Formalisation of Ground Resolution and CDCL in Isabelle/HOL

Mathias Fleury and Jasmin Blanchette

November 6, 2018

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
  ⟨proof⟩
```

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
  ⟨proof⟩
```

```
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
  ⟨proof⟩
```

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
  ⟨proof⟩
```

1.1.2 Properties of the Abstraction

First we can define simplification rules.

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

$\mathit{wf\text{-}conn}\ \mathit{CT}\ l \Longrightarrow \mathit{conn}\ \mathit{CT}\ l = \mathit{FT}$
$\mathit{wf\text{-}conn}\ \mathit{CF}\ l \Longrightarrow \mathit{conn}\ \mathit{CF}\ l = \mathit{FF}$
$\mathit{wf\text{-}conn}\ (\mathit{CVar}\ x)\ l \Longrightarrow \mathit{conn}\ (\mathit{CVar}\ x)\ l = \mathit{FVar}\ x$
$\langle \mathit{proof} \rangle$

lemma *wf-conn-list-decomp*[*simp*]:
$\mathit{wf\text{-}conn}\ \mathit{CT}\ l \longleftrightarrow l = []$
$\mathit{wf\text{-}conn}\ \mathit{CF}\ l \longleftrightarrow l = []$
$\mathit{wf\text{-}conn}\ (\mathit{CVar}\ x)\ l \longleftrightarrow l = []$
$\mathit{wf\text{-}conn}\ \mathit{CNot}\ (\xi\ @\ \varphi\ \#\ \xi') \longleftrightarrow \xi = []\ \wedge\ \xi' = []$
$\langle \mathit{proof} \rangle$

lemma *wf-conn-list*:
$\mathit{wf\text{-}conn}\ c\ l \Longrightarrow \mathit{conn}\ c\ l = \mathit{FT} \longleftrightarrow (c = \mathit{CT}\ \wedge\ l = [])$
$\mathit{wf\text{-}conn}\ c\ l \Longrightarrow \mathit{conn}\ c\ l = \mathit{FF} \longleftrightarrow (c = \mathit{CF}\ \wedge\ l = [])$
$\mathit{wf\text{-}conn}\ c\ l \Longrightarrow \mathit{conn}\ c\ l = \mathit{FVar}\ x \longleftrightarrow (c = \mathit{CVar}\ x\ \wedge\ l = [])$
$\mathit{wf\text{-}conn}\ c\ l \Longrightarrow \mathit{conn}\ c\ l = \mathit{FAnd}\ a\ b \longleftrightarrow (c = \mathit{CAnd}\ \wedge\ l = a\ \#\ b\ \#\ [])$
$\mathit{wf\text{-}conn}\ c\ l \Longrightarrow \mathit{conn}\ c\ l = \mathit{FOr}\ a\ b \longleftrightarrow (c = \mathit{COr}\ \wedge\ l = a\ \#\ b\ \#\ [])$
$\mathit{wf\text{-}conn}\ c\ l \Longrightarrow \mathit{conn}\ c\ l = \mathit{FEq}\ a\ b \longleftrightarrow (c = \mathit{CEq}\ \wedge\ l = a\ \#\ b\ \#\ [])$
$\mathit{wf\text{-}conn}\ c\ l \Longrightarrow \mathit{conn}\ c\ l = \mathit{FImp}\ a\ b \longleftrightarrow (c = \mathit{CImp}\ \wedge\ l = a\ \#\ b\ \#\ [])$
$\mathit{wf\text{-}conn}\ c\ l \Longrightarrow \mathit{conn}\ c\ l = \mathit{FNot}\ a \longleftrightarrow (c = \mathit{CNot}\ \wedge\ l = a\ \#\ [])$
$\langle \mathit{proof} \rangle$

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

lemma *list-length2-decomp*: $\mathit{length}\ l = 2 \Longrightarrow (\exists\ a\ b.\ l = a\ \#\ b\ \#\ [])$
$\langle \mathit{proof} \rangle$

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

lemma *wf-conn-bin-list-length*:
fixes $l :: {}'v\ \mathit{propo}\ \mathit{list}$
assumes *conn*: $c \in \mathit{binary\text{-}connectives}$
shows $\mathit{length}\ l = 2 \longleftrightarrow \mathit{wf\text{-}conn}\ c\ l$
$\langle \mathit{proof} \rangle$

lemma *wf-conn-not-list-length*[*iff*]:
fixes $l :: {}'v\ \mathit{propo}\ \mathit{list}$
shows $\mathit{wf\text{-}conn}\ \mathit{CNot}\ l \longleftrightarrow \mathit{length}\ l = 1$
$\langle \mathit{proof} \rangle$

Decomposing the Not into an element is moreover very useful.

lemma *wf-conn-Not-decomp*:
fixes $l :: {}'v\ \mathit{propo}\ \mathit{list}$ **and** $a :: {}'v$
assumes *corr*: $\mathit{wf\text{-}conn}\ \mathit{CNot}\ l$
shows $\exists\ a.\ l = [a]$
$\langle \mathit{proof} \rangle$

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*:
$\mathit{length}\ l = \mathit{length}\ l' \Longrightarrow \mathit{wf\text{-}conn}\ c\ l \longleftrightarrow \mathit{wf\text{-}conn}\ c\ l'$
$\langle \mathit{proof} \rangle$

lemma *wf-conn-no-arity-change-helper*:
$\mathit{length}\ (\xi\ @\ \varphi\ \#\ \xi') = \mathit{length}\ (\xi\ @\ \varphi'\ \#\ \xi')$
⟨*proof*⟩

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

lemma *conn-inj-not*:
assumes *correct*: $\mathit{wf\text{-}conn}\ c\ l$
and *conn*: $\mathit{conn}\ c\ l = \mathit{FNot}\ \psi$
shows $c = \mathit{CNot}$ **and** $l = [\psi]$
⟨*proof*⟩

lemma *conn-inj*:
fixes $c\ \mathit{ca} :: {}'v\ \mathit{connective}$ **and** $l\ \psi s :: {}'v\ \mathit{propo\ list}$
assumes *corr*: $\mathit{wf\text{-}conn}\ \mathit{ca}\ l$
and *corr'*: $\mathit{wf\text{-}conn}\ c\ \psi s$
and *eq*: $\mathit{conn}\ \mathit{ca}\ l = \mathit{conn}\ c\ \psi s$
shows $\mathit{ca} = c \land \psi s = l$
⟨*proof*⟩

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

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*: $\mathit{FNot}\ \varphi \preceq \psi \Longrightarrow \varphi \preceq \psi$
⟨*proof*⟩

lemma *subformula-in-binary-conn*:
assumes *conn*: $c \in \mathit{binary\text{-}connectives}$
shows $f \preceq \mathit{conn}\ c\ [f,\ g]$
and $g \preceq \mathit{conn}\ c\ [f,\ g]$
⟨*proof*⟩

lemma *subformula-trans*:
$\psi \preceq \psi' \Longrightarrow \varphi \preceq \psi \Longrightarrow \varphi \preceq \psi'$
⟨*proof*⟩

lemma *subformula-leaf*:
fixes $\varphi\ \psi :: {}'v\ \mathit{propo}$
assumes *incl*: $\varphi \preceq \psi$
and *simple*: $\psi = \mathit{FT} \lor \psi = \mathit{FF} \lor \psi = \mathit{FVar}\ x$
shows $\varphi = \psi$
⟨*proof*⟩

lemma *subfurmula-not-incl-eq*:

assumes $\varphi \preceq \textit{conn}\ c\ l$
and $\textit{wf-conn}\ c\ l$
and $\forall \psi.\ \psi \in \textit{set}\ l \longrightarrow \neg\ \varphi \preceq \psi$
shows $\varphi = \textit{conn}\ c\ l$
$\langle \textit{proof} \rangle$

lemma *wf-subformula-conn-cases*:
$\textit{wf-conn}\ c\ l \Longrightarrow \varphi \preceq \textit{conn}\ c\ l \longleftrightarrow (\varphi = \textit{conn}\ c\ l \vee (\exists \psi.\ \psi \in \textit{set}\ l \wedge \varphi \preceq \psi))$
$\langle \textit{proof} \rangle$

lemma *subformula-decomp-explicit*[*simp*]:
$\varphi \preceq \textit{FAnd}\ \psi\ \psi' \longleftrightarrow (\varphi = \textit{FAnd}\ \psi\ \psi' \vee \varphi \preceq \psi \vee \varphi \preceq \psi')$ (**is** *?P FAnd*)
$\varphi \preceq \textit{FOr}\ \psi\ \psi' \longleftrightarrow (\varphi = \textit{FOr}\ \psi\ \psi' \vee \varphi \preceq \psi \vee \varphi \preceq \psi')$
$\varphi \preceq \textit{FEq}\ \psi\ \psi' \longleftrightarrow (\varphi = \textit{FEq}\ \psi\ \psi' \vee \varphi \preceq \psi \vee \varphi \preceq \psi')$
$\varphi \preceq \textit{FImp}\ \psi\ \psi' \longleftrightarrow (\varphi = \textit{FImp}\ \psi\ \psi' \vee \varphi \preceq \psi \vee \varphi \preceq \psi')$
$\langle \textit{proof} \rangle$

lemma *wf-conn-helper-facts*[*iff*]:
$\textit{wf-conn}\ \textit{CNot}\ [\varphi]$
$\textit{wf-conn}\ \textit{CT}\ []$
$\textit{wf-conn}\ \textit{CF}\ []$
$\textit{wf-conn}\ (\textit{CVar}\ x)\ []$
$\textit{wf-conn}\ \textit{CAnd}\ [\varphi, \psi]$
$\textit{wf-conn}\ \textit{COr}\ [\varphi, \psi]$
$\textit{wf-conn}\ \textit{CImp}\ [\varphi, \psi]$
$\textit{wf-conn}\ \textit{CEq}\ [\varphi, \psi]$
$\langle \textit{proof} \rangle$

lemma *exists-c-conn*: $\exists\ c\ l.\ \varphi = \textit{conn}\ c\ l \wedge \textit{wf-conn}\ c\ l$
$\langle \textit{proof} \rangle$

lemma *subformula-conn-decomp*[*simp*]:
assumes *wf*: $\textit{wf-conn}\ c\ l$
shows $\varphi \preceq \textit{conn}\ c\ l \longleftrightarrow (\varphi = \textit{conn}\ c\ l \vee (\exists\ \psi \in \textit{set}\ l.\ \varphi \preceq \psi))$ (**is** $?A \longleftrightarrow ?B$)
$\langle \textit{proof} \rangle$

lemma *subformula-leaf-explicit*[*simp*]:
$\varphi \preceq \textit{FT} \longleftrightarrow \varphi = \textit{FT}$
$\varphi \preceq \textit{FF} \longleftrightarrow \varphi = \textit{FF}$
$\varphi \preceq \textit{FVar}\ x \longleftrightarrow \varphi = \textit{FVar}\ x$
$\langle \textit{proof} \rangle$

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

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

lemma *vars-of-prop-incl-conn*:
fixes $\xi\ \xi' :: 'v\ \textit{propo list}$ **and** $\psi :: 'v\ \textit{propo}$ **and** $c :: 'v\ \textit{connective}$
assumes *corr*: $\textit{wf-conn}\ c\ l$ **and** *incl*: $\psi \in \textit{set}\ l$

shows *vars-of-prop* $\psi \subseteq$ *vars-of-prop* (*conn c l*)
⟨*proof*⟩

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
⟨*proof*⟩

1.1.4 Positions

Instead of 1 or 2 we use *L* or *R*

datatype *sign* = *L* | *R*

We use *nil* instead of ε.

fun *pos* :: *'v propo* ⇒ *sign list set* **where**
pos FF = {[]} |
pos FT = {[]} |
pos (*FVar x*) = {[]} |
pos (*FAnd* φ ψ) = {[]} ∪ { *L* # *p* | *p*. *p*∈ *pos* φ} ∪ { *R* # *p* | *p*. *p*∈ *pos* ψ} |
pos (*FOr* φ ψ) = {[]} ∪ { *L* # *p* | *p*. *p*∈ *pos* φ} ∪ { *R* # *p* | *p*. *p*∈ *pos* ψ} |
pos (*FEq* φ ψ) = {[]} ∪ { *L* # *p* | *p*. *p*∈ *pos* φ} ∪ { *R* # *p* | *p*. *p*∈ *pos* ψ} |
pos (*FImp* φ ψ) = {[]} ∪ { *L* # *p* | *p*. *p*∈ *pos* φ} ∪ { *R* # *p* | *p*. *p*∈ *pos* ψ} |
pos (*FNot* φ) = {[]} ∪ { *L* # *p* | *p*. *p*∈ *pos* φ}

lemma *finite-pos*: *finite* (*pos* φ)
⟨*proof*⟩

lemma *finite-inj-comp-set*:
fixes *s* :: *'v set*
assumes *finite*: *finite s*
and *inj*: *inj f*
