atms*)})
using *Min-in atms f un* **by** *fastforce*
moreover have *disj'*: *Set.remove* (*Min atms*) *atms* $\cap$ *atms-of-s* (*Ia* $\cup$ {*Pos* (*Min atms*)}) = {}
by *simp* (*metis disj disjoint-iff-not-equal member-remove*)
moreover have *finite* (*Set.remove* (*Min atms*) *atms*) **using** *f* **by** (*simp add*: *remove-def*)
ultimately have *subtree1*: *partial-interps* (*build-sem-tree* (*Set.remove* (*Min atms*) *atms*) ψ)
(*Ia* $\cup$ {*Pos* (*Min atms*)}) ψ
using *IH1*[*of Ia* $\cup$ {*Pos* (*Min* (*atms*))}] *atms f unsat finite* **by** *metis*

have *consistent-interp* (*Ia* $\cup$ {*Neg* (*Min atms*)}) **unfolding** *consistent-interp-def*
by (*metis Int-iff Min-in Un-iff atm-of-uminus atms cons consistent-interp-def disj empty-iff f in-atms-of-s-decomp insert-iff literal.distinct*(*1*) *literal.exhaust-sel literal.sel*(*2*) *uminus-Neg*)
moreover have *atms-of-ms* ψ = *Set.remove* (*Min atms*) *atms* $\cup$ *atms-of-s* (*Ia* $\cup$ {*Neg* (*Min atms*)})
using ‹*atms-of-ms* ψ = *Set.remove* (*Min atms*) *atms* $\cup$ *atms-of-s* (*Ia* $\cup$ {*Pos* (*Min atms*)})› **by** *blast*

moreover have *disj'*: *Set.remove* (*Min atms*) *atms* $\cap$ *atms-of-s* (*Ia* $\cup$ {*Neg* (*Min atms*)}) = {}
using *disj* **by** *auto*
moreover have *finite* (*Set.remove* (*Min atms*) *atms*) **using** *f* **by** (*simp add*: *remove-def*)
ultimately have *subtree2*: *partial-interps* (*build-sem-tree* (*Set.remove* (*Min atms*) *atms*) ψ)
(*Ia* $\cup$ {*Neg* (*Min atms*)}) ψ
using *IH2*[*of Ia* $\cup$ {*Neg* (*Min* (*atms*))}] *atms f unsat finite* **by** *metis*

then have *?case*
using *IH1 subtree1 subtree2 f local.finite unsat atms* **by** *simp*
}
ultimately show *?case* **by** *metis*
qed

lemma *partial-interps-build-sem-tree-atms*:
  fixes ψ :: $'v$ :: *linorder clause-set* **and** p :: $'v$ *literal list*
  assumes *unsat*: *unsatisfiable* ψ **and** *finite*: *finite* ψ
  shows *partial-interps* (*build-sem-tree* (*atms-of-ms* ψ) ψ) {} ψ
proof −
  have *consistent-interp* {} **unfolding** *consistent-interp-def* **by** *auto*
  moreover have *atms-of-ms* ψ = *atms-of-ms* ψ ∪ *atms-of-s* {} **unfolding** *atms-of-s-def* **by** *auto*
  moreover have *atms-of-ms* ψ ∩ *atms-of-s* {} = {} **unfolding** *atms-of-s-def* **by** *auto*
  moreover have *finite* (*atms-of-ms* ψ) **unfolding** *atms-of-ms-def* **using** *finite* **by** *simp*
  ultimately show *partial-interps* (*build-sem-tree* (*atms-of-ms* ψ) ψ) {} ψ
    using *partial-interps-build-sem-tree-atms-general*[*of* ψ {} *atms-of-ms* ψ] *assms* **by** *metis*
qed

lemma *can-decrease-count*:
  fixes ψ'' :: $'v$ *clause-set* × ($'v$ *clause* × $'v$ *clause* × $'v$) *set*
  assumes *count* χ L = n
  and L ∈# χ **and** χ ∈ *fst* ψ
  shows ∃ ψ' χ'. *inference*** ψ ψ' ∧ χ' ∈ *fst* ψ' ∧ (∀ L. L ∈# χ ⟷ L ∈# χ')
                ∧ *count* χ' L = *1*
                ∧ (∀ φ. φ ∈ *fst* ψ ⟶ φ ∈ *fst* ψ')
                ∧ (I ⊨ χ ⟷ I ⊨ χ')
                ∧ (∀ I'. *total-over-m* I' {χ} ⟶ *total-over-m* I' {χ'})
  using *assms*
proof (*induct n arbitrary*: χ ψ)
  case *0*
  then show *?case* **by** (*simp add*: *not-in-iff*[*symmetric*])
next
   case (*Suc n* χ)
   note *IH* = *this*(*1*) **and** *count* = *this*(*2*) **and** L = *this*(*3*) **and** χ = *this*(*4*)
   {
     assume n = *0*
     then have *inference*** ψ ψ
     and χ ∈ *fst* ψ
     and ∀ L. (L ∈# χ) ⟷ (L ∈# χ)
     and *count* χ L = (*1*::*nat*)
     and ∀ φ. φ ∈ *fst* ψ ⟶ φ ∈ *fst* ψ
       by (*auto simp add*: *count* L χ)
     then have *?case* **by** *metis*
   }
   moreover {
     assume n > *0*
     then have ∃ C. χ = C + {#L, L#}
       by (*metis Suc-inject union-mset-add-mset-right add-mset-add-single count-add-mset count-inI less-not-refl3 local.count mset-add zero-less-Suc*)
     then obtain C **where** C: χ = C + {#L, L#} **by** *metis*
     let *?*χ' = C +{#L#}
     let *?*ψ' = (*fst* ψ ∪ {*?*χ'}, *snd* ψ)
     have φ: ∀ φ ∈ *fst* ψ. (φ ∈ *fst* ψ ∨ φ ≠ *?*χ') ⟷ φ ∈ *fst* *?*ψ' **unfolding** C **by** *auto*
     have *inf*: *inference* ψ *?*ψ'
       using C *factoring* χ *prod.collapse union-commute inference-step*
       by (*metis add-mset-add-single*)
     moreover have *count'*: *count* *?*χ' L = n **using** C *count* **by** *auto*
     moreover have $L\chi'$: L ∈# *?*χ' **by** *auto*
     moreover have $\chi'\psi'$: *?*χ' ∈ *fst* *?*ψ' **by** *auto*

ultimately obtain ψ'' **and** χ''
where
*inference*** $?\psi'$ ψ'' **and**
α: $\chi'' \in$ *fst* ψ'' **and**
$\forall$ *La*. (*La* $\in$# $?\chi'$) $\longleftrightarrow$ (*La* $\in$# χ'') **and**
β: *count* χ'' *L* = (*1*::*nat*) **and**
φ': $\forall\varphi$. $\varphi \in$ *fst* $?\psi' \longrightarrow \varphi \in$ *fst* ψ'' **and**
Iχ: *I* $\models$ $?\chi' \longleftrightarrow$ *I* $\models$ χ'' **and**
tot: $\forall$ *I'*. *total-over-m* *I'* {$?\chi'$} $\longrightarrow$ *total-over-m* *I'* {χ''}
using *IH*[*of* $?\chi'$ $?\psi'$] *count'* *L*χ' $\chi'\psi'$ **by** *blast*

then have *inference*** ψ ψ''
and $\forall$ *La*. (*La* $\in$# χ) $\longleftrightarrow$ (*La* $\in$# χ'')
using *inf* **unfolding** *C* **by** *auto*
moreover have $\forall\varphi$. $\varphi \in$ *fst* $\psi \longrightarrow \varphi \in$ *fst* ψ'' **using** φ φ' **by** *metis*
moreover have *I* $\models \chi \longleftrightarrow$ *I* $\models \chi''$ **using** *I*χ **unfolding** *true-cls-def* *C* **by** *auto*
moreover have $\forall$ *I'*. *total-over-m* *I'* {χ} $\longrightarrow$ *total-over-m* *I'* {χ''}
using *tot* **unfolding** *C* *total-over-m-def* **by** *auto*
ultimately have *?case* **using** φ φ' α β **by** *metis*
}
ultimately show *?case* **by** *auto*
qed

lemma *can-decrease-tree-size*:
fixes ψ :: *'v state* **and** *tree* :: *'v sem-tree*
assumes *finite* (*fst* ψ) **and** *already-used-inv* ψ
and *partial-interps tree I* (*fst* ψ)
shows $\exists$(*tree'*:: *'v sem-tree*) ψ'. *inference*** ψ ψ' $\wedge$ *partial-interps tree' I* (*fst* ψ')
$\wedge$ (*sem-tree-size tree'* < *sem-tree-size tree* $\vee$ *sem-tree-size tree* = *0*)
using *assms*
proof (*induct arbitrary*: *I rule*: *sem-tree-size*)
case (*bigger xs I*) **note** *IH* = *this*(*1*) **and** *finite* = *this*(*2*) **and** *a-u-i* = *this*(*3*) **and** *part* = *this*(*4*)

{
assume *sem-tree-size xs* = *0*
then have *?case* **using** *part* **by** *blast*
}

moreover {
assume *sn0*: *sem-tree-size xs* > *0*
obtain *ag ad v* **where** *xs*: *xs* = *Node v ag ad* **using** *sn0* **by** (*cases xs*, *auto*)
{
assume *sem-tree-size ag* = *0* **and** *sem-tree-size ad* = *0*
then have *ag*: *ag* = *Leaf* **and** *ad*: *ad* = *Leaf* **by** (*cases ag*, *auto*) (*cases ad*, *auto*)

then obtain χ χ' **where**
χ: $\neg$ *I* $\cup$ {*Pos v*} $\models \chi$ **and**
totχ: *total-over-m* (*I* $\cup$ {*Pos v*}) {χ} **and**
$\chi\psi$: $\chi \in$ *fst* ψ **and**
χ': $\neg$ *I* $\cup$ {*Neg v*} $\models \chi'$ **and**
totχ': *total-over-m* (*I* $\cup$ {*Neg v*}) {χ'} **and**
$\chi'\psi$: $\chi' \in$ *fst* ψ
using *part* **unfolding** *xs* **by** *auto*
have *Posv*: *Pos v* $\notin$# χ **using** χ **unfolding** *true-cls-def true-lit-def* **by** *auto*
have *Negv*: *Neg v* $\notin$# χ' **using** χ' **unfolding** *true-cls-def true-lit-def* **by** *auto*
{

assume *Negχ*: $Neg\ v \notin\#\ \chi$
have $\neg\ I \models \chi$ **using** χ *Posv* **unfolding** *true-cls-def true-lit-def* **by** *auto*
moreover have *total-over-m* $I\ \{\chi\}$
using *Posv Negχ atm-imp-pos-or-neg-lit totχ* **unfolding** *total-over-m-def total-over-set-def*
by *fastforce*
ultimately have *partial-interps Leaf* I $(fst\ \psi)$
and *sem-tree-size Leaf* $<$ *sem-tree-size xs*
and $inference^{**}\ \psi\ \psi$
unfolding *xs* **by** (*auto simp add*: $\chi\psi$)
}
moreover {
assume *Posχ*: $Pos\ v \notin\#\ \chi'$
then have $I\chi$: $\neg\ I \models \chi'$ **using** χ' *Posv* **unfolding** *true-cls-def true-lit-def* **by** *auto*
moreover have *total-over-m* $I\ \{\chi'\}$
using *Negv Posχ atm-imp-pos-or-neg-lit totχ′*
unfolding *total-over-m-def total-over-set-def* **by** *fastforce*
ultimately have *partial-interps Leaf* I $(fst\ \psi)$ **and**
sem-tree-size Leaf $<$ *sem-tree-size xs* **and**
$inference^{**}\ \psi\ \psi$
using $\chi'\psi$ $I\chi$ **unfolding** *xs* **by** *auto*
}
moreover {
assume *neg*: $Neg\ v \in\#\ \chi$ **and** *pos*: $Pos\ v \in\#\ \chi'$
then obtain $\psi'\ \chi 2$ **where** *inf*: *rtranclp inference* $\psi\ \psi'$ **and** *χ2incl*: $\chi 2 \in fst\ \psi'$
and *χχ2-incl*: $\forall L.\ L \in\#\ \chi \longleftrightarrow L \in\#\ \chi 2$
and *countχ2*: $count\ \chi 2\ (Neg\ v) = 1$
and φ: $\forall \varphi{::}'v\ literal\ multiset.\ \varphi \in fst\ \psi \longrightarrow \varphi \in fst\ \psi'$
and $I\chi$: $I \models \chi \longleftrightarrow I \models \chi 2$
and *tot-impχ*: $\forall I'.\ total\text{-}over\text{-}m\ I'\ \{\chi\} \longrightarrow total\text{-}over\text{-}m\ I'\ \{\chi 2\}$
using *can-decrease-count*[*of* χ *Neg v count* χ $(Neg\ v)$ ψ I] $\chi\psi$ $\chi'\psi$ **by** *auto*

have $\chi' \in fst\ \psi'$ **by** (*simp add*: $\chi'\psi\ \varphi$)
with *pos*
obtain $\psi''\ \chi 2'$ **where**
inf′: $inference^{**}\ \psi'\ \psi''$
and *χ2′-incl*: $\chi 2' \in fst\ \psi''$
and *χ′χ2-incl*: $\forall L{::}'v\ literal.\ (L \in\#\ \chi') = (L \in\#\ \chi 2')$
and *countχ2′*: $count\ \chi 2'\ (Pos\ v) = (1{::}nat)$
and φ': $\forall \varphi{::}'v\ literal\ multiset.\ \varphi \in fst\ \psi' \longrightarrow \varphi \in fst\ \psi''$
and $I\chi'$: $I \models \chi' \longleftrightarrow I \models \chi 2'$
and *tot-impχ′*: $\forall I'.\ total\text{-}over\text{-}m\ I'\ \{\chi'\} \longrightarrow total\text{-}over\text{-}m\ I'\ \{\chi 2'\}$
using *can-decrease-count*[*of* χ' *Pos v count* χ' $(Pos\ v)$ ψ' I] **by** *auto*

define C **where** C: $C = \chi 2 - \{\#Neg\ v\#\}$

then have $\chi 2$: $\chi 2 = C + \{\#Neg\ v\#\}$ **and** *negC*: $Neg\ v \notin\#\ C$ **and** *posC*: $Pos\ v \notin\#\ C$
using *χχ2-incl neg* **apply** *auto*[]
using *C χχ2-incl neg countχ2 count-eq-zero-iff* **apply** *fastforce*
using *C Posv χχ2-incl in-diffD* **by** *fastforce*

obtain C' **where**
$\chi 2'$: $\chi 2' = C' + \{\#Pos\ v\#\}$ **and**
posC′: $Pos\ v \notin\#\ C'$ **and**
negC′: $Neg\ v \notin\#\ C'$
proof −
assume *a1*: $\bigwedge C'.\ [\![\chi 2' = C' + \{\#Pos\ v\#\};\ Pos\ v \notin\#\ C';\ Neg\ v \notin\#\ C']\!] \Longrightarrow thesis$

have *f2*: $\bigwedge n.$ $(n::nat) - n = 0$
by *simp*
have *Neg v* $\notin\#$ $\chi 2' - \{\#Pos\ v\#\}$
using *Negv* $\chi'\chi 2$*-incl* **by** (*auto simp*: *not-in-iff*)
have *count* $\{\#Pos\ v\#\}$ $(Pos\ v) = 1$
by *simp*
then show *?thesis*
by (*metis* $\chi'\chi 2$*-incl* ‹*Neg v* $\notin\#$ $\chi 2' - \{\#Pos\ v\#\}$› *a1 count*$\chi 2'$ *count-diff f2 insert-DiffM2 less-numeral-extra*(*3*) *mem-Collect-eq pos set-mset-def*)
qed

have *already-used-inv* ψ'
using *rtranclp-inference-preserves-already-used-inv*[*of* ψ ψ'] *a-u-i inf* **by** *blast*
then have *a-u-i-*ψ'': *already-used-inv* ψ''
using *rtranclp-inference-preserves-already-used-inv a-u-i inf'* **unfolding** *tautology-def*
by *simp*

have *totC*: *total-over-m I* $\{C\}$
using *tot-imp*χ *tot*χ *tot-over-m-remove*[*of I Pos v C*] *negC posC* **unfolding** $\chi 2$
by (*metis total-over-m-sum uminus-Neg uminus-of-uminus-id*)
have *totC'*: *total-over-m I* $\{C'\}$
using *tot-imp*χ' *tot*χ' *total-over-m-sum tot-over-m-remove*[*of I Neg v C'*] *negC' posC'*
unfolding $\chi 2'$ **by** (*metis total-over-m-sum uminus-Neg*)
have $\neg$ $I \models C + C'$
using χ $I\chi$ χ' $I\chi'$ **unfolding** $\chi 2$ $\chi 2'$ *true-cls-def* **by** *auto*
then have *part-I-*ψ''': *partial-interps Leaf I* (*fst* $\psi'' \cup \{C + C'\}$)
using *totC totC'* **by** *simp*
(*metis* ‹$\neg$ $I \models C + C'$› *atms-of-ms-singleton total-over-m-def total-over-m-sum*)
{
assume $(\{\#Pos\ v\#\} + C', \{\#Neg\ v\#\} + C) \notin$ *snd* ψ''
then have *inf''*: *inference* ψ'' (*fst* $\psi'' \cup \{C + C'\}$, *snd* $\psi'' \cup \{(\chi 2', \chi 2)\}$)
using *add.commute* φ' $\chi 2$*incl* ‹$\chi 2' \in$ *fst* ψ''› **unfolding** $\chi 2$ $\chi 2'$
by (*metis prod.collapse inference-step resolution*)
have *inference*** ψ (*fst* $\psi'' \cup \{C + C'\}$, *snd* $\psi'' \cup \{(\chi 2', \chi 2)\}$)
using *inf inf' inf'' rtranclp-trans* **by** *auto*
moreover have *sem-tree-size Leaf* $<$ *sem-tree-size xs* **unfolding** *xs* **by** *auto*
ultimately have *?case* **using** *part-I-*ψ''' **by** (*metis fst-conv*)
}
moreover {
assume *a*: $(\{\#Pos\ v\#\} + C', \{\#Neg\ v\#\} + C) \in$ *snd* ψ''
then have ($\exists\chi \in$ *fst* ψ''. ($\forall I$. *total-over-m I* $\{C{+}C'\}$ $\longrightarrow$ *total-over-m I* $\{\chi\}$)
$\wedge$ ($\forall I$. *total-over-m I* $\{\chi\}$ $\longrightarrow I \models \chi \longrightarrow I \models C' + C$))
$\vee$ *tautology* $(C' + C)$
proof $-$
obtain *p* **where** *p*: *Pos p* $\in\#$ $(\{\#Pos\ v\#\} + C')$ **and**
n: *Neg p* $\in\#$ $(\{\#Neg\ v\#\} + C)$ **and**
decomp: (($\exists\chi{\in}$*fst* ψ''.
($\forall I$. *total-over-m I* $\{(\{\#Pos\ v\#\} + C') - \{\#Pos\ p\#\}$
$+ ((\{\#Neg\ v\#\} + C) - \{\#Neg\ p\#\})\}$
$\longrightarrow$ *total-over-m I* $\{\chi\}$)
$\wedge$ ($\forall I$. *total-over-m I* $\{\chi\}$ $\longrightarrow I \models \chi$
$\longrightarrow I \models (\{\#Pos\ v\#\} + C') - \{\#Pos\ p\#\} + ((\{\#Neg\ v\#\} + C) - \{\#Neg\ p\#\}))$
)
$\vee$ *tautology* $(((\{\#Pos\ v\#\} + C') - \{\#Pos\ p\#\} + ((\{\#Neg\ v\#\} + C) - \{\#Neg\ p\#\})))$
using *a* **by** (*blast intro*: *allE*[*OF a-u-i-*ψ''[*unfolded subsumes-def Ball-def*],
of $(\{\#Pos\ v\#\} + C', \{\#Neg\ v\#\} + C)$])

