Formalisation of Ground Resolution and CDCL in Isabelle/HOL

Mathias Fleury and Jasmin Blanchette

December 6, 2019

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*: ⋀*φ x*. *φ* = *FF* ∨ *φ* = *FT* ∨ *φ* = *FVar x* ⟹ *P φ*
and *unary*: ⋀*ψ*. *P ψ* ⟹ *P* (*FNot ψ*)
and *binary*: ⋀*φ ψ1 ψ2*. *P ψ1* ⟹ *P ψ2* ⟹ *φ* = *FAnd ψ1 ψ2* ∨ *φ* = *FOr ψ1 ψ2* ∨ *φ* = *FImp ψ1 ψ2*
∨ *φ* = *FEq ψ1 ψ2* ⟹ *P φ*
shows *P ψ*
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* ⇒ *'v propo list* ⇒ *'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*: ⋀*x*. *c* = *CT* ∨ *c* = *CF* ∨ *c* = *CVar x* ⟹ *P*
and *binary*: *c* ∈ *binary-connectives* ⟹ *P*
and *unary*: *c* = *CNot* ⟹ *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* ∈ *nullary-connective* ⟹ *P*
and *unary*: *c* = *CNot* ⟹ *P*
and *binary*: *c* ∈ *binary-connectives* ⟹ *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* ⇒ *'v propo list* ⇒ *bool* **for** *c* :: *'v connective* **where**
wf-conn-nullary[*simp*]: (*c* = *CT* ∨ *c* = *CF* ∨ *c* = *CVar v*) ⟹ *wf-conn c* [] |
wf-conn-unary[*simp*]: *c* = *CNot* ⟹ *wf-conn c* [*ψ*] |
wf-conn-binary[*simp*]: *c* ∈ *binary-connectives* ⟹ *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**
⋀*v*. *c* = *CT* ⟹ *P* [] **and**
⋀*v*. *c* = *CF* ⟹ *P* [] **and**
⋀*v*. *c* = *CVar v* ⟹ *P* [] **and**
⋀*ψ*. *c* = *CNot* ⟹ *P* [*ψ*] **and**
⋀*ψ ψ'*. *c* = *COr* ⟹ *P* [*ψ*, *ψ'*] **and**
⋀*ψ ψ'*. *c* = *CAnd* ⟹ *P* [*ψ*, *ψ'*] **and**
⋀*ψ ψ'*. *c* = *CImp* ⟹ *P* [*ψ*, *ψ'*] **and**
⋀*ψ ψ'*. *c* = *CEq* ⟹ *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 [ψ, ψ'] by auto
    qed
qed

lemma wf-conn-not-list-length[iff]:
  fixes l :: 'v propo list
  shows wf-conn CNot l ⟷ 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 ∃ 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' ⟹ wf-conn c l ⟷ wf-conn c l'
proof −
  {
    fix l l'
    have length l = length l' ⟹ wf-conn c l ⟹ 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' ⟹ wf-conn c l = wf-conn c l' by metis
qed

lemma wf-conn-no-arity-change-helper:
  length (ξ @ φ # ξ') = length (ξ @ φ' # ξ')
  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 = [ψ]
  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 ψs*
shows *ca* = *c* ∧ *ψs* = *l*
using *corr*
proof (*cases ca l rule*: *wf-conn.cases*)
case (*wf-conn-nullary v*)
then show *ca* = *c* ∧ *ψs* = *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 ψ′* = *conn c ψs* **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 *ψs* = *l* **using** ∗ *conn-inj-not*(*2*) *corr′ wf-conn-unary*(*1*) **by** *force*
ultimately show *ca* = *c* ∧ *ψs* = *l* **by** *auto*
next
case (*wf-conn-binary ψ′ ψ″*)
then show *ca* = *c* ∧ *ψs* = *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* ⇒ *′v propo* ⇒ *bool* (**infix** ⪯ *45*) **for** *φ* **where**
subformula-refl[*simp*]: *φ* ⪯ *φ* |
subformula-into-subformula: *ψ* ∈ *set l* ⟹ *wf-conn c l* ⟹ *φ* ⪯ *ψ* ⟹ *φ* ⪯ *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 φ* ⪯ *ψ* ⟹ *φ* ⪯ *ψ*
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* ∈ *binary-connectives*
shows *f* ⪯ *conn c* [*f*, *g*]
and *g* ⪯ *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* ⪯ *f* **using** *subformula-refl* **by** *auto*
ultimately show *f* ⪯ *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* ⪯ *g* **using** *subformula-refl* **by** *auto*
ultimately show *g* ⪯ *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 φ ψ :: *'v propo*
assumes *incl*: $\varphi \preceq \psi$
and *simple*: $\psi = FT \vee \psi = FF \vee \psi = FVar\ x$
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 $\Longrightarrow \varphi \preceq conn\ c\ l \longleftrightarrow (\varphi = conn\ c\ l \vee (\exists \psi.\ \psi \in set\ l \wedge \varphi \preceq \psi))$
apply *standard*
using *subfurmula-not-incl-eq* **apply** *metis*
by (*auto simp add*: *subformula-into-subformula*)

lemma *subformula-decomp-explicit*[*simp*]:
$\varphi \preceq FAnd\ \psi\ \psi' \longleftrightarrow (\varphi = FAnd\ \psi\ \psi' \vee \varphi \preceq \psi \vee \varphi \preceq \psi')$ (**is** *?P FAnd*)
$\varphi \preceq FOr\ \psi\ \psi' \longleftrightarrow (\varphi = FOr\ \psi\ \psi' \vee \varphi \preceq \psi \vee \varphi \preceq \psi')$
$\varphi \preceq FEq\ \psi\ \psi' \longleftrightarrow (\varphi = FEq\ \psi\ \psi' \vee \varphi \preceq \psi \vee \varphi \preceq \psi')$
$\varphi \preceq FImp\ \psi\ \psi' \longleftrightarrow (\varphi = FImp\ \psi\ \psi' \vee \varphi \preceq \psi \vee \varphi \preceq \psi')$
proof −
have *wf-conn CAnd* [ψ, ψ'] **by** (*simp add*: *binary-connectives-def*)
then have $\varphi \preceq conn\ CAnd\ [\psi, \psi'] \longleftrightarrow$
$(\varphi = conn\ CAnd\ [\psi, \psi'] \vee (\exists \psi''.\ \psi'' \in set\ [\psi, \psi'] \wedge \varphi \preceq \psi''))$
using *wf-subformula-conn-cases* **by** *metis*
then show *?P FAnd* **by** *auto*
next
have *wf-conn COr* [ψ, ψ'] **by** (*simp add*: *binary-connectives-def*)
then have $\varphi \preceq conn\ COr\ [\psi, \psi'] \longleftrightarrow$
$(\varphi = conn\ COr\ [\psi, \psi'] \vee (\exists \psi''.\ \psi'' \in set\ [\psi, \psi'] \wedge \varphi \preceq \psi''))$
using *wf-subformula-conn-cases* **by** *metis*
then show *?P FOr* **by** *auto*
next
have *wf-conn CEq* [ψ, ψ'] **by** (*simp add*: *binary-connectives-def*)
then have $\varphi \preceq conn\ CEq\ [\psi, \psi'] \longleftrightarrow$
$(\varphi = conn\ CEq\ [\psi, \psi'] \vee (\exists \psi''.\ \psi'' \in set\ [\psi, \psi'] \wedge \varphi \preceq \psi''))$
using *wf-subformula-conn-cases* **by** *metis*
then show *?P FEq* **by** *auto*
next
have *wf-conn CImp* [ψ, ψ'] **by** (*simp add*: *binary-connectives-def*)
then have $\varphi \preceq conn\ CImp\ [\psi, \psi'] \longleftrightarrow$
$(\varphi = conn\ CImp\ [\psi, \psi'] \vee (\exists \psi''.\ \psi'' \in set\ [\psi, \psi'] \wedge \varphi \preceq \psi''))$
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*: $\exists\ c\ l.\ \varphi = conn\ c\ l \wedge wf\text{-}conn\ c\ l$
by (*cases* φ) *force+*