shows *card* ({*f p* |*p*. *p* ∈ *s*}) = *card s*
⟨*proof*⟩

lemma *cons-inject*:
inj ((#) *s*)
⟨*proof*⟩

lemma *finite-insert-nil-cons*:
finite s ⟹ *card* (*insert* [] {*L* # *p* |*p*. *p* ∈ *s*}) = *1* + *card* {*L* # *p* |*p*. *p* ∈ *s*}
⟨*proof*⟩

lemma *cord-not*[*simp*]:
card (*pos* (*FNot* φ)) = *1* + *card* (*pos* φ)
⟨*proof*⟩

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*⟩

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

  ⟨proof⟩
```

```
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
  ⟨proof⟩
```

```
lemma subformula-path-exists:
  fixes φ φ':: 'v propo
  shows φ' ⪯ φ ⟹ ∃ p. path-to p φ φ'
⟨proof⟩
```

```
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
𝒜 ⊨ FT = True |
𝒜 ⊨ FF = False |
𝒜 ⊨ FVar v = (𝒜 v) |
𝒜 ⊨ FNot φ = (¬(𝒜⊨ φ)) |
𝒜 ⊨ FAnd φ₁ φ₂ = (𝒜⊨φ₁ ∧ 𝒜⊨φ₂) |
𝒜 ⊨ FOr φ₁ φ₂ = (𝒜⊨φ₁ ∨ 𝒜⊨φ₂) |
𝒜 ⊨ FImp φ₁ φ₂ = (𝒜⊨φ₁ ⟶ 𝒜⊨φ₂) |
𝒜 ⊨ FEq φ₁ φ₂ = (𝒜⊨φ₁ ⟷ 𝒜 ⊨φ₂)
```

```
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*:
  $\varphi \models f\ \psi \longleftrightarrow (\forall A.\ A \models FImp\ \varphi\ \psi)$
$\langle proof \rangle$

A shorter proof:

lemma $\varphi \models f\ \psi \longleftrightarrow (\forall A.\ A \models FImp\ \varphi\ \psi)$
  $\langle proof \rangle$

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

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

lemma *same-over-set-eval*:
  assumes $same\text{-}over\text{-}set\ A\ B\ (vars\text{-}of\text{-}prop\ \varphi)$
  shows $A \models \varphi \longleftrightarrow B \models \varphi$
  $\langle proof \rangle$

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 $propo\text{-}rew\text{-}step\ r\ \varphi\ \varphi' \Longrightarrow \exists\ \psi\ \psi'.\ \psi \preceq \varphi \land \psi' \preceq \varphi' \land r\ \psi\ \psi'$
  $\langle proof \rangle$

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

lemma *propo-rew-step-subformula-rec*:

fixes $\psi\ \psi'\ \varphi :: {'v}\ \mathit{propo}$
shows $\psi \preceq \varphi \Longrightarrow r\ \psi\ \psi' \Longrightarrow (\exists\,\varphi'.\ \psi' \preceq \varphi' \wedge \mathit{propo\text{-}rew\text{-}step}\ r\ \varphi\ \varphi')$
⟨*proof*⟩

lemma *propo-rew-step-subformula*:
$(\exists\,\psi\ \psi'.\ \psi \preceq \varphi \wedge r\ \psi\ \psi') \longleftrightarrow (\exists\,\varphi'.\ \mathit{propo\text{-}rew\text{-}step}\ r\ \varphi\ \varphi')$
⟨*proof*⟩

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 $A \models \mathit{conn}\ c\ l \longleftrightarrow A \models \mathit{conn}\ c\ l'$
⟨*proof*⟩

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

lemma *propo-rew-step-rewrite*:
fixes $\varphi\ \varphi' :: {'v}\ \mathit{propo}$ **and** $r :: {'v}\ \mathit{propo} \Rightarrow {'v}\ \mathit{propo} \Rightarrow \mathit{bool}$
assumes $\mathit{propo\text{-}rew\text{-}step}\ r\ \varphi\ \varphi'$
shows $\exists\,\psi\ \psi'\ p.\ r\ \psi\ \psi' \wedge \mathit{path\text{-}to}\ p\ \varphi\ \psi \wedge \mathit{replace\text{-}at}\ p\ \varphi\ \psi' = \varphi'$
⟨*proof*⟩

1.3.2 Consistency Preservation

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

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

lemma *propo-rew-step-preservers-val-explicit*:
$\mathit{propo\text{-}rew\text{-}step}\ r\ \varphi\ \psi \Longrightarrow \mathit{preserve\text{-}models}\ r \Longrightarrow \mathit{propo\text{-}rew\text{-}step}\ r\ \varphi\ \psi \Longrightarrow (\forall\,A.\ A \models\varphi \longleftrightarrow A\models\psi)$
⟨*proof*⟩

lemma *propo-rew-step-preservers-val'*:
assumes *preserve-models r*
shows *preserve-models* (*propo-rew-step r*)
⟨*proof*⟩

lemma *preserve-models-OO*[*intro*]:
$\mathit{preserve\text{-}models}\ f \Longrightarrow \mathit{preserve\text{-}models}\ g \Longrightarrow \mathit{preserve\text{-}models}\ (f\ \mathit{OO}\ g)$
⟨*proof*⟩

lemma *star-consistency-preservation-explicit*:
assumes $(\mathit{propo\text{-}rew\text{-}step}\ r)\hat{}{**}\ \varphi\ \psi$ **and** *preserve-models r*
shows $\forall\,A.\ A \models \varphi \longleftrightarrow A \models \psi$
⟨*proof*⟩

lemma *star-consistency-preservation*:
$\mathit{preserve\text{-}models}\ r \Longrightarrow \mathit{preserve\text{-}models}\ (\mathit{propo\text{-}rew\text{-}step}\ r)\hat{}{**}$
⟨*proof*⟩

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))
  ⟨*proof*⟩

lemma *full-propo-rew-step-subformula*:
full (*propo-rew-step* r) φ' $\varphi \Longrightarrow \neg(\exists\ \psi\ \psi'.\ \psi \preceq \varphi \land r\ \psi\ \psi')$
  ⟨*proof*⟩

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* ⇒ *bool*) ⇒ *'a propo* ⇒ *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*)
  ⟨*proof*⟩

lemma *all-subformula-st-test-symb-true-phi*:
  all-subformula-st test-symb $\varphi \Longrightarrow$ *test-symb* φ
  ⟨*proof*⟩

lemma *all-subformula-st-decomp-imp*:
  wf-conn c l $\Longrightarrow$ (*test-symb* (*conn c l*) $\land$ ($\forall\varphi\in$ *set l. all-subformula-st test-symb* φ))
  $\Longrightarrow$ *all-subformula-st test-symb* (*conn c l*)
  ⟨*proof*⟩

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*) $\land$ ($\forall\varphi\in$ *set l. all-subformula-st test-symb* φ))
  ⟨*proof*⟩

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*) $\land$ ($\forall\varphi\in$ *set l. all-subformula-st test-symb* φ))
  ⟨*proof*⟩

lemma *helper-fact*: $c \in$ *binary-connectives* $\longleftrightarrow$ (c = *COr* $\vee$ c = *CAnd* $\vee$ c = *CEq* $\vee$ c = *CImp*)
⟨*proof*⟩
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*⟩

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
⟨*proof*⟩

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* φ') $\longrightarrow$ ($\exists$ ψ. r φ' ψ) **and**
$\neg$ *all-subformula-st test-symb* φ
shows $\exists \psi\ \psi'.\ \psi \preceq \varphi \wedge r\ \psi\ \psi'$
⟨*proof*⟩

1.4.2 Invariant conservation

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

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

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

Invariant while lifting of the Rewriting Relation

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

lemma *propo-rew-step-inv-stay'*:
fixes *r*:: *'v propo* ⇒ *'v propo* ⇒ *bool* **and** *test-symb*:: *'v propo* ⇒ *bool* **and** *x* :: *'v*
and φ ψ Φ:: *'v propo*
assumes *H*: $\forall \varphi'\ \psi.\ \varphi' \preceq \Phi \longrightarrow$ *r* $\varphi'\ \psi \longrightarrow$ *all-subformula-st test-symb* φ'
$\longrightarrow$ *all-subformula-st test-symb* ψ
and *H'*: $\forall$ (*c*:: *'v connective*) $\xi\ \varphi\ \xi'\ \varphi'.\ \varphi \preceq \Phi \longrightarrow$ *propo-rew-step r* $\varphi\ \varphi'$
$\longrightarrow$ *wf-conn c* $(\xi\ @\ \varphi\ \#\ \xi') \longrightarrow$ *test-symb* (*conn c* $(\xi\ @\ \varphi\ \#\ \xi')$) $\longrightarrow$ *test-symb* φ'
$\longrightarrow$ *test-symb* (*conn c* $(\xi\ @\ \varphi'\ \#\ \xi')$) **and**
propo-rew-step r $\varphi\ \psi$ **and**
$\varphi \preceq \Phi$ **and**
all-subformula-st test-symb φ
shows *all-subformula-st test-symb* ψ
⟨*proof*⟩

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* ⇒ *'v propo* ⇒ *bool* **and** *test-symb*:: *'v propo* ⇒ *bool* **and** *x* :: *'v*
and φ ψ :: *'v propo*
assumes
H: $\forall \varphi'\ \psi.$ *r* $\varphi'\ \psi \longrightarrow$ *all-subformula-st test-symb* $\varphi' \longrightarrow$ *all-subformula-st test-symb* ψ **and**
H': $\forall$ (*c*:: *'v connective*) $\xi\ \varphi\ \xi'\ \varphi'.$ *wf-conn c* $(\xi\ @\ \varphi\ \#\ \xi') \longrightarrow$ *test-symb* (*conn c* $(\xi\ @\ \varphi\ \#\ \xi')$)
$\longrightarrow$ *test-symb* $\varphi' \longrightarrow$ *test-symb* (*conn c* $(\xi\ @\ \varphi'\ \#\ \xi')$) **and**
propo-rew-step r $\varphi\ \psi$ **and**
all-subformula-st test-symb φ
shows *all-subformula-st test-symb* ψ
⟨*proof*⟩

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* ⇒ *'v propo* ⇒ *bool* **and** *test-symb*:: *'v propo* ⇒ *bool* **and** *x* :: *'v*
and φ ψ :: *'v propo*
assumes
H: $\forall\ \varphi\ \psi.$ *propo-rew-step r* $\varphi\ \psi \longrightarrow$ *all-subformula-st test-symb* φ
$\longrightarrow$ *all-subformula-st test-symb* ψ **and**
H': $\forall$ (*c*:: *'v connective*) $\xi\ \varphi\ \xi'\ \varphi'.\ \varphi \preceq \Phi \longrightarrow$ *propo-rew-step r* $\varphi\ \varphi'$
$\longrightarrow$ *wf-conn c* $(\xi\ @\ \varphi\ \#\ \xi') \longrightarrow$ *test-symb* (*conn c* $(\xi\ @\ \varphi\ \#\ \xi')$) $\longrightarrow$ *test-symb* φ'
$\longrightarrow$ *test-symb* (*conn c* $(\xi\ @\ \varphi'\ \#\ \xi')$) **and**
$\varphi \preceq \Phi$ **and**
full: *full* (*propo-rew-step r*) $\varphi\ \psi$ **and**
init: *all-subformula-st test-symb* φ
shows *all-subformula-st test-symb* ψ
⟨*proof*⟩

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: $\forall\ \varphi\ \psi.$ *propo-rew-step r* $\varphi\ \psi \longrightarrow$ *all-subformula-st test-symb* φ
$\longrightarrow$ *all-subformula-st test-symb* ψ **and**
H': $\forall$ (*c*:: *'v connective*) $\xi\ \varphi\ \xi'\ \varphi'.$ *propo-rew-step r* $\varphi\ \varphi' \longrightarrow$ *wf-conn c* $(\xi\ @\ \varphi\ \#\ \xi')$