{ assume $p \neq v$
then have $Pos\ p \in\#\ C' \wedge Neg\ p \in\#\ C$ **using** *p n* **by** *force*
then have *?thesis* **unfolding** *Bex-def* **by** *auto*
}
moreover {
assume $p = v$
then have *?thesis* **using** *decomp* **by** (*metis add.commute add-diff-cancel-left'*)
}
ultimately show *?thesis* **by** *auto*
qed
moreover {
assume $\exists \chi \in fst\ \psi''.\ (\forall I.\ total\text{-}over\text{-}m\ I\ \{C+C'\} \longrightarrow total\text{-}over\text{-}m\ I\ \{\chi\})$
$\wedge\ (\forall I.\ total\text{-}over\text{-}m\ I\ \{\chi\} \longrightarrow I \models \chi \longrightarrow I \models C' + C)$
then obtain ϑ **where** ϑ: $\vartheta \in fst\ \psi''$ **and**
$tot\text{-}\vartheta\text{-}CC'$: $\forall I.\ total\text{-}over\text{-}m\ I\ \{C+C'\} \longrightarrow total\text{-}over\text{-}m\ I\ \{\vartheta\}$ **and**
$\vartheta\text{-}inv$: $\forall I.\ total\text{-}over\text{-}m\ I\ \{\vartheta\} \longrightarrow I \models \vartheta \longrightarrow I \models C' + C$ **by** *blast*
have *partial-interps Leaf I* $(fst\ \psi'')$
using $tot\text{-}\vartheta\text{-}CC'\ \vartheta\ \vartheta\text{-}inv$ *totC totC'* $\langle\neg\ I \models C + C'\rangle$ *total-over-m-sum* **by** *fastforce*
moreover have *sem-tree-size Leaf* $<$ *sem-tree-size xs* **unfolding** *xs* **by** *auto*
ultimately have *?case* **by** (*metis inf inf' rtranclp-trans*)
}
moreover {
assume *tautCC'*: *tautology* $(C' + C)$
have *total-over-m I* $\{C'+C\}$ **using** *totC totC' total-over-m-sum* **by** *auto*
then have $\neg$*tautology* $(C' + C)$
using $\langle\neg\ I \models C + C'\rangle$ **unfolding** *add.commute*[*of C C'*] *total-over-m-def*
unfolding *tautology-def* **by** *auto*
then have *False* **using** *tautCC'* **unfolding** *tautology-def* **by** *auto*
}
ultimately have *?case* **by** *auto*
}
ultimately have *?case* **by** *auto*
}
ultimately have *?case* **using** *part* **by** (*metis* (*no-types*) *sem-tree-size.simps*(*1*))
}
moreover {
assume *size-ag*: *sem-tree-size ag* > 0
have *sem-tree-size ag* $<$ *sem-tree-size xs* **unfolding** *xs* **by** *auto*
moreover have *partial-interps ag* $(I \cup \{Pos\ v\})\ (fst\ \psi)$
and *partad*: *partial-interps ad* $(I \cup \{Neg\ v\})\ (fst\ \psi)$
using *part partial-interps.simps*(*2*) **unfolding** *xs* **by** *metis+*
moreover have *sem-tree-size ag* $<$ *sem-tree-size xs* $\longrightarrow$ *finite* $(fst\ \psi) \longrightarrow$ *already-used-inv* ψ
$\longrightarrow$ (*partial-interps ag* $(I \cup \{Pos\ v\})\ (fst\ \psi) \longrightarrow$
$(\exists\ tree'\ \psi'.\ inference^{**}\ \psi\ \psi' \wedge partial\text{-}interps\ tree'\ (I \cup \{Pos\ v\})\ (fst\ \psi')$
$\wedge\ (sem\text{-}tree\text{-}size\ tree' < sem\text{-}tree\text{-}size\ ag \vee sem\text{-}tree\text{-}size\ ag = 0)))$
using *IH* **by** *auto*
ultimately obtain ψ' :: *'v state* **and** *tree'* :: *'v sem-tree* **where**
inf: $inference^{**}\ \psi\ \psi'$
and *part*: *partial-interps tree'* $(I \cup \{Pos\ v\})\ (fst\ \psi')$
and *size*: *sem-tree-size tree'* $<$ *sem-tree-size ag* $\vee$ *sem-tree-size ag* $= 0$
using *finite part rtranclp.rtrancl-refl a-u-i* **by** *blast*

have *partial-interps ad* $(I \cup \{Neg\ v\})\ (fst\ \psi')$
using *rtranclp-inference-preserve-partial-tree inf partad* **by** *metis*
then have *partial-interps* (*Node v tree' ad*) *I* $(fst\ \psi')$ **using** *part* **by** *auto*
then have *?case* **using** *inf size size-ag part* **unfolding** *xs* **by** *fastforce*

```
    }
    moreover {
      assume size-ad: sem-tree-size ad > 0
      have sem-tree-size ad < sem-tree-size xs unfolding xs by auto
      moreover have partag: partial-interps ag (I ∪ {Pos v}) (fst ψ) and
        partial-interps ad (I ∪ {Neg v}) (fst ψ)
        using part partial-interps.simps(2) unfolding xs by metis+
      moreover have sem-tree-size ad < sem-tree-size xs ⟶ finite (fst ψ) ⟶ already-used-inv ψ
        ⟶ ( partial-interps ad (I ∪ {Neg v}) (fst ψ)
        ⟶ (∃ tree′ ψ′. inference** ψ ψ′ ∧ partial-interps tree′ (I ∪ {Neg v}) (fst ψ′)
            ∧ (sem-tree-size tree′ < sem-tree-size ad ∨ sem-tree-size ad = 0)))
        using IH by auto
      ultimately obtain ψ′ :: ′v state and tree′ :: ′v sem-tree where
        inf: inference** ψ ψ′
        and part: partial-interps tree′ (I ∪ {Neg v}) (fst ψ′)
        and size: sem-tree-size tree′ < sem-tree-size ad ∨ sem-tree-size ad = 0
        using finite part rtranclp.rtrancl-refl a-u-i by blast

      have partial-interps ag (I ∪ {Pos v}) (fst ψ′)
        using rtranclp-inference-preserve-partial-tree inf partag by metis
      then have partial-interps (Node v ag tree′) I (fst ψ′) using part by auto
      then have ?case using inf size size-ad unfolding xs by fastforce
    }
    ultimately have ?case by auto
  }
  ultimately show ?case by auto
qed

lemma inference-completeness-inv:
  fixes ψ :: ′v ::linorder state
  assumes
    unsat: ¬satisfiable (fst ψ) and
    finite: finite (fst ψ) and
    a-u-v: already-used-inv ψ
  shows ∃ψ′. (inference** ψ ψ′ ∧ {#} ∈ fst ψ′)
proof −
  obtain tree where partial-interps tree {} (fst ψ)
    using partial-interps-build-sem-tree-atms assms by metis
  then show ?thesis
    using unsat finite a-u-v
    proof (induct tree arbitrary: ψ rule: sem-tree-size)
      case (bigger tree ψ) note H = this
      {
        fix χ
        assume tree: tree = Leaf
        obtain χ where χ: ¬ {} ⊨ χ and totχ: total-over-m {} {χ} and χψ: χ ∈ fst ψ
          using H unfolding tree by auto
        moreover have {#} = χ
          using totχ unfolding total-over-m-def total-over-set-def by fastforce
        moreover have inference** ψ ψ by auto
        ultimately have ?case by metis
      }
      moreover {
        fix v tree1 tree2
        assume tree: tree = Node v tree1 tree2
        obtain
```

tree' ψ' **where** *inf*: *inference*** ψ ψ' **and**
part': *partial-interps tree'* {} (*fst* ψ') **and**
decrease: *sem-tree-size tree'* < *sem-tree-size tree* $\vee$ *sem-tree-size tree* = *0*
using *can-decrease-tree-size*[*of* ψ] *H*(*2*,*4*,*5*) **unfolding** *tautology-def* **by** *meson*
have *sem-tree-size tree'* < *sem-tree-size tree* **using** *decrease* **unfolding** *tree* **by** *auto*
moreover have *finite* (*fst* ψ') **using** *rtranclp-inference-preserves-finite inf H*(*4*) **by** *metis*
moreover have *unsatisfiable* (*fst* ψ')
using *inference-preserves-unsat inf bigger.prems*(*2*) **by** *blast*
moreover have *already-used-inv* ψ'
using *H*(*5*) *inf rtranclp-inference-preserves-already-used-inv*[*of* ψ ψ'] **by** *auto*
ultimately have *?case* **using** *inf rtranclp-trans part' H*(*1*) **by** *fastforce*
}
ultimately show *?case* **by** (*cases tree*, *auto*)
qed
qed

lemma *inference-completeness*:
fixes ψ :: *'v* ::*linorder state*
assumes *unsat*: $\neg$*satisfiable* (*fst* ψ)
and *finite*: *finite* (*fst* ψ)
and *snd* ψ = {}
shows $\exists\psi'$. (*rtranclp inference* ψ ψ' $\wedge$ {#} $\in$ *fst* ψ')
proof −
have *already-used-inv* ψ **unfolding** *assms* **by** *auto*
then show *?thesis* **using** *assms inference-completeness-inv* **by** *blast*
qed

lemma *inference-soundness*:
fixes ψ :: *'v* ::*linorder state*
assumes *rtranclp inference* ψ ψ' **and** {#} $\in$ *fst* ψ'
shows *unsatisfiable* (*fst* ψ)
using *assms* **by** (*meson rtranclp-inference-preserve-models satisfiable-def true-cls-empty true-clss-def*)

lemma *inference-soundness-and-completeness*:
fixes ψ :: *'v* ::*linorder state*
assumes *finite*: *finite* (*fst* ψ)
and *snd* ψ = {}
shows ($\exists\psi'$. (*inference*** ψ ψ' $\wedge$ {#} $\in$ *fst* ψ')) $\longleftrightarrow$ *unsatisfiable* (*fst* ψ)
using *assms inference-completeness inference-soundness* **by** *metis*

2.1.4 Lemma about the Simplified State

abbreviation *simplified* ψ $\equiv$ (*no-step simplify* ψ)

lemma *simplified-count*:
assumes *simp*: *simplified* ψ **and** χ: $\chi \in \psi$
shows *count* χ *L* $\leq$ *1*
proof −
{
let *?*χ' = χ − {#*L*, *L*#}
assume *count* χ *L* $\geq$ *2*
then have *f1*: *count* (χ − {#*L*, *L*#} + {#*L*, *L*#}) *L* = *count* χ *L*
by *simp*
then have *L* $\in$# χ − {#*L*#}
by (*metis* (*no-types*) *add.left-neutral add-diff-cancel-left' count-union diff-diff-add*

diff-single-trivial insert-DiffM mem-Collect-eq multi-member-this not-gr0 set-mset-def)
    then have χ': {#*L*, *L*#} + *?*χ' = χ
      using *f1 in-diffD insert-DiffM* **by** *fastforce*

    have $\exists\,\psi'$. *simplify* ψ ψ'
      by (*metis* (*no-types*, *hide-lams*) χ χ' *factoring-imp-simplify*)
    then have *False* **using** *simp* **by** *auto*
  }
  then show *?thesis* **by** *arith*
qed

lemma *simplified-no-both*:
  assumes *simp*: *simplified* ψ **and** χ: $\chi \in \psi$
  shows $\neg$ (*L* $\in$# χ $\wedge$ $-$*L* $\in$# χ)
proof (*rule ccontr*)
  assume $\neg$ $\neg$ (*L* $\in$# χ $\wedge$ $-$ *L* $\in$# χ)
  then have *L* $\in$# χ $\wedge$ $-$ *L* $\in$# χ **by** *metis*
  then obtain χ' **where** χ = *add-mset* (*Pos* (*atm-of L*)) (*add-mset* (*Neg* (*atm-of L*)) χ')
    by (*cases L*) (*auto dest*!: *multi-member-split simp*: *add-eq-conv-ex*)
  then show *False* **using** χ *simp tautology-deletion* **by** *fast*
qed

lemma *add-mset-Neg-Pos-commute*[*simp*]:
  add-mset (*Neg P*) (*add-mset* (*Pos P*) *C*) = *add-mset* (*Pos P*) (*add-mset* (*Neg P*) *C*)
  by (*rule add-mset-commute*)

lemma *simplified-not-tautology*:
  assumes *simplified* {ψ}
  shows ~*tautology* ψ
proof (*rule ccontr*)
  assume ~ *?thesis*
  then obtain *p* **where** *Pos p* $\in$# ψ $\wedge$ *Neg p* $\in$# ψ **using** *tautology-decomp* **by** *metis*
  then obtain χ **where** ψ = χ + {#*Pos p*#} + {#*Neg p*#}
    by (*auto dest*!: *multi-member-split simp*: *add-eq-conv-ex*)
  then have ~ *simplified* {ψ} **by** (*auto intro*: *tautology-deletion*)
  then show *False* **using** *assms* **by** *auto*
qed

lemma *simplified-remove*:
  assumes *simplified* {ψ}
  shows *simplified* {ψ $-$ {#*l*#}}
proof (*rule ccontr*)
  assume *ns*: $\neg$ *simplified* {ψ $-$ {#*l*#}}
  {
    assume *l* $\notin$# ψ
    then have ψ $-$ {#*l*#} = ψ **by** *simp*
    then have *False* **using** *ns assms* **by** *auto*
  }
  moreover {
    assume *l*ψ: *l* $\in$# ψ
    have *A*: $\bigwedge$*A*. *A* $\in$ {ψ $-$ {#*l*#}} $\longleftrightarrow$ *add-mset l A* $\in$ {ψ} **by** (*auto simp add*: *l*ψ)
    obtain *l'* **where** *l'*: *simplify* {ψ $-$ {#*l*#}} *l'* **using** *ns* **by** *metis*
    then have $\exists$ *l'*. *simplify* {ψ} *l'*
      proof (*induction rule*: *simplify.induct*)
        case (*tautology-deletion P A*)
        then have {#*Neg P*#} + ({#*Pos P*#} + (*A* + {#*l*#})) $\in$ {ψ}

using *A* **by** *auto*
then show *?thesis*
using *simplified-no-both* **by** *fastforce*
next
case (*condensation L A*)
have *add-mset l* (*add-mset L* (*add-mset L A*)) $\in \{\psi\}$
using *condensation.hyps* **unfolding** *A* **by** *blast*
then have $\{\#L, L\#\} + (A + \{\#l\#\}) \in \{\psi\}$
by *auto*
then show *?case*
using *factoring-imp-simplify* **by** *blast*
next
case (*subsumption A B*)
then show *?case* **by** *blast*
qed
then have *False* **using** *assms*(*1*) **by** *blast*
}
ultimately show *False* **by** *auto*
qed

lemma *in-simplified-simplified*:
assumes *simp*: *simplified* ψ **and** *incl*: $\psi' \subseteq \psi$
shows *simplified* ψ'
proof (*rule ccontr*)
assume $\neg$ *?thesis*
then obtain ψ'' **where** *simplify* ψ' ψ'' **by** *metis*
then have $\exists\, l'.$ *simplify* ψ l'
proof (*induction rule*: *simplify.induct*)
case (*tautology-deletion A P*)
then show *?thesis* **using** *simplify.tautology-deletion*[*of A P* ψ] *incl* **by** *blast*
next
case (*condensation A L*)
then show *?case* **using** *simplify.condensation*[*of A L* ψ] *incl* **by** *blast*
next
case (*subsumption A B*)
then show *?case* **using** *simplify.subsumption*[*of A* ψ *B*] *incl* **by** *auto*
qed
then show *False* **using** *assms*(*1*) **by** *blast*
qed

lemma *simplified-in*:
assumes *simplified* ψ
and $N \in \psi$
shows *simplified* $\{N\}$
using *assms* **by** (*metis Set.set-insert empty-subsetI in-simplified-simplified insert-mono*)

lemma *subsumes-imp-formula*:
assumes $\psi \leq\# \varphi$
shows $\{\psi\} \models p\ \varphi$
unfolding *true-clss-cls-def* **apply** *auto*
using *assms true-cls-mono-leD* **by** *blast*

lemma *simplified-imp-distinct-mset-tauto*:
assumes *simp*: *simplified* ψ'
shows *distinct-mset-set* ψ' **and** $\forall \chi \in \psi'.\ \neg$*tautology* χ

proof −
  show $\forall\chi \in \psi'$. $\neg$*tautology* χ
    using *simp* **by** (*auto simp add*: *simplified-in simplified-not-tautology*)

  show *distinct-mset-set* ψ'
    proof (*rule ccontr*)
      assume $\neg$*?thesis*
      then obtain χ **where** $\chi \in \psi'$ **and** $\neg$*distinct-mset* χ **unfolding** *distinct-mset-set-def* **by** *auto*
      then obtain L **where** *count* χ $L \geq 2$
        unfolding *distinct-mset-def*
        by (*meson count-greater-eq-one-iff le-antisym simp simplified-count*)
      then show *False* **by** (*metis Suc-1* $\langle\chi \in \psi'\rangle$ *not-less-eq-eq simp simplified-count*)
    qed
qed

lemma *simplified-no-more-full1-simplified*:
  assumes *simplified* ψ
  shows $\neg$*full1 simplify* ψ ψ'
  using *assms* **unfolding** *full1-def* **by** (*meson tranclpD*)

2.1.5 Resolution and Invariants

inductive *resolution* :: *'v state* $\Rightarrow$ *'v state* $\Rightarrow$ *bool* **where**
full1-simp: *full1 simplify* N N' $\Longrightarrow$ *resolution* (N, *already-used*) (N', *already-used*) |
inferring: *inference* (N, *already-used*) (N', *already-used'*) $\Longrightarrow$ *simplified* N
  $\Longrightarrow$ *full simplify* N' N'' $\Longrightarrow$ *resolution* (N, *already-used*) (N'', *already-used'*)

