Propositional Proof Systems

Julius Michaelis and Tobias Nipkow

March 17, 2025

Abstract

We present a formalization of Sequent Calculus, Natural Deduction, Hilbert Calculus, and Resolution using a deep embedding of propositional formulas. We provide proofs of many of the classical results, including Cut Elimination, Craig's Interpolation, proof transformation between all calculi, and soundness and completeness. Additionally, we formalize the Model Existence Theorem.

Contents

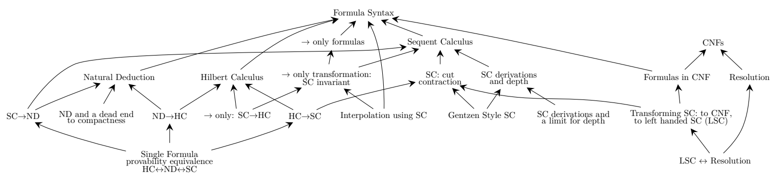

Figure 1: Overview of results considering Proof Transformation

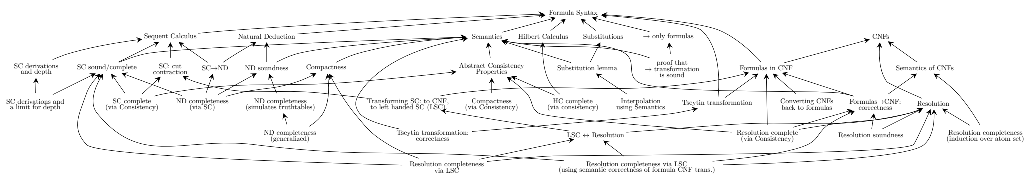

Figure 2: Overview of results considering Semantics

The files of this entry are organized as a web of results that should allow loading only that part of the formalization that the user is interested in. Special care was taken not to mix proofs that require semanics and proofs that talk about transformation between proof systems. An overview of the different theory files and their dependencies can be found in figures 1 and 2.

1 Formulas

theory *Formulas*
imports *Main HOL−Library.Countable*
begin

notation *insert* (‹- ▷ -› [*56*,*55*] *55*)

datatype (*atoms*: *'a*) *formula* =
  Atom 'a

| *Bot* (‹⊥›)
| *Not 'a formula* (‹¬›)
| *And 'a formula 'a formula* (**infix** ‹∧› *68*)
| *Or 'a formula 'a formula* (**infix** ‹∨› *68*)
| *Imp 'a formula 'a formula* (**infixr** ‹→› *68*)

lemma *atoms-finite*[*simp*,*intro*!]: *finite* (*atoms F*) **by**(*induction F*; *simp*)

primrec *subformulae* **where**
subformulae ⊥ = [⊥] |
subformulae (*Atom k*) = [*Atom k*] |
subformulae (*Not F*) = *Not F* # *subformulae F* |
subformulae (*And F G*) = *And F G* # *subformulae F* @ *subformulae G* |
subformulae (*Imp F G*) = *Imp F G* # *subformulae F* @ *subformulae G* |
subformulae (*Or F G*) = *Or F G* # *subformulae F* @ *subformulae G*

lemma *atoms-are-subformulae*: *Atom* ‘ *atoms F* ⊆ *set* (*subformulae F*)
by (*induction F*) *auto*

lemma *subsubformulae*: *G* ∈ *set* (*subformulae F*) ⟹ *H* ∈ *set* (*subformulae G*) ⟹ *H* ∈ *set* (*subformulae F*)
by(*induction F*; *force*)

lemma *subformula-atoms*: *G* ∈ *set* (*subformulae F*) ⟹ *atoms G* ⊆ *atoms F*
by(*induction F*) *auto*

lemma *subformulae-self*[*simp*,*intro*]: *F* ∈ *set* (*subformulae F*)
by(*induction F*; *simp*)

lemma *subformulas-in-subformulas*:
G ∧ *H* ∈ *set* (*subformulae F*) ⟹ *G* ∈ *set* (*subformulae F*) ∧ *H* ∈ *set* (*subformulae F*)
G ∨ *H* ∈ *set* (*subformulae F*) ⟹ *G* ∈ *set* (*subformulae F*) ∧ *H* ∈ *set* (*subformulae F*)
G → *H* ∈ *set* (*subformulae F*) ⟹ *G* ∈ *set* (*subformulae F*) ∧ *H* ∈ *set* (*subformulae F*)
¬ *G* ∈ *set* (*subformulae F*) ⟹ *G* ∈ *set* (*subformulae F*)
by(*fastforce elim*: *subsubformulae*)+

lemma *length-subformulae*: *length* (*subformulae F*) = *size F* **by**(*induction F*; *simp*)

1.1 Derived Connectives

definition *Top* (‹⊤›) **where**
⊤ ≡ ⊥ → ⊥
lemma *top-atoms-simp*[*simp*]: *atoms* ⊤ = {} **unfolding** *Top-def* **by** *simp*

primrec *BigAnd* :: *'a formula list* ⇒ *'a formula* (‹⋀-›) **where**
⋀*Nil* = (¬⊥) — essentially, it doesn't matter what I use here. But since I want to use this in CNFs, implication is not a nice thing to have. |
⋀(*F*#*Fs*) = *F* ∧ ⋀*Fs*

lemma *atoms-BigAnd*[*simp*]: *atoms* (⋀*Fs*) = ⋃(*atoms* ‘ *set Fs*)
  by(*induction Fs*; *simp*)

primrec *BigOr* :: *'a formula list* ⇒ *'a formula* (‹⋁-›) **where**
⋁*Nil* = ⊥ |
⋁(*F*#*Fs*) = *F* ∨ ⋁*Fs*

Formulas are countable if their atoms are, and *countable-datatype* is really helpful with that.

instance *formula* :: (*countable*) *countable* **by** *countable-datatype*

definition *prod-unions A B* ≡ *case A of* (*a*,*b*) ⇒ *case B of* (*c*,*d*) ⇒ (*a* ∪ *c*, *b* ∪ *d*)
primrec *pn-atoms* **where**
pn-atoms (*Atom A*) = ({*A*},{}) |
pn-atoms Bot = ({},{}) |
pn-atoms (*Not F*) = *prod.swap* (*pn-atoms F*) |
pn-atoms (*And F G*) = *prod-unions* (*pn-atoms F*) (*pn-atoms G*) |
pn-atoms (*Or F G*) = *prod-unions* (*pn-atoms F*) (*pn-atoms G*) |
pn-atoms (*Imp F G*) = *prod-unions* (*prod.swap* (*pn-atoms F*)) (*pn-atoms G*)
lemma *pn-atoms-atoms*: *atoms F* = *fst* (*pn-atoms F*) ∪ *snd* (*pn-atoms F*)
  by(*induction F*) (*auto simp add*: *prod-unions-def split*: *prod.splits*)

A very trivial simplifier. Does wonders as a postprocessor for the Harrison-style Craig interpolations.

context begin
definition *isstop F* ≡ *F* = ¬⊥ ∨ *F* = ⊤
fun *simplify-consts* **where**
simplify-consts (*Atom k*) = *Atom k* |
simplify-consts ⊥ = ⊥ |
simplify-consts (*Not F*) = (*let S* = *simplify-consts F in case S of* (*Not G*) ⇒ *G* | - ⇒
  if isstop S then ⊥ *else* ¬ *S*) |
simplify-consts (*And F G*) = (*let S* = *simplify-consts F*; *T* = *simplify-consts G in* (
  if S = ⊥ *then* ⊥ *else*
  if isstop S then T — not ⊤, *T else*
  if T = ⊥ *then* ⊥ *else*
  if isstop T then S else
  if S = *T then S else*
  S ∧ *T*)) |
simplify-consts (*Or F G*) = (*let S* = *simplify-consts F*; *T* = *simplify-consts G in* (
  if S = ⊥ *then T else*

if isstop S then ⊤ else
if T = ⊥ then S else
if isstop T then ⊤ else
if S = T then S else
S ∨ T)) |
simplify-consts (Imp F G) = (let S = simplify-consts F; T = simplify-consts G in
(
if S = ⊥ then ⊤ else
if isstop S then T else
if isstop T then ⊤ else
if T = ⊥ then ¬ S else
if S = T then ⊤ else
case S of Not H ⇒ (case T of Not I ⇒
I → H | - ⇒
H ∨ T) | - ⇒
S → T))

lemma simplify-consts-size-le: size (simplify-consts F) ≤ size F
proof −
have [simp]: Suc (Suc 0) ≤ size F + size G **for** F G :: 'a formula **by**(cases F; cases G; simp)
show ?thesis **by**(induction F; fastforce simp add: Let-def isstop-def Top-def split: formula.splits)
qed

lemma simplify-const: atoms F = {} ⟹ isstop (simplify-consts F) ∨ (simplify-consts F) = ⊥
by(induction F; fastforce simp add: Let-def isstop-def Top-def split: formula.splits)
value (size ⊤, size (¬⊥))

end

fun all-formulas-of-size **where**
all-formulas-of-size K 0 = {⊥} ∪ Atom ‘ K |
all-formulas-of-size K (Suc n) = (
let af = ⋃(set [all-formulas-of-size K m. m ← [0..<Suc n]]) in
(⋃F∈af.
(⋃G∈af. if size F + size G ≤ Suc n then {And F G, Or F G, Imp F G} else {})
∪ (if size F ≤ Suc n then {Not F} else {}))
∪ af)

lemma all-formulas-of-size: F ∈ all-formulas-of-size K n ⟷ (size F ≤ Suc n ∧ atoms F ⊆ K) (**is** ?l ⟷ ?r)
proof −
have rl: ?r ⟹ ?l
proof(induction F arbitrary: n)