```
    ⟶ 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
 ⟨proof⟩

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
 ⟨proof⟩


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⟩

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 ⇒ 'v propo ⇒ bool where
elim-equiv[simp]: elim-equiv (FEq φ ψ) (FAnd (FImp φ ψ)  (FImp ψ φ))

lemma elim-equiv-transformation-consistent:
A ⊨ FEq φ ψ ⟷ A ⊨ FAnd (FImp φ ψ) (FImp ψ φ)
```

⟨*proof*⟩

lemma *elim-equiv-explicit*: *elim-equiv φ ψ ⟹ ∀ A. A ⊨ φ ⟷ A ⊨ ψ*
⟨*proof*⟩

lemma *elim-equiv-consistent*: *preserve-models elim-equiv*
⟨*proof*⟩

lemma *elimEquv-lifted-consistant*:
preserve-models (*full* (*propo-rew-step elim-equiv*))
⟨*proof*⟩

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

fun *no-equiv-symb* :: *'v propo ⇒ 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*) ⟷ *c ≠ CEq*
⟨*proof*⟩

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

lemma *no-equiv-eq*[*simp*]:
fixes *φ ψ* :: *'v propo*
shows
¬*no-equiv* (*FEq φ ψ*)
no-equiv FT
no-equiv FF
⟨*proof*⟩

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*) ⟷ *no-equiv φ*
no-equiv (*FAnd φ ψ*) ⟷ (*no-equiv φ* ∧ *no-equiv ψ*)
no-equiv (*FOr φ ψ*) ⟷ (*no-equiv φ* ∧ *no-equiv ψ*)
no-equiv (*FImp φ ψ*) ⟷ (*no-equiv φ* ∧ *no-equiv ψ*)
⟨*proof*⟩

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*: ¬ *no-equiv φ*
shows ∃*ψ ψ'*. *ψ* ⪯ *φ* ∧ *elim-equiv ψ ψ'*
⟨*proof*⟩

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*:
 $full\ (propo\text{-}rew\text{-}step\ elim\text{-}equiv)\ \varphi\ \psi \Longrightarrow no\text{-}equiv\ \psi$
 ⟨*proof*⟩

1.5.2 Eliminate Implication

After that, we can eliminate the implication symbols.

inductive *elim-imp* :: *'v propo* ⇒ *'v propo* ⇒ *bool* **where**
[*simp*]: $elim\text{-}imp\ (FImp\ \varphi\ \psi)\ (FOr\ (FNot\ \varphi)\ \psi)$

lemma *elim-imp-transformation-consistent*:
 $A \models FImp\ \varphi\ \psi \longleftrightarrow A \models FOr\ (FNot\ \varphi)\ \psi$
 ⟨*proof*⟩

lemma *elim-imp-explicit*: $elim\text{-}imp\ \varphi\ \psi \Longrightarrow \forall A.\ A \models \varphi \longleftrightarrow A \models \psi$
 ⟨*proof*⟩

lemma *elim-imp-consistent*: *preserve-models elim-imp*
 ⟨*proof*⟩

lemma *elim-imp-lifted-consistant*:
 preserve-models (*full* (*propo-rew-step elim-imp*))
 ⟨*proof*⟩

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

lemma *no-imp-symb-conn-characterization*:
 $wf\text{-}conn\ c\ l \Longrightarrow no\text{-}imp\text{-}symb\ (conn\ c\ l) \longleftrightarrow c \neq CImp$
 ⟨*proof*⟩

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

lemma *no-imp-Imp*[*simp*]:
 $\neg no\text{-}imp\ (FImp\ \varphi\ \psi)$
 no-imp FT
 no-imp FF
 ⟨*proof*⟩

lemma *all-subformula-st-decomp-explicit-imp*[*simp*]:
 fixes $\varphi\ \psi$ :: *'v propo*
 shows
  $no\text{-}imp\ (FNot\ \varphi) \longleftrightarrow no\text{-}imp\ \varphi$
  $no\text{-}imp\ (FAnd\ \varphi\ \psi) \longleftrightarrow (no\text{-}imp\ \varphi \wedge no\text{-}imp\ \psi)$
  $no\text{-}imp\ (FOr\ \varphi\ \psi) \longleftrightarrow (no\text{-}imp\ \varphi \wedge no\text{-}imp\ \psi)$
 ⟨*proof*⟩

Invariant of the *elim-imp* transformation

lemma *elim-imp-no-equiv*:
 $elim\text{-}imp\ \varphi\ \psi \Longrightarrow no\text{-}equiv\ \varphi \Longrightarrow\ no\text{-}equiv\ \psi$

⟨*proof*⟩

lemma *elim-imp-inv*:
fixes $\varphi\ \psi$ :: *'v propo*
assumes *full* (*propo-rew-step elim-imp*) $\varphi\ \psi$ **and** *no-equiv* φ
shows *no-equiv* ψ
⟨*proof*⟩

lemma *no-no-imp-elim-imp-step-exists*:
fixes φ :: *'v propo*
assumes *no-equiv*: $\neg$ *no-imp* φ
shows $\exists \psi\ \psi'.\ \psi \preceq \varphi \land$ *elim-imp* $\psi\ \psi'$
⟨*proof*⟩

lemma *no-imp-full-propo-rew-step-elim-imp*: *full* (*propo-rew-step elim-imp*) $\varphi\ \psi \Longrightarrow$ *no-imp* ψ
⟨*proof*⟩

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*⟩

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* $l.\ \varphi \neq FT \land \varphi \neq FF)$
$\Longrightarrow$ *no-T-F-symb* (*conn* $c\ l$)

lemma *wf-conn-no-T-F-symb-iff*[*simp*]:
wf-conn $c\ \psi s \Longrightarrow$
no-T-F-symb (*conn* $c\ \psi s$) $\longleftrightarrow (c \neq CF \land c \neq CT \land (\forall \psi \in$ *set* $\psi s.\ \psi \neq FF \land \psi \neq FT))$
⟨*proof*⟩

lemma *wf-conn-no-T-F-symb-iff-explicit*[*simp*]:
no-T-F-symb (*FAnd* $\varphi\ \psi$) $\longleftrightarrow (\forall \chi \in$ *set* $[\varphi, \psi].\ \chi \neq FF \land \chi \neq FT)$
no-T-F-symb (*FOr* $\varphi\ \psi$) $\longleftrightarrow (\forall \chi \in$ *set* $[\varphi, \psi].\ \chi \neq FF \land \chi \neq FT)$
no-T-F-symb (*FEq* $\varphi\ \psi$) $\longleftrightarrow (\forall \chi \in$ *set* $[\varphi, \psi].\ \chi \neq FF \land \chi \neq FT)$

no-T-F-symb $(\mathit{FImp}\ \varphi\ \psi) \longleftrightarrow (\forall \chi \in \mathit{set}\ [\varphi, \psi].\ \chi \neq \mathit{FF} \wedge \chi \neq \mathit{FT})$
⟨*proof*⟩

lemma *no-T-F-symb-false*[*simp*]:
fixes c :: *'v connective*
shows
$\neg$*no-T-F-symb* $(\mathit{FT}$ :: *'v propo*$)$
$\neg$*no-T-F-symb* $(\mathit{FF}$ :: *'v propo*$)$
⟨*proof*⟩

lemma *no-T-F-symb-bool*[*simp*]:
fixes x :: *'v*
shows *no-T-F-symb* $(\mathit{FVar}\ x)$
⟨*proof*⟩

lemma *no-T-F-symb-fnot-imp*:
$\neg$*no-T-F-symb* $(\mathit{FNot}\ \varphi) \Longrightarrow \varphi = \mathit{FT} \vee \varphi = \mathit{FF}$
⟨*proof*⟩

lemma *no-T-F-symb-fnot*[*simp*]:
no-T-F-symb $(\mathit{FNot}\ \varphi) \longleftrightarrow \neg(\varphi = \mathit{FT} \vee \varphi = \mathit{FF})$
⟨*proof*⟩

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* $(\mathit{FVar}\ x)$
⟨*proof*⟩

lemma *no-T-F-symb-except-toplevel-not-decom*:
$\varphi \neq \mathit{FT} \Longrightarrow \varphi \neq \mathit{FF} \Longrightarrow$ *no-T-F-symb-except-toplevel* $(\mathit{FNot}\ \varphi)$
⟨*proof*⟩

lemma *no-T-F-symb-except-toplevel-bin-decom*:
fixes $\varphi\ \psi$ :: *'v propo*
assumes $\varphi \neq \mathit{FT}$ **and** $\varphi \neq \mathit{FF}$ **and** $\psi \neq \mathit{FT}$ **and** $\psi \neq \mathit{FF}$
and c: $c \in$ *binary-connectives*
shows *no-T-F-symb-except-toplevel* $(\mathit{conn}\ c\ [\varphi, \psi])$
⟨*proof*⟩

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 $\mathit{FT} \in \mathit{set}\ l \vee \mathit{FF} \in \mathit{set}\ l$
shows $\neg$*no-T-F-symb-except-toplevel* $(\mathit{conn}\ c\ l)$
⟨*proof*⟩

lemma *no-T-F-symb-except-top-level-false-example*[*simp*]:
fixes $\varphi\ \psi$ :: *'v propo*
assumes $\varphi = FT \lor \psi = FT \lor \varphi = FF \lor \psi = FF$
shows
$\neg$ *no-T-F-symb-except-toplevel* (*FAnd* $\varphi\ \psi$)
$\neg$ *no-T-F-symb-except-toplevel* (*FOr* $\varphi\ \psi$)
$\neg$ *no-T-F-symb-except-toplevel* (*FImp* $\varphi\ \psi$)
$\neg$ *no-T-F-symb-except-toplevel* (*FEq* $\varphi\ \psi$)
⟨*proof*⟩

lemma *no-T-F-symb-except-top-level-false-not*[*simp*]:
fixes $\varphi\ \psi$ :: *'v propo*
assumes $\varphi = FT \lor \varphi = FF$
shows
$\neg$ *no-T-F-symb-except-toplevel* (*FNot* φ)
⟨*proof*⟩

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* $l \lor FF \in$ *set* l
shows $\neg$*no-T-F-except-top-level* (*conn c l*)
⟨*proof*⟩

lemma *no-T-F-except-top-level-false-example*[*simp*]:
fixes $\varphi\ \psi$ :: *'v propo*
assumes $\varphi = FT \lor \psi = FT \lor \varphi = FF \lor \psi = FF$
shows
$\neg$*no-T-F-except-top-level* (*FAnd* $\varphi\ \psi$)
$\neg$*no-T-F-except-top-level* (*FOr* $\varphi\ \psi$)
$\neg$*no-T-F-except-top-level* (*FEq* $\varphi\ \psi$)
$\neg$*no-T-F-except-top-level* (*FImp* $\varphi\ \psi$)
⟨*proof*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

lemma *no-T-F-no-T-F-except-top-level*:
no-T-F $\varphi \Longrightarrow$ *no-T-F-except-top-level* φ

⟨*proof*⟩

lemma *no-T-F-except-top-level-simp*[*simp*]: *no-T-F-except-top-level FF no-T-F-except-top-level FT*
⟨*proof*⟩

lemma *no-T-F-no-T-F-except-top-level'*[*simp*]:
$\textit{no-T-F-except-top-level}\ \varphi \longleftrightarrow (\varphi = \textit{FF} \lor \varphi = \textit{FT} \lor \textit{no-T-F}\ \varphi)$
⟨*proof*⟩

lemma *no-T-F-bin-decomp*[*simp*]:
assumes *c*: $c \in \textit{binary-connectives}$
shows $\textit{no-T-F}\ (\textit{conn}\ c\ [\varphi, \psi]) \longleftrightarrow (\textit{no-T-F}\ \varphi \land \textit{no-T-F}\ \psi)$
⟨*proof*⟩

lemma *no-T-F-bin-decomp-expanded*[*simp*]:
assumes *c*: $c = \textit{CAnd} \lor c = \textit{COr} \lor c = \textit{CEq} \lor c = \textit{CImp}$
shows $\textit{no-T-F}\ (\textit{conn}\ c\ [\varphi, \psi]) \longleftrightarrow (\textit{no-T-F}\ \varphi \land \textit{no-T-F}\ \psi)$
⟨*proof*⟩

lemma *no-T-F-comp-expanded-explicit*[*simp*]:
fixes $\varphi\ \psi :: \textit{'v propo}$
shows
$\textit{no-T-F}\ (\textit{FAnd}\ \varphi\ \psi) \longleftrightarrow (\textit{no-T-F}\ \varphi \land \textit{no-T-F}\ \psi)$
$\textit{no-T-F}\ (\textit{FOr}\ \varphi\ \psi) \longleftrightarrow (\textit{no-T-F}\ \varphi \land \textit{no-T-F}\ \psi)$
$\textit{no-T-F}\ (\textit{FEq}\ \varphi\ \psi) \longleftrightarrow (\textit{no-T-F}\ \varphi \land \textit{no-T-F}\ \psi)$
$\textit{no-T-F}\ (\textit{FImp}\ \varphi\ \psi) \longleftrightarrow (\textit{no-T-F}\ \varphi \land \textit{no-T-F}\ \psi)$
⟨*proof*⟩

lemma *no-T-F-comp-not*[*simp*]:
fixes $\varphi\ \psi :: \textit{'v propo}$
shows $\textit{no-T-F}\ (\textit{FNot}\ \varphi) \longleftrightarrow \textit{no-T-F}\ \varphi$
⟨*proof*⟩

lemma *no-T-F-decomp*:
fixes $\varphi\ \psi :: \textit{'v propo}$
assumes φ: $\textit{no-T-F}\ (\textit{FAnd}\ \varphi\ \psi) \lor \textit{no-T-F}\ (\textit{FOr}\ \varphi\ \psi) \lor \textit{no-T-F}\ (\textit{FEq}\ \varphi\ \psi) \lor \textit{no-T-F}\ (\textit{FImp}\ \varphi\ \psi)$
shows $\textit{no-T-F}\ \psi$ **and** $\textit{no-T-F}\ \varphi$
⟨*proof*⟩

lemma *no-T-F-decomp-not*:
fixes $\varphi :: \textit{'v propo}$
assumes φ: $\textit{no-T-F}\ (\textit{FNot}\ \varphi)$
shows $\textit{no-T-F}\ \varphi$
⟨*proof*⟩

lemma *no-T-F-symb-except-toplevel-step-exists*:
fixes $\varphi\ \psi :: \textit{'v propo}$
assumes $\textit{no-equiv}\ \varphi$ **and** $\textit{no-imp}\ \varphi$
shows $\psi \preceq \varphi \Longrightarrow \neg\ \textit{no-T-F-symb-except-toplevel}\ \psi \Longrightarrow \exists \psi'.\ \textit{elimTB}\ \psi\ \psi'$
⟨*proof*⟩

lemma *no-T-F-except-top-level-rew*:
fixes $\varphi :: \textit{'v propo}$
assumes *noTB*: $\neg\ \textit{no-T-F-except-top-level}\ \varphi$ **and** *no-equiv*: $\textit{no-equiv}\ \varphi$ **and** *no-imp*: $\textit{no-imp}\ \varphi$
shows $\exists \psi\ \psi'.\ \psi \preceq \varphi \land \textit{elimTB}\ \psi\ \psi'$
⟨*proof*⟩

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*⟩

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
  ⟨*proof*⟩

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 ⊨ *FNot* (*FAnd* φ ψ) ⟷ *A* ⊨ (*FOr* (*FNot* φ) (*FNot* ψ))
A ⊨ *FNot* (*FOr* φ ψ) ⟷ *A* ⊨ (*FAnd* (*FNot* φ) (*FNot* ψ))
A ⊨ *FNot* (*FNot* φ) ⟷ *A* ⊨ φ
  ⟨*proof*⟩

lemma *pushNeg-explicit*: *pushNeg* φ ψ ⟹ ∀ *A*. *A* ⊨ φ ⟷ *A* ⊨ ψ
  ⟨*proof*⟩

lemma *pushNeg-consistent*: *preserve-models pushNeg*
  ⟨*proof*⟩

lemma *pushNeg-lifted-consistant*:
preserve-models (*full* (*propo-rew-step pushNeg*))
  ⟨*proof*⟩

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

lemma *simple-decomp*:
  simple φ ⟷ (φ = *FT* ∨ φ = *FF* ∨ (∃ *x*. φ = *FVar x*))
  ⟨*proof*⟩

lemma *subformula-conn-decomp-simple*:
  fixes φ ψ :: *'v propo*
  assumes *s*: *simple* ψ
  shows φ ≼ *FNot* ψ ⟷ (φ = *FNot* ψ ∨ φ = ψ)

⟨*proof*⟩

lemma *subformula-conn-decomp-explicit*[*simp*]:
fixes φ :: *'v propo* **and** x :: *'v*
shows
$\varphi \preceq \mathit{FNot}\ \mathit{FT} \longleftrightarrow (\varphi = \mathit{FNot}\ \mathit{FT} \vee \varphi = \mathit{FT})$