Invariants

lemma *resolution-finite*:
  assumes *resolution* ψ ψ' **and** *finite* (*fst* ψ)
  shows *finite* (*fst* ψ')
  using *assms* **by** (*induct rule*: *resolution.induct*)
    (*auto simp add*: *full1-def full-def rtranclp-simplify-preserves-finite*
      dest: *tranclp-into-rtranclp inference-preserves-finite*)

lemma *rtranclp-resolution-finite*:
  assumes *resolution*** ψ ψ' **and** *finite* (*fst* ψ)
  shows *finite* (*fst* ψ')
  using *assms* **by** (*induct rule*: *rtranclp-induct*, *auto simp add*: *resolution-finite*)

lemma *resolution-finite-snd*:
  assumes *resolution* ψ ψ' **and** *finite* (*snd* ψ)
  shows *finite* (*snd* ψ')
  using *assms* **apply** (*induct rule*: *resolution.induct*, *auto simp add*: *inference-preserves-finite-snd*)
  using *inference-preserves-finite-snd snd-conv* **by** *metis*

lemma *rtranclp-resolution-finite-snd*:
  assumes *resolution*** ψ ψ' **and** *finite* (*snd* ψ)
  shows *finite* (*snd* ψ')
  using *assms* **by** (*induct rule*: *rtranclp-induct*, *auto simp add*: *resolution-finite-snd*)

lemma *resolution-always-simplified*:
 assumes *resolution* ψ ψ'
 shows *simplified* (*fst* ψ')
 using *assms* **by** (*induct rule*: *resolution.induct*)

(*auto simp add*: *full1-def full-def*)

lemma *tranclp-resolution-always-simplified*:
  assumes *tranclp resolution* ψ ψ'
  shows *simplified* (*fst* ψ')
  using *assms* **by** (*induct rule*: *tranclp.induct*, *auto simp add*: *resolution-always-simplified*)

lemma *resolution-atms-of*:
  assumes *resolution* ψ ψ' **and** *finite* (*fst* ψ)
  shows *atms-of-ms* (*fst* ψ') $\subseteq$ *atms-of-ms* (*fst* ψ)
  using *assms* **apply** (*induct rule*: *resolution.induct*)
    apply(*simp add*: *rtranclp-simplify-atms-of-ms tranclp-into-rtranclp full1-def*)
  by (*metis* (*no-types*, *lifting*) *contra-subsetD fst-conv full-def*
    inference-preserves-atms-of-ms rtranclp-simplify-atms-of-ms subsetI)

lemma *rtranclp-resolution-atms-of*:
  assumes $resolution^{**}$ ψ ψ' **and** *finite* (*fst* ψ)
  shows *atms-of-ms* (*fst* ψ') $\subseteq$ *atms-of-ms* (*fst* ψ)
  using *assms* **apply** (*induct rule*: *rtranclp-induct*)
  using *resolution-atms-of rtranclp-resolution-finite* **by** *blast*+

lemma *resolution-include*:
  assumes *res*: *resolution* ψ ψ' **and** *finite*: *finite* (*fst* ψ)
  shows *fst* ψ' $\subseteq$ *simple-clss* (*atms-of-ms* (*fst* ψ))
proof −
  have *finite'*: *finite* (*fst* ψ') **using** *local.finite res resolution-finite* **by** *blast*
  have *simplified* (*fst* ψ') **using** *res finite' resolution-always-simplified* **by** *blast*
  then have *fst* ψ' $\subseteq$ *simple-clss* (*atms-of-ms* (*fst* ψ'))
    using *simplified-in-simple-clss finite' simplified-imp-distinct-mset-tauto*[*of fst* ψ'] **by** *auto*
  moreover have *atms-of-ms* (*fst* ψ') $\subseteq$ *atms-of-ms* (*fst* ψ)
    using *res finite resolution-atms-of*[*of* ψ ψ'] **by** *auto*
  ultimately show *?thesis* **by** (*meson atms-of-ms-finite local.finite order.trans rev-finite-subset*
    simple-clss-mono)
qed

lemma *rtranclp-resolution-include*:
  assumes *res*: *tranclp resolution* ψ ψ' **and** *finite*: *finite* (*fst* ψ)
  shows *fst* ψ' $\subseteq$ *simple-clss* (*atms-of-ms* (*fst* ψ))
  using *assms* **apply** (*induct rule*: *tranclp.induct*)
    apply (*simp add*: *resolution-include*)
  by (*meson simple-clss-mono order-trans resolution-include*
    rtranclp-resolution-atms-of rtranclp-resolution-finite tranclp-into-rtranclp)

abbreviation *already-used-all-simple*
  :: (*'a literal multiset* $\times$ *'a literal multiset*) *set* $\Rightarrow$ *'a set* $\Rightarrow$ *bool* **where**
already-used-all-simple already-used vars $\equiv$
($\forall$ (*A*, *B*) $\in$ *already-used*. *simplified* $\{A\}$ $\wedge$ *simplified* $\{B\}$ $\wedge$ *atms-of A* $\subseteq$ *vars* $\wedge$ *atms-of B* $\subseteq$ *vars*)

lemma *already-used-all-simple-vars-incl*:
  assumes *vars* $\subseteq$ *vars'*
  shows *already-used-all-simple a vars* $\Longrightarrow$ *already-used-all-simple a vars'*
  using *assms* **by** *fast*

lemma *inference-clause-preserves-already-used-all-simple*:
  assumes *inference-clause S S'*
  and *already-used-all-simple* (*snd S*) *vars*

```
  and simplified (fst S)
  and atms-of-ms (fst S) ⊆ vars
  shows already-used-all-simple (snd (fst S ∪ {fst S'}, snd S')) vars
  using assms
proof (induct rule: inference-clause.induct)
  case (factoring L C N already-used)
  then show ?case by (simp add: simplified-in factoring-imp-simplify)
next
  case (resolution P C N D already-used) note H = this
  show ?case apply clarify
    proof −
      fix A B v
      assume (A, B) ∈ snd (fst (N, already-used)
        ∪ {fst (C + D, already-used ∪ {({#Pos P#} + C, {#Neg P#} + D)})},
           snd (C + D, already-used ∪ {({#Pos P#} + C, {#Neg P#} + D)}))
      then have (A, B) ∈ already-used ∨ (A, B) = ({#Pos P#} + C, {#Neg P#} + D) by auto
      moreover {
        assume (A, B) ∈ already-used
        then have simplified {A} ∧ simplified {B} ∧ atms-of A ⊆ vars ∧ atms-of B ⊆ vars
          using H(4) by auto
      }
      moreover {
        assume eq: (A, B) = ({#Pos P#} + C, {#Neg P#} + D)
        then have simplified {A} using simplified-in H(1,5) by auto
        moreover have simplified {B} using eq simplified-in H(2,5) by auto
        moreover have atms-of A ⊆ atms-of-ms N
          using eq H(1)
          using atms-of-atms-of-ms-mono[of A N] by auto
        moreover have atms-of B ⊆ atms-of-ms N
          using eq H(2) atms-of-atms-of-ms-mono[of B N] by auto
        ultimately have simplified {A} ∧ simplified {B} ∧ atms-of A ⊆ vars ∧ atms-of B ⊆ vars
          using H(6) by auto
      }
      ultimately show simplified {A} ∧ simplified {B} ∧ atms-of A ⊆ vars ∧ atms-of B ⊆ vars
        by fast
    qed
qed

lemma inference-preserves-already-used-all-simple:
  assumes inference S S'
  and already-used-all-simple (snd S) vars
  and simplified (fst S)
  and atms-of-ms (fst S) ⊆ vars
  shows already-used-all-simple (snd S') vars
  using assms
proof (induct rule: inference.induct)
  case (inference-step S clause already-used)
  then show ?case
    using inference-clause-preserves-already-used-all-simple[of S (clause, already-used) vars]
    by auto
qed

lemma already-used-all-simple-inv:
  assumes resolution S S'
  and already-used-all-simple (snd S) vars
  and atms-of-ms (fst S) ⊆ vars
```

shows *already-used-all-simple* (*snd* S') *vars*
using *assms*
proof (*induct rule*: *resolution.induct*)
case (*full1-simp* N N')
then show *?case* **by** *simp*
next
case (*inferring* N *already-used* N' *already-used'* N'')
then show *already-used-all-simple* (*snd* (N'', *already-used'*)) *vars*
using *inference-preserves-already-used-all-simple*[*of* (N, *already-used*)] **by** *simp*
qed

lemma *rtranclp-already-used-all-simple-inv*:
assumes *resolution*** S S'
and *already-used-all-simple* (*snd* S) *vars*
and *atms-of-ms* (*fst* S) $\subseteq$ *vars*
and *finite* (*fst* S)
shows *already-used-all-simple* (*snd* S') *vars*
using *assms*
proof (*induct rule*: *rtranclp-induct*)
case *base*
then show *?case* **by** *simp*
next
case (*step* S' S'') **note** *infstar* = *this*(*1*) **and** *IH* = *this*(*3*) **and** *res* = *this*(*2*) **and** *already* = *this*(*4*) **and** *atms* = *this*(*5*) **and** *finite* = *this*(*6*)
have *already-used-all-simple* (*snd* S') *vars* **using** *IH already atms finite* **by** *simp*
moreover have *atms-of-ms* (*fst* S') $\subseteq$ *atms-of-ms* (*fst* S)
by (*simp add*: *infstar local.finite rtranclp-resolution-atms-of*)
then have *atms-of-ms* (*fst* S') $\subseteq$ *vars* **using** *atms* **by** *auto*
ultimately show *?case*
using *already-used-all-simple-inv*[*OF res*] **by** *simp*
qed

lemma *inference-clause-simplified-already-used-subset*:
assumes *inference-clause* S S'
and *simplified* (*fst* S)
shows *snd* $S \subset$ *snd* S'
using *assms* **apply** (*induct rule*: *inference-clause.induct*)
using *factoring-imp-simplify* **apply** (*simp*; *blast*)
using *factoring-imp-simplify* **by** *force*

lemma *inference-simplified-already-used-subset*:
assumes *inference* S S'
and *simplified* (*fst* S)
shows *snd* $S \subset$ *snd* S'
using *assms* **apply** (*induct rule*: *inference.induct*)
by (*metis inference-clause-simplified-already-used-subset snd-conv*)

lemma *resolution-simplified-already-used-subset*:
assumes *resolution* S S'
and *simplified* (*fst* S)
shows *snd* $S \subset$ *snd* S'
using *assms* **apply** (*induct rule*: *resolution.induct*, *simp-all add*: *full1-def*)
apply (*meson tranclpD*)
by (*metis inference-simplified-already-used-subset fst-conv snd-conv*)

lemma *tranclp-resolution-simplified-already-used-subset*:

assumes *tranclp resolution S S'*
and *simplified (fst S)*
shows *snd S ⊂ snd S'*
using *assms* **apply** (*induct rule: tranclp.induct*)
using *resolution-simplified-already-used-subset* **apply** *metis*
by (*meson tranclp-resolution-always-simplified resolution-simplified-already-used-subset less-trans*)

abbreviation *already-used-top vars ≡ simple-clss vars × simple-clss vars*

lemma *already-used-all-simple-in-already-used-top*:
assumes *already-used-all-simple s vars* **and** *finite vars*
shows *s ⊆ already-used-top vars*
proof
fix *x*
assume *x-s: x ∈ s*
obtain *A B* **where** *x: x = (A, B)* **by** (*cases x, auto*)
then have *simplified {A}* **and** *atms-of A ⊆ vars* **using** *assms(1) x-s* **by** *fastforce+*
then have *A: A ∈ simple-clss vars*
using *simple-clss-mono[of atms-of A vars] x assms(2)*
simplified-imp-distinct-mset-tauto[of {A}]
distinct-mset-not-tautology-implies-in-simple-clss **by** *fast*
moreover have *simplified {B}* **and** *atms-of B ⊆ vars* **using** *assms(1) x-s x* **by** *fast+*
then have *B: B ∈ simple-clss vars*
using *simplified-imp-distinct-mset-tauto[of {B}]*
distinct-mset-not-tautology-implies-in-simple-clss
simple-clss-mono[of atms-of B vars] x assms(2) **by** *fast*
ultimately show *x ∈ simple-clss vars × simple-clss vars*
unfolding *x* **by** *auto*
qed

lemma *already-used-top-finite*:
assumes *finite vars*
shows *finite (already-used-top vars)*
using *simple-clss-finite assms* **by** *auto*

lemma *already-used-top-increasing*:
assumes *var ⊆ var'* **and** *finite var'*
shows *already-used-top var ⊆ already-used-top var'*
using *assms simple-clss-mono* **by** *auto*

lemma *already-used-all-simple-finite*:
fixes *s :: ('a literal multiset × 'a literal multiset) set* **and** *vars :: 'a set*
assumes *already-used-all-simple s vars* **and** *finite vars*
shows *finite s*
using *assms already-used-all-simple-in-already-used-top[OF assms(1)]*
rev-finite-subset[OF already-used-top-finite[of vars]] **by** *auto*

abbreviation *card-simple vars ψ ≡ card (already-used-top vars − ψ)*

lemma *resolution-card-simple-decreasing*:
assumes *res: resolution ψ ψ'*
and *a-u-s: already-used-all-simple (snd ψ) vars*
and *finite-v: finite vars*
and *finite-fst: finite (fst ψ)*
and *finite-snd: finite (snd ψ)*

and *simp*: *simplified* (*fst* ψ)
and *atms-of-ms* (*fst* ψ) $\subseteq$ *vars*
shows *card-simple vars* (*snd* ψ') $<$ *card-simple vars* (*snd* ψ)
proof –
let *?vars* = *vars*
let *?top* = *simple-clss ?vars* $\times$ *simple-clss ?vars*
have *1*: *card-simple vars* (*snd* ψ) = *card ?top* $-$ *card* (*snd* ψ)
using *card-Diff-subset finite-snd already-used-all-simple-in-already-used-top*[*OF a-u-s*]
finite-v **by** *metis*
have *a-u-s'*: *already-used-all-simple* (*snd* ψ') *vars*
using *already-used-all-simple-inv res a-u-s assms*(*7*) **by** *blast*
have *f*: *finite* (*snd* ψ') **using** *already-used-all-simple-finite a-u-s' finite-v* **by** *auto*
have *2*: *card-simple vars* (*snd* ψ') = *card ?top* $-$ *card* (*snd* ψ')
using *card-Diff-subset*[*OF f*] *already-used-all-simple-in-already-used-top*[*OF a-u-s' finite-v*]
by *auto*
have *card* (*already-used-top vars*) $\geq$ *card* (*snd* ψ')
using *already-used-all-simple-in-already-used-top*[*OF a-u-s' finite-v*]
card-mono[*of already-used-top vars snd* ψ'] *already-used-top-finite*[*OF finite-v*] **by** *metis*
then show *?thesis*
using *psubset-card-mono*[*OF f resolution-simplified-already-used-subset*[*OF res simp*]]
unfolding *1 2* **by** *linarith*
qed

lemma *tranclp-resolution-card-simple-decreasing*:
assumes *tranclp resolution* ψ ψ' **and** *finite-fst*: *finite* (*fst* ψ)
and *already-used-all-simple* (*snd* ψ) *vars*
and *atms-of-ms* (*fst* ψ) $\subseteq$ *vars*
and *finite-v*: *finite vars*
and *finite-snd*: *finite* (*snd* ψ)
and *simplified* (*fst* ψ)
shows *card-simple vars* (*snd* ψ') $<$ *card-simple vars* (*snd* ψ)
using *assms*
proof (*induct rule*: *tranclp-induct*)
case (*base* ψ')
then show *?case* **by** (*simp add*: *resolution-card-simple-decreasing*)
next
case (*step* ψ' ψ'') **note** *res* = *this*(*1*) **and** *res'* = *this*(*2*) **and** *a-u-s* = *this*(*5*) **and**
atms = *this*(*6*) **and** *f-v* = *this*(*7*) **and** *f-fst* = *this*(*4*) **and** *H* = *this*
then have *card-simple vars* (*snd* ψ') $<$ *card-simple vars* (*snd* ψ) **by** *auto*
moreover have *a-u-s'*: *already-used-all-simple* (*snd* ψ') *vars*
using *rtranclp-already-used-all-simple-inv*[*OF tranclp-into-rtranclp*[*OF res*] *a-u-s atms f-fst*] **.**
have *finite* (*fst* ψ')
by (*meson finite-fst res rtranclp-resolution-finite tranclp-into-rtranclp*)
moreover have *finite* (*snd* ψ') **using** *already-used-all-simple-finite*[*OF a-u-s' f-v*] **.**
moreover have *simplified* (*fst* ψ') **using** *res tranclp-resolution-always-simplified* **by** *blast*
moreover have *atms-of-ms* (*fst* ψ') $\subseteq$ *vars*
by (*meson atms f-fst order.trans res rtranclp-resolution-atms-of tranclp-into-rtranclp*)
ultimately show *?case*
using *resolution-card-simple-decreasing*[*OF res' a-u-s' f-v*] *f-v*
less-trans[*of card-simple vars* (*snd* ψ'') *card-simple vars* (*snd* ψ')
card-simple vars (*snd* ψ)]
by *blast*
qed

lemma *tranclp-resolution-card-simple-decreasing-2*:
assumes *tranclp resolution* ψ ψ'
and *finite-fst*: *finite* (*fst* ψ)
and *empty-snd*: *snd* ψ = {}
and *simplified* (*fst* ψ)
shows *card-simple* (*atms-of-ms* (*fst* ψ)) (*snd* ψ') < *card-simple* (*atms-of-ms* (*fst* ψ)) (*snd* ψ)
proof −
let *?vars* = *atms-of-ms* (*fst* ψ)
have *already-used-all-simple* (*snd* ψ) *?vars* **unfolding** *empty-snd* **by** *auto*
moreover have *atms-of-ms* (*fst* ψ) ⊆ *?vars* **by** *auto*
moreover have *finite-v*: *finite ?vars* **using** *finite-fst* **by** *auto*
moreover have *finite-snd*: *finite* (*snd* ψ) **unfolding** *empty-snd* **by** *auto*
ultimately show *?thesis*
using *assms*(*1*,*2*,*4*) *tranclp-resolution-card-simple-decreasing*[*of* ψ ψ'] **by** *presburger*
qed