case (*Atom x*)
have ∗: *Atom x* ∈ *all-formulas-of-size K 0* **using** *Atom* **by** *simp*
hence ∗∗: *Atom x* ∈ ⋃ (*all-formulas-of-size K* ‘ *set* [*0*..<*Suc m*]) **for** *m*
by (*simp*; *metis atLeastLessThan-iff le-zero-eq not-le*)
thus *?case* **using** *Atom* **by**(*cases n*; *simp*)
next
case *Bot*
have ∗: *Bot* ∈ *all-formulas-of-size K 0* **by** *simp*
hence ∗∗: *Bot* ∈ ⋃ (*all-formulas-of-size K* ‘ *set* [*0*..<*Suc m*]) **for** *m*
by (*simp*; *metis atLeastLessThan-iff le-zero-eq not-le*)
then show *?case* **using** *Bot* **by**(*cases n*; *simp*)
next
case (*Not F*)
have ∗: *size F* ≤ *n* **using** *Not* **by** *simp*
then obtain *m* **where** *n*[*simp*]: *n* = *Suc m* **by** (*metis Suc-diff-1 formula.size-neq leD neq0-conv*)
with *Not* **have** *IH*: *F* ∈ *all-formulas-of-size K m* **by** *simp*
then show *?case* **using** ∗ **by**(*simp add*: *bexI*[**where** *x*=*F*])
next
case (*And F G*)
with *And* **have** ∗: *size F* + *size G* ≤ *n* **by** *simp*
then obtain *m* **where** *n*[*simp*]: *n* = *Suc m*
by (*metis Suc-diff-1 add-is-0 formula.size-neq le-zero-eq neq0-conv*)
then obtain *nF nG* **where** *nFG*[*simp*]: *size F* ≤ *nF size G* ≤ *nG n* = *nF* + *nG*
by (*metis* ∗ *add.assoc nat-le-iff-add order-refl*)
then obtain *mF mG* **where** *mFG*[*simp*]: *nF* = *Suc mF nG* = *Suc mG*
by (*metis Suc-diff-1 formula.size-neq leD neq0-conv*)
with *And* **have** *IH*: *F* ∈ *all-formulas-of-size K mF G* ∈ *all-formulas-of-size K mG*
using *nFG* **by** *simp*+
let *?af* = ⋃(*set* [*all-formulas-of-size K m*. *m* ← [*0*..<*Suc m*]])
have *r*: *F* ∈ *all-formulas-of-size K mF* ⟹ *mF* ≤ *n* ⟹ *F* ∈ ⋃(*set* (*map* (*all-formulas-of-size K*) [*0*..<*Suc n*]))
for *F mF n* **by** *fastforce*
have *af*: *F* ∈ *?af G* ∈ *?af* **using** *nFG*(*3*) **by**(*intro IH*[*THEN r*]; *simp*)+
have *m*: *F* ∧ *G* ∈ (*if size F* + *size G* ≤ *Suc m then* {*F* ∧ *G*, *F* ∨ *G*, *F* → *G*} *else* {}) **using** ∗ **by** *simp*
from *IH* ∗ **show** *?case* **using** *af* **by**(*simp only*: *n all-formulas-of-size.simps Let-def*, *insert m*) *fast*
next
case (*Or F G*) **case** (*Imp F G*) — analogous **qed**
have *lr*: *?r* **if** *l*: *?l*
proof
have ∗: *F* ∈ *all-formulas-of-size K x* ⟹ *F* ∈ *all-formulas-of-size K* (*x* + *n*) **for** *x n*
by(*induction n*; *simp*)
show *size F* ≤ *Suc n* **using** *l*
by(*induction n*; *auto split*: *if-splits*) (*metis* ∗ *le-SucI le-eq-less-or-eq le-iff-add*)

show *atoms* $F \subseteq K$ **using** *l*
proof(*induction n arbitrary*: *F rule*: *less-induct*)
case (*less x*)
then show *?case* **proof**(*cases x*)
case *0* **with** *less* **show** *?thesis* **by** *force*
next
case (*Suc y*) **with** *less* **show** *?thesis*
by(*simp only*: *all-formulas-of-size.simps Let-def*) (*fastforce simp add*: *less-Suc-eq split*: *if-splits*)
qed
qed
qed
from *lr rl* **show** *?thesis* **proof qed**
qed

end

1.2 Semantics

theory *Sema*
imports *Formulas*
begin

type-synonym *'a valuation* = *'a* ⇒ *bool*

The implicit statement here is that an assignment or valuation is always defined on all atoms (because HOL is a total logic). Thus, there are no unsuitable assignments.

primrec *formula-semantics* :: *'a valuation* ⇒ *'a formula* ⇒ *bool* (**infix** ‹⊨› *51*)
where
$\mathcal{A} \models$ *Atom k* = $\mathcal{A}$ *k* |
\- ⊨ ⊥ = *False* |
$\mathcal{A} \models$ *Not F* = (¬ $\mathcal{A} \models F$) |
$\mathcal{A} \models$ *And F G* = ($\mathcal{A} \models F \land \mathcal{A} \models G$) |
$\mathcal{A} \models$ *Or F G* = ($\mathcal{A} \models F \lor \mathcal{A} \models G$) |
$\mathcal{A} \models$ *Imp F G* = ($\mathcal{A} \models F \longrightarrow \mathcal{A} \models G$)

abbreviation *valid* (‹⊨ -› *51*) **where**
$\models F \equiv \forall A.\ A \models F$

lemma *irrelevant-atom*[*simp*]: $A \notin$ *atoms* $F \Longrightarrow (\mathcal{A}(A := V)) \models F \longleftrightarrow \mathcal{A} \models F$
by (*induction F*; *simp*)
lemma *relevant-atoms-same-semantics*: $\forall k \in$ *atoms* $F.\ \mathcal{A}_1\ k = \mathcal{A}_2\ k \Longrightarrow \mathcal{A}_1 \models F \longleftrightarrow \mathcal{A}_2 \models F$
by(*induction F*; *simp*)

context begin

Just a definition more similar to [9, p. 5]. Unfortunately, using this as the main definition would get in the way of automated reasoning all the time.

private primrec *formula-semantics-alt* **where**
formula-semantics-alt 𝒜 (*Atom k*) = 𝒜 *k* |
formula-semantics-alt 𝒜 (*Bot*) = *False* |
formula-semantics-alt 𝒜 (*Not a*) = (*if formula-semantics-alt* 𝒜 *a then False else True*) |
formula-semantics-alt 𝒜 (*And a b*) = (*if formula-semantics-alt* 𝒜 *a then formula-semantics-alt* 𝒜 *b else False*) |
formula-semantics-alt 𝒜 (*Or a b*) = (*if formula-semantics-alt* 𝒜 *a then True else formula-semantics-alt* 𝒜 *b*) |
formula-semantics-alt 𝒜 (*Imp a b*) = (*if formula-semantics-alt* 𝒜 *a then formula-semantics-alt* 𝒜 *b else True*)
private lemma *formula-semantics-alt* 𝒜 *F* ⟷ 𝒜 ⊨ *F*
  by(*induction F*; *simp*)

If you fancy a definition more similar to [3, p. 39], this is probably the closest you can go without going incredibly ugly.

private primrec *formula-semantics-tt* **where**
formula-semantics-tt 𝒜 (*Atom k*) = 𝒜 *k* |
formula-semantics-tt 𝒜 (*Bot*) = *False* |
formula-semantics-tt 𝒜 (*Not a*) = (*case formula-semantics-tt* 𝒜 *a of True* ⇒ *False* | *False* ⇒ *True*) |
formula-semantics-tt 𝒜 (*And a b*) = (*case* (*formula-semantics-tt* 𝒜 *a*, *formula-semantics-tt* 𝒜 *b*) *of*
  (*False*, *False*) ⇒ *False*
| (*False*, *True*) ⇒ *False*
| (*True*, *False*) ⇒ *False*
| (*True*, *True*) ⇒ *True*) |
formula-semantics-tt 𝒜 (*Or a b*) = (*case* (*formula-semantics-tt* 𝒜 *a*, *formula-semantics-tt* 𝒜 *b*) *of*
  (*False*, *False*) ⇒ *False*
| (*False*, *True*) ⇒ *True*
| (*True*, *False*) ⇒ *True*
| (*True*, *True*) ⇒ *True*) |
formula-semantics-tt 𝒜 (*Imp a b*) = (*case* (*formula-semantics-tt* 𝒜 *a*, *formula-semantics-tt* 𝒜 *b*) *of*
  (*False*, *False*) ⇒ *True*
| (*False*, *True*) ⇒ *True*
| (*True*, *False*) ⇒ *False*
| (*True*, *True*) ⇒ *True*)
private lemma *A* ⊨ *F* ⟷ *formula-semantics-tt A F*
  by(*induction F*; *simp split*: *prod.splits bool.splits*)
end

definition *entailment* :: *'a formula set* ⇒ *'a formula* ⇒ *bool* (‹(- ⊫/ -)› [*53*,*53*] *53*) **where**
Γ ⊫ *F* ≡ (∀𝒜. ((∀ *G* ∈ Γ. 𝒜 ⊨ *G*) ⟶ (𝒜 ⊨ *F*)))

We write entailment differently than semantics (⊨ vs. ⊫). For humans, it is usually pretty clear what is meant in a specific situation, but it often needs to be decided from context that Isabelle/HOL does not have.