$\varphi \preceq \mathit{FNot}\ \mathit{FF} \longleftrightarrow (\varphi = \mathit{FNot}\ \mathit{FF} \vee \varphi = \mathit{FF})$
$\varphi \preceq \mathit{FNot}\ (\mathit{FVar}\ x) \longleftrightarrow (\varphi = \mathit{FNot}\ (\mathit{FVar}\ x) \vee \varphi = \mathit{FVar}\ x)$
⟨*proof*⟩

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* φ ψ))
¬ *simple-not* (*FNot* (*FOr* φ ψ))
⟨*proof*⟩

lemma *simple-not-step-exists*:
fixes φ ψ :: *'v propo*
assumes *no-equiv* φ **and** *no-imp* φ
shows $\psi \preceq \varphi \Longrightarrow \neg$ *simple-not-symb* $\psi \Longrightarrow \exists \psi'.$ *pushNeg* ψ ψ'
⟨*proof*⟩

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* ψ ψ'
⟨*proof*⟩

lemma *no-T-F-except-top-level-pushNeg1*:
no-T-F-except-top-level (*FNot* (*FAnd* φ ψ)) $\Longrightarrow$ *no-T-F-except-top-level* (*FOr* (*FNot* φ) (*FNot* ψ))
⟨*proof*⟩

lemma *no-T-F-except-top-level-pushNeg2*:
no-T-F-except-top-level (*FNot* (*FOr* φ ψ)) $\Longrightarrow$ *no-T-F-except-top-level* (*FAnd* (*FNot* φ) (*FNot* ψ))
⟨*proof*⟩

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'))
⟨*proof*⟩

lemma *propo-rew-step-pushNeg-no-T-F-symb*:
propo-rew-step pushNeg φ ψ $\Longrightarrow$ *no-T-F-except-top-level* φ $\Longrightarrow$ *no-T-F-symb* φ $\Longrightarrow$ *no-T-F-symb* ψ
⟨*proof*⟩

lemma *propo-rew-step-pushNeg-no-T-F*:
propo-rew-step pushNeg φ ψ $\Longrightarrow$ *no-T-F* φ $\Longrightarrow$ *no-T-F* ψ

⟨*proof*⟩

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*⟩

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
⟨*proof*⟩

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*]: $c = CAnd \vee c = COr \Longrightarrow c' = CAnd \vee c' = COr$
$\Longrightarrow$ *push-conn-inside c c'* (*conn c* [*conn c'* [$\varphi 1$, $\varphi 2$], ψ])
(*conn c'* [*conn c* [$\varphi 1$, ψ], *conn c* [$\varphi 2$, ψ]]) |
push-conn-inside-r[*simp*]: $c = CAnd \vee c = COr \Longrightarrow c' = CAnd \vee c' = COr$
$\Longrightarrow$ *push-conn-inside c c'* (*conn c* [ψ, *conn c'* [$\varphi 1$, $\varphi 2$]])
(*conn c'* [*conn c* [ψ, $\varphi 1$], *conn c* [ψ, $\varphi 2$]])

lemma *push-conn-inside-explicit*: *push-conn-inside c c'* φ $\psi \Longrightarrow \forall A.\ A{\models}\varphi \longleftrightarrow A{\models}\psi$
⟨*proof*⟩

lemma *push-conn-inside-consistent*: *preserve-models* (*push-conn-inside c c'*)
⟨*proof*⟩

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*⟩

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

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

lemma *c-in-c'-symb-simp*:
not-c-in-c'-symb c c' $\xi \Longrightarrow \xi = FF \vee \xi = FT \vee \xi = FVar\ x \vee \xi = FNot\ FF \vee \xi = FNot\ FT$
$\vee\ \xi = FNot\ (FVar\ x) \Longrightarrow False$

⟨*proof*⟩

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*))
⟨*proof*⟩

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*))
⟨*proof*⟩

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*⟩

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*])
⟨*proof*⟩

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*⟩

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*])
⟨*proof*⟩

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*⟩

lemma *c-in-c′-symb-step-exists*:
fixes *φ* :: *′v propo*
assumes *c*: *c* = *CAnd* ∨ *c* = *COr* **and** *c′*: *c′* = *CAnd* ∨ *c′* = *COr*

shows $\psi \preceq \varphi \Longrightarrow \neg$ *c-in-c′-symb c c′* $\psi \Longrightarrow \exists \psi'.$ *push-conn-inside c c′* ψ ψ'
⟨*proof*⟩

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 $\exists \psi\ \psi'.\ \psi \preceq \varphi \wedge$ *push-conn-inside c c′* ψ ψ'
⟨*proof*⟩

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*⟩

lemma *simple-propo-rew-step-push-conn-inside-inv*:
propo-rew-step (*push-conn-inside c c′*) φ ψ ⟹ *simple* φ ⟹ *simple* ψ
⟨*proof*⟩

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*⟩

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'))
⟨*proof*⟩

lemma *push-conn-inside-not-true-false*:
push-conn-inside c c′ φ ψ ⟹ $\psi \neq$ *FT* ∧ $\psi \neq$ *FF*
⟨*proof*⟩

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*⟩

lemma *push-conn-inside-full-propo-rew-step*:
fixes φ ψ :: *′v propo*
assumes
no-equiv φ **and**
no-imp φ **and**
full (*propo-rew-step* (*push-conn-inside c c′*)) φ ψ **and**
no-T-F-except-top-level φ **and**

simple-not φ **and**
$c = CAnd \lor c = COr$ **and**
$c' = CAnd \lor c' = COr$
shows *c-in-c'-only* $c\ c'\ \psi$
⟨*proof*⟩

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

inductive *only-c-inside-symb* :: *'v connective* ⇒ *'v propo* ⇒ *bool* **for** c :: *'v connective* **where**
simple-only-c-inside[*simp*]: *simple* $\varphi \Longrightarrow$ *only-c-inside-symb* $c\ \varphi$ |
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) ⟨*proof*⟩

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

lemma *only-c-inside-symb-decomp*:
only-c-inside-symb $c\ \psi \longleftrightarrow$ (*simple* ψ
$\lor$ ($\exists\ \varphi'.\ \psi =$ *FNot* $\varphi' \land$ *simple* φ')
$\lor$ ($\exists\, l.\ \psi =$ *conn* $c\ l \land$ *wf-conn* $c\ l$))
⟨*proof*⟩

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
⟨*proof*⟩

lemma *only-c-inside-decomp-not*[*simp*]:
assumes c: $c \neq$ *CNot*
shows *only-c-inside* c (*FNot* ψ) $\longleftrightarrow$ *simple* ψ
⟨*proof*⟩

lemma *only-c-inside-decomp*:
only-c-inside $c\ \varphi \longleftrightarrow$
($\forall\,\psi.\ \psi \preceq \varphi \longrightarrow$ (*simple* $\psi \lor$ ($\exists\ \varphi'.\ \psi =$ *FNot* $\varphi' \land$ *simple* φ')
$\lor$ ($\exists\, l.\ \psi =$ *conn* $c\ l \land$ *wf-conn* $c\ l$)))
⟨*proof*⟩

lemma *only-c-inside-c-c'-false*:
fixes $c\ c'$ :: *'v connective* **and** l :: *'v propo list* **and** φ :: *'v propo*
assumes *cc'*: $c \neq c'$ **and** c: $c = CAnd \lor c = COr$ **and** c': $c' = CAnd \lor c' = COr$
and *only*: *only-c-inside* $c\ \varphi$ **and** *incl*: *conn* $c'\ l \preceq \varphi$ **and** *wf*: *wf-conn* $c'\ l$
shows *False*
⟨*proof*⟩

lemma *only-c-inside-implies-c-in-c'-symb*:
assumes δ: $c \neq c'$ **and** c: $c = CAnd \lor c = COr$ **and** c': $c' = CAnd \lor c' = COr$
shows *only-c-inside* $c\ \varphi \Longrightarrow$ *c-in-c'-symb* $c\ c'\ \varphi$
⟨*proof*⟩

lemma *c-in-c′-symb-decomp-level1*:
  fixes *l* :: *′v propo list* **and** *c c′ ca* :: *′v connective*
  shows *wf-conn ca l* ⟹ *ca* ≠ *c* ⟹ *c-in-c′-symb c c′* (*conn ca l*)
⟨*proof*⟩

lemma *only-c-inside-implies-c-in-c′-only*:
  assumes *δ*: *c* ≠ *c′* **and** *c*: *c* = *CAnd* ∨ *c* = *COr* **and** *c′*: *c′* = *CAnd* ∨ *c′* = *COr*
  shows *only-c-inside c φ* ⟹ *c-in-c′-only c c′ φ*
  ⟨*proof*⟩

lemma *c-in-c′-symb-c-implies-only-c-inside*:
  assumes *δ*: *c* = *CAnd* ∨ *c* = *COr c′* = *CAnd* ∨ *c′* = *COr c* ≠ *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* ⟹ *c-in-c′-only c c′* (*conn c l*) ⟹ (∀*ψ*∈ *set l*. *only-c-inside c ψ*)
⟨*proof*⟩

Push Conjunction

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

lemma *pushConj-consistent*: *preserve-models pushConj*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

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*
  ⟨*proof*⟩

Push Disjunction

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

lemma *pushDisj-consistent*: *preserve-models pushDisj*
  ⟨*proof*⟩

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')
⟨*proof*⟩

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
⟨*proof*⟩

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
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *only-c-inside-symb-c-eq-c′*:
only-c-inside-symb c (*conn c′* [$\varphi 1$, $\varphi 2$]) ⟹ $c' = CAnd \lor c' = COr$ ⟹ *wf-conn c′* [$\varphi 1$, $\varphi 2$]
⟹ $c' = c$
⟨*proof*⟩

lemma *only-c-inside-c-eq-c′*:

only-c-inside c (*conn c'* [$\varphi 1$, $\varphi 2$]) $\Longrightarrow$ $c' = CAnd \lor c' = COr$ $\Longrightarrow$ *wf-conn c'* [$\varphi 1$, $\varphi 2$] $\Longrightarrow$ $c = c'$
⟨*proof*⟩

lemma *only-c-inside-imp-grouped-by*:
assumes *c*: $c \neq CNot$ **and** *c'*: $c' = CAnd \lor c' = COr$
shows *only-c-inside c* φ $\Longrightarrow$ *grouped-by c* φ (**is** *?O* φ $\Longrightarrow$ *?G* φ)
⟨*proof*⟩

lemma *grouped-by-false*:
grouped-by c (*conn c'* [φ, ψ]) $\Longrightarrow$ $c \neq c'$ $\Longrightarrow$ *wf-conn c'* [φ, ψ] $\Longrightarrow$ *False*
⟨*proof*⟩

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*))
⟨*proof*⟩

lemma *c-in-c'-only-super-grouped-by*:
assumes *c*: $c = CAnd \lor c = COr$ **and** *c'*: $c' = CAnd \lor c' = COr$ **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*⟩

1.6.2 Conjunctive Normal Form

Definition

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

lemma *or-in-and-only-conjunction-in-disj*:
shows *no-equiv* φ $\Longrightarrow$ *no-imp* φ $\Longrightarrow$ *simple-not* φ $\Longrightarrow$ *or-in-and-only* φ $\Longrightarrow$ *is-conj-with-TF* φ
⟨*proof*⟩

definition *is-cnf* **where**
is-cnf φ $\equiv$ *is-conj-with-TF* φ $\land$ *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*
⟨*proof*⟩

lemma *cnf-rew-is-cnf*: *cnf-rew* φ φ' $\Longrightarrow$ *is-cnf* φ'
⟨*proof*⟩

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* φ $\Longrightarrow$ *no-imp* φ $\Longrightarrow$ *simple-not* φ $\Longrightarrow$ *and-in-or-only* φ $\Longrightarrow$ *is-disj-with-TF* φ
⟨*proof*⟩

definition *is-dnf* :: *'a propo* ⇒ *bool* **where**
is-dnf φ $\longleftrightarrow$ *is-disj-with-TF* φ ∧ *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*
⟨*proof*⟩

theorem *dnf-transformation-correction*:
dnf-rew φ φ' $\Longrightarrow$ *is-dnf* φ'
⟨*proof*⟩

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*⟩

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*⟩

The same applies here. We do not need the assumption, but the deep link between ¬ *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*: ¬ *no-T-F-except-top-level* φ
  shows $\exists\,\psi\ \psi'.\ \psi \preceq \varphi \land$ *elimTBFull* ψ ψ'
⟨*proof*⟩

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
  ⟨*proof*⟩

1.7.2 More invariants

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

lemma *propo-rew-step-ElimEquiv-no-T-F*: *propo-rew-step elim-equiv* φ $\psi \Longrightarrow$ *no-T-F* $\varphi \Longrightarrow$ *no-T-F* ψ
⟨*proof*⟩

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*⟩

lemma *propo-rew-step-ElimImp-no-T-F*: *propo-rew-step elim-imp* φ ψ $\Longrightarrow$ *no-T-F* φ $\Longrightarrow$ *no-T-F* ψ
⟨*proof*⟩

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*⟩

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'*
  ⟨*proof*⟩

theorem *cnf-transformation-correction*:
    dnf-rew' φ φ' $\Longrightarrow$ *is-dnf* φ'
  ⟨*proof*⟩

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'*
  ⟨*proof*⟩

theorem *cnf'-transformation-correction*:
  cnf-rew' φ φ' $\Longrightarrow$ *is-cnf* φ'
  ⟨*proof*⟩

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*)
  ⟨*proof*⟩

lemma *grouped-by-COr-FNot*:
  grouped-by *COr* (*FNot* φ) ⟷ (∃ *v*. φ = *FVar* *v* ∨ φ = *FF* ∨ φ = *FT*)
  ⟨*proof*⟩

lemma
  shows *no-T-F-FF*[*simp*]: ¬*no-T-F* *FF* **and**
    no-T-F-FT[*simp*]: ¬*no-T-F* *FT*
  ⟨*proof*⟩

lemma *grouped-by-CAnd-FAnd*:
  grouped-by *CAnd* (*FAnd* φ*1* φ*2*) ⟷ *grouped-by* *CAnd* φ*1* ∧ *grouped-by* *CAnd* φ*2*
  ⟨*proof*⟩

lemma *grouped-by-COr-FOr*:
  grouped-by *COr* (*FOr* φ*1* φ*2*) ⟷ *grouped-by* *COr* φ*1* ∧ *grouped-by* *COr* φ*2*
  ⟨*proof*⟩

lemma *grouped-by-COr-FAnd*[*simp*]: ¬ *grouped-by* *COr* (*FAnd* φ*1* φ*2*)
  ⟨*proof*⟩

lemma *grouped-by-COr-FEq*[*simp*]: ¬ *grouped-by* *COr* (*FEq* φ*1* φ*2*)
  ⟨*proof*⟩

lemma [*simp*]: ¬*grouped-by* *COr* (*FImp* φ ψ)
  ⟨*proof*⟩

lemma [*simp*]: ¬ *is-conj-with-TF* (*FImp* φ ψ)
  ⟨*proof*⟩