lemma *subformula-conn-decomp*[*simp*]:
assumes *wf*: *wf-conn c l*
shows $\varphi \preceq conn\ c\ l \longleftrightarrow (\varphi = conn\ c\ l \vee (\exists\ \psi \in set\ l.\ \varphi \preceq \psi))$ (**is** *?A* $\longleftrightarrow$ *?B*)
proof (*rule iffI*)
{
fix ξ
have $\varphi \preceq \xi \Longrightarrow \xi = conn\ c\ l \Longrightarrow wf\text{-}conn\ c\ l \Longrightarrow \forall x::'a\ propo \in set\ l.\ \neg\ \varphi \preceq x \Longrightarrow \varphi = conn\ c\ l$
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 $\psi = a \vee \psi = b$ **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 $\varphi ::$ *'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* = $L \mid R$

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 | p. $p \in$ *pos* φ} $\cup$ { R # p | p. $p \in$ *pos* ψ} |
pos (*FOr* φ ψ) = {[]} $\cup$ { L # p | p. $p \in$ *pos* φ} $\cup$ { R # p | p. $p \in$ *pos* ψ} |
pos (*FEq* φ ψ) = {[]} $\cup$ { L # p | p. $p \in$ *pos* φ} $\cup$ { R # p | p. $p \in$ *pos* ψ} |
pos (*FImp* φ ψ) = {[]} $\cup$ { L # p | p. $p \in$ *pos* φ} $\cup$ { R # p | p. $p \in$ *pos* ψ} |
pos (*FNot* φ) = {[]} $\cup$ { L # p | p. $p \in$ *pos* φ}

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* |*p*. $p \in s$}) = *card s*
using *finite*
proof (*induct s rule*: *finite-induct*)
show *card* {*f p* |*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* |*p*. $p \in s$} = *card s*

```
  have f': finite {f p |p. p ∈ insert x s} using f by auto
  have notin': f x ∉ {f p |p. p ∈ s} using notin inj injD by fastforce
  have {f p |p. p ∈ insert x s} = insert (f x) {f p |p. p∈ s} by auto
  then have card {f p |p. p ∈ insert x s} = 1 + card {f p |p. p ∈ s}
    using finite card-insert-disjoint f' notin' by auto
  moreover have ... = card (insert x s) using notin f IH by auto
  finally show card {f p |p. p ∈ insert x s} = card (insert x s) .
qed

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

lemma finite-insert-nil-cons:
  finite s ⟹ card (insert [] {L # p |p. p ∈ s}) = 1 + card {L # p |p. p ∈ s}
  using card-insert-disjoint by auto


lemma cord-not[simp]:
  card (pos (FNot φ)) = 1 + card (pos φ)
by (simp add: cons-inject finite-inj-comp-set finite-pos)

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

definition prop-size where prop-size φ = card (pos φ)

lemma prop-size-vars-of-prop:
  fixes φ :: 'v propo
  shows card (vars-of-prop φ) ≤ prop-size φ

  unfolding prop-size-def apply (induct φ, auto simp add: cons-inject finite-inj-comp-set finite-pos)
proof −
  fix φ1 φ2 :: 'v propo
  assume IH1: card (vars-of-prop φ1) ≤ card (pos φ1)
  and IH2: card (vars-of-prop φ2) ≤ card (pos φ2)
  let ?L = {L # p |p. p ∈ pos φ1}
  let ?R = {R # p |p. p ∈ pos φ2}
  have card (?L ∪ ?R) = card ?L +  card ?R
    using card-seperate finite-pos by blast
  moreover have ... = card (pos φ1) + card (pos φ2)
    by (simp add: cons-inject finite-inj-comp-set finite-pos)
  moreover have ... ≥ card (vars-of-prop φ1) + card (vars-of-prop φ2) using IH1 IH2 by arith
  then have ... ≥ card (vars-of-prop φ1 ∪ vars-of-prop φ2) using card-Un-le le-trans by blast
  ultimately
    show card (vars-of-prop φ1 ∪ vars-of-prop φ2) ≤ Suc (card (?L ∪ ?R))
      card (vars-of-prop φ1 ∪ vars-of-prop φ2) ≤ Suc (card (?L ∪ ?R))
      card (vars-of-prop φ1 ∪ vars-of-prop φ2) ≤ Suc (card (?L ∪ ?R))
```

```
    card (vars-of-prop φ1 ∪ vars-of-prop φ2) ≤ Suc (card (?L ∪ ?R))
  by auto
qed

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

inductive path-to :: sign list ⇒ 'v propo ⇒ 'v propo ⇒ bool where
path-to-refl[intro]: path-to [] φ φ |
path-to-l: c∈binary-connectives ∨ c = CNot ⟹ wf-conn c (φ#l) ⟹ path-to p φ φ'⟹
  path-to (L#p) (conn c (φ#l)) φ' |
path-to-r: c∈binary-connectives ⟹ wf-conn c (ψ#φ#[]) ⟹ path-to p φ φ' ⟹
  path-to (R#p) (conn c (ψ#φ#[])) φ'
```