Well-Foundness of the Relation

lemma *wf-simplified-resolution*:
assumes *f-vars*: *finite vars*
shows *wf* {(*y*:: ′*v*:: *linorder state*, *x*). (*atms-of-ms* (*fst x*) ⊆ *vars* ∧ *simplified* (*fst x*)
∧ *finite* (*snd x*) ∧ *finite* (*fst x*) ∧ *already-used-all-simple* (*snd x*) *vars*) ∧ *resolution x y*}
proof −
{
fix *a b* :: ′*v*::*linorder state*
assume (*b*, *a*) ∈ {(*y*, *x*). (*atms-of-ms* (*fst x*) ⊆ *vars* ∧ *simplified* (*fst x*) ∧ *finite* (*snd x*)
∧ *finite* (*fst x*) ∧ *already-used-all-simple* (*snd x*) *vars*) ∧ *resolution x y*}
then have
atms-of-ms (*fst a*) ⊆ *vars* **and**
simp: *simplified* (*fst a*) **and**
finite (*snd a*) **and**
finite (*fst a*) **and**
a-u-v: *already-used-all-simple* (*snd a*) *vars* **and**
res: *resolution a b* **by** *auto*
have *finite* (*already-used-top vars*) **using** *f-vars already-used-top-finite* **by** *blast*
moreover have *already-used-top vars* ⊆ *already-used-top vars* **by** *auto*
moreover have *snd b* ⊆ *already-used-top vars*
using *already-used-all-simple-in-already-used-top*[*of snd b vars*]
a-u-v already-used-all-simple-inv[*OF res*] ‹*finite* (*fst a*)› ‹*atms-of-ms* (*fst a*) ⊆ *vars*› *f-vars*
by *presburger*
moreover have *snd a* ⊂ *snd b* **using** *resolution-simplified-already-used-subset*[*OF res simp*] **.**
ultimately have *finite* (*already-used-top vars*) ∧ *already-used-top vars* ⊆ *already-used-top vars*
∧ *snd b* ⊆ *already-used-top vars* ∧ *snd a* ⊂ *snd b* **by** *metis*
}
then show *?thesis* **using** *wf-bounded-set*[*of* {(*y*:: ′*v*:: *linorder state*, *x*).
(*atms-of-ms* (*fst x*) ⊆ *vars*
∧ *simplified* (*fst x*) ∧ *finite* (*snd x*) ∧ *finite* (*fst x*)∧ *already-used-all-simple* (*snd x*) *vars*)
∧ *resolution x y*} λ-. *already-used-top vars snd*] **by** *auto*
qed

lemma *wf-simplified-resolution*′:
assumes *f-vars*: *finite vars*
shows *wf* {(*y*:: ′*v*:: *linorder state*, *x*). (*atms-of-ms* (*fst x*) ⊆ *vars* ∧ ¬*simplified* (*fst x*)
∧ *finite* (*snd x*) ∧ *finite* (*fst x*) ∧ *already-used-all-simple* (*snd x*) *vars*) ∧ *resolution x y*}
unfolding *wf-def*
apply (*simp add*: *resolution-always-simplified*)

by (*metis* (*mono-tags*, *hide-lams*) *fst-conv resolution-always-simplified*)

lemma *wf-resolution*:
assumes *f-vars*: *finite vars*
shows *wf* ({(*y*:: *'v*:: *linorder state*, *x*). (*atms-of-ms* (*fst x*) $\subseteq$ *vars* $\wedge$ *simplified* (*fst x*)
$\wedge$ *finite* (*snd x*) $\wedge$ *finite* (*fst x*) $\wedge$ *already-used-all-simple* (*snd x*) *vars*) $\wedge$ *resolution x y*}
$\cup$ {(*y*, *x*). (*atms-of-ms* (*fst x*) $\subseteq$ *vars* $\wedge$ $\neg$ *simplified* (*fst x*) $\wedge$ *finite* (*snd x*) $\wedge$ *finite* (*fst x*)
$\wedge$ *already-used-all-simple* (*snd x*) *vars*) $\wedge$ *resolution x y*}) (**is** *wf* (*?R* $\cup$ *?S*))
proof −
have *Domain ?R Int Range ?S* = {} **using** *resolution-always-simplified* **by** *auto blast*
then show *wf* (*?R* $\cup$ *?S*)
using *wf-simplified-resolution*[*OF f-vars*] *wf-simplified-resolution'*[*OF f-vars*] *wf-Un*[*of ?R ?S*]
by *fast*
qed

lemma *rtrancp-simplify-already-used-inv*:
assumes *simplify*** *S S'*
and *already-used-inv* (*S*, *N*)
shows *already-used-inv* (*S'*, *N*)
using *assms* **apply** *induction*
using *simplify-preserves-already-used-inv* **by** *fast+*

lemma *full1-simplify-already-used-inv*:
assumes *full1 simplify S S'*
and *already-used-inv* (*S*, *N*)
shows *already-used-inv* (*S'*, *N*)
using *assms tranclp-into-rtranclp*[*of simplify S S'*] *rtrancp-simplify-already-used-inv*
unfolding *full1-def* **by** *fast*

lemma *full-simplify-already-used-inv*:
assumes *full simplify S S'*
and *already-used-inv* (*S*, *N*)
shows *already-used-inv* (*S'*, *N*)
using *assms rtrancp-simplify-already-used-inv* **unfolding** *full-def* **by** *fast*
lemma *resolution-already-used-inv*:
assumes *resolution S S'*
and *already-used-inv S*
shows *already-used-inv S'*
using *assms*
proof *induction*
case (*full1-simp N N' already-used*)
then show *?case* **using** *full1-simplify-already-used-inv* **by** *fast*
next
case (*inferring N already-used N' already-used' N''*) **note** *inf* = *this*(*1*) **and** *full* = *this*(*3*) **and**
a-u-v = *this*(*4*)
then show *?case*
using *inference-preserves-already-used-inv*[*OF inf a-u-v*] *full-simplify-already-used-inv full*
by *fast*
qed

lemma *rtranclp-resolution-already-used-inv*:
assumes *resolution*** *S S'*
and *already-used-inv S*
shows *already-used-inv S'*
using *assms* **apply** *induction*
using *resolution-already-used-inv* **by** *fast+*

lemma *rtanclp-simplify-preserves-unsat*:
assumes *simplify*** ψ ψ'
shows *satisfiable* ψ' $\longrightarrow$ *satisfiable* ψ
using *assms* **apply** *induction*
using *simplify-clause-preserves-sat* **by** *blast+*

lemma *full1-simplify-preserves-unsat*:
assumes *full1 simplify* ψ ψ'
shows *satisfiable* ψ' $\longrightarrow$ *satisfiable* ψ
using *assms rtanclp-simplify-preserves-unsat*[*of* ψ ψ'] *tranclp-into-rtranclp*
unfolding *full1-def* **by** *metis*

lemma *full-simplify-preserves-unsat*:
assumes *full simplify* ψ ψ'
shows *satisfiable* ψ' $\longrightarrow$ *satisfiable* ψ
using *assms rtanclp-simplify-preserves-unsat*[*of* ψ ψ'] **unfolding** *full-def* **by** *metis*

lemma *resolution-preserves-unsat*:
assumes *resolution* ψ ψ'
shows *satisfiable* (*fst* ψ') $\longrightarrow$ *satisfiable* (*fst* ψ)
using *assms* **apply** (*induct rule*: *resolution.induct*)
using *full1-simplify-preserves-unsat* **apply** (*metis fst-conv*)
using *full-simplify-preserves-unsat simplify-preserves-unsat* **by** *fastforce*

lemma *rtranclp-resolution-preserves-unsat*:
assumes *resolution*** ψ ψ'
shows *satisfiable* (*fst* ψ') $\longrightarrow$ *satisfiable* (*fst* ψ)
using *assms* **apply** *induction*
using *resolution-preserves-unsat* **by** *fast+*

lemma *rtranclp-simplify-preserve-partial-tree*:
assumes *simplify*** N N'
and *partial-interps* t I N
shows *partial-interps* t I N'
using *assms* **apply** (*induction*, *simp*)
using *simplify-preserve-partial-tree* **by** *metis*

lemma *full1-simplify-preserve-partial-tree*:
assumes *full1 simplify* N N'
and *partial-interps* t I N
shows *partial-interps* t I N'
using *assms rtranclp-simplify-preserve-partial-tree*[*of* N N' t I] *tranclp-into-rtranclp*
unfolding *full1-def* **by** *fast*

lemma *full-simplify-preserve-partial-tree*:
assumes *full simplify* N N'
and *partial-interps* t I N
shows *partial-interps* t I N'
using *assms rtranclp-simplify-preserve-partial-tree*[*of* N N' t I] *tranclp-into-rtranclp*
unfolding *full-def* **by** *fast*

lemma *resolution-preserve-partial-tree*:
assumes *resolution* S S'
and *partial-interps* t I (*fst* S)
shows *partial-interps* t I (*fst* S')

using *assms* **apply** *induction*
using *full1-simplify-preserve-partial-tree fst-conv* **apply** *metis*
using *full-simplify-preserve-partial-tree inference-preserve-partial-tree* **by** *fastforce*

lemma *rtranclp-resolution-preserve-partial-tree*:
assumes $resolution^{**}\ S\ S'$
and *partial-interps t I* (*fst S*)
shows *partial-interps t I* (*fst S'*)
using *assms* **apply** *induction*
using *resolution-preserve-partial-tree* **by** *fast+*
thm *nat-less-induct nat.induct*

lemma *nat-ge-induct*[*case-names 0 Suc*]:
assumes *P 0*
and $\bigwedge n.\ (\bigwedge m.\ m<Suc\ n \Longrightarrow P\ m) \Longrightarrow P\ (Suc\ n)$
shows *P n*
using *assms* **apply** (*induct rule*: *nat-less-induct*)
by (*rename-tac n*, *case-tac n*) *auto*

lemma *wf-always-more-step-False*:
assumes *wf R*
shows $(\forall x.\ \exists z.\ (z, x)\in R) \Longrightarrow False$
using *assms* **unfolding** *wf-def* **by** (*meson Domain.DomainI assms wfE-min*)

lemma *finite-finite-mset-element-of-mset*[*simp*]:
assumes *finite N*
shows *finite* $\{f\ \varphi\ L\ |\varphi\ L.\ \varphi \in N \wedge L \in\#\ \varphi \wedge P\ \varphi\ L\}$
using *assms*
proof (*induction N rule*: *finite-induct*)
case *empty*
show *?case* **by** *auto*
next
case (*insert x N*) **note** *finite* = *this*(*1*) **and** *IH* = *this*(*3*)
have $\{f\ \varphi\ L\ |\varphi\ L.\ (\varphi = x \vee \varphi \in N) \wedge L \in\#\ \varphi \wedge P\ \varphi\ L\} \subseteq \{f\ x\ L\ |\ L.\ L \in\#\ x \wedge P\ x\ L\}$
$\cup \{f\ \varphi\ L\ |\varphi\ L.\ \varphi \in N \wedge L \in\#\ \varphi \wedge P\ \varphi\ L\}$ **by** *auto*
moreover have *finite* $\{f\ x\ L\ |\ L.\ L \in\#\ x\}$ **by** *auto*
ultimately show *?case* **using** *IH finite-subset* **by** *fastforce*
qed

definition *sum-count-ge-2* :: *'a multiset set* ⇒ *nat* (Ξ) **where**
sum-count-ge-2 ≡ *folding.F* ($\lambda\varphi.$ (+)(*sum-mset* $\{\#count\ \varphi\ L\ |L \in\#\ \varphi.\ 2 \le count\ \varphi\ L\#\}$)) *0*

interpretation *sum-count-ge-2*:
folding $\lambda\varphi.$ (+)(*sum-mset* $\{\#count\ \varphi\ L\ |L \in\#\ \varphi.\ 2 \le count\ \varphi\ L\#\}$) *0*
rewrites
folding.F ($\lambda\varphi.$ (+)(*sum-mset* $\{\#count\ \varphi\ L\ |L \in\#\ \varphi.\ 2 \le count\ \varphi\ L\#\}$)) *0* = *sum-count-ge-2*
proof −
show *folding* ($\lambda\varphi.$ (+) (*sum-mset* (*image-mset* (*count* φ) $\{\#\ L \in\#\ \varphi.\ 2 \le count\ \varphi\ L\#\}$)))
by *standard auto*
then interpret *sum-count-ge-2*:
folding $\lambda\varphi.$ (+)(*sum-mset* $\{\#count\ \varphi\ L\ |L \in\#\ \varphi.\ 2 \le count\ \varphi\ L\#\}$) *0* **.**
show *folding.F* ($\lambda\varphi.$ (+) (*sum-mset* (*image-mset* (*count* φ) $\{\#\ L \in\#\ \varphi.\ 2 \le count\ \varphi\ L\#\}$))) *0*
= *sum-count-ge-2* **by** (*auto simp add*: *sum-count-ge-2-def*)
qed

```
lemma finite-incl-le-setsum:
 finite (B::'a multiset set) ⟹ A ⊆ B ⟹ Ξ A ≤ Ξ B
proof (induction arbitrary:A rule: finite-induct)
  case empty
  then show ?case by simp
next
  case (insert a F) note finite = this(1) and aF = this(2) and IH = this(3) and AF = this(4)
  show ?case
    proof (cases a ∈ A)
      assume a ∉ A
      then have A ⊆ F using AF by auto
      then show ?case using IH[of A] by (simp add: aF local.finite)
    next
      assume aA: a ∈ A
      then have A − {a} ⊆ F using AF by auto
      then have Ξ (A − {a}) ≤ Ξ F using IH by blast
      then show ?case
        proof −
          obtain nn :: nat ⇒ nat ⇒ nat where
            ∀ x0 x1. (∃ v2. x0 = x1 + v2) = (x0 = x1 + nn x0 x1)
            by moura
          then have Ξ F = Ξ (A − {a}) + nn (Ξ F) (Ξ (A − {a}))
            by (meson ‹Ξ (A − {a}) ≤ Ξ F› le-iff-add)
          then show ?thesis
            by (metis (no-types) le-iff-add aA aF add.assoc finite.insertI finite-subset
              insert.prems local.finite sum-count-ge-2.insert sum-count-ge-2.remove)
        qed
    qed
qed

lemma simplify-finite-measure-decrease:
  simplify N N' ⟹ finite N ⟹ card N' + Ξ N' < card N + Ξ N
proof (induction rule: simplify.induct)
  case (tautology-deletion P A) note an = this(1) and fin = this(2)
  let ?N' = N − {add-mset (Pos P) (add-mset (Neg P) A)}
  have card ?N' < card N
    by (meson card-Diff1-less tautology-deletion.hyps tautology-deletion.prems)
  moreover have ?N' ⊆ N by auto
  then have sum-count-ge-2 ?N' ≤ sum-count-ge-2 N using finite-incl-le-setsum[OF fin] by blast
  ultimately show ?case by linarith
next
  case (condensation L A) note AN = this(1) and fin = this(2)
  let ?C' = add-mset L A
  let ?C = add-mset L ?C'
  let ?N' = N − {?C} ∪ {?C'}
  have card ?N' ≤ card N
    using AN by (metis (no-types, lifting) Diff-subset Un-empty-right Un-insert-right card.remove
      card-insert-if card-mono fin finite-Diff order-refl)
  moreover have Ξ {?C'} < Ξ {?C}
  proof −
    have mset-decomp:
      {# La ∈# A. (L = La ⟶ La ∈# A) ∧ (L ≠ La ⟶ 2 ≤ count A La)#}
        = {# La ∈# A. L ≠ La ∧ 2 ≤ count A La#} +
          {# La ∈# A. L = La ∧ Suc 0 ≤ count A L#}
      by (auto simp: multiset-eq-iff ac-simps)
```

have *mset-decomp2*: $\{\# La \in\# A.\ L \neq La \longrightarrow 2 \leq count\ A\ La\#\} =$
$\{\# La \in\# A.\ L \neq La \wedge 2 \leq count\ A\ La\#\} + replicate\text{-}mset\ (count\ A\ L)\ L$
by (*auto simp*: *multiset-eq-iff*)
have $*$: $(\sum x{\in}\#B.\ if\ L = x\ then\ Suc\ (count\ A\ x)\ else\ count\ A\ x) \leq$
$(\sum x{\in}\#B.\ if\ L = x\ then\ Suc\ (count\ (add\text{-}mset\ L\ A)\ x)\ else\ count\ (add\text{-}mset\ L\ A)\ x)$
for *B*
by (*auto intro*!: *sum-mset-mono*)
show *?thesis*
using $*[of\ \{\#La \in\# A.\ L \neq La \wedge 2 \leq count\ A\ La\#\}]$
by (*auto simp*: *mset-decomp mset-decomp2 filter-mset-eq*)
qed
have $\Xi\ ?N' < \Xi\ N$
proof *cases*
assume *a1*: $?C' \in N$
then show *?thesis*
proof $-$
have *f2*: $\bigwedge m\ M.\ insert\ (m{::}'a\ literal\ multiset)\ (M - \{m\}) = M \cup \{\} \vee m \notin M$
using *Un-empty-right insert-Diff* **by** *blast*
have *f3*: $\bigwedge m\ M\ Ma.\ insert\ (m{::}'a\ literal\ multiset)\ M - insert\ m\ Ma = M - insert\ m\ Ma$
by *simp*
then have *f4*: $\bigwedge M\ m.\ M - \{m{::}'a\ literal\ multiset\} = M \cup \{\} \vee m \in M$
using *Diff-insert-absorb Un-empty-right* **by** *fastforce*
have *f5*: $insert\ ?C\ N = N$
using *f3 f2 Un-empty-right condensation.hyps insert-iff* **by** *fastforce*
have $\bigwedge m\ M.\ insert\ (m{::}'a\ literal\ multiset)\ M = M \cup \{\} \vee m \notin M$
using *f3 f2 Un-empty-right add.right-neutral insert-iff* **by** *fastforce*
then have $\Xi\ (N - \{?C\}) < \Xi\ N$
using *f5 f4* **by** (*metis Un-empty-right* $\langle \Xi\ \{?C'\} < \Xi\ \{?C\} \rangle$
add.right-neutral add-diff-cancel-left′ add-gr-0 diff-less fin finite.emptyI not-le
sum-count-ge-2.empty sum-count-ge-2.insert-remove trans-le-add2)
then show *?thesis*
using *f3 f2 a1* **by** (*metis* (*no-types*) *Un-empty-right Un-insert-right condensation.hyps*
insert-iff multi-self-add-other-not-self)
qed
next
assume $?C' \notin N$
have *mset-decomp*:
$\{\# La \in\# A.\ (L = La \longrightarrow Suc\ 0 \leq count\ A\ La) \wedge (L \neq La \longrightarrow 2 \leq count\ A\ La)\#\}$
$= \{\# La \in\# A.\ L \neq La \wedge 2 \leq count\ A\ La\#\} +$
$\{\# La \in\# A.\ L = La \wedge Suc\ 0 \leq count\ A\ L\#\}$
by (*auto simp*: *multiset-eq-iff ac-simps*)
have *mset-decomp2*: $\{\# La \in\# A.\ L \neq La \longrightarrow 2 \leq count\ A\ La\#\} =$
$\{\# La \in\# A.\ L \neq La \wedge 2 \leq count\ A\ La\#\} + replicate\text{-}mset\ (count\ A\ L)\ L$
by (*auto simp*: *multiset-eq-iff*)

show *?thesis*
using $\langle \Xi\ \{?C'\} < \Xi\ \{?C\} \rangle$ *condensation.hyps fin*
sum-count-ge-2.remove[*of* - *?C*] $\langle ?C' \notin N \rangle$
by (*auto simp*: *mset-decomp mset-decomp2 filter-mset-eq*)
qed
ultimately show *?case* **by** *linarith*
next
case (*subsumption A B*) **note** $AN = this(1)$ **and** $AB = this(2)$ **and** $BN = this(3)$ **and** $fin = this(4)$
have $card\ (N - \{B\}) < card\ N$ **using** *BN* **by** (*meson card-Diff1-less subsumption.prems*)
moreover have $\Xi\ (N - \{B\}) \leq \Xi\ N$
by (*simp add*: *Diff-subset finite-incl-le-setsum subsumption.prems*)

ultimately show *?case* **by** *linarith*
qed

lemma *simplify-terminates*:
wf $\{(N', N).$ *finite* $N \wedge$ *simplify* $N\ N'\}$
apply (*rule wfP-if-measure*[*of finite simplify* $\lambda N.$ *card* $N + \Xi\ N$])
using *simplify-finite-measure-decrease* **by** *blast*