```
lemma [simp]: ¬ is-conj-with-TF (FEq φ ψ)
  ⟨proof⟩

lemma is-conj-with-TF-Fand:
  is-conj-with-TF (FAnd φ1 φ2) ⟹ is-conj-with-TF φ1 ∧ is-conj-with-TF φ2
  ⟨proof⟩

lemma is-conj-with-TF-FOr:
  is-conj-with-TF (FOr φ1 φ2) ⟹ grouped-by COr φ1 ∧ grouped-by COr φ2
  ⟨proof⟩

lemma grouped-by-COr-mset-of-formula:
  grouped-by COr φ ⟹ mset-of-formula φ = (if φ = FT then {} else {mset-of-conj φ})
  ⟨proof⟩
```

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: (⊨) uses a function assigning *True* or *False*, while (⊨*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 φ ⟷ Partial-Herbrand-Interpretation.true-clss ({Pos v|v. A v} ∪ {Neg v|v. ¬A v})
    (mset-of-formula φ)
  ⟨proof⟩

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

lemma formula-of-mset-empty[simp]: ‹formula-of-mset {#} = FF›
  ⟨proof⟩

lemma formula-of-mset-empty-iff[iff]: ‹formula-of-mset φ = FF ⟷ φ = {#}›
  ⟨proof⟩

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})))›
  ⟨proof⟩
termination
  ⟨proof⟩

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

lemma remove1-mset-empty-iff:
  ‹remove1-mset v φ = {#} ⟷ (φ = {#} ∨ φ = {#v#})›
  ⟨proof⟩

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⟩
lemma no-T-F-formula-of-mset: ‹no-T-F (formula-of-mset φ)› if ‹formula-of-mset φ ≠ FF› for φ
  ⟨proof⟩

lemma mset-of-conj-formula-of-mset[simp]: ‹mset-of-conj(formula-of-mset φ) = φ› for φ
⟨proof⟩

lemma mset-of-formula-formula-of-mset [simp]: ‹mset-of-formula (formula-of-mset φ) = {φ}› for φ
⟨proof⟩

lemma formula-of-mset-is-cnf: ‹is-cnf (formula-of-mset φ)›
  ⟨proof⟩

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}›
  ⟨proof⟩

lemma is-conj-with-TF-Fand-iff:
  is-conj-with-TF (FAnd φ1 φ2) ⟷ is-conj-with-TF φ1 ∧ is-conj-with-TF φ2
  ⟨proof⟩

lemma is-CNF-Fand:
  ‹is-cnf (FAnd φ ψ) ⟷ (is-cnf φ ∧ no-T-F φ) ∧ is-cnf ψ ∧ no-T-F ψ›
  ⟨proof⟩

lemma no-T-F-formula-of-mset-iff: ‹no-T-F (formula-of-mset φ) ⟷ φ ≠ {#}›
⟨proof⟩

lemma no-T-F-formula-of-msets:
  assumes ‹finite φ› and ‹{#} ∉ φ› and ‹φ ≠ {}›
  shows ‹no-T-F (formula-of-msets (φ))›
  ⟨proof⟩

lemma is-cnf-formula-of-msets:
  assumes ‹finite φ› and ‹{#} ∉ φ›
  shows ‹is-cnf (formula-of-msets φ)›
  ⟨proof⟩

lemma mset-of-formula-formula-of-msets:
  assumes ‹finite φ›
  shows ‹mset-of-formula (formula-of-msets φ) = φ›
```

⟨*proof*⟩

lemma
assumes ‹*consistent-interp A*› **and** ‹*total-over-set A UNIV*› **and** ‹*finite* φ› **and** ‹{#} $\notin$ φ›
shows ‹*eval* (*fun-of-set A*) (*formula-of-msets* φ) $\longleftrightarrow$ *Partial-Herbrand-Interpretation.true-clss A* φ›
⟨*proof*⟩

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

begin

Chapter 2

Resolution-based techniques

This chapter contains the formalisation of resolution and superposition.

2.1 Resolution

2.1.1 Simplification Rules

inductive *simplify* :: *'v clause-set* $\Rightarrow$ *'v clause-set* $\Rightarrow$ *bool* **for** *N* :: *'v clause set* **where**
tautology-deletion:
add-mset (*Pos P*) (*add-mset* (*Neg P*) *A*) $\in$ *N* $\Longrightarrow$ *simplify N* (*N* $-$ {*add-mset* (*Pos P*) (*add-mset* (*Neg P*) *A*)})|
condensation:
add-mset L (*add-mset L A*) $\in$ *N* $\Longrightarrow$ *simplify N* (*N* $-$ {*add-mset L* (*add-mset L A*)} $\cup$ {*add-mset L A*}) |
subsumption:
A $\in$ *N* $\Longrightarrow$ *A* $\subset\#$ *B* $\Longrightarrow$ *B* $\in$ *N* $\Longrightarrow$ *simplify N* (*N* $-$ {*B*})

lemma *simplify-preserve-models'*:
fixes *N N'* :: *'v clause-set*
assumes *simplify N N'*
and *total-over-m I N*
shows $I \models s\ N' \longrightarrow I \models s\ N$
⟨*proof*⟩

lemma *simplify-preserve-models*:
fixes *N N'* :: *'v clause-set*
assumes *simplify N N'*
and *total-over-m I N*
shows $I \models s\ N \longrightarrow I \models s\ N'$
⟨*proof*⟩

lemma *simplify-preserve-models''*:
fixes *N N'* :: *'v clause-set*
assumes *simplify N N'*
and *total-over-m I N'*
shows $I \models s\ N \longrightarrow I \models s\ N'$
⟨*proof*⟩

lemma *simplify-preserve-models-eq*:
fixes *N N'* :: *'v clause-set*
assumes *simplify N N'*

and *total-over-m I N*
shows $I \models s\ N \longleftrightarrow I \models s\ N'$
⟨*proof*⟩

lemma *simplify-preserves-finite*:
assumes *simplify* $\psi\ \psi'$
shows *finite* $\psi \longleftrightarrow$ *finite* ψ'
⟨*proof*⟩

lemma *rtranclp-simplify-preserves-finite*:
assumes *rtranclp simplify* $\psi\ \psi'$
shows *finite* $\psi \longleftrightarrow$ *finite* ψ'
⟨*proof*⟩

lemma *simplify-atms-of-ms*:
assumes *simplify* $\psi\ \psi'$
shows *atms-of-ms* $\psi' \subseteq$ *atms-of-ms* ψ
⟨*proof*⟩

lemma *rtranclp-simplify-atms-of-ms*:
assumes *rtranclp simplify* $\psi\ \psi'$
shows *atms-of-ms* $\psi' \subseteq$ *atms-of-ms* ψ
⟨*proof*⟩

lemma *factoring-imp-simplify*:
assumes $\{\#L,\ L\#\} + C \in N$
shows $\exists N'.$ *simplify* $N\ N'$
⟨*proof*⟩

2.1.2 Unconstrained Resolution

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

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

lemma *uncon-res-increasing*:
assumes *uncon-res* $S\ S'$ **and** $\psi \in S$
shows $\psi \in S'$
⟨*proof*⟩

lemma *rtranclp-uncon-inference-increasing*:
assumes *rtranclp uncon-res* $S\ S'$ **and** $\psi \in S$
shows $\psi \in S'$
⟨*proof*⟩

Subsumption

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

lemma *subsumes-refl*[*simp*]:
subsumes χ χ
⟨*proof*⟩

lemma *subsumes-subsumption*:
assumes *subsumes D* χ
and *C* ⊂# *D* **and** ¬*tautology* χ
shows *subsumes C* χ ⟨*proof*⟩

lemma *subsumes-tautology*:
assumes *subsumes* (*add-mset* (*Pos P*) (*add-mset* (*Neg P*) *C*)) χ
shows *tautology* χ
⟨*proof*⟩

2.1.3 Inference Rule

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

inductive *inference-clause* :: ′*v state* ⇒ ′*v clause* × (′*v clause* × ′*v clause*) *set* ⇒ *bool*
(**infix** $\Rightarrow_{\mathrm{Res}}$ *100*) **where**
resolution:
{#*Pos p*#} + *C* ∈ *N* ⟹ {#*Neg p*#} + *D* ∈ *N* ⟹ ({#*Pos p*#} + *C*, {#*Neg p*#} + *D*) ∉ *already-used*
⟹ *inference-clause* (*N*, *already-used*) (*C* + *D*, *already-used* ∪ {({#*Pos p*#} + *C*, {#*Neg p*#} + *D*)}) |
factoring: {#*L*, *L*#} + *C* ∈ *N* ⟹ *inference-clause* (*N*, *already-used*) (*C* + {#*L*#}, *already-used*)

inductive *inference* :: ′*v state* ⇒ ′*v state* ⇒ *bool* **where**
inference-step: *inference-clause S* (*clause*, *already-used*)
⟹ *inference S* (*fst S* ∪ {*clause*}, *already-used*)

abbreviation *already-used-inv*
:: ′*a literal multiset set* × (′*a literal multiset* × ′*a literal multiset*) *set* ⇒ *bool* **where**
already-used-inv state ≡
(∀ (*A*, *B*) ∈ *snd state*. ∃ *p*. *Pos p* ∈# *A* ∧ *Neg p* ∈# *B* ∧
((∃ χ ∈ *fst state*. *subsumes* χ ((*A* − {#*Pos p*#}) + (*B* − {#*Neg p*#})))
∨ *tautology* ((*A* − {#*Pos p*#}) + (*B* − {#*Neg p*#}))))

lemma *inference-clause-preserves-already-used-inv*:
assumes *inference-clause S S*′
and *already-used-inv S*
shows *already-used-inv* (*fst S* ∪ {*fst S*′}, *snd S*′)
⟨*proof*⟩

lemma *inference-preserves-already-used-inv*:
assumes *inference S S*′
and *already-used-inv S*
shows *already-used-inv S*′
⟨*proof*⟩

lemma *rtranclp-inference-preserves-already-used-inv*:
assumes *rtranclp inference S S*′
and *already-used-inv S*

shows *already-used-inv S′*
⟨*proof*⟩

lemma *subsumes-condensation*:
assumes *subsumes* (*C* + {#*L*#} + {#*L*#}) *D*
shows *subsumes* (*C* + {#*L*#}) *D*
⟨*proof*⟩

lemma *simplify-preserves-already-used-inv*:
assumes *simplify N N′*
and *already-used-inv* (*N*, *already-used*)
shows *already-used-inv* (*N′*, *already-used*)
⟨*proof*⟩

lemma
factoring-satisfiable: *I* ⊨ *add-mset L* (*add-mset L C*) ⟷ *I* ⊨ *add-mset L C* **and**
resolution-satisfiable:
consistent-interp I ⟹ *I* ⊨ *add-mset* (*Pos p*) *C* ⟹ *I* ⊨ *add-mset* (*Neg p*) *D* ⟹ *I* ⊨ *C* + *D* **and**
factoring-same-vars: *atms-of* (*add-mset L* (*add-mset L C*)) = *atms-of* (*add-mset L C*)
⟨*proof*⟩

lemma *inference-increasing*:
assumes *inference S S′* **and** *ψ* ∈ *fst S*
shows *ψ* ∈ *fst S′*
⟨*proof*⟩

lemma *rtranclp-inference-increasing*:
assumes *rtranclp inference S S′* **and** *ψ* ∈ *fst S*
shows *ψ* ∈ *fst S′*
⟨*proof*⟩

lemma *inference-clause-already-used-increasing*:
assumes *inference-clause S S′*
shows *snd S* ⊆ *snd S′*
⟨*proof*⟩

lemma *inference-already-used-increasing*:
assumes *inference S S′*
shows *snd S* ⊆ *snd S′*
⟨*proof*⟩

lemma *inference-clause-preserve-models*:
fixes *N N′* :: *′v clause-set*
assumes *inference-clause T T′*
and *total-over-m I* (*fst T*)
and *consistent*: *consistent-interp I*
shows *I* ⊨*s fst T* ⟷ *I* ⊨*s fst T* ∪ {*fst T′*}
⟨*proof*⟩

lemma *inference-preserve-models*:
fixes *N N′* :: *′v clause-set*
assumes *inference T T′*
and *total-over-m I* (*fst T*)

and *consistent*: *consistent-interp I*
shows *I* ⊨*s fst T* ⟷ *I* ⊨*s fst T′*
⟨*proof*⟩

lemma *inference-clause-preserves-atms-of-ms*:
assumes *inference-clause S S′*
shows *atms-of-ms* (*fst* (*fst S* ∪ {*fst S′*}, *snd S′*)) ⊆ *atms-of-ms* (*fst S*)
⟨*proof*⟩

lemma *inference-preserves-atms-of-ms*:
fixes *N N′* :: ′*v clause-set*
assumes *inference T T′*
shows *atms-of-ms* (*fst T′*) ⊆ *atms-of-ms* (*fst T*)
⟨*proof*⟩

lemma *inference-preserves-total*:
fixes *N N′* :: ′*v clause-set*
assumes *inference* (*N*, *already-used*) (*N′*, *already-used′*)
shows *total-over-m I N* ⟹ *total-over-m I N′*
⟨*proof*⟩

lemma *rtranclp-inference-preserves-total*:
assumes *rtranclp inference T T′*
shows *total-over-m I* (*fst T*) ⟹ *total-over-m I* (*fst T′*)
⟨*proof*⟩

lemma *rtranclp-inference-preserve-models*:
assumes *rtranclp inference N N′*
and *total-over-m I* (*fst N*)
and *consistent*: *consistent-interp I*
shows *I* ⊨*s fst N* ⟷ *I* ⊨*s fst N′*
⟨*proof*⟩

lemma *inference-preserves-finite*:
assumes *inference ψ ψ′* **and** *finite* (*fst ψ*)
shows *finite* (*fst ψ′*)
⟨*proof*⟩

lemma *inference-clause-preserves-finite-snd*:
assumes *inference-clause ψ ψ′* **and** *finite* (*snd ψ*)
shows *finite* (*snd ψ′*)
⟨*proof*⟩

lemma *inference-preserves-finite-snd*:
assumes *inference ψ ψ′* **and** *finite* (*snd ψ*)
shows *finite* (*snd ψ′*)
⟨*proof*⟩

lemma *rtranclp-inference-preserves-finite*:
assumes *rtranclp inference ψ ψ′* **and** *finite* (*fst ψ*)
shows *finite* (*fst ψ′*)
⟨*proof*⟩

lemma *consistent-interp-insert*:
  assumes *consistent-interp I*
  and $\mathit{atm\text{-}of}\ P \notin \mathit{atm\text{-}of}\ `\ I$
  shows *consistent-interp* (*insert P I*)
⟨*proof*⟩

lemma *simplify-clause-preserves-sat*:
  assumes *simp*: $\mathit{simplify}\ \psi\ \psi'$
  and $\mathit{satisfiable}\ \psi'$
  shows $\mathit{satisfiable}\ \psi$
  ⟨*proof*⟩

lemma *simplify-preserves-unsat*:
  assumes $\mathit{inference}\ \psi\ \psi'$
  shows $\mathit{satisfiable}\ (\mathit{fst}\ \psi') \longrightarrow \mathit{satisfiable}\ (\mathit{fst}\ \psi)$
  ⟨*proof*⟩

lemma *inference-preserves-unsat*:
  assumes $\mathit{inference}^{**}\ S\ S'$
  shows $\mathit{satisfiable}\ (\mathit{fst}\ S') \longrightarrow\ \mathit{satisfiable}\ (\mathit{fst}\ S)$
  ⟨*proof*⟩

datatype $'v\ \mathit{sem\text{-}tree} = \mathit{Node}\ 'v\ 'v\ \mathit{sem\text{-}tree}\ 'v\ \mathit{sem\text{-}tree} \mid \mathit{Leaf}$