Some helpers for the derived connectives

lemma *top-semantics*[*simp*,*intro*!]: $A \models \top$ **unfolding** *Top-def* **by** *simp*
lemma *BigAnd-semantics*[*simp*]: $A \models \bigwedge F \longleftrightarrow (\forall f \in set\ F.\ A \models f)$ **by**(*induction F*; *simp*)
lemma *BigOr-semantics*[*simp*]: $A \models \bigvee F \longleftrightarrow (\exists f \in set\ F.\ A \models f)$ **by**(*induction F*; *simp*)

Definitions for sets of formulae, used for compactness and model existence.

definition $sat\ S \equiv \exists \mathcal{A}.\ \forall F \in S.\ \mathcal{A} \models F$
definition $fin\text{-}sat\ S \equiv (\forall s \subseteq S.\ finite\ s \longrightarrow sat\ s)$

lemma *entail-sat*: $\Gamma \models \bot \longleftrightarrow \neg\ sat\ \Gamma$
  unfolding *sat-def entailment-def* **by** *simp*

lemma *pn-atoms-updates*: $p \notin snd\ (pn\text{-}atoms\ F) \Longrightarrow n \notin fst\ (pn\text{-}atoms\ F) \Longrightarrow$
  $((M \models F \longrightarrow (M(p := True) \models F \wedge M(n := False) \models F))) \wedge ((\neg(M \models F)) \longrightarrow \neg(M(n := True) \models F) \wedge \neg(M(p := False) \models F))$
proof(*induction F arbitrary*: *n p*)
  case (*Imp F G*)
  from *Imp.prems* **have** *prems*:
    $p \notin fst\ (pn\text{-}atoms\ F)$ $p \notin snd\ (pn\text{-}atoms\ G)$
    $n \notin snd\ (pn\text{-}atoms\ F)$ $n \notin fst\ (pn\text{-}atoms\ G)$
   by(*simp-all add*: *prod-unions-def split*: *prod.splits*)
  have *IH1*: $M \models F \Longrightarrow M(n := True) \models F$ $M \models F \Longrightarrow M(p := False) \models F$ $\neg M \models F \Longrightarrow \neg M(p := True) \models F$ $\neg M \models F \Longrightarrow \neg M(n := False) \models F$
   using *Imp.IH*(*1*)[*OF prems*(*3*) *prems*(*1*)] **by** *blast+*
  have *IH2*: $M \models G \Longrightarrow M(p := True) \models G$ $M \models G \Longrightarrow M(n := False) \models G$ $\neg M \models G \Longrightarrow \neg M(n := True) \models G$ $\neg M \models G \Longrightarrow \neg M(p := False) \models G$
   using *Imp.IH*(*2*)[*OF prems*(*2*) *prems*(*4*)] **by** *blast+*
   show *?case* **proof**(*intro conjI*; *intro impI*)
    assume $M \models F \rightarrow G$
    then consider $\neg M \models F \mid M \models G$ **by** *auto*
    thus $M(p := True) \models F \rightarrow G \wedge M(n := False) \models F \rightarrow G$ **using** *IH1*(*3*,*4*) *IH2*(*1*,*2*) **by** *cases simp-all*
   next
    assume $\neg(M \models F \rightarrow G)$
    hence $M \models F$ $\neg M \models G$ **by** *simp-all*
   thus $\neg M(n := True) \models F \rightarrow G \wedge \neg M(p := False) \models F \rightarrow G$ **using** *IH1*(*1*,*2*) *IH2*(*3*,*4*) **by** *simp*
   qed
 next
  case (*And F G*)
  from *And.prems* **have** *prems*:
    $p \notin snd\ (pn\text{-}atoms\ F)$ $p \notin snd\ (pn\text{-}atoms\ G)$
    $n \notin fst\ (pn\text{-}atoms\ F)$ $n \notin fst\ (pn\text{-}atoms\ G)$
   by(*simp-all add*: *prod-unions-def split*: *prod.splits*)

have *IH1*: $M \models F \Longrightarrow M(p := True) \models F$ $M \models F \Longrightarrow M(n := False) \models F$ $\neg$ $M \models F \Longrightarrow \neg\ M(n := True) \models F$ $\neg\ M \models F \Longrightarrow \neg\ M(p := False) \models F$
using *And.IH(1)[OF prems(1) prems(3)]* **by** *blast+*
have *IH2*: $M \models G \Longrightarrow M(p := True) \models G$ $M \models G \Longrightarrow M(n := False) \models G$ $\neg$ $M \models G \Longrightarrow \neg\ M(n := True) \models G$ $\neg\ M \models G \Longrightarrow \neg\ M(p := False) \models G$
using *And.IH(2)[OF prems(2) prems(4)]* **by** *blast+*
show *?case* **proof**(*intro conjI*; *intro impI*)
assume $\neg M \models F \wedge G$
then consider $\neg\ M \models F \mid \neg\ M \models G$ **by** *auto*
thus $\neg\ M(n := True) \models F \wedge G \wedge \neg\ M(p := False) \models F \wedge G$ **using** *IH1 IH2* **by** *cases simp-all*
next
assume $M \models F \wedge G$
hence $M \models F$ $M \models G$ **by** *simp-all*
thus $M(p := True) \models F \wedge G \wedge M(n := False) \models F \wedge G$ **using** *IH1 IH2* **by** *simp*
qed
next
case (*Or F G*)
from *Or.prems* **have** *prems*:
$p \notin snd\ (pn\text{-}atoms\ F)$ $p \notin snd\ (pn\text{-}atoms\ G)$
$n \notin fst\ (pn\text{-}atoms\ F)$ $n \notin fst\ (pn\text{-}atoms\ G)$
by(*simp-all add*: *prod-unions-def split*: *prod.splits*)
have *IH1*: $M \models F \Longrightarrow M(p := True) \models F$ $M \models F \Longrightarrow M(n := False) \models F$ $\neg$ $M \models F \Longrightarrow \neg\ M(n := True) \models F$ $\neg\ M \models F \Longrightarrow \neg\ M(p := False) \models F$
using *Or.IH(1)[OF prems(1) prems(3)]* **by** *blast+*
have *IH2*: $M \models G \Longrightarrow M(p := True) \models G$ $M \models G \Longrightarrow M(n := False) \models G$ $\neg$ $M \models G \Longrightarrow \neg\ M(n := True) \models G$ $\neg\ M \models G \Longrightarrow \neg\ M(p := False) \models G$
using *Or.IH(2)[OF prems(2) prems(4)]* **by** *blast+*
show *?case* **proof**(*intro conjI*; *intro impI*)
assume $M \models F \vee G$
then consider $M \models F \mid M \models G$ **by** *auto*
thus $M(p := True) \models F \vee G \wedge M(n := False) \models F \vee G$ **using** *IH1 IH2* **by** *cases simp-all*
next
assume $\neg M \models F \vee G$
hence $\neg M \models F$ $\neg M \models G$ **by** *simp-all*
thus $\neg M(n := True) \models F \vee G \wedge \neg M(p := False) \models F \vee G$ **using** *IH1 IH2* **by** *simp*
qed
qed *simp-all*

lemma *const-simplifier-correct*: $\mathcal{A} \models simplify\text{-}consts\ F \longleftrightarrow \mathcal{A} \models F$
by (*induction F*) (*auto simp add*: *Let-def isstop-def Top-def split*: *formula.splits*)

end

1.3 Substitutions

theory *Substitution*
imports *Formulas*
begin

primrec *subst* **where**
subst A F (Atom B) = (if A = B then F else Atom B) |
subst - - $\bot = \bot$ |
subst A F (G $\vee$ *H) = (subst A F G* $\vee$ *subst A F H)* |
subst A F (G $\wedge$ *H) = (subst A F G* $\wedge$ *subst A F H)* |
subst A F (G $\rightarrow$ *H) = (subst A F G* $\rightarrow$ *subst A F H)* |
subst A F ($\neg$ *H) = (*$\neg$ *(subst A F H))*
term *subst*
abbreviation *subst-syntax* (‹(-[/(-/′//-)])› [*70*,*70*] *69*) **where**
A[B / C] $\equiv$ *subst C B A*

lemma *no-subst*[*simp*]: $k \notin$ *atoms F* $\Longrightarrow$ *F[G / k] = F* **by**(*induction F*; *simp*)
lemma *subst-atoms*: $k \in$ *atoms F* $\Longrightarrow$ *atoms (F[G / k]) = atoms F* $-$ $\{k\}$ $\cup$ *atoms G*
proof(*induction F*)
  case (*And F G*) **thus** *?case* **by**(*cases k* $\in$ *atoms F*; *force*) **next**
  case (*Or F G*) **thus** *?case* **by**(*cases k* $\in$ *atoms F*; *force*) **next**
  case (*Imp F G*) **thus** *?case* **by**(*cases k* $\in$ *atoms F*; *force*) **next**
qed *simp-all*

lemma *subst-atoms-simp*: *atoms (F[G / k]) = atoms F* $-$ $\{k\}$ $\cup$ *(if k* $\in$ *atoms F then atoms G else* $\{\}$*)*
  by(*simp add*: *subst-atoms*)

end
theory *Substitution-Sema*
imports *Substitution Sema*
begin

lemma *substitution-lemma*: $\mathcal{A} \models F[G\ /\ n] \longleftrightarrow \mathcal{A}(n := \mathcal{A} \models G) \models F$ **by**(*induction F*; *simp*)

end

1.4 Conjunctive Normal Forms

theory *CNF*
imports *Main HOL−Library.Simps-Case-Conv*
begin

datatype *′a literal = Pos ′a* (‹(-$^{+}$)› [*1000*] *999*) | *Neg ′a* (‹(-$^{-1}$)› [*1000*] *999*)

type-synonym *′a clause = ′a literal set*
abbreviation *empty-clause* (‹□›) **where** □ $\equiv$ $\{\}$ :: *′a clause*

primrec *atoms-of-lit* **where**
atoms-of-lit (*Pos k*) = *k* |
atoms-of-lit (*Neg k*) = *k*
case-of-simps *lit-atoms-cases*: *atoms-of-lit.simps*

definition *atoms-of-cnf c* = *atoms-of-lit* ‘ $\bigcup$ *c*
lemma *atoms-of-cnf-alt*: *atoms-of-cnf c* = $\bigcup$ (((‘) *atoms-of-lit*) ‘ *c*)
  unfolding *atoms-of-cnf-def* **by** *blast*

lemma *atoms-of-cnf-Un*: *atoms-of-cnf* (*S* ∪ *T*) = *atoms-of-cnf S* ∪ *atoms-of-cnf T*
  unfolding *atoms-of-cnf-def* **by** *auto*

term {*0*$^+$}::*nat clause*
translations
  {*x*} <= *CONST insert x* □
term {*0*$^+$}::*nat clause*

end

CNFs alone are nice, but now we want to relate between CNFs and formulas.