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-to p φ φ' ⟹ φ' ⪯ φ
  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 φ φ':: 'v propo
  shows φ' ⪯ φ ⟹ ∃ p. path-to p φ φ'
proof (induct rule: subformula.induct)
  case subformula-refl
  have path-to [] φ' φ' by auto
  then show ∃ p. path-to p φ' φ' by metis
next
  case (subformula-into-subformula ψ l c)
  note wf = this(2) and IH = this(4) and ψ = this(1)
  then obtain p where p: path-to p ψ φ' by metis
  {
    fix x :: 'v
    assume c = CT ∨ c = CF ∨ c = CVar x
    then have False using subformula-into-subformula by auto
    then have ∃ p. path-to p (conn c l) φ' by blast
  }
  moreover {
    assume c: c = CNot
    then have l = [ψ] using wf ψ wf-conn-Not-decomp by fastforce
    then have path-to (L # p) (conn c l) φ' by (metis c wf-conn-unary p path-to-l)
   then have ∃ p. path-to p (conn c l) φ' by blast
  }
  moreover {
    assume c: c∈ binary-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 = ψ ∨ b = ψ using ψ by auto
    then have path-to (L # p) (conn c l) φ' ∨ path-to (R # p) (conn c l) φ' using c path-to-l
      path-to-r p ab by (metis wf-conn-binary)
    then have ∃ p. path-to p (conn c l) φ' by blast
  }
  ultimately show ∃ p. path-to p (conn c l) φ' using connective-cases-arity by metis
qed
```

fun *replace-at* :: *sign list* ⇒ *'v propo* ⇒ *'v propo* ⇒ *'v propo* **where**
replace-at [] - ψ = ψ |
replace-at (*L* # *l*) (*FAnd* φ φ') ψ = *FAnd* (*replace-at l* φ ψ) φ'|
replace-at (*R* # *l*) (*FAnd* φ φ') ψ = *FAnd* φ (*replace-at l* φ' ψ) |
replace-at (*L* # *l*) (*FOr* φ φ') ψ = *FOr* (*replace-at l* φ ψ) φ' |
replace-at (*R* # *l*) (*FOr* φ φ') ψ = *FOr* φ (*replace-at l* φ' ψ) |
replace-at (*L* # *l*) (*FEq* φ φ') ψ = *FEq* (*replace-at l* φ ψ) φ'|
replace-at (*R* # *l*) (*FEq* φ φ') ψ = *FEq* φ (*replace-at l* φ' ψ) |
replace-at (*L* # *l*) (*FImp* φ φ') ψ = *FImp* (*replace-at l* φ ψ) φ'|
replace-at (*R* # *l*) (*FImp* φ φ') ψ = *FImp* φ (*replace-at l* φ' ψ) |
replace-at (*L* # *l*) (*FNot* φ) ψ = *FNot* (*replace-at l* φ ψ)

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**
$\mathcal{A}$ ⊨ *FT* = *True* |
$\mathcal{A}$ ⊨ *FF* = *False* |
$\mathcal{A}$ ⊨ *FVar v* = ($\mathcal{A}$ *v*) |
$\mathcal{A}$ ⊨ *FNot* φ = (¬($\mathcal{A}$⊨ φ)) |
$\mathcal{A}$ ⊨ *FAnd* φ_1 φ_2 = ($\mathcal{A}$⊨φ_1 ∧ $\mathcal{A}$⊨φ_2) |
$\mathcal{A}$ ⊨ *FOr* φ_1 φ_2 = ($\mathcal{A}$⊨φ_1 ∨ $\mathcal{A}$⊨φ_2) |
$\mathcal{A}$ ⊨ *FImp* φ_1 φ_2 = ($\mathcal{A}$⊨φ_1 ⟶ $\mathcal{A}$⊨φ_2) |
$\mathcal{A}$ ⊨ *FEq* φ_1 φ_2 = ($\mathcal{A}$⊨φ_1 ⟷ $\mathcal{A}$ ⊨φ_2)

definition *evalf* (**infix** ⊨*f 50*) **where**
evalf φ ψ = (∀ *A*. *A* ⊨ φ ⟶ *A* ⊨ ψ)

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

theorem *deduction-theorem*:
φ ⊨*f* ψ ⟷ (∀ *A*. *A* ⊨ *FImp* φ ψ)
proof
assume *H*: φ ⊨*f* ψ
{
fix *A*
have *A* ⊨ *FImp* φ ψ
proof (*cases A* ⊨ φ)
case *True*
then have *A* ⊨ ψ **using** *H* **unfolding** *evalf-def* **by** *metis*
then show *A* ⊨ *FImp* φ ψ **by** *auto*
next
case *False*
then show *A* ⊨ *FImp* φ ψ **by** *auto*
qed
}
then show ∀ *A*. *A* ⊨ *FImp* φ ψ **by** *blast*
next
assume *A*: ∀ *A*. *A* ⊨ *FImp* φ ψ
show φ ⊨*f* ψ
proof (*rule ccontr*)
assume ¬ φ ⊨*f* ψ
then obtain *A* **where** *A* ⊨ φ **and** ¬ *A* ⊨ ψ **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-over-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 ψ ψ' φ :: *'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** ψ'' = *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: $l = \xi$ @ ψ'' # ξ' **using** *List.split-list* ψ'' **by** *metis*
ultimately have *propo-rew-step* r (*conn* c l) (*conn* c (ξ @ φ' # ξ'))
using *propo-rew-step.intros*(*2*) *wf* **by** *metis*
moreover have $\psi' \preceq$ *conn* c (ξ @ φ' # ξ')
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 φ φ' :: *'v propo* **and** r :: *'v propo* $\Rightarrow$ *'v propo* $\Rightarrow$ *bool*

```
  assumes propo-rew-step r φ φ'
  shows ∃ψ ψ' p. r ψ ψ' ∧ path-to p φ ψ ∧ replace-at p φ ψ' = φ'
  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 ψ = ψ 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 ψ ψ' ∧ path-to p φ ψ ∧ replace-at p φ ψ' = φ' using IH0 by metis

  {
    fix x :: 'v
    assume c = CT ∨ c = CF ∨ c = CVar x
    then have False using corr by auto
    then have ∃ψ ψ' p. r ψ ψ' ∧ path-to p (conn c (ξ@ (φ # ξ'))) ψ
                    ∧ 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 ∃ψ ψ' p. r ψ ψ' ∧ path-to p (conn c (ξ@ (φ # ξ'))) ψ
                          ∧ replace-at p (conn c (ξ@ (φ # ξ'))) ψ' = conn c (ξ@ (φ' # ξ'))
    using IH by metis
  }
  moreover {
    assume c: c ∈ 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 ∧ length ξ' = 0) ∨ (length ξ = 0 ∧ length ξ' = 1)  by arith
    obtain a b where ab: (ξ=[] ∧ ξ'=[b]) ∨ (ξ=[a] ∧ ξ'=[])
      using ld by (case-tac ξ, case-tac ξ', auto)
    {
      assume φ: ξ=[] ∧ 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 ∃ψ ψ' p. r ψ ψ' ∧ path-to p (conn c (ξ@ (φ # ξ'))) ψ
        ∧ 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-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-rew-step r $\varphi\ \psi \Longrightarrow$ *preserve-models r* $\Longrightarrow$ *propo-rew-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-rew-one-step-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 $(A \models \mathit{conn}\ c\ (\xi\ @\ \varphi\ \#\ \xi')) = (A \models \mathit{conn}\ c\ (\xi\ @\ \varphi'\ \#\ \xi'))$
by (*meson consistency-decompose-into-list wf wf-conn-no-arity-change-helper wf-conn-no-arity-change*)
}
then show $\forall A.\ A \models \mathit{conn}\ c\ (\xi\ @\ \varphi\ \#\ \xi') \longleftrightarrow A \models \mathit{conn}\ c\ (\xi\ @\ \varphi'\ \#\ \xi')$ **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-models f $\Longrightarrow$ *preserve-models g* $\Longrightarrow$ *preserve-models* (*f OO g*)
unfolding *preserve-models-def* **by** *auto*

lemma *star-consistency-preservation-explicit*:
assumes (*propo-rew-step r*)$\widehat{}$** $\varphi\ \psi$ **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-models r $\Longrightarrow$ *preserve-models* (*propo-rew-step r*)$\widehat{}$**
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 φ $\Longrightarrow$ *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 *φ1 φ2*)
note *IHφ1-0* = *this*(*1*)[*OF this*(*4*)] **and** *IHφ2-0* = *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* [φ*1*, φ*2*] = φ
using φ **by** *fastforce*