lemma *wf-terminates*:
assumes *wf r*
shows $\exists N'.(N', N) \in r^* \wedge (\forall N''. (N'', N') \notin r)$
proof –
let $?P = \lambda N. (\exists N'.(N', N) \in r^* \wedge (\forall N''. (N'', N') \notin r))$
have $\forall x. (\forall y. (y, x) \in r \longrightarrow ?P\ y) \longrightarrow ?P\ x$
proof *clarify*
fix x
assume H: $\forall y. (y, x) \in r \longrightarrow ?P\ y$
{ assume $\exists y. (y, x) \in r$
then obtain y **where** y: $(y, x) \in r$ **by** *blast*
then have $?P\ y$ **using** H **by** *blast*
then have $?P\ x$ **using** y **by** (*meson rtrancl.rtrancl-into-rtrancl*)
}
moreover {
assume $\neg(\exists y. (y, x) \in r)$
then have $?P\ x$ **by** *auto*
}
ultimately show $?P\ x$ **by** *blast*
qed
moreover have $(\forall x. (\forall y. (y, x) \in r \longrightarrow ?P\ y) \longrightarrow ?P\ x) \longrightarrow$ *All* $?P$
using *assms* **unfolding** *wf-def* **by** (*rule allE*)
ultimately have *All* $?P$ **by** *blast*
then show $?P\ N$ **by** *blast*
qed

lemma *rtranclp-simplify-terminates*:
assumes *fin*: *finite* N
shows $\exists N'.$ *simplify*** $N\ N' \wedge$ *simplified* N'
proof –
have H: $\{(N', N).$ *finite* $N \wedge$ *simplify* $N\ N'\} = \{(N', N).$ *simplify* $N\ N' \wedge$ *finite* $N\}$ **by** *auto*
then have *wf*: *wf* $\{(N', N).$ *simplify* $N\ N' \wedge$ *finite* $N\}$
using *simplify-terminates* **by** (*simp add*: H)
obtain N' **where** N': $(N', N) \in \{(b, a).$ *simplify* $a\ b \wedge$ *finite* $a\}^*$ **and**
more: $\forall N''. (N'', N') \notin \{(b, a).$ *simplify* $a\ b \wedge$ *finite* $a\}$
using *Prop-Resolution.wf-terminates*[*OF wf*, *of* N] **by** *blast*
have *1*: *simplify*** $N\ N'$
using N' **by** (*induction rule*: *rtrancl.induct*) *auto*
then have *finite* N' **using** *fin rtranclp-simplify-preserves-finite* **by** *blast*
then have *2*: $\forall N''. \neg$*simplify* $N'\ N''$ **using** *more* **by** *auto*

show *?thesis* **using** *1 2* **by** *blast*
qed

lemma *finite-simplified-full1-simp*:
assumes *finite* N
shows *simplified* $N \vee (\exists N'.$ *full1 simplify* $N\ N')$

using *rtranclp-simplify-terminates*[*OF assms*] **unfolding** *full1-def*
by (*metis Nitpick.rtranclp-unfold*)

lemma *finite-simplified-full-simp*:
assumes *finite N*
shows $\exists N'$. *full simplify* N N'
using *rtranclp-simplify-terminates*[*OF assms*] **unfolding** *full-def* **by** *metis*

lemma *can-decrease-tree-size-resolution*:
fixes ψ :: *'v state* **and** *tree* :: *'v sem-tree*
assumes *finite* (*fst* ψ) **and** *already-used-inv* ψ
and *partial-interps tree I* (*fst* ψ)
and *simplified* (*fst* ψ)
shows $\exists$(*tree'*:: *'v sem-tree*) ψ'. *resolution*** ψ ψ' $\wedge$ *partial-interps tree' I* (*fst* ψ')
$\wedge$ (*sem-tree-size tree'* $<$ *sem-tree-size tree* $\vee$ *sem-tree-size tree* $=$ *0*)
using *assms*
proof (*induct arbitrary*: *I rule*: *sem-tree-size*)
case (*bigger xs I*) **note** *IH* = *this*(*1*) **and** *finite* = *this*(*2*) **and** *a-u-i* = *this*(*3*) **and** *part* = *this*(*4*)
and *simp* = *this*(*5*)

{ assume *sem-tree-size xs* = *0*
then have *?case* **using** *part* **by** *blast*
}

moreover {
assume *sn0*: *sem-tree-size xs* > *0*
obtain *ag ad v* **where** *xs*: *xs* = *Node v ag ad* **using** *sn0* **by** (*cases xs*, *auto*)
{
assume *sem-tree-size ag* = *0* $\wedge$ *sem-tree-size ad* = *0*
then have *ag*: *ag* = *Leaf* **and** *ad*: *ad* = *Leaf* **by** (*cases ag*, *auto*, *cases ad*, *auto*)

then obtain χ χ' **where**
χ: $\neg$ *I* $\cup$ {*Pos v*} $\models$ χ **and**
totχ: *total-over-m* (*I* $\cup$ {*Pos v*}) {χ} **and**
$\chi\psi$: $\chi \in$ *fst* ψ **and**
χ': $\neg$ *I* $\cup$ {*Neg v*} $\models$ χ' **and**
totχ': *total-over-m* (*I* $\cup$ {*Neg v*}) {χ'} **and** $\chi'\psi$: $\chi' \in$ *fst* ψ
using *part* **unfolding** *xs* **by** *auto*
have *Posv*: *Pos v* $\notin$# χ **using** χ **unfolding** *true-cls-def true-lit-def* **by** *auto*
have *Negv*: *Neg v* $\notin$# χ' **using** χ' **unfolding** *true-cls-def true-lit-def* **by** *auto*
{
assume *Neg*χ: *Neg v* $\notin$# χ
then have $\neg$ *I* $\models$ χ **using** χ *Posv* **unfolding** *true-cls-def true-lit-def* **by** *auto*
moreover have *total-over-m I* {χ}
using *Posv Neg*χ *atm-imp-pos-or-neg-lit tot*χ **unfolding** *total-over-m-def total-over-set-def*
by *fastforce*
ultimately have *partial-interps Leaf I* (*fst* ψ)
and *sem-tree-size Leaf* < *sem-tree-size xs*
and *resolution*** ψ ψ
unfolding *xs* **by** (*auto simp add*: $\chi\psi$)
}
moreover {
assume *Pos*χ: *Pos v* $\notin$# χ'
then have *I*χ: $\neg$ *I* $\models$ χ' **using** χ' *Posv* **unfolding** *true-cls-def true-lit-def* **by** *auto*
moreover have *total-over-m I* {χ'}
using *Negv Pos*χ *atm-imp-pos-or-neg-lit tot*χ'

unfolding *total-over-m-def total-over-set-def* **by** *fastforce*
ultimately have *partial-interps Leaf I* $(fst\ \psi)$
and *sem-tree-size Leaf* $<$ *sem-tree-size xs*
and *resolution*** $\psi\ \psi$
using $\chi'\psi\ I\chi$ **unfolding** *xs* **by** *auto*
}
moreover {
assume *neg*: *Neg v* $\in\#\ \chi$ **and** *pos*: *Pos v* $\in\#\ \chi'$
have *count* χ (*Neg v*) = *1*
using *simplified-count*[*OF simp* $\chi\psi$] *neg*
by (*simp add*: *dual-order.antisym*)
have *count* χ' (*Pos v*) = *1*
using *simplified-count*[*OF simp* $\chi'\psi$] *pos*
by (*simp add*: *dual-order.antisym*)

obtain *C* **where** χC: χ = *add-mset* (*Neg v*) *C* **and** *negC*: *Neg v* $\notin\#$ *C* **and** *posC*: *Pos v* $\notin\#$
C
by (*metis* (*no-types*, *lifting*) *One-nat-def Posv* ‹*count* χ (*Neg v*) = *1*›
‹*count* χ' (*Pos v*) = *1*› *count-add-mset count-greater-eq-Suc-zero-iff insert-DiffM*
le-numeral-extra(*2*) *nat.inject pos*)

obtain C' **where**
$\chi C'$: χ' = *add-mset* (*Pos v*) C' **and**
posC': *Pos v* $\notin\#$ C' **and**
negC': *Neg v* $\notin\#$ C'
by (*metis* (*no-types*, *lifting*) *Negv One-nat-def* ‹*count* χ' (*Pos v*) = *1*› *count-add-mset*
count-eq-zero-iff mset-add nat.inject pos)

have *totC*: *total-over-m I* $\{C\}$
using *tot*χ *tot-over-m-remove*[*of I Pos v C*] *negC posC* **unfolding** χC **by** *auto*
have *totC'*: *total-over-m I* $\{C'\}$
using *tot*χ' *total-over-m-sum tot-over-m-remove*[*of I Neg v C'*] *negC' posC'*
unfolding $\chi C'$ **by** *auto*
have $\neg\ I \models C + C'$
using $\chi\ \chi'\ \chi C\ \chi C'$ **by** *auto*
then have *part-I-*ψ''': *partial-interps Leaf I* $(fst\ \psi \cup \{C + C'\})$
using *totC totC'* ‹$\neg\ I \models C + C'$› **by** (*metis Un-insert-right insertI1*
partial-interps.simps(*1*) *total-over-m-sum*)
{
assume (*add-mset* (*Pos v*) C', *add-mset* (*Neg v*) *C*) $\notin$ *snd* ψ
then have *inf''*: *inference* ψ $(fst\ \psi \cup \{C + C'\}, snd\ \psi \cup \{(\chi', \chi)\})$
by (*metis* $\chi'\psi\ \chi C\ \chi C'\ \chi\psi$ *add-mset-add-single inference-clause.resolution*
inference-step prod.collapse union-commute)
obtain N' **where** *full*: *full simplify* $(fst\ \psi \cup \{C + C'\})$ N'
by (*metis finite-simplified-full-simp fst-conv inf'' inference-preserves-finite*
local.finite)
have *resolution* ψ $(N', snd\ \psi \cup \{(\chi', \chi)\})$
using *resolution.intros*(*2*)[*OF - simp full*, *of snd* ψ *snd* $\psi \cup \{(\chi', \chi)\}$] *inf''*
by (*metis surjective-pairing*)
moreover have *partial-interps Leaf I* N'
using *full-simplify-preserve-partial-tree*[*OF full part-I-*ψ'''] **.**
moreover have *sem-tree-size Leaf* $<$ *sem-tree-size xs* **unfolding** *xs* **by** *auto*
ultimately have *?case*
by (*metis* (*no-types*) *prod.sel*(*1*) *rtranclp.rtrancl-into-rtrancl rtranclp.rtrancl-refl*)
}
moreover {

assume a: $(\{\#Pos\ v\#\} + C', \{\#Neg\ v\#\} + C) \in snd\ \psi$
then have $(\exists \chi \in fst\ \psi.\ (\forall I.\ total\text{-}over\text{-}m\ I\ \{C+C'\} \longrightarrow total\text{-}over\text{-}m\ I\ \{\chi\})$
$\wedge (\forall I.\ total\text{-}over\text{-}m\ I\ \{\chi\} \longrightarrow I \models \chi \longrightarrow I \models C' + C)) \vee tautology\ (C' + C)$
proof $-$
obtain p **where** p: $Pos\ p \in\#\ (\{\#Pos\ v\#\} + C') \wedge Neg\ p \in\#\ (\{\#Neg\ v\#\} + C)$
$\wedge((\exists \chi{\in}fst\ \psi.\ (\forall I.\ total\text{-}over\text{-}m\ I\ \{(\{\#Pos\ v\#\} + C') - \{\#Pos\ p\#\} + ((\{\#Neg\ v\#\} + C) - \{\#Neg\ p\#\})\} \longrightarrow total\text{-}over\text{-}m\ I\ \{\chi\}) \wedge (\forall I.\ total\text{-}over\text{-}m\ I\ \{\chi\} \longrightarrow I \models \chi \longrightarrow I \models (\{\#Pos\ v\#\} + C') - \{\#Pos\ p\#\} + ((\{\#Neg\ v\#\} + C) - \{\#Neg\ p\#\}))) \vee tautology\ ((\{\#Pos\ v\#\} + C') - \{\#Pos\ p\#\} + ((\{\#Neg\ v\#\} + C) - \{\#Neg\ p\#\})))$
using a **by** $(blast\ intro$: $allE[OF\ a\text{-}u\text{-}i[unfolded\ subsumes\text{-}def\ Ball\text{-}def]$,
$of\ \ (\{\#Pos\ v\#\} + C', \{\#Neg\ v\#\} + C)])$
{ **assume** $p \neq v$
then have $Pos\ p \in\#\ C' \wedge Neg\ p \in\#\ C$ **using** p **by** *force*
then have *?thesis* **by** *auto*
}
moreover {
assume $p = v$
then have *?thesis* **using** p **by** $(metis\ add.commute\ add\text{-}diff\text{-}cancel\text{-}left')$
}
ultimately show *?thesis* **by** *auto*
qed
moreover {
assume $\exists \chi \in fst\ \psi.\ (\forall I.\ total\text{-}over\text{-}m\ I\ \{C+C'\} \longrightarrow total\text{-}over\text{-}m\ I\ \{\chi\})$
$\wedge (\forall I.\ total\text{-}over\text{-}m\ I\ \{\chi\} \longrightarrow I \models \chi \longrightarrow I \models C' + C)$
then obtain ϑ **where**
ϑ: $\vartheta \in fst\ \psi$ **and**
$tot\text{-}\vartheta\text{-}CC'$: $\forall I.\ total\text{-}over\text{-}m\ I\ \{C+C'\} \longrightarrow total\text{-}over\text{-}m\ I\ \{\vartheta\}$ **and**
$\vartheta\text{-}inv$: $\forall I.\ total\text{-}over\text{-}m\ I\ \{\vartheta\} \longrightarrow I \models \vartheta \longrightarrow I \models C' + C$ **by** *blast*
have $partial\text{-}interps\ Leaf\ I\ (fst\ \psi)$
using $tot\text{-}\vartheta\text{-}CC'\ \vartheta\ \vartheta\text{-}inv\ totC\ totC'\ \langle\neg\ I \models C + C'\rangle\ total\text{-}over\text{-}m\text{-}sum$ **by** *fastforce*
moreover have $sem\text{-}tree\text{-}size\ Leaf < sem\text{-}tree\text{-}size\ xs$ **unfolding** xs **by** *auto*
ultimately have *?case* **by** *blast*
}
moreover {
assume $tautCC'$: $tautology\ (C' + C)$
have $total\text{-}over\text{-}m\ I\ \{C'+C\}$ **using** $totC\ totC'\ total\text{-}over\text{-}m\text{-}sum$ **by** *auto*
then have $\neg tautology\ (C' + C)$
using $\langle\neg\ I \models C + C'\rangle$ **unfolding** $add.commute[of\ C\ C']\ total\text{-}over\text{-}m\text{-}def$
unfolding $tautology\text{-}def$ **by** *auto*
then have *False* **using** $tautCC'$ **unfolding** $tautology\text{-}def$ **by** *auto*
}
ultimately have *?case* **by** *auto*
}
ultimately have *?case* **by** *auto*
}
ultimately have *?case* **using** *part* **by** $(metis\ (no\text{-}types)\ sem\text{-}tree\text{-}size.simps(1))$
}
moreover {
assume $size\text{-}ag$: $sem\text{-}tree\text{-}size\ ag > 0$
have $sem\text{-}tree\text{-}size\ ag < sem\text{-}tree\text{-}size\ xs$ **unfolding** xs **by** *auto*
moreover have $partial\text{-}interps\ ag\ (I \cup \{Pos\ v\})\ (fst\ \psi)$
and $partad$: $partial\text{-}interps\ ad\ (I \cup \{Neg\ v\})\ (fst\ \psi)$
using $part\ partial\text{-}interps.simps(2)$ **unfolding** xs **by** $metis+$
moreover
have $sem\text{-}tree\text{-}size\ ag < sem\text{-}tree\text{-}size\ xs \Longrightarrow finite\ (fst\ \psi) \Longrightarrow already\text{-}used\text{-}inv\ \psi$
$\Longrightarrow partial\text{-}interps\ ag\ (I \cup \{Pos\ v\})\ (fst\ \psi) \Longrightarrow simplified\ (fst\ \psi)$

```
        ⟹ ∃tree' ψ'. resolution** ψ ψ' ∧ partial-interps tree' (I ∪ {Pos v}) (fst ψ')
            ∧ (sem-tree-size tree' < sem-tree-size ag ∨ sem-tree-size ag = 0)
        using IH[of ag I ∪ {Pos v}]  by auto
      ultimately obtain ψ' :: 'v state and tree' :: 'v sem-tree where
        inf: resolution** ψ ψ'
        and part: partial-interps tree' (I ∪ {Pos v}) (fst ψ')
        and size: sem-tree-size tree' < sem-tree-size ag ∨ sem-tree-size ag = 0
        using finite part rtranclp.rtrancl-refl a-u-i simp by blast

      have partial-interps ad (I ∪ {Neg v}) (fst ψ')
        using rtranclp-resolution-preserve-partial-tree inf partad by fast
      then have partial-interps (Node v tree' ad) I (fst ψ') using part by auto
      then have ?case using inf size size-ag part unfolding xs by fastforce
    }
    moreover {
      assume size-ad: sem-tree-size ad > 0
      have sem-tree-size ad < sem-tree-size xs unfolding xs by auto
      moreover
        have
          partag: partial-interps ag (I ∪ {Pos v}) (fst ψ) and
          partial-interps ad (I ∪ {Neg v}) (fst ψ)
          using part partial-interps.simps(2) unfolding xs by metis+
      moreover have sem-tree-size ad < sem-tree-size xs ⟶ finite (fst ψ) ⟶ already-used-inv ψ
        ⟶ ( partial-interps ad (I ∪ {Neg v}) (fst ψ) ⟶ simplified (fst ψ)
        ⟶ (∃tree' ψ'. resolution** ψ ψ' ∧ partial-interps tree' (I ∪ {Neg v}) (fst ψ')
            ∧ (sem-tree-size tree' < sem-tree-size ad ∨ sem-tree-size ad = 0)))
        using IH by blast
      ultimately obtain ψ' :: 'v state and tree' :: 'v sem-tree where
        inf: resolution** ψ ψ'
        and part: partial-interps tree' (I ∪ {Neg v}) (fst ψ')
        and size: sem-tree-size tree' < sem-tree-size ad ∨ sem-tree-size ad = 0
        using finite part rtranclp.rtrancl-refl a-u-i simp by blast

      have partial-interps ag (I ∪ {Pos v}) (fst ψ')
        using rtranclp-resolution-preserve-partial-tree inf partag by fast
      then have partial-interps (Node v ag tree') I (fst ψ') using part by auto
      then have ?case using inf size size-ad unfolding xs by fastforce
    }
     ultimately have ?case by auto
  }
  ultimately show ?case by auto
qed

lemma resolution-completeness-inv:
  fixes ψ :: 'v ::linorder state
  assumes
    unsat: ¬satisfiable (fst ψ) and
    finite: finite (fst ψ) and
    a-u-v: already-used-inv ψ
  shows ∃ψ'. (resolution** ψ ψ' ∧ {#} ∈ fst ψ')
proof -
  obtain tree where partial-interps tree {} (fst ψ)
    using partial-interps-build-sem-tree-atms assms by metis
  then show ?thesis
    using unsat finite a-u-v
    proof (induct tree arbitrary: ψ rule: sem-tree-size)
```