theory *CNF-Formulas*
imports *Formulas CNF*
begin

context begin

function (*sequential*) *nnf* **where**
nnf (*Atom k*) = (*Atom k*) |
nnf ⊥ = ⊥ |
nnf (*Not* (*And F G*)) = *Or* (*nnf* (*Not F*)) (*nnf* (*Not G*)) |
nnf (*Not* (*Or F G*)) = *And* (*nnf* (*Not F*)) (*nnf* (*Not G*)) |
nnf (*Not* (*Not F*)) = *nnf F* |
nnf (*Not* (*Imp F G*)) = *And* (*nnf F*) (*nnf* (*Not G*)) |
nnf (*Not F*) = (*Not F*) |
nnf (*And F G*) = *And* (*nnf F*) (*nnf G*) |
nnf (*Or F G*) = *Or* (*nnf F*) (*nnf G*) |
nnf (*Imp F G*) = *Or* (*nnf* (*Not F*)) (*nnf G*)
  by(*pat-completeness*) *auto*

private fun *nnf-cost* **where**
nnf-cost (*Atom* -) = *42* |
nnf-cost ⊥ = *42* |
nnf-cost (*Not F*) = *Suc* (*nnf-cost F*) |

nnf-cost (*And F G*) = *Suc* (*nnf-cost F* + *nnf-cost G*) |
nnf-cost (*Or F G*) = *Suc* (*nnf-cost F* + *nnf-cost G*) |
nnf-cost (*Imp F G*) = *Suc* (*Suc* (*nnf-cost F* + *nnf-cost G*))

termination *nnf* **by**(*relation measure* (λ*F*. *nnf-cost F*); *simp*)

lemma *nnf* ((*Atom* (*k*::*nat*)) → (*Not* ((*Atom l*) ∨ (*Not* (*Atom m*))))) = ¬ (*Atom k*) ∨ (¬ (*Atom l*) ∧ *Atom m*)
  by *code-simp*

fun *is-lit-plus* **where**
is-lit-plus ⊥ = *True* |
is-lit-plus (*Not* ⊥) = *True* |
is-lit-plus (*Atom* -) = *True* |
is-lit-plus (*Not* (*Atom* -)) = *True* |
is-lit-plus - = *False*
case-of-simps *is-lit-plus-cases*: *is-lit-plus.simps*
fun *is-disj* **where**

is-disj (*Or F G*) = (*is-lit-plus F* ∧ *is-disj G*) |
is-disj F = *is-lit-plus F*
fun *is-cnf* **where**

is-cnf (*And F G*) = (*is-cnf F* ∧ *is-cnf G*) |
is-cnf H = *is-disj H*
fun *is-nnf* **where**
is-nnf (*Imp F G*) = *False* |
is-nnf (*And F G*) = (*is-nnf F* ∧ *is-nnf G*) |
is-nnf (*Or F G*) = (*is-nnf F* ∧ *is-nnf G*) |
is-nnf F = *is-lit-plus F*

lemma *is-nnf-nnf*: *is-nnf* (*nnf F*)
  by(*induction F rule*: *nnf.induct*; *simp*)
lemma *nnf-no-imp*: *A* → *B* ∉ *set* (*subformulae* (*nnf F*))
  by(*induction F rule*: *nnf.induct*; *simp*)
lemma *subformulae-nnf*: *is-nnf F* ⟹ *G* ∈ *set* (*subformulae F*) ⟹ *is-nnf G*
  by(*induction F rule*: *is-nnf.induct*; *simp add*: *is-lit-plus-cases split*: *formula.splits*; *elim disjE conjE*; *simp*)
lemma *is-nnf-NotD*: *is-nnf* (¬ *F*) ⟹ (∃ *k*. *F* = *Atom k*) ∨ *F* = ⊥
  by(*cases F*; *simp*)

fun *cnf* :: *'a formula* ⇒ *'a clause set* **where**
cnf (*Atom k*) = {{ k^+ }} |
cnf (*Not* (*Atom k*)) = {{ k^{-1} }} |
cnf ⊥ = {□} |
cnf (*Not* ⊥) = {} |
cnf (*And F G*) = *cnf F* ∪ *cnf G* |
cnf (*Or F G*) = {*C* ∪ *D*| *C D*. *C* ∈ (*cnf F*) ∧ *D* ∈ (*cnf G*)}

lemma *cnf-fin*:
assumes *is-nnf F*
shows *finite* (*cnf F*) $C \in$ *cnf F* $\Longrightarrow$ *finite C*
proof −
  have *finite* (*cnf F*) $\wedge$ ($C \in$ *cnf F* $\longrightarrow$ *finite C*) **using** *assms*
   by(*induction F arbitrary*: *C rule*: *cnf.induct*; *clarsimp simp add*: *finite-image-set2*)
  thus *finite* (*cnf F*) $C \in$ *cnf F* $\Longrightarrow$ *finite C* **by** *simp*+
qed

fun *cnf-lists* :: *'a formula* $\Rightarrow$ *'a literal list list* **where**
cnf-lists (*Atom k*) = [[k^+]] |
cnf-lists (*Not* (*Atom k*)) = [[k^{-1}]] |
cnf-lists $\bot$ = [[]] |
cnf-lists (*Not* $\bot$) = [] |
cnf-lists (*And F G*) = *cnf-lists F* @ *cnf-lists G* |
cnf-lists (*Or F G*) = [*f* @ *g*. *f* $\leftarrow$ (*cnf-lists F*), *g* $\leftarrow$ (*cnf-lists G*)]

primrec *form-of-lit* **where**
form-of-lit (*Pos k*) = *Atom k* |
form-of-lit (*Neg k*) = $\neg$(*Atom k*)
case-of-simps *form-of-lit-cases*: *form-of-lit.simps*

definition *disj-of-clause c* $\equiv$ $\bigvee$[*form-of-lit l*. *l* $\leftarrow$ *c*]
definition *form-of-cnf F* $\equiv$ $\bigwedge$[*disj-of-clause c*. *c* $\leftarrow$ *F*]
definition *cnf-form-of* $\equiv$ *form-of-cnf* $\circ$ *cnf-lists*
lemmas *cnf-form-of-defs* = *cnf-form-of-def form-of-cnf-def disj-of-clause-def*

lemma *disj-of-clause-simps*[*simp*]:
  disj-of-clause [] = $\bot$
  disj-of-clause (*F*#*FF*) = *form-of-lit F* $\vee$ *disj-of-clause FF*
by(*simp-all add*: *disj-of-clause-def*)

lemma *is-cnf-BigAnd*: *is-cnf* ($\bigwedge$*ls*) $\longleftrightarrow$ ($\forall$ *l* $\in$ *set ls*. *is-cnf l*)
  by(*induction ls*; *simp*)

private lemma *BigOr-is-not-cnf''*: *is-cnf* ($\bigvee$*ls*) $\Longrightarrow$ ($\forall$ *l* $\in$ *set ls*. *is-lit-plus l*)
proof(*induction ls*)
  case (*Cons l ls*)
  from *Cons.prems* **have** *is-cnf* ($\bigvee$ *ls*)
    by (*metis BigOr.simps is-cnf.simps*(*3*,*5*) *is-disj.simps*(*1*) *list.exhaust*)
  thus *?case* **using** *Cons* **by** *simp*
qed *simp*
private lemma *BigOr-is-not-cnf'*: ($\forall$ *l* $\in$ *set ls*. *is-lit-plus l*) $\Longrightarrow$ *is-cnf* ($\bigvee$*ls*)
 by(*induction ls*; *simp*) (*metis BigOr.simps*(*1*, *2*) *formula.distinct*(*25*) *is-cnf.elims*(*2*) *is-cnf.simps*(*3*) *list.exhaust*)

lemma *BigOr-is-not-cnf*: *is-cnf* ($\bigvee$*ls*) $\longleftrightarrow$ ($\forall$ *l* $\in$ *set ls*. *is-lit-plus l*)
  using *BigOr-is-not-cnf' BigOr-is-not-cnf''* **by** *blast*

lemma *is-nnf-BigAnd*[*simp*]: *is-nnf* ($\bigwedge$*ls*) $\longleftrightarrow$ ($\forall$ *l* $\in$ *set ls*. *is-nnf l*)
by(*induction ls*; *simp*)
lemma *is-nnf-BigOr*[*simp*]: *is-nnf* ($\bigvee$*ls*) $\longleftrightarrow$ ($\forall$ *l* $\in$ *set ls*. *is-nnf l*)
by(*induction ls*; *simp*)
lemma *form-of-lit-is-nnf*[*simp*,*intro*!]: *is-nnf* (*form-of-lit x*)
by(*cases x*; *simp*)
lemma *form-of-lit-is-lit*[*simp*,*intro*!]: *is-lit-plus* (*form-of-lit x*)
by(*cases x*; *simp*)
lemma *disj-of-clause-is-nnf*[*simp*,*intro*!]: *is-nnf* (*disj-of-clause F*)
unfolding *disj-of-clause-def* **by** *simp*