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

from *r IHφ1-0* **have** *IHφ1*: ¬ *all-subformula-st test-symb φ1* ⟹ ∃ *ψ ψ'*. *ψ* ≼ *φ1* ∧ *r ψ ψ'*
using *inc(1) subformula-trans le* **by** *blast*
from *r IHφ2-0* **have** *IHφ2*: ¬ *all-subformula-st test-symb φ2* ⟹ ∃ *ψ*. *ψ* ≼ *φ2* ∧ (∃ *ψ'*. *r ψ ψ'*)
using *inc(2) subformula-trans le* **by** *blast*
have *cases*: ¬*test-symb φ* ∨ ¬*all-subformula-st test-symb φ1* ∨ ¬*all-subformula-st test-symb φ2*
using *c nst* **by** *auto*
show ∃ *ψ ψ'*. *ψ* ≼ *φ* ∧ *r ψ ψ'*
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 ∀ *φ' ψ*. *φ'* ≼ Φ ⟶ *r φ' ψ* ⟶ *all-subformula-st test-symb φ'* ⟶ *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: ∀ *c ξ φ ξ' φ'*. *φ* ≼ 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'*))

Invariant while lifting of the Rewriting Relation

The condition *φ* ≼ Φ (that will by used with Φ = *φ* 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* ⇒ *'v propo* ⇒ *bool* **and** *test-symb*:: *'v propo* ⇒ *bool* **and** *x* :: *'v*
and *φ ψ* Φ:: *'v propo*
assumes *H*: ∀ *φ' ψ*. *φ'* ≼ Φ ⟶ *r φ' ψ* ⟶ *all-subformula-st test-symb φ'*
⟶ *all-subformula-st test-symb ψ*
and *H'*: ∀ (*c*:: *'v connective*) *ξ φ ξ' φ'*. *φ* ≼ 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**
propo-rew-step r φ ψ **and**
φ ≼ Φ **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 φ φ' c ξ ξ'*)
note *rel* = *this*(*1*) **and** *φ* = *this*(*2*) **and** *corr* = *this*(*3*) **and** Φ = *this*(*4*) **and** *nst* = *this*(*5*)
have *sq*: *φ* ≼ Φ
using Φ *corr subformula-into-subformula subformula-refl subformula-trans*
by (*metis in-set-conv-decomp*)
from *corr* **have** ∀ *ψ*. *ψ* ∈ *set* (*ξ* @ *φ* # *ξ'*) ⟶ *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* (ξ @ φ # ξ'))
using *all-subformula-st-decomp* **by** *blast*
ultimately have *test-symb*: *test-symb* (*conn c* (ξ @ φ' # ξ')) **using** *H' sq corr rel* **by** *blast*

have *wf-conn c* (ξ @ φ' # ξ')
by (*metis wf-conn-no-arity-change-helper corr wf-conn-no-arity-change*)
then show *all-subformula-st test-symb* (*conn c* (ξ @ φ' # ξ'))
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 φ ψ :: *'v propo*
assumes
H: $\forall \varphi'\ \psi.$ *r* φ' ψ $\longrightarrow$ *all-subformula-st test-symb* φ' $\longrightarrow$ *all-subformula-st test-symb* ψ **and**
H': $\forall$ (*c*:: *'v connective*) ξ φ ξ' φ'. *wf-conn c* (ξ @ φ # ξ') $\longrightarrow$ *test-symb* (*conn c* (ξ @ φ # ξ'))
$\longrightarrow$ *test-symb* φ' $\longrightarrow$ *test-symb* (*conn c* (ξ @ φ' # ξ')) **and**
propo-rew-step r φ ψ **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* φ ψ] *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 φ ψ :: *'v propo*
assumes
H: $\forall$ φ ψ. *propo-rew-step r* φ ψ $\longrightarrow$ *all-subformula-st test-symb* φ
$\longrightarrow$ *all-subformula-st test-symb* ψ **and**
H': $\forall$ (*c*:: *'v connective*) ξ φ ξ' φ'. $\varphi \preceq \Phi \longrightarrow$ *propo-rew-step r* φ φ'
$\longrightarrow$ *wf-conn c* (ξ @ φ # ξ') $\longrightarrow$ *test-symb* (*conn c* (ξ @ φ # ξ')) $\longrightarrow$ *test-symb* φ'
$\longrightarrow$ *test-symb* (*conn c* (ξ @ φ' # ξ')) **and**
$\varphi \preceq \Phi$ **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 *assms* **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' 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-conn c* ($\xi$ @ φ # ξ')
$\Longrightarrow$ *test-symb* (*conn c* (ξ @ φ # ξ')) $\Longrightarrow$ *test-symb* φ' $\Longrightarrow$ *test-symb* (*conn c* (ξ @ φ' # ξ'))
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* φ ψ) (*FAnd* (*FImp* φ ψ) (*FImp* ψ φ))

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

lemma *elim-equiv-explicit*: *elim-equiv* φ ψ $\Longrightarrow$ $\forall A.\ A \models \varphi \longleftrightarrow A \models \psi$
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* φ $\psi \longleftrightarrow A \models$ *FOr* (*FNot* φ) ψ
by *auto*

lemma *elim-imp-explicit*: *elim-imp* φ $\psi \Longrightarrow \forall A.\ A \models \varphi \longleftrightarrow A \models \psi$
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 ⟹ *no-imp-symb* (*conn c l*) ⟷ $c \neq$ *CImp*
by (*induction rule*: *wf-conn-induct*) *auto*

definition *no-imp* **where** *no-imp* ≡ *all-subformula-st no-imp-symb*
declare *no-imp-def*[*simp*]

lemma *no-imp-Imp*[*simp*]:
¬*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) ⟷ *no-imp* φ
no-imp (*FAnd* φ ψ) ⟷ (*no-imp* φ ∧ *no-imp* ψ)
no-imp (*FOr* φ ψ) ⟷ (*no-imp* φ ∧ *no-imp* ψ)
by *auto*

Invariant of the *elim-imp* transformation

lemma *elim-imp-no-equiv*:
elim-imp φ $\psi \Longrightarrow$ *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 $\exists\,\psi\ \psi'.\ \psi \preceq \varphi \wedge$ *elim-imp* $\psi\ \psi'$
proof −
have *test-symb-false-nullary*: $\forall\, x.$ *no-imp-symb FF* $\wedge$ *no-imp-symb FT* $\wedge$ *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* $\psi\ \forall\,\psi.\ \neg$*elim-imp FF* $\psi\ \forall\,\psi\ x.\ \neg$*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 φ ψ:: *'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* ⇒ *bool* **where**
no-T-F-symb-comp: $c \neq CF \Longrightarrow c \neq CT \Longrightarrow$ *wf-conn c l* $\Longrightarrow (\forall \varphi \in set\ l.\ \varphi \neq FT \wedge \varphi \neq FF)$
$\Longrightarrow$ *no-T-F-symb* (*conn c l*)