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

lemma *sem-tree-size*[*case-names bigger*]:
  $(\bigwedge xs{::}\ 'v\ \mathit{sem\text{-}tree}.\ (\bigwedge ys{::}\ 'v\ \mathit{sem\text{-}tree}.\ \mathit{sem\text{-}tree\text{-}size}\ ys < \mathit{sem\text{-}tree\text{-}size}\ xs \Longrightarrow P\ ys) \Longrightarrow P\ xs)$
  $\Longrightarrow P\ xs$
  ⟨*proof*⟩

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

lemma *simplify-preserve-partial-leaf*:
  $\mathit{simplify}\ N\ N' \Longrightarrow \mathit{partial\text{-}interps}\ \mathit{Leaf}\ I\ N \Longrightarrow \mathit{partial\text{-}interps}\ \mathit{Leaf}\ I\ N'$
  ⟨*proof*⟩

lemma *simplify-preserve-partial-tree*:
  assumes $\mathit{simplify}\ N\ N'$
  and *partial-interps t I N*
  shows $\mathit{partial\text{-}interps}\ t\ I\ N'$
  ⟨*proof*⟩

lemma *inference-preserve-partial-tree*:
  assumes $\mathit{inference}\ S\ S'$
  and *partial-interps t I* (*fst S*)
  shows $\mathit{partial\text{-}interps}\ t\ I\ (\mathit{fst}\ S')$

⟨*proof*⟩

lemma *rtranclp-inference-preserve-partial-tree*:
assumes *rtranclp inference N N′*
and *partial-interps t I* (*fst N*)
shows *partial-interps t I* (*fst N′*)
⟨*proof*⟩

function *build-sem-tree* :: $'v$ :: *linorder set* ⇒ $'v$ *clause-set* ⇒ $'v$ *sem-tree* **where**
build-sem-tree atms ψ =
(*if atms* = {} ∨ ¬ *finite atms*
then Leaf
else Node (*Min atms*) (*build-sem-tree* (*Set.remove* (*Min atms*) *atms*) ψ)
(*build-sem-tree* (*Set.remove* (*Min atms*) *atms*) ψ))
⟨*proof*⟩
termination
⟨*proof*⟩
declare *build-sem-tree.induct*[*case-names tree*]

lemma *unsatisfiable-empty*[*simp*]:
¬*unsatisfiable* {}
⟨*proof*⟩

lemma *partial-interps-build-sem-tree-atms-general*:
fixes ψ :: $'v$ :: *linorder clause-set* **and** *p* :: $'v$ *literal list*
assumes *unsat*: *unsatisfiable* ψ **and** *finite* ψ **and** *consistent-interp I*
and *finite atms*
and *atms-of-ms* ψ = *atms* ∪ *atms-of-s I* **and** *atms* ∩ *atms-of-s I* = {}
shows *partial-interps* (*build-sem-tree atms* ψ) *I* ψ
⟨*proof*⟩

lemma *partial-interps-build-sem-tree-atms*:
fixes ψ :: $'v$ :: *linorder clause-set* **and** *p* :: $'v$ *literal list*
assumes *unsat*: *unsatisfiable* ψ **and** *finite*: *finite* ψ
shows *partial-interps* (*build-sem-tree* (*atms-of-ms* ψ) ψ) {} ψ
⟨*proof*⟩

lemma *can-decrease-count*:
fixes ψ'' :: $'v$ *clause-set* × ($'v$ *clause* × $'v$ *clause* × $'v$) *set*
assumes *count* χ *L* = *n*
and *L* ∈# χ **and** χ ∈ *fst* ψ
shows $\exists\,\psi'\ \chi'.$ *inference*** $\psi\ \psi'$ ∧ χ' ∈ *fst* ψ' ∧ (∀ *L*. *L* ∈# χ ⟷ *L* ∈# χ')
∧ *count* χ' *L* = *1*
∧ (∀ φ. φ ∈ *fst* ψ ⟶ φ ∈ *fst* ψ')
∧ (*I* ⊨ χ ⟷ *I* ⊨ χ')
∧ (∀ *I′*. *total-over-m I′* {χ} ⟶ *total-over-m I′* {χ'})
⟨*proof*⟩

lemma *can-decrease-tree-size*:
fixes ψ :: $'v$ *state* **and** *tree* :: $'v$ *sem-tree*
assumes *finite* (*fst* ψ) **and** *already-used-inv* ψ
and *partial-interps tree I* (*fst* ψ)
shows ∃(*tree′*:: $'v$ *sem-tree*) ψ'. *inference*** $\psi\ \psi'$ ∧ *partial-interps tree′ I* (*fst* ψ')

$\wedge$ (*sem-tree-size tree′* $<$ *sem-tree-size tree* $\vee$ *sem-tree-size tree* $=$ *0*)
⟨*proof*⟩

lemma *inference-completeness-inv*:
fixes ψ :: *′v ::linorder state*
assumes
unsat: $\neg$*satisfiable* (*fst* ψ) **and**
finite: *finite* (*fst* ψ) **and**
a-u-v: *already-used-inv* ψ
shows $\exists\,\psi'$. (*inference*** ψ ψ' $\wedge$ {#} $\in$ *fst* ψ')
⟨*proof*⟩

lemma *inference-completeness*:
fixes ψ :: *′v ::linorder state*
assumes *unsat*: $\neg$*satisfiable* (*fst* ψ)
and *finite*: *finite* (*fst* ψ)
and *snd* ψ = {}
shows $\exists\,\psi'$. (*rtranclp inference* ψ ψ' $\wedge$ {#} $\in$ *fst* ψ')
⟨*proof*⟩

lemma *inference-soundness*:
fixes ψ :: *′v ::linorder state*
assumes *rtranclp inference* ψ ψ' **and** {#} $\in$ *fst* ψ'
shows *unsatisfiable* (*fst* ψ)
⟨*proof*⟩

lemma *inference-soundness-and-completeness*:
fixes ψ :: *′v ::linorder state*
assumes *finite*: *finite* (*fst* ψ)
and *snd* ψ = {}
shows ($\exists\,\psi'$. (*inference*** ψ ψ' $\wedge$ {#} $\in$ *fst* ψ')) $\longleftrightarrow$ *unsatisfiable* (*fst* ψ)
⟨*proof*⟩

2.1.4 Lemma about the Simplified State

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

lemma *simplified-count*:
assumes *simp*: *simplified* ψ **and** χ: $\chi \in \psi$
shows *count* χ *L* $\leq$ *1*
⟨*proof*⟩

lemma *simplified-no-both*:
assumes *simp*: *simplified* ψ **and** χ: $\chi \in \psi$
shows $\neg$ ($L \in\# \chi \wedge -L \in\# \chi$)
⟨*proof*⟩

lemma *add-mset-Neg-Pos-commute*[*simp*]:
add-mset (*Neg P*) (*add-mset* (*Pos P*) *C*) = *add-mset* (*Pos P*) (*add-mset* (*Neg P*) *C*)
⟨*proof*⟩

lemma *simplified-not-tautology*:
assumes *simplified* {ψ}
shows ~*tautology* ψ
⟨*proof*⟩

lemma *simplified-remove*:
  assumes *simplified* $\{\psi\}$
  shows *simplified* $\{\psi - \{\#l\#\}\}$
⟨*proof*⟩

lemma *in-simplified-simplified*:
  assumes *simp*: *simplified* ψ **and** *incl*: $\psi' \subseteq \psi$
  shows *simplified* ψ'
⟨*proof*⟩

lemma *simplified-in*:
  assumes *simplified* ψ
  and $N \in \psi$
  shows *simplified* $\{N\}$
  ⟨*proof*⟩

lemma *subsumes-imp-formula*:
  assumes $\psi \leq\# \varphi$
  shows $\{\psi\} \models p\ \varphi$
  ⟨*proof*⟩

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

lemma *simplified-no-more-full1-simplified*:
  assumes *simplified* ψ
  shows $\neg$*full1 simplify* $\psi\ \psi'$
  ⟨*proof*⟩

2.1.5 Resolution and Invariants

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

Invariants

lemma *resolution-finite*:
  assumes *resolution* $\psi\ \psi'$ **and** *finite* (*fst* ψ)
  shows *finite* (*fst* ψ')
  ⟨*proof*⟩

lemma *rtranclp-resolution-finite*:
  assumes $resolution^{**}\ \psi\ \psi'$ **and** *finite* (*fst* ψ)
  shows *finite* (*fst* ψ')
  ⟨*proof*⟩

lemma *resolution-finite-snd*:
  assumes *resolution* $\psi\ \psi'$ **and** *finite* (*snd* ψ)
  shows *finite* (*snd* ψ')
  ⟨*proof*⟩

lemma *rtranclp-resolution-finite-snd*:
assumes *resolution*** ψ ψ' **and** *finite* (*snd* ψ)
shows *finite* (*snd* ψ')
⟨*proof*⟩

lemma *resolution-always-simplified*:
assumes *resolution* ψ ψ'
shows *simplified* (*fst* ψ')
⟨*proof*⟩

lemma *tranclp-resolution-always-simplified*:
assumes *tranclp resolution* ψ ψ'
shows *simplified* (*fst* ψ')
⟨*proof*⟩

lemma *resolution-atms-of*:
assumes *resolution* ψ ψ' **and** *finite* (*fst* ψ)
shows *atms-of-ms* (*fst* ψ') $\subseteq$ *atms-of-ms* (*fst* ψ)
⟨*proof*⟩

lemma *rtranclp-resolution-atms-of*:
assumes *resolution*** ψ ψ' **and** *finite* (*fst* ψ)
shows *atms-of-ms* (*fst* ψ') $\subseteq$ *atms-of-ms* (*fst* ψ)
⟨*proof*⟩

lemma *resolution-include*:
assumes *res*: *resolution* ψ ψ' **and** *finite*: *finite* (*fst* ψ)
shows *fst* ψ' $\subseteq$ *simple-clss* (*atms-of-ms* (*fst* ψ))
⟨*proof*⟩

lemma *rtranclp-resolution-include*:
assumes *res*: *tranclp resolution* ψ ψ' **and** *finite*: *finite* (*fst* ψ)
shows *fst* ψ' $\subseteq$ *simple-clss* (*atms-of-ms* (*fst* ψ))
⟨*proof*⟩

abbreviation *already-used-all-simple*
:: (*'a literal multiset* × *'a literal multiset*) *set* ⇒ *'a set* ⇒ *bool* **where**
already-used-all-simple already-used vars ≡
(∀ (*A*, *B*) ∈ *already-used*. *simplified* {*A*} ∧ *simplified* {*B*} ∧ *atms-of A* ⊆ *vars* ∧ *atms-of B* ⊆ *vars*)

lemma *already-used-all-simple-vars-incl*:
assumes *vars* ⊆ *vars'*
shows *already-used-all-simple a vars* ⟹ *already-used-all-simple a vars'*
⟨*proof*⟩

lemma *inference-clause-preserves-already-used-all-simple*:
assumes *inference-clause S S'*
and *already-used-all-simple* (*snd S*) *vars*
and *simplified* (*fst S*)
and *atms-of-ms* (*fst S*) ⊆ *vars*
shows *already-used-all-simple* (*snd* (*fst S* ∪ {*fst S'*}, *snd S'*)) *vars*
⟨*proof*⟩

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

and *simplified* (*fst S*)
and *atms-of-ms* (*fst S*) $\subseteq$ *vars*
shows *already-used-all-simple* (*snd* S') *vars*
⟨*proof*⟩

lemma *already-used-all-simple-inv*:
assumes *resolution S* S'
and *already-used-all-simple* (*snd S*) *vars*
and *atms-of-ms* (*fst S*) $\subseteq$ *vars*
shows *already-used-all-simple* (*snd* S') *vars*
⟨*proof*⟩

lemma *rtranclp-already-used-all-simple-inv*:
assumes *resolution*** *S* S'
and *already-used-all-simple* (*snd S*) *vars*
and *atms-of-ms* (*fst S*) $\subseteq$ *vars*
and *finite* (*fst S*)
shows *already-used-all-simple* (*snd* S') *vars*
⟨*proof*⟩

lemma *inference-clause-simplified-already-used-subset*:
assumes *inference-clause S* S'
and *simplified* (*fst S*)
shows *snd S* $\subset$ *snd* S'
⟨*proof*⟩

lemma *inference-simplified-already-used-subset*:
assumes *inference S* S'
and *simplified* (*fst S*)
shows *snd S* $\subset$ *snd* S'
⟨*proof*⟩

lemma *resolution-simplified-already-used-subset*:
assumes *resolution S* S'
and *simplified* (*fst S*)
shows *snd S* $\subset$ *snd* S'
⟨*proof*⟩

lemma *tranclp-resolution-simplified-already-used-subset*:
assumes *tranclp resolution S* S'
and *simplified* (*fst S*)
shows *snd S* $\subset$ *snd* S'
⟨*proof*⟩

abbreviation *already-used-top vars* $\equiv$ *simple-clss vars* × *simple-clss vars*

lemma *already-used-all-simple-in-already-used-top*:
assumes *already-used-all-simple s vars* **and** *finite vars*
shows *s* $\subseteq$ *already-used-top vars*
⟨*proof*⟩

lemma *already-used-top-finite*:
assumes *finite vars*
shows *finite* (*already-used-top vars*)
⟨*proof*⟩

lemma *already-used-top-increasing*:
assumes *var* $\subseteq$ *var′* **and** *finite var′*
shows *already-used-top var* $\subseteq$ *already-used-top var′*
⟨*proof*⟩

lemma *already-used-all-simple-finite*:
fixes *s* :: (*′a literal multiset* $\times$ *′a literal multiset*) *set* **and** *vars* :: *′a set*
assumes *already-used-all-simple s vars* **and** *finite vars*
shows *finite s*
⟨*proof*⟩

abbreviation *card-simple vars* $\psi \equiv$ *card* (*already-used-top vars* $-$ ψ)

lemma *resolution-card-simple-decreasing*:
assumes *res*: *resolution* ψ ψ'
and *a-u-s*: *already-used-all-simple* (*snd* ψ) *vars*
and *finite-v*: *finite vars*
and *finite-fst*: *finite* (*fst* ψ)
and *finite-snd*: *finite* (*snd* ψ)
and *simp*: *simplified* (*fst* ψ)
and *atms-of-ms* (*fst* ψ) $\subseteq$ *vars*
shows *card-simple vars* (*snd* ψ') $<$ *card-simple vars* (*snd* ψ)
⟨*proof*⟩

lemma *tranclp-resolution-card-simple-decreasing*:
assumes *tranclp resolution* ψ ψ' **and** *finite-fst*: *finite* (*fst* ψ)
and *already-used-all-simple* (*snd* ψ) *vars*
and *atms-of-ms* (*fst* ψ) $\subseteq$ *vars*
and *finite-v*: *finite vars*
and *finite-snd*: *finite* (*snd* ψ)
and *simplified* (*fst* ψ)
shows *card-simple vars* (*snd* ψ') $<$ *card-simple vars* (*snd* ψ)
⟨*proof*⟩

lemma *tranclp-resolution-card-simple-decreasing-2*:
assumes *tranclp resolution* ψ ψ'
and *finite-fst*: *finite* (*fst* ψ)
and *empty-snd*: *snd* $\psi = \{\}$
and *simplified* (*fst* ψ)
shows *card-simple* (*atms-of-ms* (*fst* ψ)) (*snd* ψ') $<$ *card-simple* (*atms-of-ms* (*fst* ψ)) (*snd* ψ)
⟨*proof*⟩