lemma *cnf-form-of-is*: *is-nnf F* $\Longrightarrow$ *is-cnf* (*cnf-form-of F*)
by(*cases F*) (*auto simp*: *cnf-form-of-defs is-cnf-BigAnd BigOr-is-not-cnf*)

lemma *nnf-cnf-form*: *is-nnf F* $\Longrightarrow$ *is-nnf* (*cnf-form-of F*)
by(*cases F*) (*auto simp add*: *cnf-form-of-defs*)

lemma *cnf-BigAnd*: *cnf* ($\bigwedge$*ls*) = ($\bigcup$ *x* $\in$ *set ls*. *cnf x*)
by(*induction ls*; *simp*)

lemma *cnf-BigOr*: *cnf* ($\bigvee$ (*x* @ *y*)) = $\{f \cup g \mid f\ g.\ f \in cnf\ (\bigvee x) \wedge g \in cnf\ (\bigvee y)\}$
by(*induction x arbitrary*: *y*; *simp*) (*metis* (*no-types*, *opaque-lifting*) *sup.assoc*)

lemma *cnf-cnf*: *is-nnf F* $\Longrightarrow$ *cnf* (*cnf-form-of F*) = *cnf F*
by(*induction F rule*: *cnf.induct*;
fastforce simp add: *cnf-form-of-defs cnf-BigAnd cnf-BigOr*)

lemma *is-nnf-nnf-id*: *is-nnf F* $\Longrightarrow$ *nnf F* = *F*
proof(*induction rule*: *is-nnf.induct*)
fix *v* **assume** *is-nnf* ($\neg$ *v*)
thus *nnf* ($\neg$ *v*) = $\neg$ *v* **by**(*cases v rule*: *is-lit-plus.cases*; *simp*)
qed *simp-all*

lemma *disj-of-clause-is*: *is-disj* (*disj-of-clause R*)
by(*induction R*; *simp*)

lemma *form-of-cnf-is-nnf*: *is-nnf* (*form-of-cnf R*)
unfolding *form-of-cnf-def* **by** *simp*

lemma *cnf-disj*: *cnf* (*disj-of-clause R*) = {*set R*}
by(*induction R*; *simp add*: *form-of-lit-cases split*: *literal.splits*)
lemma *cnf-disj-ex*: *is-disj F* $\Longrightarrow$ $\exists$ *R*. *cnf F* = {*R*} $\vee$ *cnf F* = {}
by(*induction F rule*: *is-disj.induct*; *clarsimp simp*: *is-lit-plus-cases split*: *formula.splits*)

lemma *cnf-form-of-cnf*: *cnf* (*form-of-cnf S*) = *set* (*map set S*)

unfolding *form-of-cnf-def* **by** (*simp add*: *cnf-BigAnd cnf-disj*) *blast*

lemma *disj-is-nnf*: *is-disj F* $\Longrightarrow$ *is-nnf F*
by(*induction F rule*: *is-disj.induct*; *simp add*: *is-lit-plus-cases split*: *formula.splits*)

lemma *nnf-BigAnd*: *nnf* ($\bigwedge F$) = $\bigwedge$(*map nnf F*)
by(*induction F*; *simp*)

end

end
theory *CNF-Sema*
imports *CNF*
begin

primrec *lit-semantics* :: ('*a* $\Rightarrow$ *bool*) $\Rightarrow$ '*a literal* $\Rightarrow$ *bool* **where**
lit-semantics $\mathcal{A}$ (k^+) = $\mathcal{A}$ *k* |
lit-semantics $\mathcal{A}$ (k^{-1}) = ($\neg\mathcal{A}$ *k*)
case-of-simps *lit-semantics-cases*: *lit-semantics.simps*
definition *clause-semantics* **where**
clause-semantics $\mathcal{A}$ *C* $\equiv \exists$ *L* $\in$ *C*. *lit-semantics* $\mathcal{A}$ *L*
definition *cnf-semantics* **where**
cnf-semantics $\mathcal{A}$ *S* $\equiv \forall$ *C* $\in$ *S*. *clause-semantics* $\mathcal{A}$ *C*

end
theory *CNF-Formulas-Sema*
imports *CNF-Sema CNF-Formulas Sema*
begin

lemma *nnf-semantics*: $\mathcal{A} \models$ *nnf F* $\longleftrightarrow \mathcal{A} \models F$
by(*induction F rule*: *nnf.induct*; *simp*)

lemma *cnf-semantics*: *is-nnf F* $\Longrightarrow$ *cnf-semantics* $\mathcal{A}$ (*cnf F*) $\longleftrightarrow \mathcal{A} \models F$
by(*induction F rule*: *cnf.induct*; *simp add*: *cnf-semantics-def clause-semantics-def ball-Un*; *metis Un-iff*)

lemma *cnf-form-semantics*:
fixes *F* :: '*a formula*
assumes *nnf*: *is-nnf F*
shows $\mathcal{A} \models$ *cnf-form-of F* $\longleftrightarrow \mathcal{A} \models F$
proof –
define *cnf-semantics-list*
where *cnf-semantics-list* $\mathcal{A}$ *S* $\longleftrightarrow$ ($\forall$ *s* $\in$ *set S*. $\exists$ *l* $\in$ *set s*. *lit-semantics* $\mathcal{A}$ (*l* :: '*a literal*))
for $\mathcal{A}$ *S*
have *tcn*: *cnf F* = *set* (*map set* (*cnf-lists F*)) **using** *nnf*

by(*induction F rule*: *cnf.induct*) (*auto simp add*: *image-UN image-comp comp-def*
)
have $\mathcal{A} \models F \longleftrightarrow$ *cnf-semantics* $\mathcal{A}$ (*cnf F*) **using** *cnf-semantics*[*OF nnf*] **by**
simp
also have ... = *cnf-semantics* $\mathcal{A}$ (*set* (*map set* (*cnf-lists F*))) **unfolding** *tcn* **..**
also have ... = *cnf-semantics-list* $\mathcal{A}$ (*cnf-lists F*)
unfolding *cnf-semantics-def clause-semantics-def cnf-semantics-list-def* **by**
fastforce
also have ... = $\mathcal{A} \models$ (*cnf-form-of F*) **using** *nnf*
by(*induction F rule*: *cnf-lists.induct*;
simp add: *cnf-semantics-list-def cnf-form-of-defs ball-Un bex-Un*)
finally show *?thesis* **by** *simp*
qed

corollary $\exists G.\ \mathcal{A} \models F \longleftrightarrow \mathcal{A} \models G \wedge$ *is-cnf G*
using *cnf-form-of-is cnf-form-semantics is-nnf-nnf nnf-semantics* **by** *blast*

end

1.4.1 Going back: CNFs to formulas

theory *CNF-To-Formula*
imports *CNF-Formulas HOL−Library.List-Lexorder*
begin

One downside of CNFs is that they cannot be converted back to formulas as-is in full generality. If we assume an order on the atoms, we can convert finite CNFs.

instantiation *literal* :: (*ord*) *ord*
begin

definition
literal-less-def: *xs* < *ys* ⟷ (
if atoms-of-lit xs = *atoms-of-lit ys*
then (*case xs of Neg -* ⇒ (*case ys of Pos -* ⇒ *True* | *-* ⇒ *False*) | *-* ⇒ *False*)
else atoms-of-lit xs < *atoms-of-lit ys*)

definition
literal-le-def: (*xs* :: *- literal*) ≤ *ys* ⟷ *xs* < *ys* ∨ *xs* = *ys*

instance ..

end

instance *literal* :: (*linorder*) *linorder*
by *standard* (*auto simp add*: *literal-less-def literal-le-def split*: *literal.splits if-splits*)

definition *formula-of-cnf* **where**

formula-of-cnf S ≡ *form-of-cnf* (*sorted-list-of-set* (*sorted-list-of-set* ‘ *S*))

To use the lexicographic order on lists, we first have to convert the clauses to lists, then the set of lists of literals to a list.

lemma *simplify-consts* (*formula-of-cnf* ({{*Pos 0*}} :: *nat clause set*)) = *Atom 0*
by *code-simp*

lemma *cnf-formula-of-cnf*:
assumes *finite S* ∀ *C* ∈ *S*. *finite C*
shows *cnf* (*formula-of-cnf S*) = *S*
using *assms* **by**(*simp add*: *cnf-BigAnd formula-of-cnf-def form-of-cnf-def cnf-disj*)

end

1.4.2 Tseytin transformation

theory *Tseytin*
imports *Formulas CNF-Formulas*
begin

The *cnf* transformation clearly has exponential complexity. If the intention is to use Resolution to decide validity of a formula, that is clearly a deal-breaker for any practical implementation, since validity can be decided by brute force in exponential time. This theory pair shows the Tseytin transformation, a way to transform a formula while preserving validity. The *cnf* of the transformed formula will have clauses with maximally 3 atoms, and an amount of clauses linear in the size of the formula, at the cost of introducing one new atom for each subformula of *F* (i.e. *size F* many).