lemma *wf-conn-no-T-F-symb-iff*[*simp*]:
wf-conn c ψs $\Longrightarrow$
no-T-F-symb (*conn c ψs*) $\longleftrightarrow (c \neq CF \wedge c \neq CT \wedge (\forall \psi \in set\ \psi s.\ \psi \neq FF \wedge \psi \neq FT))$
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-T-F-symb (*FAnd φ ψ*) $\longleftrightarrow (\forall \chi \in set\ [\varphi, \psi].\ \chi \neq FF \wedge \chi \neq FT)$
no-T-F-symb (*FOr φ ψ*) $\longleftrightarrow (\forall \chi \in set\ [\varphi, \psi].\ \chi \neq FF \wedge \chi \neq FT)$
no-T-F-symb (*FEq φ ψ*) $\longleftrightarrow (\forall \chi \in set\ [\varphi, \psi].\ \chi \neq FF \wedge \chi \neq FT)$
no-T-F-symb (*FImp φ ψ*) $\longleftrightarrow (\forall \chi \in set\ [\varphi, \psi].\ \chi \neq FF \wedge \chi \neq FT)$
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
¬*no-T-F-symb* (*FT* :: *'v propo*)
¬*no-T-F-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*:
¬*no-T-F-symb* (*FNot φ*) $\Longrightarrow \varphi = FT \vee \varphi = FF$
proof (*rule ccontr*)
assume *n*: ¬ *no-T-F-symb* (*FNot φ*)
assume ¬ $(\varphi = FT \vee \varphi = FF)$
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) $\longleftrightarrow \neg(\varphi = FT \vee \varphi = FF)$
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 $\varphi = FT \vee \psi = FT \vee \varphi = FF \vee \psi = FF$
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 $\varphi = FT \vee \varphi = FF$
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 $\varphi = FT \vee \psi = FT \vee \varphi = FF \vee \psi = FF$
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 $\varphi \longleftrightarrow (\varphi = FF \vee \varphi = FT \vee$ *no-T-F* $\varphi)$
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 $\psi = FF \vee \psi = FT$ **using** *no-T-F-symb-except-toplevel-not-decom* **by** *blast*

```
  then show ?case using ElimTB5 ElimTB6 by blast
next
  case (binary φ' ψ1 ψ2)
  note IH1 = this(1) and IH2 = this(2) and φ' = this(3) and Fφ = this(4) and n = this(5)
  {
    assume φ' = FImp ψ1 ψ2 ∨ φ' = FEq ψ1 ψ2
    then have False using n Fφ 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': φ' = FAnd ψ1 ψ2 ∨ φ' = FOr ψ1 ψ2
    then have ψ1 = FT ∨ ψ2 = FT ∨ ψ1 = FF ∨ ψ2 = 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: ¬ no-T-F-except-top-level φ and no-equiv: no-equiv φ and no-imp: no-imp φ
  shows ∃ψ ψ'. ψ ≼ φ ∧ elimTB ψ ψ'
proof −
  have test-symb-false-nullary: ∀x. no-T-F-symb-except-toplevel (FF:: 'v propo)
    ∧ no-T-F-symb-except-toplevel FT ∧ 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) ψ ⟹ ¬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 ψ ≼ φ ⟹ ¬ no-T-F-symb-except-toplevel ψ ⟹ ∃ψ'. elimTB ψ ψ'
      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 φ ψ :: 'v propo
  assumes full (propo-rew-step elimTB) φ ψ
  and no-equiv φ and no-imp φ
  shows no-equiv ψ and no-imp ψ
proof −
  {
    fix φ ψ :: 'v propo
    have H: elimTB φ ψ ⟹ no-equiv φ ⟹  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 $\psi \Longrightarrow$ *no-imp* $\varphi \Longrightarrow$ *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 $\psi \Longrightarrow \forall A.\ A \models \varphi \longleftrightarrow A \models \psi$
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 $\varphi \longleftrightarrow (\varphi = FT \vee \varphi = FF \vee (\exists x.\ \varphi = FVar\ x))$
  by (*cases* φ) *auto*

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

lemma *subformula-conn-decomp-explicit*[*simp*]:
  fixes φ :: *'v propo* **and** *x* :: *'v*
  shows
    $\varphi \preceq FNot\ FT \longleftrightarrow (\varphi = FNot\ FT \vee \varphi = FT)$
    $\varphi \preceq FNot\ FF \longleftrightarrow (\varphi = FNot\ FF \vee \varphi = FF)$
    $\varphi \preceq FNot\ (FVar\ x) \longleftrightarrow (\varphi = FNot\ (FVar\ x) \vee \varphi = FVar\ x)$
  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 $\exists \psi\ \psi'.\ \psi \preceq \varphi \wedge$ *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: ζ ≼ φ
    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: ∀φ∈set (ξ @ ζ # ξ'). φ ≠ FT ∧ φ ≠ FF
        using corr wf-conn-no-T-F-symb-iff p by blast
      from rel incl have ζ'≠FT ∧ζ'≠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 ∀φ ∈ set (ξ @ ζ' # ξ'). φ ≠ FT ∧ φ ≠ FF using l by auto
      moreover have c ≠ CT ∧ c ≠ 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 ψ ⟹ no-equiv φ ⟹ 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 ψ ⟹ no-imp φ ⟹  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*]: $c = CAnd \vee c = COr \Longrightarrow c' = CAnd \vee c' = COr$
⟹ *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*]: $c = CAnd \vee c = COr \Longrightarrow c' = CAnd \vee c' = COr$
⟹ *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 $\psi \Longrightarrow \varphi = FT \vee \varphi = FF \Longrightarrow False$
by (*induct rule*: *push-conn-inside.induct*, *auto*)
} note H = *this*
fix φ
have *propo-rew-step* (*push-conn-inside c c'*) φ $\psi \Longrightarrow \varphi = FT \vee \varphi = FF \Longrightarrow False$
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] ⟹ *wf-conn c'* [φ, φ']
⟹ *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']] ⟹ *wf-conn c'* [φ, φ']
⟹ *not-c-in-c'-symb c c'* (*conn c* [ψ, *conn c'* [φ, φ']])

abbreviation *c-in-c'-symb c c'* $\varphi \equiv$ ¬*not-c-in-c'-symb c c'* φ

lemma *c-in-c'-symb-simp*:
not-c-in-c'-symb c c' $\xi \Longrightarrow \xi = FF \vee \xi = FT \vee \xi = FVar\ x \vee \xi = FNot\ FF \vee \xi = FNot\ FT$
$\vee\ \xi = FNot\ (FVar\ x) \Longrightarrow False$
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*]:
¬*not-c-in-c′-symb c c′ FF*
¬*not-c-in-c′-symb c c′ FT*
¬*not-c-in-c′-symb c c′* (*FVar x*)
¬*not-c-in-c′-symb c c′* (*FNot FF*)
¬*not-c-in-c′-symb c c′* (*FNot FT*)
¬*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′ ≡ *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′ ξ ⟹ *wf-conn c* [*φ*, *ψ*] ⟹ *ξ* = *conn c* [*φ*, *ψ*]
⟹ *not-c-in-c′-symb c c′* (*conn c* [*ψ*, *φ*])
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*: *ψ* = *conn c′* [*φ″*, *ψ′*] **using** *conn-inj* **by** *auto*
have *wf-conn c* [*conn c′* [*φ″*, *ψ′*], *φ*]
using *H*(1) *wf-conn-no-arity-change length-Cons* **by** *metis*
then show *not-c-in-c′-symb c c′* (*conn c* [*ψ*, *φ*])
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 *φ* = *conn c′* [*φ′*, *φ″*] **using** *conn-inj* **by** *auto*
moreover have *wf-conn c* [*ψ′*, *conn c′* [*φ′*, *φ″*]]
using *H*(1) *wf-conn-no-arity-change length-Cons* **by** *metis*
ultimately show *not-c-in-c′-symb c c′* (*conn c* [*ψ*, *φ*])
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 [*φ*, *ψ*] ⟹ *c-in-c′-symb c c′* (*conn c* [*φ*, *ψ*]) ⟷ *c-in-c′-symb c c′* (*conn c* [*ψ*, *φ*])
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* [*φ*, *ψ*]
shows *c-in-c′-only c c′* (*conn c* [*φ*, *ψ*]) ⟷ *c-in-c′-only c c′* (*conn c* [*ψ*, *φ*]) (**is** *?A* ⟷ *?B*)
proof −
have *?A* ⟷ (*c-in-c′-symb c c′* (*conn c* [*φ*, *ψ*])
∧ (∀ *ξ* ∈ *set* [*φ*, *ψ*]. *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 ... ⟷ (*c-in-c′-symb c c′* (*conn c* [*ψ*, *φ*])