case (*bigger tree* ψ) **note** H = *this*
{
fix χ
assume *tree*: *tree* = *Leaf*
obtain χ **where** χ: $\neg$ {} $\models \chi$ **and** *tot*χ: *total-over-m* {} {χ} **and** $\chi\psi$: $\chi \in$ *fst* ψ
using *H* **unfolding** *tree* **by** *auto*
moreover have {#} = χ
using *H atms-empty-iff-empty tot*χ
unfolding *true-cls-def total-over-m-def total-over-set-def* **by** *fastforce*
moreover have *resolution*** ψ ψ **by** *auto*
ultimately have *?case* **by** *metis*
}
moreover {
fix *v tree1 tree2*
assume *tree*: *tree* = *Node v tree1 tree2*
obtain ψ_0 **where** ψ_0: *resolution*** ψ ψ_0 **and** *simp*: *simplified* (*fst* ψ_0)
proof −
{ assume *simplified* (*fst* ψ)
moreover have *resolution*** ψ ψ **by** *auto*
ultimately have *thesis* **using** *that* **by** *blast*
}
moreover {
assume $\neg$*simplified* (*fst* ψ)
then have $\exists\psi'$. *full1 simplify* (*fst* ψ) ψ'
by (*metis Nitpick.rtranclp-unfold bigger.prems*(*3*) *full1-def*
rtranclp-simplify-terminates)
then obtain *N* **where** *full1 simplify* (*fst* ψ) *N* **by** *metis*
then have *resolution* ψ (*N*, *snd* ψ)
using *resolution.intros*(*1*)[*of fst* ψ *N snd* ψ] **by** *auto*
moreover have *simplified N*
using ‹*full1 simplify* (*fst* ψ) *N*› **unfolding** *full1-def* **by** *blast*
ultimately have *?thesis* **using** *that* **by** *force*
}
ultimately show *?thesis* **by** *auto*
qed

have *p*: *partial-interps tree* {} (*fst* ψ_0)
and *uns*: *unsatisfiable* (*fst* ψ_0)
and *f*: *finite* (*fst* ψ_0)
and *a-u-v*: *already-used-inv* ψ_0
using ψ_0 *bigger.prems*(*1*) *rtranclp-resolution-preserve-partial-tree* **apply** *blast*
using ψ_0 *bigger.prems*(*2*) *rtranclp-resolution-preserves-unsat* **apply** *blast*
using ψ_0 *bigger.prems*(*3*) *rtranclp-resolution-finite* **apply** *blast*
using *rtranclp-resolution-already-used-inv*[*OF* ψ_0 *bigger.prems*(*4*)] **by** *blast*
obtain *tree'* ψ' **where**
inf: *resolution*** ψ_0 ψ' **and**
part': *partial-interps tree'* {} (*fst* ψ') **and**
decrease: *sem-tree-size tree'* < *sem-tree-size tree* $\vee$ *sem-tree-size tree* = *0*
using *can-decrease-tree-size-resolution*[*OF f a-u-v p simp*] **unfolding** *tautology-def*
by *meson*
have *s*: *sem-tree-size tree'* < *sem-tree-size tree* **using** *decrease* **unfolding** *tree* **by** *auto*
have *fin*: *finite* (*fst* ψ')
using *f inf rtranclp-resolution-finite* **by** *blast*
have *unsat*: *unsatisfiable* (*fst* ψ')
using *rtranclp-resolution-preserves-unsat inf uns* **by** *metis*

have *a-u-i'*: *already-used-inv* ψ'
using *a-u-v inf rtranclp-resolution-already-used-inv*[*of* ψ_0 ψ'] **by** *auto*
have *?case*
using *inf rtranclp-trans*[*of resolution*] *H*(*1*)[*OF s part' unsat fin a-u-i'*] ψ_0 **by** *blast*
}
ultimately show *?case* **by** (*cases tree*, *auto*)
qed
qed

lemma *resolution-preserves-already-used-inv*:
assumes *resolution S S'*
and *already-used-inv S*
shows *already-used-inv S'*
using *assms*
apply (*induct rule*: *resolution.induct*)
apply (*rule full1-simplify-already-used-inv*; *simp*)
apply (*rule full-simplify-already-used-inv*, *simp*)
apply (*rule inference-preserves-already-used-inv*, *simp*)
apply *blast*
done

lemma *rtranclp-resolution-preserves-already-used-inv*:
assumes *resolution*** *S S'*
and *already-used-inv S*
shows *already-used-inv S'*
using *assms*
apply (*induct rule*: *rtranclp-induct*)
apply *simp*
using *resolution-preserves-already-used-inv* **by** *fast*

lemma *resolution-completeness*:
fixes ψ :: *'v* ::*linorder state*
assumes *unsat*: ¬*satisfiable* (*fst* ψ)
and *finite*: *finite* (*fst* ψ)
and *snd* $\psi = \{\}$
shows $\exists \psi'$. (*resolution*** ψ ψ' $\wedge$ {#} $\in$ *fst* ψ')
proof −
have *already-used-inv* ψ **unfolding** *assms* **by** *auto*
then show *?thesis* **using** *assms resolution-completeness-inv* **by** *blast*
qed

lemma *rtranclp-preserves-sat*:
assumes *simplify*** *S S'*
and *satisfiable S*
shows *satisfiable S'*
using *assms* **apply** *induction*
apply *simp*
by (*meson satisfiable-carac satisfiable-def simplify-preserve-models-eq*)

lemma *resolution-preserves-sat*:
assumes *resolution S S'*
and *satisfiable* (*fst S*)
shows *satisfiable* (*fst S'*)
using *assms* **apply** (*induction rule*: *resolution.induct*)
using *rtranclp-preserves-sat tranclp-into-rtranclp* **unfolding** *full1-def* **apply** *fastforce*
by (*metis fst-conv full-def inference-preserve-models rtranclp-preserves-sat*

satisfiable-carac′ satisfiable-def)

lemma *rtranclp-resolution-preserves-sat*:
assumes *resolution*** *S S′*
and *satisfiable* (*fst S*)
shows *satisfiable* (*fst S′*)
using *assms* **apply** (*induction rule*: *rtranclp-induct*)
apply *simp*
using *resolution-preserves-sat* **by** *blast*

lemma *resolution-soundness*:
fixes ψ :: *′v* ::*linorder state*
assumes *resolution*** ψ ψ' **and** $\{\#\} \in$ *fst* ψ'
shows *unsatisfiable* (*fst* ψ)
using *assms* **by** (*meson rtranclp-resolution-preserves-sat satisfiable-def true-cls-empty true-clss-def*)

lemma *resolution-soundness-and-completeness*:
fixes ψ :: *′v* ::*linorder state*
assumes *finite*: *finite* (*fst* ψ)
and *snd*: *snd* $\psi = \{\}$
shows $(\exists\, \psi'.\ (resolution^{**}\ \psi\ \psi' \wedge \{\#\} \in fst\ \psi')) \longleftrightarrow unsatisfiable\ (fst\ \psi)$
using *assms resolution-completeness resolution-soundness* **by** *metis*

lemma *simplified-falsity*:
assumes *simp*: *simplified* ψ
and $\{\#\} \in \psi$
shows $\psi = \{\{\#\}\}$
proof (*rule ccontr*)
assume *H*: $\neg$ *?thesis*
then obtain χ **where** $\chi \in \psi$ **and** $\chi \neq \{\#\}$ **using** *assms*(*2*) **by** *blast*
then have $\{\#\} \subset\# \chi$ **by** (*simp add*: *subset-mset.zero-less-iff-neq-zero*)
then have *simplify* ψ $(\psi - \{\chi\})$
using *simplify.subsumption*[*OF assms*(*2*) ‹$\{\#\} \subset\# \chi$› ‹$\chi \in \psi$›] **by** *blast*
then show *False* **using** *simp* **by** *blast*
qed

lemma *simplify-falsity-in-preserved*:
assumes *simplify* χs $\chi s'$
and $\{\#\} \in \chi s$
shows $\{\#\} \in \chi s'$
using *assms*
by *induction auto*

lemma *rtranclp-simplify-falsity-in-preserved*:
assumes *simplify*** χs $\chi s'$
and $\{\#\} \in \chi s$
shows $\{\#\} \in \chi s'$
using *assms*
by *induction* (*auto intro*: *simplify-falsity-in-preserved*)

lemma *resolution-falsity-get-falsity-alone*:
assumes *finite* (*fst* ψ)
shows $(\exists\, \psi'.\ (resolution^{**}\ \psi\ \psi' \wedge \{\#\} \in fst\ \psi')) \longleftrightarrow (\exists\, a\text{-}u\text{-}v.\ resolution^{**}\ \psi\ (\{\{\#\}\},\ a\text{-}u\text{-}v))$
(**is** *?A* $\longleftrightarrow$ *?B*)

proof
  assume *?B*
  then show *?A* **by** *auto*
next
  assume *?A*
  then obtain χs *a-u-v* **where** χs: *resolution*** ψ (χs, *a-u-v*) **and** *F*: {#} $\in$ χs **by** *auto*
  { assume *simplified* χs
    then have *?B* **using** *simplified-falsity*[*OF - F*] χs **by** *blast*
  }
  moreover {
    assume $\neg$ *simplified* χs
    then obtain $\chi s'$ **where** *full1 simplify* χs $\chi s'$
      by (*metis* χs *assms finite-simplified-full1-simp fst-conv rtranclp-resolution-finite*)
    then have {#} $\in$ $\chi s'$
      unfolding *full1-def* **by** (*meson F rtranclp-simplify-falsity-in-preserved*
        tranclp-into-rtranclp)
    then have *?B*
      by (*metis* χs ‹*full1 simplify* χs $\chi s'$› *fst-conv full1-simp resolution-always-simplified*
        rtranclp.rtrancl-into-rtrancl simplified-falsity)
  }
  ultimately show *?B* **by** *blast*
qed

theorem *resolution-soundness-and-completeness'*:
  fixes ψ :: *'v* ::*linorder state*
  assumes
    finite: *finite* (*fst* ψ)**and**
    snd: *snd* ψ = {}
  shows ($\exists$ *a-u-v*. (*resolution*** ψ ({{#}}, *a-u-v*))) $\longleftrightarrow$ *unsatisfiable* (*fst* ψ)
  using *assms resolution-completeness resolution-soundness resolution-falsity-get-falsity-alone*
  by *metis*

end
theory *Prop-Superposition*
imports *Entailment-Definition.Partial-Herbrand-Interpretation Ordered-Resolution-Prover.Herbrand-Interpretation*
begin

2.2 Superposition

no-notation *Herbrand-Interpretation.true-cls* (**infix** $\models$ *50*)
notation *Herbrand-Interpretation.true-cls* (**infix** $\models$*h 50*)

no-notation *Herbrand-Interpretation.true-clss* (**infix** $\models$*s 50*)
notation *Herbrand-Interpretation.true-clss* (**infix** $\models$*hs 50*)

lemma *herbrand-interp-iff-partial-interp-cls*:
  S $\models$*h* *C* $\longleftrightarrow$ {*Pos P*|*P*. *P*$\in$*S*} $\cup$ {*Neg P*|*P*. *P*$\notin$*S*} $\models$ *C*
  unfolding *Herbrand-Interpretation.true-cls-def Partial-Herbrand-Interpretation.true-cls-def*
  by *auto*

lemma *herbrand-consistent-interp*:
  consistent-interp ({*Pos P*|*P*. *P*$\in$*S*} $\cup$ {*Neg P*|*P*. *P*$\notin$*S*})
  unfolding *consistent-interp-def* **by** *auto*

lemma *herbrand-total-over-set*:

total-over-set $(\{Pos\ P|P.\ P{\in}S\} \cup \{Neg\ P|P.\ P{\notin}S\})\ T$
unfolding *total-over-set-def* **by** *auto*

lemma *herbrand-total-over-m*:
total-over-m $(\{Pos\ P|P.\ P{\in}S\} \cup \{Neg\ P|P.\ P{\notin}S\})\ T$
unfolding *total-over-m-def* **by** (*auto simp add*: *herbrand-total-over-set*)

lemma *herbrand-interp-iff-partial-interp-clss*:
$S \models hs\ C \longleftrightarrow \{Pos\ P|P.\ P{\in}S\} \cup \{Neg\ P|P.\ P{\notin}S\} \models s\ C$
unfolding *true-clss-def Ball-def herbrand-interp-iff-partial-interp-cls Partial-Herbrand-Interpretation.true-clss-def* **by** *auto*

definition *clss-lt* :: *'a*::*wellorder clause-set* ⇒ *'a clause* ⇒ *'a clause-set* **where**
clss-lt $N\ C = \{D \in N.\ D < C\}$

notation (*latex* **output**)
clss-lt (-<^bsup>-<^esup>)

locale *selection* =
fixes *S* :: *'a clause* ⇒ *'a clause*
assumes
S-selects-subseteq: $\bigwedge C.\ S\ C \leq\#\ C$ **and**
S-selects-neg-lits: $\bigwedge C\ L.\ L \in\#\ S\ C \Longrightarrow$ *is-neg* L

locale *ground-resolution-with-selection* =
selection S **for** *S* :: (*'a* :: *wellorder*) *clause* ⇒ *'a clause*
begin

context
fixes *N* :: *'a clause set*
begin

We do not create an equivalent of δ, but we directly defined N_C by inlining the definition.

function
production :: *'a clause* ⇒ *'a interp*
where
production $C =$
$\{A.\ C \in N \land C \neq \{\#\} \land Max\text{-}mset\ C = Pos\ A \land count\ C\ (Pos\ A) \leq 1$
$\land \neg (\bigcup D \in \{D.\ D < C\}.\ production\ D) \models h\ C \land S\ C = \{\#\}\}$
by *auto*
termination by (*relation* $\{(D,\ C).\ D < C\}$) (*auto simp*: *wf-less-multiset*)

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

definition *interp* :: *'a clause* ⇒ *'a interp* **where**
interp $C = (\bigcup D \in \{D.\ D < C\}.\ production\ D)$

lemma *production-unfold*:
production $C = \{A.\ C \in N \land C \neq \{\#\} \land Max\text{-}mset\ C = Pos\ A \land count\ C\ (Pos\ A) \leq 1 \land \neg\ interp\ C \models h\ C \land S\ C = \{\#\}\}$
unfolding *interp-def* **by** (*rule production.simps*)

abbreviation *productive* $A \equiv (production\ A \neq \{\})$

abbreviation *produces* :: *'a clause* ⇒ *'a* ⇒ *bool* **where**
produces $C\ A \equiv production\ C = \{A\}$

lemma *producesD*:
  produces C A $\Longrightarrow$ $C \in N \wedge C \neq \{\#\} \wedge$ *Pos A* = *Max-mset C* $\wedge$ *count C* (*Pos A*) $\leq$ *1* $\wedge$
  $\neg$ *interp C* $\models$*h C* $\wedge$ *S C* = $\{\#\}$
  unfolding *production-unfold* **by** *auto*

lemma *produces C A* $\Longrightarrow$ *Pos A* $\in\#$ *C*
  by (*simp add*: *Max-in-lits producesD*)

lemma *interp′-def-in-set*:
  interp C = ($\bigcup$ *D* $\in$ {*D* $\in$ *N*. *D* < *C*}. *production D*)
  unfolding *interp-def* **apply** *auto*
  unfolding *production-unfold* **apply** *auto*
  done

lemma *production-iff-produces*:
  produces D A $\longleftrightarrow$ *A* $\in$ *production D*
  unfolding *production-unfold* **by** *auto*

definition *Interp* :: *′a clause* $\Rightarrow$ *′a interp* **where**
  Interp C = *interp C* $\cup$ *production C*

lemma
  assumes *produces C P*
  shows *Interp C* $\models$*h C*
  unfolding *Interp-def assms* **using** *producesD*[*OF assms*]
  by (*metis Max-in-lits Un-insert-right insertI1 pos-literal-in-imp-true-cls*)

definition *INTERP* :: *′a interp* **where**
INTERP = ($\bigcup$ *D* $\in$*N*. *production D*)

lemma *interp-subseteq-Interp*[*simp*]: *interp C* $\subseteq$ *Interp C*
  unfolding *Interp-def* **by** *simp*

lemma *Interp-as-UNION*: *Interp C* = ($\bigcup$ *D* $\in$ {*D*. *D* $\leq$ *C*}. *production D*)
  unfolding *Interp-def interp-def less-eq-multiset-def* **by** *fast*

lemma *productive-not-empty*: *productive C* $\Longrightarrow$ *C* $\neq$ $\{\#\}$
  unfolding *production-unfold* **by** *auto*

lemma *productive-imp-produces-Max-literal*: *productive C* $\Longrightarrow$ *produces C* (*atm-of* (*Max-mset C*))
  unfolding *production-unfold* **by** (*auto simp del*: *atm-of-Max-lit*)

lemma *productive-imp-produces-Max-atom*: *productive C* $\Longrightarrow$ *produces C* (*Max* (*atms-of C*))
  unfolding *atms-of-def Max-atm-of-set-mset-commute*[*OF productive-not-empty*]
  by (*rule productive-imp-produces-Max-literal*)

lemma *produces-imp-Max-literal*: *produces C A* $\Longrightarrow$ *A* = *atm-of* (*Max-mset C*)
  by (*metis Max-singleton insert-not-empty productive-imp-produces-Max-literal*)

lemma *produces-imp-Max-atom*: *produces C A* $\Longrightarrow$ *A* = *Max* (*atms-of C*)
  by (*metis Max-singleton insert-not-empty productive-imp-produces-Max-atom*)

lemma *produces-imp-Pos-in-lits*: *produces C A* $\Longrightarrow$ *Pos A* $\in\#$ *C*
  by (*auto intro*: *Max-in-lits dest*!: *producesD*)

lemma *productive-in-N*: *productive* $C \Longrightarrow C \in N$
unfolding *production-unfold* **by** *auto*

lemma *produces-imp-atms-leq*: *produces* C $A \Longrightarrow B \in$ *atms-of* $C \Longrightarrow B \leq A$
by (*metis Max-ge finite-atms-of insert-not-empty productive-imp-produces-Max-atom singleton-inject*)

lemma *produces-imp-neg-notin-lits*: *produces* C $A \Longrightarrow$ *Neg* $A \notin\# C$
by (*rule pos-Max-imp-neg-notin*) (*auto dest*: *producesD*)

lemma *less-eq-imp-interp-subseteq-interp*: $C \leq D \Longrightarrow$ *interp* $C \subseteq$ *interp* D
unfolding *interp-def* **by** *auto* (*metis order.strict-trans2*)

lemma *less-eq-imp-interp-subseteq-Interp*: $C \leq D \Longrightarrow$ *interp* $C \subseteq$ *Interp* D
unfolding *Interp-def* **using** *less-eq-imp-interp-subseteq-interp* **by** *blast*

lemma *less-imp-production-subseteq-interp*: $C < D \Longrightarrow$ *production* $C \subseteq$ *interp* D
unfolding *interp-def* **by** *fast*

lemma *less-eq-imp-production-subseteq-Interp*: $C \leq D \Longrightarrow$ *production* $C \subseteq$ *Interp* D
unfolding *Interp-def* **using** *less-imp-production-subseteq-interp*
by (*metis le-imp-less-or-eq le-supI1 sup-ge2*)

lemma *less-imp-Interp-subseteq-interp*: $C < D \Longrightarrow$ *Interp* $C \subseteq$ *interp* D
unfolding *Interp-def*
by (*auto simp*: *less-eq-imp-interp-subseteq-interp less-imp-production-subseteq-interp*)

lemma *less-eq-imp-Interp-subseteq-Interp*: $C \leq D \Longrightarrow$ *Interp* $C \subseteq$ *Interp* D
using *less-imp-Interp-subseteq-interp*
unfolding *Interp-def* **by** (*metis le-imp-less-or-eq le-supI2 subset-refl sup-commute*)

lemma *false-Interp-to-true-interp-imp-less-multiset*: $A \notin$ *Interp* $C \Longrightarrow A \in$ *interp* $D \Longrightarrow C < D$
using *less-eq-imp-interp-subseteq-Interp not-less* **by** *blast*

lemma *false-interp-to-true-interp-imp-less-multiset*: $A \notin$ *interp* $C \Longrightarrow A \in$ *interp* $D \Longrightarrow C < D$
using *less-eq-imp-interp-subseteq-interp not-less* **by** *blast*

lemma *false-Interp-to-true-Interp-imp-less-multiset*: $A \notin$ *Interp* $C \Longrightarrow A \in$ *Interp* $D \Longrightarrow C < D$
using *less-eq-imp-Interp-subseteq-Interp not-less* **by** *blast*

lemma *false-interp-to-true-Interp-imp-le-multiset*: $A \notin$ *interp* $C \Longrightarrow A \in$ *Interp* $D \Longrightarrow C \leq D$
using *less-imp-Interp-subseteq-interp not-less* **by** *blast*

lemma *interp-subseteq-INTERP*: *interp* $C \subseteq$ *INTERP*
unfolding *interp-def INTERP-def* **by** (*auto simp*: *production-unfold*)

lemma *production-subseteq-INTERP*: *production* $C \subseteq$ *INTERP*
unfolding *INTERP-def* **using** *production-unfold* **by** *blast*

lemma *Interp-subseteq-INTERP*: *Interp* $C \subseteq$ *INTERP*
unfolding *Interp-def* **by** (*auto intro*!: *interp-subseteq-INTERP production-subseteq-INTERP*)