definition *pair-fun-upd f p* ≡ (*case p of* (*k*,*v*) ⇒ *fun-upd f k v*)

lemma *fold-pair-upd-triv*: *A* ∉ *fst* ‘ *set U* ⟹ *foldl pair-fun-upd F U A* = *F A*
by(*induction U arbitrary*: *F*; *simp*)
(*metis fun-upd-apply pair-fun-upd-def prod.simps*(*2*) *surjective-pairing*)

lemma *distinct-pair-update-one*: (*k*,*v*) ∈ *set U* ⟹ *distinct* (*map fst U*) ⟹ *foldl pair-fun-upd F U k* = *v*
by(*induction U arbitrary*: *F*; *clarsimp simp add*: *pair-fun-upd-def fold-pair-upd-triv split*: *prod.splits*)
(*insert fold-pair-upd-triv*, *fastforce*)

lemma *distinct-zipunzip*: *distinct xs* ⟹ *distinct* (*map fst* (*zip xs ys*)) **by** (*simp add*: *distinct-conv-nth*)

lemma *foldl-pair-fun-upd-map-of*: *distinct* (*map fst U*) ⟹ *foldl pair-fun-upd F U* = (λ*k*. *case map-of U k of Some v* ⇒ *v* | *None* ⇒ *F k*)

by(*unfold fun-eq-iff*; *induction U arbitrary*: *F*; *clarsimp split*: *option.splits simp*: *pair-fun-upd-def rev-image-eqI*)

lemma *map-of-map-apsnd*: *map-of* (*map* (*apsnd t*) *M*) = *map-option t* ∘ (*map-of* *M*)
by(*unfold fun-eq-iff comp-def*; *induction M*; *simp*)

definition *biimp* (**infix** ‹↔› *67*) **where** *F* ↔ *G* ≡ (*F* → *G*) ∧ (*G* → *F*)
lemma *atoms-biimp*[*simp*]: *atoms* (*F* ↔ *G*) = *atoms F* ∪ *atoms G*
unfolding *biimp-def* **by** *auto*
lemma *biimp-size*[*simp*]: *size* (*F* ↔ *G*) = (*2* ∗ (*size F* + *size G*)) + *3*
by(*simp add*: *biimp-def*)

locale *freshstuff* =
fixes *fresh* :: *'a set* ⇒ *'a*
assumes *isfresh*: *finite S* ⟹ *fresh S* ∉ *S*
begin

primrec *nfresh* **where**
nfresh S 0 = [] |
nfresh S (*Suc n*) = (*let f* = *fresh S in f* # *nfresh* (*f* ▷ *S*) *n*)

lemma *length-nfresh*: *length* (*nfresh S n*) = *n*
by(*induction n arbitrary*: *S*; *simp add*: *Let-def*)

lemma *nfresh-isfresh*: *finite S* ⟹ *set* (*nfresh S n*) ∩ *S* = {}
by(*induction n arbitrary*: *S*; *auto simp add*: *Let-def isfresh*)

lemma *nfresh-distinct*: *finite S* ⟹ *distinct* (*nfresh S n*)
by(*induction n arbitrary*: *S*; *simp add*: *Let-def*; *meson disjoint-iff-not-equal finite-insert insertI1 nfresh-isfresh*)

definition *tseytin-assmt F* ≡ *let SF* = *remdups* (*subformulae F*) *in zip* (*nfresh* (*atoms F*) (*length SF*)) *SF*

lemma *tseytin-assmt-distinct*: *distinct* (*map fst* (*tseytin-assmt F*))
unfolding *tseytin-assmt-def* **using** *nfresh-distinct* **by** (*simp add*: *Let-def length-nfresh*)

lemma *tseytin-assmt-has*: *G* ∈ *set* (*subformulae F*) ⟹ ∃ *n*. (*n*,*G*) ∈ *set* (*tseytin-assmt* *F*)
proof −
assume *G* ∈ *set* (*subformulae F*)
then have ∃ *n*. *G* = *subformulae F* ! *n* ∧ *n* < *length* (*subformulae F*)
by (*simp add*: *set-conv-nth*)
then have ∃ *a*. (*a*, *G*) ∈ *set* (*zip* (*nfresh* (*atoms F*) (*length* (*subformulae F*))) (*subformulae F*))
by (*metis* (*no-types*) *in-set-zip length-nfresh prod.sel*(*1*) *prod.sel*(*2*))
thus *?thesis* **by**(*simp add*: *tseytin-assmt-def Let-def*) (*metis fst-conv in-set-conv-nth*

in-set-zip length-nfresh set-remdups snd-conv)
qed

lemma *tseytin-assmt-new-atoms*: $(k,l) \in set\ (tseytin\text{-}assmt\ F) \Longrightarrow k \notin atoms\ F$
unfolding *tseytin-assmt-def Let-def* **using** *nfresh-isfresh* **by** (*fastforce dest*: *set-zip-leftD*)

primrec *tseytin-tran1* **where**
tseytin-tran1 S (*Atom k*) = [*Atom k* ↔ *S* (*Atom k*)] |
tseytin-tran1 S ⊥ = [⊥ ↔ *S* ⊥] |
tseytin-tran1 S (*Not F*) = [*S* (*Not F*) ↔ *Not* (*S F*)] @ *tseytin-tran1 S F* |
tseytin-tran1 S (*And F G*) = [*S* (*And F G*) ↔ *And* (*S F*) (*S G*)] @ *tseytin-tran1 S F* @ *tseytin-tran1 S G* |
tseytin-tran1 S (*Or F G*) = [*S* (*Or F G*) ↔ *Or* (*S F*) (*S G*)] @ *tseytin-tran1 S F* @ *tseytin-tran1 S G* |
tseytin-tran1 S (*Imp F G*) = [*S* (*Imp F G*) ↔ *Imp* (*S F*) (*S G*)] @ *tseytin-tran1 S F* @ *tseytin-tran1 S G*
definition *tseytin-toatom F* ≡ *Atom* ∘ *the* ∘ *map-of* (*map* (λ(*a*,*b*). (*b*,*a*)) (*tseytin-assmt F*))
definition *tseytin-tran F* ≡ ⋀(*let S* = *tseytin-toatom F in S F* # *tseytin-tran1 S F*)

lemma *distinct-snd-tseytin-assmt*: *distinct* (*map snd* (*tseytin-assmt F*))
unfolding *tseytin-assmt-def* **by**(*simp add*: *Let-def length-nfresh*)

lemma *tseytin-assmt-backlookup*: **assumes** $J \in set\ (subformulae\ F)$
shows (*the* (*map-of* (*map* (λ(*a*, *b*). (*b*, *a*)) (*tseytin-assmt F*)) *J*), *J*) ∈ *set* (*tseytin-assmt F*)
proof −
have *1*: *distinct* (*map snd M*) ⟹ *J* ∈ *snd* ‘ *set M* ⟹ (*the* (*map-of* (*map* (λ(*a*, *b*). (*b*, *a*)) *M*) *J*), *J*) ∈ *set M* **for** *J M*
by(*induction M*; *clarsimp split*: *prod.splits*)
have *2*: *J* ∈ *set* (*subformulae F*) ⟹ *J* ∈ *snd* ‘ *set* (*tseytin-assmt F*) **for** *J* **using** *image-iff tseytin-assmt-has* **by** *fastforce*
from *1*[*OF distinct-snd-tseytin-assmt 2*, *OF assms*] **show** *?thesis* **.**
qed

lemma *tseytin-tran-small-clauses*: ∀ *C* ∈ *cnf* (*nnf* (*tseytin-tran F*)). *card C* ≤ *3*
proof −
have *3*: *card S* ≤ *2* ⟹ *card* (*a* ▷ *S*) ≤ *3* **for** *a S*
by(*cases finite S*; *simp add*: *card-insert-le-m1*)
have *2*: *card S* ≤ *1* ⟹ *card* (*a* ▷ *S*) ≤ *2* **for** *a S*
by(*cases finite S*; *simp add*: *card-insert-le-m1*)
have *1*: *card S* ≤ *0* ⟹ *card* (*a* ▷ *S*) ≤ *1* **for** *a S*
by(*cases finite S*; *simp add*: *card-insert-le-m1*)
have ∗: ⟦*G* ∈ *set* (*tseytin-tran1* (*Atom* ∘ *S*) *F*); *C* ∈ *cnf* (*nnf G*)⟧ ⟹ *card C* ≤ *3* **for** *G C S*
by(*induction F arbitrary*: *G*; *simp add*: *biimp-def*; (*elim disjE exE conjE* | *intro 1 2 3* | *simp*)+)
show *?thesis*

unfolding *tseytin-tran-def tseytin-toatom-def Let-def*
by(*clarsimp simp add*: *cnf-BigAnd nnf-BigAnd comp-assoc* $*$)
qed

lemma *tseytin-tran-few-clauses*: *card* (*cnf* (*nnf* (*tseytin-tran F*))) $\le$ *3* $*$ *size F* + *1*
proof −
have *size Bot* = *1* **by** *simp*
have *ws*: {*c* ▷ *D* |*D*. *D* = {*c1*} ∨ *D* = {*c2*}} = {{*c*,*c1*},{*c*,*c2*}} **for** *c1 c2 c* **by** *auto*
have *grr*: *Suc* (*card S*) $\le$ *c* ⟹ *card* (*a* ▷ *S*) $\le$ *c* **for** *a S c*
by(*cases finite S*; *simp add*: *card-insert-le-m1*)
have $*$: *card* (⋃ *a*∈*set* (*tseytin-tran1* (*Atom* ∘ *S*) *F*). *cnf* (*nnf a*)) $\le$ *3* $*$ *size F* **for** *S*
by(*induction F*; *simp add*: *biimp-def*; ((*intro grr card-Un-le*[*THEN le-trans*] | *simp add*: *ws*)+)?)
show *?thesis*
unfolding *tseytin-tran-def tseytin-toatom-def Let-def*
by(*clarsimp simp*: *nnf-BigAnd cnf-BigAnd*; *intro grr*; *simp add*: *comp-assoc* $*$)
qed