```
                ∧ (∀ξ ∈ 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])
          ∧ (∀ξ ∈ set [ψ, φ]. all-subformula-st (c-in-c'-symb c c') ξ))
        ⟷ ?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 φ1 φ2 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' [φ1, φ2], ψ] ⟹ wf-conn c' [φ1, φ2]
    ⟹ ¬ c-in-c'-only c c' (conn c [conn c' [φ1, φ2], 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) ⟹ 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 ∨ c = COr and c': c' = CAnd ∨ c' = COr
  shows ψ ≼ φ ⟹ ¬ c-in-c'-symb c c' ψ ⟹ ∃ψ'. push-conn-inside c c' ψ ψ'
  apply (induct ψ rule: propo-induct-arity)
  apply auto[2]
proof −
  fix ψ1 ψ2 φ':: 'v propo
  assume IHψ1: ψ1 ≼ φ ⟹ ¬ c-in-c'-symb c c' ψ1 ⟹ Ex (push-conn-inside c c' ψ1)
  and IHψ2: ψ1 ≼ φ ⟹ ¬ c-in-c'-symb c c' ψ1 ⟹ Ex (push-conn-inside c c' ψ1)
  and φ': φ' = FAnd ψ1 ψ2 ∨ φ' = FOr ψ1 ψ2 ∨ φ' = FImp ψ1 ψ2 ∨ φ' = FEq ψ1 ψ2
  and inφ: φ' ≼ φ and n0: ¬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 [ψ1, ψ2]
    obtain a b where ψ1 = conn c' [a, b] ∨ ψ2 = 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': φ' ≠ conn c [ψ1, ψ2]
    have ∀φ c ca. ∃φ1 ψ1 ψ2 ψ1' ψ2' φ2'. conn (c::'v connective) [φ1, conn ca [ψ1, ψ2]] = φ
          ∨ conn c [conn ca [ψ1', ψ2'], φ2'] = φ ∨ 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: ¬c-in-c'-only c c' φ
  and c: c = CAnd ∨ c = COr and c': c' = CAnd ∨ c' = COr
  shows ∃ψ ψ'. ψ ≼ φ ∧ push-conn-inside c c' ψ ψ'
proof −
  have test-symb-false-nullary:
    ∀x. c-in-c'-symb c c' (FF:: 'v propo) ∧ c-in-c'-symb c c' FT
      ∧ 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 ψ ≼ φ ⟹ ¬ c-in-c'-symb c c' ψ ⟹ ∃ψ'. push-conn-inside c c' ψ ψ'
      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 φ ψ :: 'v propo
  shows propo-rew-step (push-conn-inside c c') φ ψ ⟹ no-T-F φ ⟹ no-T-F ψ
proof (induct rule: propo-rew-step.induct)
  case (global-rel φ ψ)
  then show no-T-F ψ
    by (cases rule: push-conn-inside.cases, auto)
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)
  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 ∀ζ ∈ 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: ∀ζ ∈ set (ξ @ φ' # ξ'). no-T-F ζ using φ' by auto
  then have n': ∀ζ ∈ set (ξ @ φ' # ξ'). ζ ≠  FF ∧ ζ ≠ 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 (ξ @ φ' # ξ')
    using wf wf-conn-no-arity-change by (metis wf-conn-no-arity-change-helper)
  {
    fix x :: 'v
    assume c = CT ∨ c = CF ∨ c = CVar x
    then have False using wf by auto
    then have no-T-F (conn c (ξ @ φ' # ξ')) by blast
  }
  moreover {
    assume c: c = CNot
    then have ξ = [] ξ' = [] using wf by auto
    then have no-T-F (conn c (ξ @ φ' # ξ'))
      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 ∈ binary-connectives
    then have no-T-F-symb (conn c (ξ @ φ' # ξ')) using wf' n' no-T-F-symb.simps by fastforce
    then have no-T-F (conn c (ξ @ φ' # ξ'))
      by (metis all-subformula-st-decomp-imp wf' n no-T-F-def)
  }
  ultimately show no-T-F (conn c (ξ @ φ' # ξ')) using connective-cases-arity by auto
qed


lemma simple-propo-rew-step-push-conn-inside-inv:
propo-rew-step (push-conn-inside c c') φ ψ ⟹ simple φ ⟹ 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 φ ψ :: 'v propo
  shows propo-rew-step (push-conn-inside c c') φ ψ ⟹ simple-not φ ⟹ simple-not ψ
proof (induct rule: propo-rew-step.induct)
  case (global-rel φ ψ)
  then show ?case by (cases φ, auto simp: push-conn-inside.simps)
next
  case (propo-rew-one-step-lift φ φ' ca ξ ξ') 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' # ξ' = [a, b]
        using wf ca list-length2-decomp wf-conn-bin-list-length
        by (metis (no-types) wf-conn-no-arity-change-helper)
      have ∀ζ ∈ set (ξ @ φ # ξ'). simple-not ζ
        by (metis wf all-subformula-st-decomp simple simple-not-def)
      then have ∀ζ ∈ set (ξ @ φ' # ξ'). 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 χs') 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' # χs') # 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 = [] χs' = [] 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 ⟹ ψ ≠ FT ∧ ψ ≠ 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 $c = CT \vee c = CF \vee c = CVar\ x$
then have ξ @ φ # $\xi' = []$ **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*: $\xi = []$ $\xi' = []$ **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*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* $\in$ *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 nφ by blast
      obtain a b where ab: [a, b] = (ξ @ φ # ξ')
        using corr c list-length2-decomp wf-conn-bin-list-length by metis
      then have ξ @ φ' # ξ' = [a, φ'] ∨ (ξ @ φ' # ξ') = [φ', 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 χ ∨ χ ∉ set (ξ @ φ' # ξ')
          using wf' by (metis (no-types) φ' all-subformula-st-decomp calculation insert-iff
            list.set(2))
      }
      then have ∀φ. φ ∈ set (ξ @ φ' # ξ') ⟶ all-subformula-st simple-not-symb φ
          by (metis (no-types))