This lemma corresponds to theorem 2.7.7 page 77 of Weidenbach's book.

lemma *produces-imp-in-interp*:
assumes *a-in-c*: *Neg* $A \in\# C$ **and** *d*: *produces* D A

shows $A \in$ *interp* C
proof –
from d **have** *Max-mset* $D =$ *Pos* A
using *production-unfold* **by** *blast*
then have $D < \{\#Neg\ A\#\}$
by (*meson Max-pos-neg-less-multiset multi-member-last*)
moreover have $\{\#Neg\ A\#\} \leq C$
by (*rule subset-eq-imp-le-multiset*) (*rule mset-subset-eq-single*[*OF a-in-c*])
ultimately show *?thesis*
using d **by** (*blast dest*: *less-eq-imp-interp-subseteq-interp less-imp-production-subseteq-interp*)
qed

lemma *neg-notin-Interp-not-produce*: $Neg\ A \in\#\ C \Longrightarrow A \notin Interp\ D \Longrightarrow C \leq D \Longrightarrow \neg\ produces\ D''\ A$
by (*auto dest*: *produces-imp-in-interp less-eq-imp-interp-subseteq-Interp*)

lemma *in-production-imp-produces*: $A \in production\ C \Longrightarrow produces\ C\ A$
by (*metis insert-absorb productive-imp-produces-Max-atom singleton-insert-inj-eq'*)

lemma *not-produces-imp-notin-production*: $\neg\ produces\ C\ A \Longrightarrow A \notin production\ C$
by (*metis in-production-imp-produces*)

lemma *not-produces-imp-notin-interp*: $(\bigwedge D.\ \neg\ produces\ D\ A) \Longrightarrow A \notin interp\ C$
unfolding *interp-def* **by** (*fast intro*!: *in-production-imp-produces*)

The results below corresponds to Lemma 3.4.

Nitpicking 0.1. *If $D = D'$ and D is productive, $I^D \subseteq I_{D'}$ does not hold.*

lemma *true-Interp-imp-general*:
assumes
c-le-d: $C \leq D$ **and**
d-lt-d': $D < D'$ **and**
c-at-d: $Interp\ D \models h\ C$ **and**
subs: $interp\ D' \subseteq (\bigcup C \in CC.\ production\ C)$
shows $(\bigcup C \in CC.\ production\ C) \models h\ C$
proof (*cases* $\exists A.\ Pos\ A \in\#\ C \wedge A \in Interp\ D$)
case *True*
then obtain A **where** *a-in-c*: $Pos\ A \in\#\ C$ **and** *a-at-d*: $A \in Interp\ D$
by *blast*
from *a-at-d* **have** $A \in interp\ D'$
using *d-lt-d' less-imp-Interp-subseteq-interp* **by** *blast*
then show *?thesis*
using *subs a-in-c* **by** (*blast dest*: *contra-subsetD*)
next
case *False*
then obtain A **where** *a-in-c*: $Neg\ A \in\#\ C$ **and** $A \notin Interp\ D$
using *c-at-d* **unfolding** *true-cls-def* **by** *blast*
then have $\bigwedge D''.\ \neg\ produces\ D''\ A$
using *c-le-d neg-notin-Interp-not-produce* **by** *simp*
then show *?thesis*
using *a-in-c subs not-produces-imp-notin-production* **by** *auto*
qed

lemma *true-Interp-imp-interp*: $C \leq D \Longrightarrow D < D' \Longrightarrow Interp\ D \models h\ C \Longrightarrow interp\ D' \models h\ C$

using *interp-def true-Interp-imp-general* **by** *simp*

lemma *true-Interp-imp-Interp*: $C \le D \Longrightarrow D < D' \Longrightarrow$ *Interp* $D \models h\ C \Longrightarrow$ *Interp* $D' \models h\ C$
using *Interp-as-UNION interp-subseteq-Interp true-Interp-imp-general* **by** *simp*

lemma *true-Interp-imp-INTERP*: $C \le D \Longrightarrow$ *Interp* $D \models h\ C \Longrightarrow$ *INTERP* $\models h\ C$
using *INTERP-def interp-subseteq-INTERP*
true-Interp-imp-general[*OF - le-multiset-right-total*]
by *simp*

lemma *true-interp-imp-general*:
assumes
c-le-d: $C \le D$ **and**
d-lt-d': $D < D'$ **and**
c-at-d: *interp* $D \models h\ C$ **and**
subs: *interp* $D' \subseteq (\bigcup C \in CC.$ *production* $C)$
shows $(\bigcup C \in CC.$ *production* $C) \models h\ C$
proof (*cases* $\exists A.$ *Pos* $A \in\# C \wedge A \in$ *interp* D)
case *True*
then obtain A **where** *a-in-c*: *Pos* $A \in\# C$ **and** *a-at-d*: $A \in$ *interp* D
by *blast*
from *a-at-d* **have** $A \in$ *interp* D'
using *d-lt-d' less-eq-imp-interp-subseteq-interp*[*OF less-imp-le*] **by** *blast*
then show *?thesis*
using *subs a-in-c* **by** (*blast dest*: *contra-subsetD*)
next
case *False*
then obtain A **where** *a-in-c*: *Neg* $A \in\# C$ **and** $A \notin$ *interp* D
using *c-at-d* **unfolding** *true-cls-def* **by** *blast*
then have $\bigwedge D''.\ \neg$ *produces* $D''\ A$
using *c-le-d* **by** (*auto dest*: *produces-imp-in-interp less-eq-imp-interp-subseteq-interp*)
then show *?thesis*
using *a-in-c subs not-produces-imp-notin-production* **by** *auto*
qed

This lemma corresponds to theorem 2.7.7 page 77 of Weidenbach's book. Here the strict maximality is important

lemma *true-interp-imp-interp*: $C \le D \Longrightarrow D < D' \Longrightarrow$ *interp* $D \models h\ C \Longrightarrow$ *interp* $D' \models h\ C$
using *interp-def true-interp-imp-general* **by** *simp*

lemma *true-interp-imp-Interp*: $C \le D \Longrightarrow D < D' \Longrightarrow$ *interp* $D \models h\ C \Longrightarrow$ *Interp* $D' \models h\ C$
using *Interp-as-UNION interp-subseteq-Interp*[*of* D'] *true-interp-imp-general* **by** *simp*

lemma *true-interp-imp-INTERP*: $C \le D \Longrightarrow$ *interp* $D \models h\ C \Longrightarrow$ *INTERP* $\models h\ C$
using *INTERP-def interp-subseteq-INTERP*
true-interp-imp-general[*OF - le-multiset-right-total*]
by *simp*

lemma *productive-imp-false-interp*: *productive* $C \Longrightarrow \neg$ *interp* $C \models h\ C$
unfolding *production-unfold* **by** *auto*

This lemma corresponds to theorem 2.7.7 page 77 of Weidenbach's book. Here the strict maximality is important

lemma *cls-gt-double-pos-no-production*:
assumes D: $\{\#Pos\ P,\ Pos\ P\#\} < C$

shows $\neg$*produces C P*
proof $-$
let *?D* = {#*Pos P*, *Pos P*#}
note $D' = D$[*unfolded less-multiset*$_{HO}$]
consider
(*P*) *count C* (*Pos P*) $\geq$ *2*
| (*Q*) *Q* **where** *Q* > *Pos P* **and** *Q* $\in$# *C*
using *HOL.spec*[*OF HOL.conjunct2*[*OF D'*], *of Pos P*] **by** (*auto split*: *if-split-asm*)
then show *?thesis*
proof *cases*
case *Q*
have *Q* $\in$ *set-mset C*
using *Q*(*2*) **by** (*auto split*: *if-split-asm*)
then have *Max-mset C* > *Pos P*
using *Q*(*1*) *Max-gr-iff* **by** *blast*
then show *?thesis*
unfolding *production-unfold* **by** *auto*
next
case *P*
then show *?thesis*
unfolding *production-unfold* **by** *auto*
qed
qed

This lemma corresponds to theorem 2.7.7 page 77 of Weidenbach's book.

lemma
assumes *D*: *C*+{#*Neg P*#} < *D*
shows *production D* $\neq$ {*P*}
proof $-$
note $D' = D$[*unfolded less-multiset*$_{HO}$]
consider
(*P*) *Neg P* $\in$# *D*
| (*Q*) *Q* **where** *Q* > *Neg P* **and** *count D Q* > *count* (*C* + {#*Neg P*#}) *Q*
using *HOL.spec*[*OF HOL.conjunct2*[*OF D'*], *of Neg P*] *count-greater-zero-iff* **by** *fastforce*
then show *?thesis*
proof *cases*
case *Q*
have *Q* $\in$ *set-mset D*
using *Q*(*2*) *gr-implies-not0* **by** *fastforce*
then have *Max-mset D* > *Neg P*
using *Q*(*1*) *Max-gr-iff* **by** *blast*
then have *Max-mset D* > *Pos P*
using *less-trans*[*of Pos P Neg P Max-mset D*] **by** *auto*
then show *?thesis*
unfolding *production-unfold* **by** *auto*
next
case *P*
then have *Max-mset D* > *Pos P*
by (*meson Max-ge finite-set-mset le-less-trans linorder-not-le pos-less-neg*)
then show *?thesis*
unfolding *production-unfold* **by** *auto*
qed
qed

lemma *in-interp-is-produced*:
assumes *P* $\in$ *INTERP*

shows $\exists D.\ D +\{\#Pos\ P\#\} \in N \wedge produces\ (D +\{\#Pos\ P\#\})\ P$
using *assms* **unfolding** *INTERP-def UN-iff production-iff-produces Ball-def*
by (*metis ground-resolution-with-selection.produces-imp-Pos-in-lits insert-DiffM2 ground-resolution-with-selection-axioms not-produces-imp-notin-production*)

end
end

2.2.1 We can now define the rules of the calculus

inductive *superposition-rules* :: $'a\ clause \Rightarrow\ 'a\ clause \Rightarrow\ 'a\ clause \Rightarrow bool$ **where**
factoring: $superposition\text{-}rules\ (C + \{\#Pos\ P\#\} + \{\#Pos\ P\#\})\ B\ (C + \{\#Pos\ P\#\})\ |$
superposition-l: $superposition\text{-}rules\ (C_1 + \{\#Pos\ P\#\})\ (C_2 + \{\#Neg\ P\#\})\ (C_1 + C_2)$

inductive *superposition* :: $'a\ clause\text{-}set \Rightarrow\ 'a\ clause\text{-}set \Rightarrow bool$ **where**
superposition: $A \in N \Longrightarrow B \in N \Longrightarrow superposition\text{-}rules\ A\ B\ C$
$\Longrightarrow superposition\ N\ (N \cup \{C\})$

definition *abstract-red* :: $'a{::}wellorder\ clause \Rightarrow\ 'a\ clause\text{-}set \Rightarrow bool$ **where**
$abstract\text{-}red\ C\ N = (clss\text{-}lt\ N\ C \models p\ C)$

lemma *herbrand-true-clss-true-clss-cls-herbrand-true-clss*:
assumes
AB: $A \models hs\ B$ **and**
BC: $B \models p\ C$
shows $A \models h\ C$
proof −
let $?I = \{Pos\ P\ |P.\ P \in A\} \cup \{Neg\ P\ |P.\ P \notin A\}$
have *B*: $?I \models s\ B$ **using** *AB*
by (*auto simp add*: *herbrand-interp-iff-partial-interp-clss*)

have *IH*: $\bigwedge I.\ total\text{-}over\text{-}set\ I\ (atms\text{-}of\ C) \Longrightarrow total\text{-}over\text{-}m\ I\ B \Longrightarrow consistent\text{-}interp\ I$
$\Longrightarrow I \models s\ B \Longrightarrow I \models C$ **using** *BC*
by (*auto simp add*: *true-clss-cls-def*)
show *?thesis*
unfolding *herbrand-interp-iff-partial-interp-cls*
by (*auto intro*: *IH*[*of ?I*] *simp add*: *herbrand-total-over-set herbrand-total-over-m herbrand-consistent-interp B*)
qed

lemma *abstract-red-subset-mset-abstract-red*:
assumes
abstr: *abstract-red C N* **and**
c-lt-d: $C \subseteq\#\ D$
shows *abstract-red D N*
proof −
have $\{D \in N.\ D < C\} \subseteq \{D' \in N.\ D' < D\}$
using *subset-eq-imp-le-multiset*[*OF c-lt-d*]
by (*metis* (*no-types*, *lifting*) *Collect-mono order.strict-trans2*)
then show *?thesis*
using *abstr* **unfolding** *abstract-red-def clss-lt-def*
by (*metis* (*no-types*, *lifting*) *c-lt-d subset-mset.diff-add true-clss-cls-mono-r′ true-clss-cls-subset*)
qed

lemma *true-clss-cls-extended*:
  assumes
    $A \models p\ B$ **and**
    tot: *total-over-m* $I\ A$ **and**
    cons: *consistent-interp* I **and**
    I-A: $I \models s\ A$
  shows $I \models B$
proof −
  let $?I = I \cup \{Pos\ P|P.\ P \in atms\text{-}of\ B \wedge P \notin atms\text{-}of\text{-}s\ I\}$
  have *consistent-interp ?I*
    using *cons* **unfolding** *consistent-interp-def atms-of-s-def atms-of-def*
      apply (*auto 1 5 simp add*: *image-iff*)
    by (*metis atm-of-uminus literal.sel*(*1*))
  moreover have *tot-I*: *total-over-m* $?I\ (A \cup \{B\})$
  proof −
    obtain $aa :: 'a\ set \Rightarrow 'a\ literal\ set \Rightarrow 'a$ **where**
      $f2: \forall x0\ x1.\ (\exists v2.\ v2 \in x0 \wedge Pos\ v2 \notin x1 \wedge Neg\ v2 \notin x1)$
          $\longleftrightarrow (aa\ x0\ x1 \in x0 \wedge Pos\ (aa\ x0\ x1) \notin x1 \wedge Neg\ (aa\ x0\ x1) \notin x1)$
      by *moura*
    have $\forall a.\ a \notin atms\text{-}of\text{-}ms\ A \vee Pos\ a \in I \vee Neg\ a \in I$
      using *tot* **by** (*simp add*: *total-over-m-def total-over-set-def*)
    then have $aa\ (atms\text{-}of\text{-}ms\ A \cup atms\text{-}of\text{-}ms\ \{B\})\ (I \cup \{Pos\ a\ |a.\ a \in atms\text{-}of\ B \wedge a \notin atms\text{-}of\text{-}s\ I\})$
        $\notin atms\text{-}of\text{-}ms\ A \cup atms\text{-}of\text{-}ms\ \{B\} \vee Pos\ (aa\ (atms\text{-}of\text{-}ms\ A \cup atms\text{-}of\text{-}ms\ \{B\})$
          $(I \cup \{Pos\ a\ |a.\ a \in atms\text{-}of\ B \wedge a \notin atms\text{-}of\text{-}s\ I\})) \in I$
            $\cup \{Pos\ a\ |a.\ a \in atms\text{-}of\ B \wedge a \notin atms\text{-}of\text{-}s\ I\}$
          $\vee Neg\ (aa\ (atms\text{-}of\text{-}ms\ A \cup atms\text{-}of\text{-}ms\ \{B\})$
            $(I \cup \{Pos\ a\ |a.\ a \in atms\text{-}of\ B \wedge a \notin atms\text{-}of\text{-}s\ I\})) \in I$
            $\cup \{Pos\ a\ |a.\ a \in atms\text{-}of\ B \wedge a \notin atms\text{-}of\text{-}s\ I\}$
      by *auto*
    then have *total-over-set* $(I \cup \{Pos\ a\ |a.\ a \in atms\text{-}of\ B \wedge a \notin atms\text{-}of\text{-}s\ I\})$
        $(atms\text{-}of\text{-}ms\ A \cup atms\text{-}of\text{-}ms\ \{B\})$
      using *f2* **by** (*meson total-over-set-def*)
    then show *?thesis*
      by (*simp add*: *total-over-m-def*)
  qed
  moreover have $?I \models s\ A$
    using *I-A* **by** *auto*
  ultimately have *1*: $?I \models B$
    using ‹$A \models pB$› **unfolding** *true-clss-cls-def* **by** *auto*

  let $?I' = I \cup \{Neg\ P|P.\ P \in atms\text{-}of\ B \wedge P \notin atms\text{-}of\text{-}s\ I\}$
  have *consistent-interp* $?I'$
    using *cons* **unfolding** *consistent-interp-def atms-of-s-def atms-of-def*
    apply (*auto 1 5 simp add*: *image-iff*)
    by (*metis atm-of-uminus literal.sel*(*2*))
  moreover have *tot*: *total-over-m* $?I'\ (A \cup \{B\})$
    by (*smt Un-iff in-atms-of-s-decomp mem-Collect-eq tot total-over-m-empty total-over-m-insert*
        total-over-m-union total-over-set-def total-union)
  moreover have $?I' \models s\ A$
    using *I-A* **by** *auto*
  ultimately have *2*: $?I' \models B$
    using ‹$A \models pB$› **unfolding** *true-clss-cls-def* **by** *auto*
  define *BB* **where**
    ‹$BB = \{P.\ P \in atms\text{-}of\ B \wedge P \notin atms\text{-}of\text{-}s\ I\}$›
  have *1*: ‹$I \cup Pos\ `\ BB \models B$›