lemma *tseytin-tran-new-atom-count*: *card* (*atoms* (*tseytin-tran F*)) $\le$ *size F* + *card* (*atoms F*)
proof −
have *tseytin-tran1-atoms*: *H* ∈ *set* (*tseytin-tran1* (*tseytin-toatom F*) *G*) ⟹ *G* ∈ *set* (*subformulae F*) ⟹
atoms H ⊆ *atoms F* ∪ (⋃ *I* ∈ *set* (*subformulae F*). *atoms* (*tseytin-toatom F I*)) **for** *G H*
proof(*induction G arbitrary*: *H*)
case (*Atom k*)
hence *k* ∈ *atoms F*
by *simp* (*meson formula.set-intros*(*1*) *rev-subsetD subformula-atoms*)
with *Atom* **show** *?case* **by** *simp blast*
next
case *Bot* **then show** *?case* **by** *simp blast*
next
case (*Not G*)
show *?case* **by**(*insert Not.prems*(*1*,*2*);
frule subformulas-in-subformulas; *simp*; *elim disjE*; (*elim Not.IH* | *force*))
next
case (*And G1 G2*)
show *?case* **by**(*insert And.prems*(*1*,*2*);
frule subformulas-in-subformulas; *simp*; *elim disjE*; (*elim And.IH*; *simp* | *force*))
next
case (*Or G1 G2*)
show *?case* **by**(*insert Or.prems*(*1*,*2*);
frule subformulas-in-subformulas; *simp*; *elim disjE*; (*elim Or.IH*; *simp* | *force*))

next
case (*Imp G1 G2*)
show *?case* **by**(*insert Imp.prems*(*1*,*2*);
frule subformulas-in-subformulas; *simp*; *elim disjE*; (*elim Imp.IH*; *simp* | *force*))
qed
have *tseytin-tran1-atoms*: ($\bigcup$ *G*∈*set* (*tseytin-tran1* (*tseytin-toatom F*) *F*). *atoms G*) ⊆
atoms F ∪ ($\bigcup$ *I*∈*set* (*subformulae F*). *atoms* (*tseytin-toatom F I*))
using *tseytin-tran1-atoms*[*OF - subformulae-self*] **by** *blast*
have *1*: *card* (*atoms* (*tseytin-tran F*)) ≤
card (*atoms* (*tseytin-toatom F F*) ∪ ($\bigcup$ *x*∈*set* (*tseytin-tran1* (*tseytin-toatom F*) *F*). *atoms x*))
unfolding *tseytin-tran-def* **by**(*simp add*: *Let-def tseytin-tran1-atoms*)
have *2*: *atoms* (*tseytin-toatom F F*) ∪ ($\bigcup$ *x*∈*set* (*tseytin-tran1* (*tseytin-toatom F*) *F*). *atoms x*) ⊆
(*atoms F* ∪ ($\bigcup$ *I*∈*set* (*subformulae F*). *atoms* (*tseytin-toatom F I*)))
using *tseytin-tran1-atoms* **by** *blast*
have *twofin*: *finite* (*atoms F* ∪ ($\bigcup$ *I*∈*set* (*subformulae F*). *atoms* (*tseytin-toatom F I*))) **by** *simp*
have *card-subformulae*: *card* (*set* (*subformulae F*)) ≤ *size F* **using** *length-subformulae* **by** (*metis card-length*)
have *card-singleton-union*: *finite S* ⟹ *card* ($\bigcup$ *x*∈*S*. {*f x*}) ≤ *card S* **for** *f S*
by(*induction S rule*: *finite-induct*; *simp add*: *card-insert-if*)
have *3*: *card* ($\bigcup$ *I*∈*set* (*subformulae F*). *atoms* (*tseytin-toatom F I*)) ≤ *size F*
unfolding *tseytin-toatom-def* **using** *le-trans*[*OF card-singleton-union card-subformulae*]
by *simp fast*
have *4*: *card* (*atoms* (*tseytin-tran F*)) ≤ *card* (*atoms F*) + *card* ($\bigcup$ *f*∈*set* (*subformulae F*). *atoms* (*tseytin-toatom F f*))
using *le-trans*[*OF 1 card-mono*[*OF twofin 2*]] *card-Un-le le-trans* **by** *blast*
show *?thesis* **using** *3 4* **by** *linarith*
qed

end

definition *freshnat S* ≡ *Suc* (*Max* (*0* ⊳ *S*))
primrec *nfresh-natcode* **where**
nfresh-natcode S 0 = [] |
nfresh-natcode S (*Suc n*) = (*let f* = *freshnat S in f* # *nfresh-natcode* (*f* ⊳ *S*) *n*)
interpretation *freshnats*: *freshstuff freshnat* **unfolding** *freshnat-def* **by** *standard* (*meson Max-ge Suc-n-not-le-n finite-insert insertCI*)

lemma [*code-unfold*]: *freshnats.nfresh* = *nfresh-natcode*
proof −
have *freshnats.nfresh S n* = *nfresh-natcode S n* **for** *S n* **by**(*induction n arbitrary*: *S*; *simp*)
thus *?thesis* **by** *auto*
qed
lemmas *freshnats-code*[*code-unfold*] = *freshnats.tseytin-tran-def freshnats.tseytin-toatom-def*

```
freshnats.tseytin-assmt-def freshnats.nfresh.simps

lemma freshnats.tseytin-tran (Atom 0 → (¬ (Atom 1))) = ⋀[
  Atom 2,
  Atom 2 ↔ Atom 3 → Atom 4,
  Atom 0 ↔ Atom 3,
  Atom 4 ↔ ¬ (Atom 5),
  Atom 1 ↔ Atom 5
] (is ?l = ?r)
proof −
  have cnf (nnf ?r) =
    {{Pos 2},
     {Neg 4, Pos 2}, {Pos 3, Pos 2}, {Neg 2, Neg 3, Pos 4},
     {Neg 3, Pos 0}, {Neg 0, Pos 3},
     {Pos 5, Pos 4}, {Neg 4, Neg 5},
     {Neg 5, Pos 1}, {Neg 1, Pos 5}} by eval
  have ?thesis by eval
  show ?thesis by code-simp
qed

end
theory Tseytin-Sema
imports Sema Tseytin
begin

lemma biimp-simp[simp]: 𝒜 ⊨ F ↔ G ⟷ (𝒜 ⊨ F ⟷ 𝒜 ⊨ G)
  unfolding biimp-def by auto

locale freshstuff-sema = freshstuff
begin

definition tseytin-update 𝒜 F ≡ (let U = map (apsnd (formula-semantics 𝒜))
(tseytin-assmt F) in foldl pair-fun-upd 𝒜 U)

lemma tseyting-update-keep-subformula-sema: G ∈ set (subformulae F) ⟹ tseytin-update
𝒜 F ⊨ G ⟷ 𝒜 ⊨ G
proof −
  assume G ∈ set (subformulae F)
  hence sub: atoms G ⊆ atoms F by(fact subformula-atoms)
  have natoms: k ∈ atoms F ⟹ k ∉ fst ‘ set (tseytin-assmt F) for k l
    using tseytin-assmt-new-atoms by force
  have k ∈ atoms F ⟹ tseytin-update 𝒜 F k = 𝒜 k for k
    unfolding tseytin-update-def Let-def
    by(force intro!: fold-pair-upd-triv dest!: natoms)
  thus ?thesis using relevant-atoms-same-semantics sub by (metis subsetCE)
qed

lemma (k,G) ∈ set (tseytin-assmt F) ⟹ tseytin-update 𝒜 F k ⟷ tseytin-update
𝒜 F ⊨ G
```

proof(*induction F arbitrary*: *G*)
case (*Atom x*)
then show *?case* **by**(*simp add*: *tseytin-update-def tseytin-assmt-def Let-def pair-fun-upd-def*)
next
case *Bot*
then show *?case* **by**(*simp add*: *tseytin-update-def tseytin-assmt-def Let-def pair-fun-upd-def*)
next
case (*Not F*)
then show *?case*
oops

lemma *tseytin-updates*: $(k,G) \in$ *set* (*tseytin-assmt F*) $\Longrightarrow$ *tseytin-update* $\mathcal{A}$ *F k* $\longleftrightarrow$ *tseytin-update* $\mathcal{A}$ *F* $\models$ *G*
apply(*subst tseytin-update-def*)
apply(*simp add*: *tseytin-assmt-def Let-def foldl-pair-fun-upd-map-of map-of-map-apsnd distinct-zipunzip*[*OF nfresh-distinct*[*OF atoms-finite*]])
apply(*subst tseyting-update-keep-subformula-sema*)
apply(*erule in-set-zipE*; *simp*; *fail*)
..