Well-Foundness of the Relation

lemma *wf-simplified-resolution*:
assumes *f-vars*: *finite vars*
shows *wf* {(*y*:: *′v*:: *linorder state*, *x*). (*atms-of-ms* (*fst x*) $\subseteq$ *vars* $\wedge$ *simplified* (*fst x*)
$\wedge$ *finite* (*snd x*) $\wedge$ *finite* (*fst x*) $\wedge$ *already-used-all-simple* (*snd x*) *vars*) $\wedge$ *resolution x y*}
⟨*proof*⟩

lemma *wf-simplified-resolution′*:
assumes *f-vars*: *finite vars*
shows *wf* {(*y*:: *′v*:: *linorder state*, *x*). (*atms-of-ms* (*fst x*) $\subseteq$ *vars* $\wedge$ $\neg$*simplified* (*fst x*)
$\wedge$ *finite* (*snd x*) $\wedge$ *finite* (*fst x*) $\wedge$ *already-used-all-simple* (*snd x*) *vars*) $\wedge$ *resolution x y*}

⟨*proof*⟩

lemma *wf-resolution*:
assumes *f-vars*: *finite vars*
shows *wf* ({(*y*:: ′*v*:: *linorder state*, *x*). (*atms-of-ms* (*fst x*) ⊆ *vars* ∧ *simplified* (*fst x*)
∧ *finite* (*snd x*) ∧ *finite* (*fst x*) ∧ *already-used-all-simple* (*snd x*) *vars*) ∧ *resolution x y*}
∪ {(*y*, *x*). (*atms-of-ms* (*fst x*) ⊆ *vars* ∧ ¬ *simplified* (*fst x*) ∧ *finite* (*snd x*) ∧ *finite* (*fst x*)
∧ *already-used-all-simple* (*snd x*) *vars*) ∧ *resolution x y*}) (**is** *wf* (*?R* ∪ *?S*))
⟨*proof*⟩

lemma *rtrancp-simplify-already-used-inv*:
assumes *simplify*** *S S*′
and *already-used-inv* (*S*, *N*)
shows *already-used-inv* (*S*′, *N*)
⟨*proof*⟩

lemma *full1-simplify-already-used-inv*:
assumes *full1 simplify S S*′
and *already-used-inv* (*S*, *N*)
shows *already-used-inv* (*S*′, *N*)
⟨*proof*⟩

lemma *full-simplify-already-used-inv*:
assumes *full simplify S S*′
and *already-used-inv* (*S*, *N*)
shows *already-used-inv* (*S*′, *N*)
⟨*proof*⟩
lemma *resolution-already-used-inv*:
assumes *resolution S S*′
and *already-used-inv S*
shows *already-used-inv S*′
⟨*proof*⟩

lemma *rtranclp-resolution-already-used-inv*:
assumes *resolution*** *S S*′
and *already-used-inv S*
shows *already-used-inv S*′
⟨*proof*⟩

lemma *rtanclp-simplify-preserves-unsat*:
assumes *simplify*** ψ ψ'
shows *satisfiable* ψ' ⟶ *satisfiable* ψ
⟨*proof*⟩

lemma *full1-simplify-preserves-unsat*:
assumes *full1 simplify* ψ ψ'
shows *satisfiable* ψ' ⟶ *satisfiable* ψ
⟨*proof*⟩

lemma *full-simplify-preserves-unsat*:
assumes *full simplify* ψ ψ'
shows *satisfiable* ψ' ⟶ *satisfiable* ψ
⟨*proof*⟩

lemma *resolution-preserves-unsat*:
assumes *resolution* ψ ψ'

shows *satisfiable* (*fst* ψ') $\longrightarrow$ *satisfiable* (*fst* ψ)
⟨*proof*⟩

lemma *rtranclp-resolution-preserves-unsat*:
assumes *resolution*** ψ ψ'
shows *satisfiable* (*fst* ψ') $\longrightarrow$ *satisfiable* (*fst* ψ)
⟨*proof*⟩

lemma *rtranclp-simplify-preserve-partial-tree*:
assumes *simplify*** N N'
and *partial-interps t I N*
shows *partial-interps t I* N'
⟨*proof*⟩

lemma *full1-simplify-preserve-partial-tree*:
assumes *full1 simplify* N N'
and *partial-interps t I N*
shows *partial-interps t I* N'
⟨*proof*⟩

lemma *full-simplify-preserve-partial-tree*:
assumes *full simplify* N N'
and *partial-interps t I N*
shows *partial-interps t I* N'
⟨*proof*⟩

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

lemma *rtranclp-resolution-preserve-partial-tree*:
assumes *resolution*** S S'
and *partial-interps t I* (*fst S*)
shows *partial-interps t I* (*fst* S')
⟨*proof*⟩
thm *nat-less-induct nat.induct*

lemma *nat-ge-induct*[*case-names 0 Suc*]:
assumes *P 0*
and $\bigwedge n.$ ($\bigwedge m.$ $m<Suc\ n \Longrightarrow P\ m$) $\Longrightarrow$ P ($Suc\ n$)
shows *P n*
⟨*proof*⟩

lemma *wf-always-more-step-False*:
assumes *wf R*
shows ($\forall x.\ \exists z.\ (z, x) \in R$) $\Longrightarrow$ *False*
⟨*proof*⟩

lemma *finite-finite-mset-element-of-mset*[*simp*]:
assumes *finite N*
shows *finite* $\{f\ \varphi\ L\ |\varphi\ L.\ \varphi \in N \land L \in\#\ \varphi \land P\ \varphi\ L\}$
⟨*proof*⟩

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

interpretation *sum-count-ge-2*:
$\quad\textit{folding}\ \lambda\varphi.\ (+)(\textit{sum-mset}\ \{\#\textit{count}\ \varphi\ L\ |L \in\#\ \varphi.\ 2 \le \textit{count}\ \varphi\ L\#\})\ 0$
rewrites
$\quad\textit{folding.F}\ (\lambda\varphi.\ (+)(\textit{sum-mset}\ \{\#\textit{count}\ \varphi\ L\ |L \in\#\ \varphi.\ 2 \le \textit{count}\ \varphi\ L\#\}))\ 0 = \textit{sum-count-ge-2}$
⟨*proof*⟩

lemma *finite-incl-le-setsum*:
$\textit{finite}\ (B{::}'a\ \textit{multiset set}) \Longrightarrow A \subseteq B \Longrightarrow \Xi\ A \le \Xi\ B$
⟨*proof*⟩

lemma *simplify-finite-measure-decrease*:
$\quad\textit{simplify}\ N\ N' \Longrightarrow \textit{finite}\ N \Longrightarrow \textit{card}\ N' + \Xi\ N' < \textit{card}\ N + \Xi\ N$
⟨*proof*⟩

lemma *simplify-terminates*:
$\quad\textit{wf}\ \{(N', N).\ \textit{finite}\ N \wedge \textit{simplify}\ N\ N'\}$
  ⟨*proof*⟩

lemma *wf-terminates*:
  assumes $\textit{wf}\ r$
  shows $\exists N'.(N', N) \in r^* \wedge (\forall N''.\ (N'', N') \notin r)$
⟨*proof*⟩

lemma *rtranclp-simplify-terminates*:
  assumes *fin*: $\textit{finite}\ N$
  shows $\exists N'.\ \textit{simplify}^{**}\ N\ N' \wedge \textit{simplified}\ N'$
⟨*proof*⟩

lemma *finite-simplified-full1-simp*:
  assumes $\textit{finite}\ N$
  shows $\textit{simplified}\ N \vee (\exists N'.\ \textit{full1 simplify}\ N\ N')$
  ⟨*proof*⟩

lemma *finite-simplified-full-simp*:
  assumes $\textit{finite}\ N$
  shows $\exists N'.\ \textit{full simplify}\ N\ N'$
  ⟨*proof*⟩

lemma *can-decrease-tree-size-resolution*:
  fixes $\psi :: {}'v\ \textit{state}$ **and** $\textit{tree} :: {}'v\ \textit{sem-tree}$
  assumes $\textit{finite}\ (\textit{fst}\ \psi)$ **and** $\textit{already-used-inv}\ \psi$
  and $\textit{partial-interps tree}\ I\ (\textit{fst}\ \psi)$
  and $\textit{simplified}\ (\textit{fst}\ \psi)$
  shows $\exists (\textit{tree}'{::}\ 'v\ \textit{sem-tree})\ \psi'.\ \textit{resolution}^{**}\ \psi\ \psi' \wedge \textit{partial-interps tree}'\ I\ (\textit{fst}\ \psi')$
    $\wedge\ (\textit{sem-tree-size tree}' < \textit{sem-tree-size tree} \vee \textit{sem-tree-size tree} = 0)$
  ⟨*proof*⟩

lemma *resolution-completeness-inv*:
  fixes $\psi :: {}'v\ {::}\textit{linorder state}$
  assumes
    unsat: $\neg\textit{satisfiable}\ (\textit{fst}\ \psi)$ **and**

finite: *finite* (*fst* ψ) **and**
a-u-v: *already-used-inv* ψ
shows $\exists\,\psi'$. (*resolution*** ψ ψ' $\wedge$ $\{\#\} \in$ *fst* ψ')
⟨*proof*⟩

lemma *resolution-preserves-already-used-inv*:
assumes *resolution* S S'
and *already-used-inv* S
shows *already-used-inv* S'
⟨*proof*⟩

lemma *rtranclp-resolution-preserves-already-used-inv*:
assumes *resolution*** S S'
and *already-used-inv* S
shows *already-used-inv* S'
⟨*proof*⟩

lemma *resolution-completeness*:
fixes ψ :: $'v$ ::*linorder state*
assumes *unsat*: $\neg$*satisfiable* (*fst* ψ)
and *finite*: *finite* (*fst* ψ)
and *snd* $\psi = \{\}$
shows $\exists\,\psi'$. (*resolution*** ψ ψ' $\wedge$ $\{\#\} \in$ *fst* ψ')
⟨*proof*⟩

lemma *rtranclp-preserves-sat*:
assumes *simplify*** S S'
and *satisfiable* S
shows *satisfiable* S'
⟨*proof*⟩

lemma *resolution-preserves-sat*:
assumes *resolution* S S'
and *satisfiable* (*fst* S)
shows *satisfiable* (*fst* S')
⟨*proof*⟩

lemma *rtranclp-resolution-preserves-sat*:
assumes *resolution*** S S'
and *satisfiable* (*fst* S)
shows *satisfiable* (*fst* S')
⟨*proof*⟩

lemma *resolution-soundness*:
fixes ψ :: $'v$ ::*linorder state*
assumes *resolution*** ψ ψ' **and** $\{\#\} \in$ *fst* ψ'
shows *unsatisfiable* (*fst* ψ)
⟨*proof*⟩

lemma *resolution-soundness-and-completeness*:
fixes ψ :: $'v$ ::*linorder state*
assumes *finite*: *finite* (*fst* ψ)
and *snd*: *snd* $\psi = \{\}$
shows ($\exists\,\psi'$. (*resolution*** ψ ψ' $\wedge$ $\{\#\} \in$ *fst* ψ')) $\longleftrightarrow$ *unsatisfiable* (*fst* ψ)
⟨*proof*⟩

lemma *simplified-falsity*:
  assumes *simp*: *simplified* ψ
  and $\{\#\} \in \psi$
  shows $\psi = \{\{\#\}\}$
⟨*proof*⟩

lemma *simplify-falsity-in-preserved*:
  assumes *simplify* χs $\chi s'$
  and $\{\#\} \in \chi s$
  shows $\{\#\} \in \chi s'$
  ⟨*proof*⟩

lemma *rtranclp-simplify-falsity-in-preserved*:
  assumes *simplify*** χs $\chi s'$
  and $\{\#\} \in \chi s$
  shows $\{\#\} \in \chi s'$
  ⟨*proof*⟩

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

theorem *resolution-soundness-and-completeness'*:
  fixes ψ :: *'v* ::*linorder state*
  assumes
    finite: *finite* (*fst* ψ)**and**
    snd: *snd* $\psi = \{\}$
  shows ($\exists$ *a-u-v*. (*resolution*** ψ ($\{\{\#\}\}$, *a-u-v*))) $\longleftrightarrow$ *unsatisfiable* (*fst* ψ)
  ⟨*proof*⟩

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

2.2 Superposition

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

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

lemma *herbrand-interp-iff-partial-interp-cls*:
  $S \models h\ C \longleftrightarrow \{Pos\ P|P.\ P{\in}S\} \cup \{Neg\ P|P.\ P{\notin}S\} \models C$
  ⟨*proof*⟩

lemma *herbrand-consistent-interp*:
  consistent-interp $(\{Pos\ P|P.\ P{\in}S\} \cup \{Neg\ P|P.\ P{\notin}S\})$
  ⟨*proof*⟩

lemma *herbrand-total-over-set*:

$\mathit{total\text{-}over\text{-}set}\ (\{\mathit{Pos}\ P|P.\ P{\in}S\} \cup \{\mathit{Neg}\ P|P.\ P{\notin}S\})\ T$
$\langle \mathit{proof} \rangle$

lemma *herbrand-total-over-m*:
$\mathit{total\text{-}over\text{-}m}\ (\{\mathit{Pos}\ P|P.\ P{\in}S\} \cup \{\mathit{Neg}\ P|P.\ P{\notin}S\})\ T$
$\langle \mathit{proof} \rangle$

lemma *herbrand-interp-iff-partial-interp-clss*:
$S \models\!\mathit{hs}\ C \longleftrightarrow \{\mathit{Pos}\ P|P.\ P{\in}S\} \cup \{\mathit{Neg}\ P|P.\ P{\notin}S\} \models\!s\ C$
$\langle \mathit{proof} \rangle$