using *1* **unfolding** *BB-def* **by** (*simp add*: *setcompr-eq-image*)
have *2*: ‹$I \cup Neg$ ‘ $BB \models B$›
using *2* **unfolding** *BB-def* **by** (*simp add*: *setcompr-eq-image*)
have ‹*finite BB*›
unfolding *BB-def* **by** *auto*
then show *?thesis*
using *1 2* **apply** (*induction BB*)
subgoal by *auto*
subgoal for *x BB*
using *remove-literal-in-model-tautology*[*of* ‹$I \cup Pos$ ‘ BB›]
apply −
apply (*rule ccontr*)
apply (*auto simp*: *Partial-Herbrand-Interpretation.true-cls-def total-over-set-def total-over-m-def atms-of-ms-def*)

oops
lemma
assumes
CP: $\neg$ *clss-lt* N ({#*C*#} + {#*E*#}) $\models p$ {#*C*#} + {#*Neg P*#} **and**
clss-lt N ({#*C*#} + {#*E*#}) $\models p$ {#*E*#} + {#*Pos P*#} $\vee$ *clss-lt* N ({#*C*#} + {#*E*#}) $\models p$ {#*C*#} + {#*Neg P*#}
shows *clss-lt* N ({#*C*#} + {#*E*#}) $\models p$ {#*E*#} + {#*Pos P*#}

oops

locale *ground-ordered-resolution-with-redundancy* =
ground-resolution-with-selection +
fixes *redundant* :: *'a::wellorder clause* ⇒ *'a clause-set* ⇒ *bool*
assumes
redundant-iff-abstract: *redundant A N* ⟷ *abstract-red A N*
begin

definition *saturated* :: *'a clause-set* ⇒ *bool* **where**
saturated N ⟷
(∀ *A B C*. *A* ∈ *N* ⟶ *B* ∈ *N* ⟶ ¬*redundant A N* ⟶ ¬*redundant B N* ⟶
superposition-rules A B C ⟶ *redundant C N* ∨ *C* ∈ *N*)
lemma (**in** −)
assumes ‹$A \models p\ C + E$›
shows ‹$A \models p$ *add-mset L C* ∨ $A \models p$ *add-mset* (−*L*) *E*›
proof *clarify*
assume ‹¬ $A \models p$ *add-mset* (− *L*) *E*›
then obtain I' **where**
tot': ‹*total-over-m* I' ($A \cup$ {*add-mset* (−*L*) *E*})› **and**
cons': ‹*consistent-interp* I'› **and**
I'-A: ‹$I' \models s\ A$› **and**
I'-uL-E: ‹$\neg I' \models$ *add-mset* (−*L*) *E*›
unfolding *true-clss-cls-def* **by** *auto*
have ‹− $L \notin I'$› ‹¬ $I' \models E$›
using *I'-uL-E* **by** *auto*
moreover have ‹*atm-of L* ∈ *atm-of* ‘ I'›
using *tot'* **unfolding** *total-over-m-def total-over-set-def*
by (*cases L*) *force+*
ultimately have ‹$L \in I'$›
by (*auto simp*: *image-iff atm-of-eq-atm-of*)

show ‹$A \models p$ *add-mset L C*›

unfolding *true-clss-cls-def*
proof (*intro allI impI conjI*)
fix *I*
assume
tot: ‹*total-over-m I* ($A \cup \{$*add-mset L C*$\}$)› **and**
cons: ‹*consistent-interp I*› **and**
I-A: ‹$I \models s\ A$›
let $?I = I \cup \{Pos\ P|P.\ P \in atms\text{-}of\ E \wedge P \notin atms\text{-}of\text{-}s\ I\}$
have *in-C-pm-I*: ‹$L \in\#\ C \Longrightarrow L \in I \vee -L \in I$› **for** *L*
using *tot* **by** (*cases L*) (*force simp*: *total-over-m-def total-over-set-def atms-of-def*)+
have *consistent-interp ?I*
using *cons* **unfolding** *consistent-interp-def atms-of-s-def atms-of-def*
apply (*auto 1 5 simp add*: *image-iff*)
by (*metis atm-of-uminus literal.sel*(*1*))
moreover **{**
have *tot-I*: *total-over-m ?I* ($A \cup \{E\}$)
using *tot total-over-set-def total-union* **by** *force*
then have *tot-I*: *total-over-m ?I* ($A \cup \{C+E\}$)
using *total-union*[*OF tot*] **by** *auto***}**
moreover have $?I \models s\ A$
using *I-A* **by** *auto*
ultimately have *1*: $?I \models C + E$
using *assms* **unfolding** *true-clss-cls-def* **by** *auto*

then show ‹$I \models$ *add-mset L C*›
unfolding *Partial-Herbrand-Interpretation.true-cls-def*
apply (*auto simp*: *true-cls-def dest*: *in-C-pm-I*)
oops

lemma
assumes
saturated: *saturated N* **and**
finite: *finite N* **and**
empty: $\{\#\} \notin N$
shows *INTERP N* $\models hs$ *N*
proof (*rule ccontr*)
let $?N_{\mathcal{I}}$ = *INTERP N*
assume $\neg$ *?thesis*
then have *not-empty*: $\{E \in N.\ \neg ?N_{\mathcal{I}} \models h\ E\} \neq \{\}$
unfolding *true-clss-def Ball-def* **by** *auto*
define *D* **where** $D = Min\ \{E \in N.\ \neg ?N_{\mathcal{I}} \models h\ E\}$
have [*simp*]: $D \in N$
unfolding *D-def*
by (*metis* (*mono-tags*, *lifting*) *Min-in not-empty finite mem-Collect-eq rev-finite-subset subsetI*)
have *not-d-interp*: $\neg ?N_{\mathcal{I}} \models h\ D$
unfolding *D-def*
by (*metis* (*mono-tags*, *lifting*) *Min-in finite mem-Collect-eq not-empty rev-finite-subset subsetI*)
have *cls-not-D*: $\bigwedge E.\ E \in N \Longrightarrow E \neq D \Longrightarrow \neg ?N_{\mathcal{I}} \models h\ E \Longrightarrow D \leq E$
using *finite D-def* **by** *auto*
obtain *C L* **where** *D*: $D = C + \{\#L\#\}$ **and** *LSD*: $L \in\#\ S\ D \vee (S\ D = \{\#\} \wedge Max\text{-}mset\ D = L)$
proof (*cases* $S\ D = \{\#\}$)
case *False*
then obtain *L* **where** $L \in\#\ S\ D$
using *Max-in-lits* **by** *blast*
moreover **{**
then have $L \in\#\ D$

using *S-selects-subseteq*[*of D*] **by** *auto*
then have $D = (D - \{\#L\#\}) + \{\#L\#\}$
by *auto* **}**
ultimately show *?thesis* **using** *that* **by** *blast*
next
let $?L = Max\text{-}mset\ D$
case *True*
moreover {
have $?L \in\#\ D$
by (*metis* (*no-types*, *lifting*) *Max-in-lits* ‹$D \in N$› *empty*)
then have $D = (D - \{\#?L\#\}) + \{\#?L\#\}$
by *auto* **}**
ultimately show *?thesis* **using** *that* **by** *blast*
qed
have *red*: ¬*redundant D N*
proof (*rule ccontr*)
assume *red*[*simplified*]: ~~*redundant D N*
have $\forall E < D.\ E \in N \longrightarrow ?N_{\mathcal{I}} \models h\ E$
using *cls-not-D* **unfolding** *not-le*[*symmetric*] **by** *fastforce*
then have $?N_{\mathcal{I}} \models hs\ clss\text{-}lt\ N\ D$
unfolding *clss-lt-def true-clss-def Ball-def* **by** *blast*
then show *False*
using *red not-d-interp* **unfolding** *abstract-red-def redundant-iff-abstract*
using *herbrand-true-clss-true-clss-cls-herbrand-true-clss* **by** *fast*
qed

consider
(*L*) *P* **where** $L = Pos\ P$ **and** $S\ D = \{\#\}$ **and** $Max\text{-}mset\ D = Pos\ P$
| (*Lneg*) *P* **where** $L = Neg\ P$
using *LSD S-selects-neg-lits*[*of L D*] **by** (*cases L*) *auto*
then show *False*
proof *cases*
case *L* **note** *P* = *this*(*1*) **and** *S* = *this*(*2*) **and** *max* = *this*(*3*)
have $count\ D\ L > 1$
proof (*rule ccontr*)
assume ~ *?thesis*
then have *count*: $count\ D\ L = 1$
unfolding *D* **by** (*auto simp*: *not-in-iff*)
have $\neg ?N_{\mathcal{I}} \models h\ D$
using *not-d-interp true-interp-imp-INTERP ground-resolution-with-selection-axioms*
by *blast*
then have *produces N D P*
using *not-empty empty finite* ‹$D \in N$› *count L*
true-interp-imp-INTERP **unfolding** *production-iff-produces* **unfolding** *production-unfold*
by (*auto simp add*: *max not-empty*)
then have $INTERP\ N \models h\ D$
unfolding *D*
by (*metis pos-literal-in-imp-true-cls produces-imp-Pos-in-lits*
production-subseteq-INTERP singletonI subsetCE)
then show *False*
using *not-d-interp* **by** *blast*
qed
then have $Pos\ P \in\#\ C$
by (*simp add*: *P D*)
then obtain C' **where** $C'{:}D = C' + \{\#Pos\ P\#\} + \{\#Pos\ P\#\}$
unfolding *D* **by** (*metis* (*full-types*) *P insert-DiffM2*)

have *sup*: *superposition-rules D D* (*D* − {#*L*#})
unfolding *C′ L* **by** (*auto simp add*: *superposition-rules.simps*)
have *C′* + {#*Pos P*#} < *C′* + {#*Pos P*#} + {#*Pos P*#}
by *auto*
moreover have ¬?$N_{\mathcal{I}}$ ⊨*h* (*D* − {#*L*#})
using *not-d-interp* **unfolding** *C′ L* **by** *auto*
ultimately have *C′* + {#*Pos P*#} ∉ *N*
using *C′ P cls-not-D* **by** *fastforce*
have *D* − {#*L*#} < *D*
unfolding *C′ L* **by** *auto*
have *c′-p-p*: *C′* + {#*Pos P*#} + {#*Pos P*#} − {#*Pos P*#} = *C′* + {#*Pos P*#}
by *auto*
have *redundant* (*C′* + {#*Pos P*#}) *N*
using *saturated red sup* ‹*D* ∈ *N*›‹*C′* + {#*Pos P*#} ∉ *N*› **unfolding** *saturated-def C′ L c′-p-p*
by *blast*
moreover have *C′* + {#*Pos P*#} ⊆# *C′* + {#*Pos P*#} + {#*Pos P*#}
by *auto*
ultimately show *False*
using *red* **unfolding** *C′ redundant-iff-abstract* **by** (*blast dest*:
abstract-red-subset-mset-abstract-red)
next
case *Lneg* **note** *L* = *this*(*1*)
have *P*: *P* ∈ ?$N_{\mathcal{I}}$
using *not-d-interp* **unfolding** *D true-cls-def L* **by** (*auto split*: *if-split-asm*)
then obtain *E* **where**
DPN: *E* + {#*Pos P*#} ∈ *N* **and**
prod: *production N* (*E* + {#*Pos P*#}) = {*P*}
using *in-interp-is-produced* **by** *blast*
have ‹¬ ?$N_{\mathcal{I}}$ ⊨*h C*›
using *not-d-interp P* **unfolding** *D Lneg* **by** *auto*
then have *uL-C*: ‹*Pos P* ∉# *C*›
using *P* **unfolding** *Lneg* **by** *blast*

have *sup-EC*: *superposition-rules* (*E* + {#*Pos P*#}) (*C* + {#*Neg P*#}) (*E* + *C*)
using *superposition-l* **by** *fast*
then have *superposition N* (*N* ∪ {*E*+*C*})
using *DPN* ‹*D* ∈ *N*› **unfolding** *D L* **by** (*auto simp add*: *superposition.simps*)
have
PMax: *Pos P* = *Max-mset* (*E* + {#*Pos P*#}) **and**
count (*E* + {#*Pos P*#}) (*Pos P*) ≤ *1* **and**
S (*E* + {#*Pos P*#}) = {#} **and**
¬*interp N* (*E* + {#*Pos P*#}) ⊨*h E* + {#*Pos P*#}
using *prod* **unfolding** *production-unfold* **by** *auto*
have *Neg P* ∉# *E*
using *prod produces-imp-neg-notin-lits* **by** *force*
then have ⋀*y*. *y* ∈# (*E* + {#*Pos P*#}) ⟹
count (*E* + {#*Pos P*#}) (*Neg P*) < *count* (*C* + {#*Neg P*#}) (*Neg P*)
using *count-greater-zero-iff* **by** *fastforce*
moreover have ⋀*y*. *y* ∈# (*E* + {#*Pos P*#}) ⟹ *y* < *Neg P*
using *PMax* **by** (*metis DPN Max-less-iff empty finite-set-mset pos-less-neg*
set-mset-eq-empty-iff)
moreover have *E* + {#*Pos P*#} ≠ *C* + {#*Neg P*#}
using *prod produces-imp-neg-notin-lits* **by** *force*
ultimately have *E* + {#*Pos P*#} < *C* + {#*Neg P*#}
unfolding *less-multiset*$_{HO}$ **by** (*metis count-greater-zero-iff less-iff-Suc-add zero-less-Suc*)
have *ce-lt-d*: *C* + *E* < *D*

```
 unfolding D L by (simp add: ‹⋀y. y ∈# E + {#Pos P#} ⟹ y < Neg P› ex-gt-imp-less-multiset)
have ?N_𝓘 ⊨h E + {#Pos P#}
  using ‹P ∈ ?N_𝓘› by blast
have ?N_𝓘 ⊨h C+E ∨ C+E ∉ N
  using ce-lt-d cls-not-D unfolding D-def by fastforce
have Pos-P-C-E: Pos P ∉# C+E
  using D ‹P ∈ ground-resolution-with-selection.INTERP S N›
    ‹count (E + {#Pos P#}) (Pos P) ≤ 1› multi-member-skip not-d-interp
  by (auto simp: not-in-iff)
then have ⋀y. y ∈# C+E ⟹ count (C+E) (Pos P) < count (E + {#Pos P#}) (Pos P)
  using set-mset-def by fastforce
have ¬redundant (C + E) N
proof (rule ccontr)
  assume red'[simplified]: ¬ ?thesis
  have abs: clss-lt N (C + E) ⊨p C + E
    using redundant-iff-abstract red' unfolding abstract-red-def by auto
  moreover
  have ‹clss-lt N (C + E)  ⊆ clss-lt N (E + {#Pos P#})›
    using ce-lt-d Pos-P-C-E uL-C apply (auto simp: clss-lt-def D L)

    using Pos-P-C-E unfolding less-multiset_HO
    apply (auto split: if-splits)
    sorry
  then have clss-lt N (E + {#Pos P#}) ⊨p E + {#Pos P#} ∨
      clss-lt N (C + {#Neg P#}) ⊨p C + {#Neg P#}
  proof clarify
    assume CP: ¬ clss-lt N (C + {#Neg P#}) ⊨p C + {#Neg P#}
    { fix I
      assume
        total-over-m I (clss-lt N (C + E) ∪ {E + {#Pos P#}}) and
        consistent-interp I and
        I ⊨s clss-lt N (C + E)
      then have I ⊨ C + E
        using  abs sorry
      moreover have ¬ I ⊨ C + {#Neg P#}
        using CP unfolding true-clss-cls-def
        sorry
      ultimately have I ⊨ E + {#Pos P#} by auto
    }
    then show clss-lt N (E + {#Pos P#}) ⊨p E + {#Pos P#}
      unfolding true-clss-cls-def sorry
  qed
  then have clss-lt N (C + E) ⊨p E + {#Pos P#} ∨ clss-lt N (C + E) ⊨p C + {#Neg P#}
  proof clarify
    assume CP: ¬ clss-lt N (C + E) ⊨p C + {#Neg P#}
    { fix I
      assume
        total-over-m I (clss-lt N (C + E) ∪ {E + {#Pos P#}}) and
        consistent-interp I and
        I ⊨s clss-lt N (C + E)
      then have I ⊨ C + E
        using  abs sorry
      moreover have ¬ I ⊨ C + {#Neg P#}
        using CP unfolding true-clss-cls-def
        sorry
      ultimately have I ⊨ E + {#Pos P#} by auto
```

}
then show $clss\text{-}lt\ N\ (C + E) \models p\ E + \{\#Pos\ P\#\}$
unfolding *true-clss-cls-def* **by** *auto*
qed
moreover have $clss\text{-}lt\ N\ (C + E) \subseteq clss\text{-}lt\ N\ (C + \{\#Neg\ P\#\})$
using *ce-lt-d order.strict-trans2* **unfolding** *clss-lt-def D L*
by (*blast dest*: *less-imp-le*)
ultimately have $redundant\ (C + \{\#Neg\ P\#\})\ N \lor clss\text{-}lt\ N\ (C + E) \models p\ E + \{\#Pos\ P\#\}$
unfolding *redundant-iff-abstract abstract-red-def* **using** *true-clss-cls-subset* **by** *blast*
show *False*

sorry
qed
moreover have $\neg\ redundant\ (E + \{\#Pos\ P\#\})\ N$
sorry
ultimately have CEN: $C + E \in N$
using ‹$D \in N$› ‹$E + \{\#Pos\ P\#\} \in N$› *saturated sup-EC red* **unfolding** *saturated-def D L*
by (*metis union-commute*)
have CED: $C + E \neq D$
using *D ce-lt-d* **by** *auto*
have *interp*: $\neg\ INTERP\ N \models h\ C + E$
sorry
show *False*
using *cls-not-D*[*OF CEN CED interp*] *ce-lt-d* **unfolding** *INTERP-def less-eq-multiset-def* **by** *auto*
qed
qed

end

lemma *tautology-is-redundant*:
assumes *tautology C*
shows *abstract-red C N*
using *assms* **unfolding** *abstract-red-def true-clss-cls-def tautology-def* **by** *auto*

lemma *subsumed-is-redundant*:
assumes AB: $A \subset\# B$
and AN: $A \in N$
shows *abstract-red B N*
proof −
have $A \in clss\text{-}lt\ N\ B$ **using** *AN AB* **unfolding** *clss-lt-def*
by (*auto dest*: *subset-eq-imp-le-multiset simp add*: *dual-order.order-iff-strict*)
then show *?thesis*
using *AB* **unfolding** *abstract-red-def true-clss-cls-def Partial-Herbrand-Interpretation.true-clss-def*
by *blast*
qed

inductive *redundant* :: $'a\ clause \Rightarrow 'a\ clause\text{-}set \Rightarrow bool$ **where**
subsumption: $A \in N \Longrightarrow A \subset\# B \Longrightarrow redundant\ B\ N$

lemma *redundant-is-redundancy-criterion*:
fixes A :: $'a$ :: *wellorder clause* **and** N :: $'a$ :: *wellorder clause-set*
assumes *redundant A N*
shows *abstract-red A N*
using *assms*
proof (*induction rule*: *redundant.induct*)
case (*subsumption A B N*)

then show *?case*
using *subsumed-is-redundant*[*of A N B*] **unfolding** *abstract-red-def clss-lt-def* **by** *auto*
qed

lemma *redundant-mono*:
redundant A N $\Longrightarrow$ *A* $\subseteq$# *B* $\Longrightarrow$ *redundant B N*
apply (*induction rule*: *redundant.induct*)
by (*meson subset-mset.less-le-trans subsumption*)

locale *truc* =
selection S **for** *S* :: *nat clause* $\Rightarrow$ *nat clause*
begin

end

end