      moreover have simple-not-symb (conn ca (ξ @ φ' # ξ'))
        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 (ξ @ φ' # ξ') using c calculation(1) by auto
      ultimately show all-subformula-st simple-not-symb (conn ca (ξ @ φ' # ξ'))
        by (metis all-subformula-st-decomp-imp)
    qed
  }
  moreover {
    fix ca :: 'v connective and ξ ξ' :: 'v propo list and φ φ' :: 'v propo
    have propo-rew-step (push-conn-inside c c') φ φ' ⟹ wf-conn ca (ξ @ φ # ξ')
      ⟹ simple-not-symb (conn ca (ξ @ φ # ξ')) ⟹ simple-not-symb φ'
      ⟹ simple-not-symb (conn ca (ξ @ φ' # ξ'))
      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 φ ψ :: 'v propo
    have H: propo-rew-step (push-conn-inside c c') φ ψ ⟹ no-T-F-except-top-level φ
      ⟹ no-T-F-except-top-level ψ
      proof −
        assume rel: propo-rew-step (push-conn-inside c c') φ ψ
        and no-T-F-except-top-level φ
        then have no-T-F φ ∨ φ = FF ∨ φ = FT
          by (metis no-T-F-symb-except-toplevel-all-subformula-st-no-T-F-symb)
        moreover {
          assume φ = FF ∨ φ = 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 φ ∧ φ ≠ FF ∧ φ ≠ 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 ≠ CT ∧ ca ≠ 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 ≠ CT ∧ ca ≠ CF using corr by auto
    have ζ: ∀ζ∈ set (ξ @ φ # ξ'). ζ≠FT ∧ ζ ≠ FF
      using corr no-T-F no-T-F-symb-except-toplevel-if-is-a-true-false by blast
    then have φ ≠ FT ∧ φ ≠ FF by auto
    from rel this have φ' ≠ FT ∧ φ' ≠ 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 ∀ζ ∈ set (ξ @ φ' # ξ'). ζ ≠ FT ∧ ζ ≠ 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 ⟹ no-equiv φ ⟹ 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 ⟹ no-imp φ ⟹ 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**
$c = CAnd \vee c = COr$ **and**
$c' = CAnd \vee c' = COr$
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
$\vee$ ($\exists$ φ'. ψ = *FNot* $\varphi' \wedge$ *simple* φ')
$\vee$ ($\exists$ *l*. ψ = *conn c l* $\wedge$ *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* $\psi \vee$ ($\exists$ φ'. ψ = *FNot* $\varphi' \wedge$ *simple* φ')
$\vee$ ($\exists$ *l*. ψ = *conn c l* $\wedge$ *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** $\varphi :: 'v$ *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* φ **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 ¬ *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'* φ
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'* φ
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* [φ, ψ]
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-c-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 $c\text{-}in\text{-}only\varphi 1$: $c\text{-}in\text{-}c'\text{-}only\ c\ c'\ (conn\ c1\ l1)$ **and** $c\text{-}in\text{-}c'\text{-}only\ c\ c'\ (conn\ c2\ l2)$
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*
$c\text{-}in\text{-}only\varphi 1$ *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 φ* ⟹ *grouped-by c φ* |
simple-not-is-grouped[*simp*]: *simple φ* ⟹ *grouped-by c* (*FNot φ*) |
connected-is-group[*simp*]: *grouped-by c φ* ⟹ *grouped-by c ψ* ⟹ *wf-conn c* [*φ*, *ψ*]
⟹ *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'* [*φ1*, *φ2*]) ⟹ *c' = CAnd* ∨ *c' = COr* ⟹ *wf-conn c'* [*φ1*, *φ2*]
⟹ *c' = c*
by (*induct conn c'* [*φ1*, *φ2*] *rule*: *only-c-inside-symb.induct*, *auto simp*: *conn-inj*)

lemma *only-c-inside-c-eq-c'*:
only-c-inside c (*conn c'* [*φ1*, *φ2*]) ⟹ *c' = CAnd* ∨ *c' = COr* ⟹ *wf-conn c'* [*φ1*, *φ2*] ⟹ *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* ≠ *CNot* **and** *c'*: *c' = CAnd* ∨ *c' = COr*
shows *only-c-inside c φ* ⟹ *grouped-by c φ* (**is** *?O φ* ⟹ *?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 φ1 φ2*)
note *IHφ1* = *this*(*1*) **and** *IHφ2* = *this*(*2*) **and** *φ* = *this*(*3*) **and** *only* = *this*(*4*)
have *φ-conn*: *φ* = *conn c* [*φ1*, *φ2*] **and** *wf*: *wf-conn c* [*φ1*, *φ2*]
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''* = [*φ1*, *φ2*] **using** *φ* **by** (*metis wf-conn-list*(*4*−*7*))
have *only-c-inside-symb c* (*conn c''* [*φ1*, *φ2*])
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 IH*φ1 *IH*φ2 φ-*conn only* φ **unfolding** *only-c-inside-def* **by** *auto*
moreover have *grouped-by c* φ2
using *wf* φ *IH*φ1 *IH*φ2 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]) $\Longrightarrow$ *c* $\neq$ *c'* $\Longrightarrow$ *wf-conn c'* [φ, ψ] $\Longrightarrow$ *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* $\Rightarrow$ *'a connective* $\Rightarrow$ *'a propo* $\Rightarrow$ *bool* **for** *c c'* **where**
grouped-is-super-grouped[*simp*]: *grouped-by c* φ $\Longrightarrow$ *super-grouped-by c c'* φ |
connected-is-super-group: *super-grouped-by c c'* φ $\Longrightarrow$ *super-grouped-by c c'* ψ $\Longrightarrow$ *wf-conn c* [φ, ψ]
$\Longrightarrow$ *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* $\vee$ *c* = *COr* **and** *c'*: *c'* = *CAnd* $\vee$ *c'* = *COr* **and** *cc'*: *c* $\neq$ *c'*
shows *no-equiv* φ $\Longrightarrow$ *no-imp* φ $\Longrightarrow$ *simple-not* φ $\Longrightarrow$ *c-in-c'-only c c'* φ
$\Longrightarrow$ *super-grouped-by c c'* φ
(**is** *?NE* φ $\Longrightarrow$ *?NI* φ $\Longrightarrow$ *?SN* φ $\Longrightarrow$ *?C* φ $\Longrightarrow$ *?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* $\vee$ φ = *FF* $\vee$ ($\exists$ *x*. φ = *FVar x*) **by** (*cases* φ, *auto*)
then show *?S* (*FNot* φ) **by** *auto*
next
case (*binary* φ φ1 φ2)
note *IH*φ1 = *this*(1) **and** *IH*φ2 = *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 $\vee$ φ = *FEq* φ1 φ2
then have *False* **using** *no-equiv no-imp* **by** *auto*
then have *?S* φ **by** *auto*
}