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

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

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

locale *ground-resolution-with-selection* =
selection S **for** $S :: ({'a} :: \mathit{wellorder})\ \mathit{clause} \Rightarrow {'a}\ \mathit{clause}$
begin

context
fixes $N :: {'a}\ \mathit{clause}\ \mathit{set}$
begin

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

function
$\mathit{production} :: {'a}\ \mathit{clause} \Rightarrow {'a}\ \mathit{interp}$
where
$\mathit{production}\ C =$
$\{A.\ C \in N \land C \neq \{\#\} \land \mathit{Max\text{-}mset}\ C = \mathit{Pos}\ A \land \mathit{count}\ C\ (\mathit{Pos}\ A) \leq \mathit{1}$
$\land \neg\ (\bigcup D \in \{D.\ D < C\}.\ \mathit{production}\ D) \models\!h\ C \land S\ C = \{\#\}\}$
$\langle \mathit{proof} \rangle$
termination $\langle \mathit{proof} \rangle$

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

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

lemma *production-unfold*:
$\mathit{production}\ C = \{A.\ C \in N \land C \neq \{\#\} \land \mathit{Max\text{-}mset}\ C = \mathit{Pos}\ A{\land}\ \mathit{count}\ C\ (\mathit{Pos}\ A) \leq \mathit{1} \land \neg\ \mathit{interp}\ C \models\!h\ C \land S\ C = \{\#\}\}$
$\langle \mathit{proof} \rangle$

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

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

lemma *producesD*:
$\mathit{produces}\ C\ A \Longrightarrow C \in N \wedge C \neq \{\#\} \wedge \mathit{Pos}\ A = \mathit{Max\text{-}mset}\ C \wedge \mathit{count}\ C\ (\mathit{Pos}\ A) \leq \mathit{1} \wedge$
$\neg\ \mathit{interp}\ C \models h\ C \wedge S\ C = \{\#\}$
⟨*proof*⟩

lemma $\mathit{produces}\ C\ A \Longrightarrow \mathit{Pos}\ A \in\#\ C$
⟨*proof*⟩

lemma *interp′-def-in-set*:
$\mathit{interp}\ C = (\bigcup D \in \{D \in N.\ D < C\}.\ \mathit{production}\ D)$
⟨*proof*⟩

lemma *production-iff-produces*:
$\mathit{produces}\ D\ A \longleftrightarrow A \in \mathit{production}\ D$
⟨*proof*⟩

definition *Interp* :: *′a clause* ⇒ *′a interp* **where**
$\mathit{Interp}\ C = \mathit{interp}\ C \cup \mathit{production}\ C$

lemma
assumes *produces C P*
shows $\mathit{Interp}\ C \models h\ C$
⟨*proof*⟩

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

lemma *interp-subseteq-Interp*[*simp*]: $\mathit{interp}\ C \subseteq \mathit{Interp}\ C$
⟨*proof*⟩

lemma *Interp-as-UNION*: $\mathit{Interp}\ C = (\bigcup D \in \{D.\ D \leq C\}.\ \mathit{production}\ D)$
⟨*proof*⟩

lemma *productive-not-empty*: $\mathit{productive}\ C \Longrightarrow C \neq \{\#\}$
⟨*proof*⟩

lemma *productive-imp-produces-Max-literal*: $\mathit{productive}\ C \Longrightarrow \mathit{produces}\ C\ (\mathit{atm\text{-}of}\ (\mathit{Max\text{-}mset}\ C))$
⟨*proof*⟩

lemma *productive-imp-produces-Max-atom*: $\mathit{productive}\ C \Longrightarrow \mathit{produces}\ C\ (\mathit{Max}\ (\mathit{atms\text{-}of}\ C))$
⟨*proof*⟩

lemma *produces-imp-Max-literal*: $\mathit{produces}\ C\ A \Longrightarrow A = \mathit{atm\text{-}of}\ (\mathit{Max\text{-}mset}\ C)$
⟨*proof*⟩

lemma *produces-imp-Max-atom*: $\mathit{produces}\ C\ A \Longrightarrow A = \mathit{Max}\ (\mathit{atms\text{-}of}\ C)$
⟨*proof*⟩

lemma *produces-imp-Pos-in-lits*: $\mathit{produces}\ C\ A \Longrightarrow \mathit{Pos}\ A \in\#\ C$
⟨*proof*⟩

lemma *productive-in-N*: $\mathit{productive}\ C \Longrightarrow C \in N$
⟨*proof*⟩

lemma *produces-imp-atms-leq*: $\mathit{produces}\ C\ A \Longrightarrow B \in \mathit{atms\text{-}of}\ C \Longrightarrow B \leq A$

$\langle \mathit{proof} \rangle$

lemma *produces-imp-neg-notin-lits*: $\mathit{produces}\ C\ A \Longrightarrow \mathit{Neg}\ A \notin\#\ C$
$\langle \mathit{proof} \rangle$

lemma *less-eq-imp-interp-subseteq-interp*: $C \leq D \Longrightarrow \mathit{interp}\ C \subseteq \mathit{interp}\ D$
$\langle \mathit{proof} \rangle$

lemma *less-eq-imp-interp-subseteq-Interp*: $C \leq D \Longrightarrow \mathit{interp}\ C \subseteq \mathit{Interp}\ D$
$\langle \mathit{proof} \rangle$

lemma *less-imp-production-subseteq-interp*: $C < D \Longrightarrow \mathit{production}\ C \subseteq \mathit{interp}\ D$
$\langle \mathit{proof} \rangle$

lemma *less-eq-imp-production-subseteq-Interp*: $C \leq D \Longrightarrow \mathit{production}\ C \subseteq \mathit{Interp}\ D$
$\langle \mathit{proof} \rangle$

lemma *less-imp-Interp-subseteq-interp*: $C < D \Longrightarrow \mathit{Interp}\ C \subseteq \mathit{interp}\ D$
$\langle \mathit{proof} \rangle$

lemma *less-eq-imp-Interp-subseteq-Interp*: $C \leq D \Longrightarrow \mathit{Interp}\ C \subseteq \mathit{Interp}\ D$
$\langle \mathit{proof} \rangle$

lemma *false-Interp-to-true-interp-imp-less-multiset*: $A \notin \mathit{Interp}\ C \Longrightarrow A \in \mathit{interp}\ D \Longrightarrow C < D$
$\langle \mathit{proof} \rangle$

lemma *false-interp-to-true-interp-imp-less-multiset*: $A \notin \mathit{interp}\ C \Longrightarrow A \in \mathit{interp}\ D \Longrightarrow C < D$
$\langle \mathit{proof} \rangle$

lemma *false-Interp-to-true-Interp-imp-less-multiset*: $A \notin \mathit{Interp}\ C \Longrightarrow A \in \mathit{Interp}\ D \Longrightarrow C < D$
$\langle \mathit{proof} \rangle$

lemma *false-interp-to-true-Interp-imp-le-multiset*: $A \notin \mathit{interp}\ C \Longrightarrow A \in \mathit{Interp}\ D \Longrightarrow C \leq D$
$\langle \mathit{proof} \rangle$

lemma *interp-subseteq-INTERP*: $\mathit{interp}\ C \subseteq \mathit{INTERP}$
$\langle \mathit{proof} \rangle$

lemma *production-subseteq-INTERP*: $\mathit{production}\ C \subseteq \mathit{INTERP}$
$\langle \mathit{proof} \rangle$

lemma *Interp-subseteq-INTERP*: $\mathit{Interp}\ C \subseteq \mathit{INTERP}$
$\langle \mathit{proof} \rangle$

This lemma corresponds to theorem 2.7.6 page 67 of Weidenbach's book.

lemma *produces-imp-in-interp*:
assumes *a-in-c*: $\mathit{Neg}\ A \in\#\ C$ **and** *d*: $\mathit{produces}\ D\ A$
shows $A \in \mathit{interp}\ C$
$\langle \mathit{proof} \rangle$

lemma *neg-notin-Interp-not-produce*: $\mathit{Neg}\ A \in\#\ C \Longrightarrow A \notin \mathit{Interp}\ D \Longrightarrow C \leq D \Longrightarrow \neg\ \mathit{produces}\ D''\ A$
$\langle \mathit{proof} \rangle$

lemma *in-production-imp-produces*: $A \in \mathit{production}\ C \Longrightarrow \mathit{produces}\ C\ A$
$\langle \mathit{proof} \rangle$

lemma *not-produces-imp-notin-production*: $\neg$ *produces C A* $\Longrightarrow$ *A* $\notin$ *production C*
⟨*proof*⟩

lemma *not-produces-imp-notin-interp*: $(\bigwedge D.\ \neg$ *produces D A*$)$ $\Longrightarrow$ *A* $\notin$ *interp C*
⟨*proof*⟩

The results below corresponds to Lemma 3.4.

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

lemma *true-Interp-imp-general*:
assumes
c-le-d: $C \leq D$ **and**
d-lt-d': $D < D'$ **and**
c-at-d: *Interp D* $\models h$ *C* **and**
subs: *interp D'* $\subseteq (\bigcup C \in CC.$ *production C*$)$
shows $(\bigcup C \in CC.$ *production C*$)$ $\models h$ *C*
⟨*proof*⟩

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

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

lemma *true-Interp-imp-INTERP*: $C \leq D \Longrightarrow$ *Interp D* $\models h$ *C* $\Longrightarrow$ *INTERP* $\models h$ *C*
⟨*proof*⟩

lemma *true-interp-imp-general*:
assumes
c-le-d: $C \leq D$ **and**
d-lt-d': $D < D'$ **and**
c-at-d: *interp D* $\models h$ *C* **and**
subs: *interp D'* $\subseteq (\bigcup C \in CC.$ *production C*$)$
shows $(\bigcup C \in CC.$ *production C*$)$ $\models h$ *C*
⟨*proof*⟩

This lemma corresponds to theorem 2.7.6 page 67 of Weidenbach's book. Here the strict maximality is important

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

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

lemma *true-interp-imp-INTERP*: $C \leq D \Longrightarrow$ *interp D* $\models h$ *C* $\Longrightarrow$ *INTERP* $\models h$ *C*
⟨*proof*⟩

lemma *productive-imp-false-interp*: *productive C* $\Longrightarrow \neg$ *interp C* $\models h$ *C*
⟨*proof*⟩

This lemma corresponds to theorem 2.7.6 page 67 of Weidenbach's book. Here the strict maximality is important

lemma *cls-gt-double-pos-no-production*:
assumes D: $\{\#Pos\ P,\ Pos\ P\#\} < C$
shows $\neg$*produces* $C\ P$
⟨*proof*⟩

This lemma corresponds to theorem 2.7.6 page 67 of Weidenbach's book.

lemma
assumes D: $C{+}\{\#Neg\ P\#\} < D$
shows *production* $D \neq \{P\}$
⟨*proof*⟩

lemma *in-interp-is-produced*:
assumes $P \in INTERP$
shows $\exists\, D.\ D\ {+}\{\#Pos\ P\#\} \in N \wedge$ *produces* $(D\ {+}\{\#Pos\ P\#\})\ P$
⟨*proof*⟩

end
end

2.2.1 We can now define the rules of the calculus

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

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

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

lemma *herbrand-true-clss-true-clss-cls-herbrand-true-clss*:
assumes
AB: $A \models hs\ B$ **and**
BC: $B \models p\ C$
shows $A \models h\ C$
⟨*proof*⟩

lemma *abstract-red-subset-mset-abstract-red*:
assumes
abstr: *abstract-red* $C\ N$ **and**
c-lt-d: $C \subseteq\#\ D$
shows *abstract-red* $D\ N$
⟨*proof*⟩

lemma *true-clss-cls-extended*:
assumes
$A \models p\ B$ **and**
tot: *total-over-m* $I\ A$ **and**
cons: *consistent-interp* I **and**
I-A: $I \models s\ A$
shows $I \models B$
⟨*proof*⟩

lemma
assumes
CP: ¬ *clss-lt N* ({#*C*#} + {#*E*#}) ⊨*p* {#*C*#} + {#*Neg P*#} **and**
clss-lt N ({#*C*#} + {#*E*#}) ⊨*p* {#*E*#} + {#*Pos P*#} ∨ *clss-lt N* ({#*C*#} + {#*E*#}) ⊨*p* {#*C*#} + {#*Neg P*#}
shows *clss-lt N* ({#*C*#} + {#*E*#}) ⊨*p* {#*E*#} + {#*Pos P*#}

⟨*proof*⟩

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

definition *saturated* :: *'a clause-set* ⇒ *bool* **where**
saturated N ⟷
(∀ *A B C*. *A* ∈ *N* ⟶ *B* ∈ *N* ⟶ ¬*redundant A N* ⟶ ¬*redundant B N* ⟶
superposition-rules A B C ⟶ *redundant C N* ∨ *C* ∈ *N*)
lemma (**in** −)
assumes ‹*A* ⊨*p C* + *E*›
shows ‹*A* ⊨*p add-mset L C* ∨ *A* ⊨*p add-mset* (−*L*) *E*›
⟨*proof*⟩

lemma
assumes
saturated: *saturated N* **and**
finite: *finite N* **and**
empty: {#} ∉ *N*
shows *INTERP N* ⊨*hs N*
⟨*proof*⟩

end

lemma *tautology-is-redundant*:
assumes *tautology C*
shows *abstract-red C N*
⟨*proof*⟩

lemma *subsumed-is-redundant*:
assumes *AB*: *A* ⊂# *B*
and *AN*: *A* ∈ *N*
shows *abstract-red B N*
⟨*proof*⟩

inductive *redundant* :: *'a clause* ⇒ *'a clause-set* ⇒ *bool* **where**
subsumption: *A* ∈ *N* ⟹ *A* ⊂# *B* ⟹ *redundant B N*

lemma *redundant-is-redundancy-criterion*:
fixes *A* :: *'a* :: *wellorder clause* **and** *N* :: *'a* :: *wellorder clause-set*
assumes *redundant A N*
shows *abstract-red A N*
⟨*proof*⟩

lemma *redundant-mono*:

redundant A N $\Longrightarrow$ *A* $\subseteq\#$ *B* $\Longrightarrow$ *redundant B N*
⟨*proof*⟩

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

end

end