moreover {
assume φ: $\varphi = conn\ c'\ [\varphi 1, \varphi 2] \wedge wf\text{-}conn\ c'\ [\varphi 1, \varphi 2]$
have *c-in-c'-only*: $c\text{-}in\text{-}c'\text{-}only\ c\ c'\ \varphi 1 \wedge c\text{-}in\text{-}c'\text{-}only\ c\ c'\ \varphi 2 \wedge c\text{-}in\text{-}c'\text{-}symb\ c\ c'\ \varphi$
using *c-in-c'-only* φ' **unfolding** *c-in-c'-only-def* **by** *auto*
have *super-grouped-by c c'* $\varphi 1$ **using** φ *c' no-equiv no-imp simpleN IH*$\varphi 1$ *c-in-c'-only* **by** *auto*
moreover have *super-grouped-by c c'* $\varphi 2$
using φ *c' no-equiv no-imp simpleN IH*$\varphi 2$ *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: $\varphi = conn\ c\ [\varphi 1, \varphi 2] \wedge wf\text{-}conn\ c\ [\varphi 1, \varphi 2]$
then have *only-c-inside c* $\varphi 1 \wedge$ *only-c-inside c* $\varphi 2$
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* [$\varphi 1$, $\varphi 2$])
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* $\varphi \Longrightarrow$ *no-imp* $\varphi \Longrightarrow$ *simple-not* $\varphi \Longrightarrow$ *or-in-and-only* $\varphi \Longrightarrow$ *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 $\varphi \equiv$ *is-conj-with-TF* $\varphi \wedge$ *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 $\varphi' \Longrightarrow$ *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* ≡ *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* ⇒ *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* ≡
(*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-rew φ $\varphi' \Longrightarrow$ *is-dnf* φ'
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* φ $\psi \Longrightarrow \forall A.\ A \models \varphi \longleftrightarrow A \models \psi$

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 $\psi = FF \vee \psi = FT$ **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 $\psi 1 = FT \vee \psi 2 = FT \vee \psi 1 = FF \vee \psi 2 = FF$
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 $\exists \psi\ \psi'$. $\psi \preceq \varphi \wedge$ *elimTBFull* ψ ψ'
proof −
have *test-symb-false-nullary*:
$\forall x$. *no-T-F-symb-except-toplevel* (*FF*:: *'v propo*) $\wedge$ *no-T-F-symb-except-toplevel FT*
$\wedge$ *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* $\varphi\ \psi \Longrightarrow$ *no-T-F* $\varphi \Longrightarrow$ *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* $\varphi'\ \psi'$
  have $\forall$ *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 $\forall$ *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 $\varphi\ \varphi'$ :: *'v propo* **and** $\xi\ \xi'$ :: *'v propo list* **and** *c* :: *'v connective*
  assume *rel*: *propo-rew-step elim-equiv* $\varphi\ \varphi'$
  and *IH*: *no-T-F* $\varphi \Longrightarrow$ *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: $\forall \zeta \in$ *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 $\forall \zeta \in$ *set* (ξ @ φ' # ξ'). *no-T-F* ζ **using** ζ φ' *ab* **by** *fastforce*
    moreover
      have $\forall \zeta \in$ *set* (ξ @ φ # ξ'). $\zeta \neq$ *FT* ∧ $\zeta \neq$ *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 ∨ c = CF ∨ 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 ⟹ no-T-F-except-top-level φ
      ⟹ 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 φ = FT ∨ φ = 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 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-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: ζ ≼ φ
    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: ∀φ∈set (ξ @ ζ # ξ'). φ ≠ FT ∧ φ ≠ FF
        using corr wf-conn-no-T-F-symb-iff p by blast
      from rel incl have ζ'≠FT ∧ζ'≠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 ∀φ ∈ set (ξ @ ζ' # ξ'). φ ≠ FT ∧ φ ≠ FF using l by auto
   moreover have c ≠ CT ∧ c ≠ CF using corr by auto
   ultimately show no-T-F-symb (conn c (ξ @ ζ' # ξ'))
    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 ψ ⟹ no-T-F φ ⟹ 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 (ξ @ φ' # ξ')) using c empty no-T-F-comp-not IH by auto
 }
 moreover {
  assume c: c ∈ 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: φ = a ∨ φ = 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: ∀ζ ∈ set (ξ @ φ # ξ'). 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] ∨ 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 no-T-F (last (ξ @ φ' # ξ')) by (simp add: calculation)
   then have no-T-F-symb (conn c (ξ @ φ' # ξ'))
    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 (ξ @ φ' # ξ')) using c χ by fastforce
 }
 moreover {
  fix x
  assume c = CVar x ∨ c = CF ∨ 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 ⟹ no-T-F-except-top-level φ ⟹ 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 ⟹ no-T-F-except-top-level φ ⟹ 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 φ = FT ∨ φ = 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 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 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: ζ ≼ φ
    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: ∀φ∈set (ξ @ ζ # ξ'). φ ≠ FT ∧ φ ≠ FF
        using corr wf-conn-no-T-F-symb-iff p by blast
      from rel incl have ζ'≠FT ∧ζ'≠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* ($\xi\ @\ \zeta' \#\ \xi'$))
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' $\varphi\ \varphi' \Longrightarrow$ *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' $\varphi\ \varphi' \Longrightarrow$ *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* φ*1* φ*2*) ⟷ *grouped-by CAnd* φ*1* ∧ *grouped-by CAnd* φ*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* $\varphi\}$ *mset-of-formula* ψ = $\{$*mset-of-conj* $\psi\}$
    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* $\in\#$ 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* ⟷ φ = {#}›
by (*induction* φ) (*auto simp*: *Let-def*)

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

function *formula-of-msets* :: *'a literal multiset set* ⇒ *'a propo* **where**
‹*formula-of-msets* φs =
(*if* φs = {} ∨ *infinite* φs *then FT*
else
let v = (*SOME v*. *v* ∈ 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 = {#} ⟷ (φ = {#} ∨ φ = {#*v*#})›
using *remove1-mset-eqE* **by** *force*

definition *fun-of-set* **where**
‹*fun-of-set A x* = (*if Pos x* ∈ *A then True else if Neg x* ∈ *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* ∈# φ) φ)› **if** ‹φ ≠ {#}›
using *that* **by** (*auto simp*: *size-Diff-singleton-if some-in-eq*)
then show *?case*
using *IH*[*of* ‹*remove1-mset* (*SOME v*. *v* ∈# 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* φ ≠ *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* $\varphi 1$ $\varphi 2$) $\longleftrightarrow$ *is-conj-with-TF* $\varphi 1 \wedge$ *is-conj-with-TF* $\varphi 2$
unfolding *is-conj-with-TF-def* **by** (*subst super-grouped-by.simps*) *auto*

lemma *is-CNF-Fand*:
‹*is-cnf* (*FAnd* $\varphi\ \psi$) $\longleftrightarrow$ (*is-cnf* $\varphi \wedge$ *no-T-F* φ) $\wedge$ *is-cnf* $\psi \wedge$ *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) $\longleftrightarrow$ $\varphi \neq \{\#\}$›
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* $\in\#$ φ) φ)› **if** ‹$\varphi \neq \{\#\}$›
using *that* **by** (*auto simp*: *size-Diff-singleton-if some-in-eq*)
moreover have ‹*no-T-F* (*FVar* (*atm-of* (*SOME v*. *v* $\in\#$ φ)))›
by (*auto simp*: *no-T-F-def*)
ultimately show *?case*
using *IH*[*of* ‹*remove1-mset* (*SOME v*. *v* $\in\#$ 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** ‹$\{\#\} \notin \varphi$› **and** ‹$\varphi \neq \{\}$›
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** ‹$\{\#\} \notin \varphi$›
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