lemma *tseytin-tran1*: $G \in$ *set* (*subformulae F*) $\Longrightarrow$ $H \in$ *set* (*tseytin-tran1 S G*) $\Longrightarrow$ $\forall J \in$ *set* (*subformulae F*). *tseytin-update* $\mathcal{A}$ *F* $\models$ *J* $\longleftrightarrow$ *tseytin-update* $\mathcal{A}$ *F* $\models$ (*S J*) $\Longrightarrow$ *tseytin-update* $\mathcal{A}$ *F* $\models$ *H*
proof(*induction G arbitrary*: *H*)
case *Bot* **thus** *?case* **by** *auto*
next
case (*Atom k*) **thus** *?case* **by** *fastforce*
next
case (*Not G*)
consider $H = S\ (\neg\ G) \leftrightarrow \neg\ (S\ G)$ | $H \in$ *set* (*tseytin-tran1 S G*) **using** *Not.prems*(*2*) **by** *auto*
then show *?case* **proof** *cases*
case *1* **then show** *?thesis* **using** *Not.prems*(*3*)
by (*metis Not.prems*(*1*) *biimp-simp formula-semantics.simps*(*3*) *set-subset-Cons subformulae.simps*(*3*) *subformulae-self subsetCE subsubformulae*)
next
have *D*: $\neg G \in$ *set* (*subformulae F*) $\Longrightarrow$ $G \in$ *set* (*subformulae F*)
by(*elim subsubformulae*; *simp*)
case *2* **then show** *?thesis* **using** *D Not.IH Not.prems*(*1*,*3*) **by** *blast*
qed
next
case (*And G1 G2*)
have *el*: $G1 \in$ *set* (*subformulae F*) $G2 \in$ *set* (*subformulae F*) **using** *subsubformulae And.prems*(*1*) **by** *fastforce+*
with *And.IH And.prems*(*3*) **have** *IH*: $H \in$ *set* (*tseytin-tran1 S G1*) $\Longrightarrow$ *tseytin-update* $\mathcal{A}$ *F* $\models$ *H*
$H \in$ *set* (*tseytin-tran1 S G2*) $\Longrightarrow$ *tseytin-update* $\mathcal{A}$ *F* $\models$ *H* **for** *H*
by *blast+*

show *?case* **using** *And.prems IH el* **by**(*simp*; *elim disjE*; *simp*; *insert And.prems*(*1*) *formula-semantics.simps*(*4*), *blast*)
next
case (*Or G1 G2*)
have *el*: *G1* ∈ *set* (*subformulae F*) *G2* ∈ *set* (*subformulae F*) **using** *subsubformulae Or.prems*(*1*) **by** *fastforce*+
with *Or.IH Or.prems*(*3*) **have** *IH*: *H* ∈ *set* (*tseytin-tran1 S G1*) ⟹ *tseytin-update* 𝒜 *F* ⊨ *H*
H ∈ *set* (*tseytin-tran1 S G2*) ⟹ *tseytin-update* 𝒜 *F* ⊨ *H* **for** *H*
by *blast*+
show *?case* **using** *Or.prems*(*3*,*2*) *IH el* **by**(*simp*; *elim disjE*; *simp*; *metis Or.prems*(*1*) *formula-semantics.simps*(*5*))
next
case (*Imp G1 G2*)
have *el*: *G1* ∈ *set* (*subformulae F*) *G2* ∈ *set* (*subformulae F*) **using** *subsubformulae Imp.prems*(*1*) **by** *fastforce*+
with *Imp.IH Imp.prems*(*3*) **have** *IH*: *H* ∈ *set* (*tseytin-tran1 S G1*) ⟹ *tseytin-update* 𝒜 *F* ⊨ *H*
H ∈ *set* (*tseytin-tran1 S G2*) ⟹ *tseytin-update* 𝒜 *F* ⊨ *H* **for** *H*
by *blast*+
show *?case* **using** *Imp.prems*(*3*,*2*) *IH el* **by**(*simp*; *elim disjE*; *simp*; *metis Imp.prems*(*1*) *formula-semantics.simps*(*6*))
qed

lemma *all-tran-formulas-validated*: ∀ *J* ∈ *set* (*subformulae F*). *tseytin-update* 𝒜 *F* ⊨ *J* ⟷ *tseytin-update* 𝒜 *F* ⊨ (*tseytin-toatom F J*)
apply(*simp add*: *tseytin-toatom-def*)
apply(*intro ballI*)
apply(*subst tseytin-updates*)
apply(*erule tseytin-assmt-backlookup*)
..

lemma *tseytin-tran-equisat*: 𝒜 ⊨ *F* ⟷ *tseytin-update* 𝒜 *F* ⊨ (*tseytin-tran F*)
using *all-tran-formulas-validated tseytin-tran1 all-tran-formulas-validated tseyting-update-keep-subformula-sema* **by**(*simp add*: *tseytin-tran-def Let-def*) *blast*

lemma *tseytin-tran1-orig-connection*: *G* ∈ *set* (*subformulae F*) ⟹ (∀ *H*∈*set* (*tseytin-tran1* (*tseytin-toatom F*) *G*). 𝒜 ⊨ *H*) ⟹
𝒜 ⊨ *G* ⟷ 𝒜 ⊨ *tseytin-toatom F G*
by(*induction G*; *simp*; *drule subformulas-in-subformulas*; *simp*)

lemma *tseytin-untran*: 𝒜 ⊨ (*tseytin-tran F*) ⟹ 𝒜 ⊨ *F*
proof −
have *1*: ⟦𝒜 ⊨ *tseytin-toatom F F*; 𝒜 ⊨ *F*⟧ ⟹ *tseytin-update* 𝒜 *F* ⊨ *tseytin-toatom F F*
using *all-tran-formulas-validated tseyting-update-keep-subformula-sema* **by** *blast*

let *?C* = $\lambda\mathcal{A}$. ($\forall$ *H* $\in$ *set* (*tseytin-tran1* (*tseytin-toatom F*) *F*). $\mathcal{A} \models H$)
have *2*: *?C* $\mathcal{A} \Longrightarrow$ *?C* (*tseytin-update* $\mathcal{A}$ *F*)
using *all-tran-formulas-validated tseytin-tran1* **by** *blast*
assume $\mathcal{A} \models$ *tseytin-tran F*
hence *tseytin-update* $\mathcal{A}$ *F* $\models$ *tseytin-tran F*
unfolding *tseytin-tran-def*
apply(*simp add*: *Let-def*)
apply(*intro conjI*)
apply(*elim conjE*)
apply(*drule tseytin-tran1-orig-connection*[*OF subformulae-self*])
apply(*clarsimp simp add*: *tseytin-assmt-distinct foldl-pair-fun-upd-map-of 1 2*)+
done
thus *?thesis* **using** *tseytin-tran-equisat* **by** *blast*
qed
lemma *tseytin-tran-equiunsatisfiable*: $\models \neg F \longleftrightarrow \models \neg$ (*tseytin-tran F*) (**is** *?l* $\longleftrightarrow$ *?r*)
proof(*rule iffI*; *erule contrapos-pp*)
assume $\neg$ *?l*
then obtain $\mathcal{A}$ **where** $\mathcal{A} \models F$ **by** *auto*
hence *tseytin-update* $\mathcal{A}$ *F* $\models$ (*tseytin-tran F*) **using** *tseytin-tran-equisat* **by** *simp*
thus $\neg$ *?r* **by** *simp blast*
next
assume $\neg$ *?r*
then obtain $\mathcal{A}$ **where** $\mathcal{A} \models$ *tseytin-tran F* **by** *auto*
thus ~ *?l* **using** *tseytin-untran* **by** *simp blast*
qed

end

interpretation *freshsemanats*: *freshstuff-sema freshnat*
by (*simp add*: *freshnats.freshstuff-axioms freshstuff-sema-def*)
print-theorems

end

1.5 Implication-only formulas

theory *MiniFormulas*
imports *Formulas*
begin

fun *is-mini-formula* **where**
is-mini-formula (*Atom* -) = *True* |
is-mini-formula $\bot$ = *True* |
is-mini-formula (*Imp F G*) = (*is-mini-formula F* $\wedge$ *is-mini-formula G*) |
is-mini-formula - = *False*

The similarity between these "mini" formulas and Johansson's minimal calculus of implications [8] is mostly in name. Johansson does replace $\neg F$ by $F \to \bot$ in one place, but generally keeps it. The main focus of [8] is on removing rules from Calculi anyway, not on removing connectives. We are only borrowing the name.

primrec *to-mini-formula* **where**
to-mini-formula (*Atom k*) = *Atom k* |
to-mini-formula ⊥ = ⊥ |
to-mini-formula (*Imp F G*) = *to-mini-formula F* → *to-mini-formula G* |
to-mini-formula (*Not F*) = *to-mini-formula F* → ⊥ |
to-mini-formula (*And F G*) = (*to-mini-formula F* → (*to-mini-formula G* → ⊥))
→ ⊥ |
to-mini-formula (*Or F G*) = (*to-mini-formula F* → ⊥) → *to-mini-formula G*

lemma *to-mini-is-mini*[*simp*,*intro*!]: *is-mini-formula* (*to-mini-formula F*)
by(*induction F*; *simp*)
lemma *mini-to-mini*: *is-mini-formula F* ⟹ *to-mini-formula F* = *F*
by(*induction F*; *simp*)
corollary *mini-mini*[*simp*]: *to-mini-formula* (*to-mini-formula F*) = *to-mini-formula F*
using *mini-to-mini*[*OF to-mini-is-mini*] **.**

We could have used an arbitrary other combination, e.g. *Atom*, ¬, and (∧). The choice for *Atom*, ⊥, and (→) was made because it is (to the best of my knowledge) the only combination that only requires three elements and verifies:

lemma *mini-formula-atoms*: *atoms* (*to-mini-formula F*) = *atoms F*
by(*induction F*; *simp*)

(The story would be different if we had different junctors, e.g. if we allowed a NAND.)

end
theory *MiniFormulas-Sema*
imports *MiniFormulas Sema*
begin

lemma *A* ⊨ *F* ⟷ *A* ⊨ *to-mini-formula F*
by(*induction F*) *auto*

end

1.6 Consistency

We follow the proofs by Melvin Fitting [2].

theory *Consistency*
imports *Sema*
begin

```
definition Hintikka S ≡ (
  ⊥ ∉ S
∧ (∀k. Atom k ∈ S ⟶ ¬ (Atom k) ∈ S ⟶ False)
∧ (∀F G. F ∧ G ∈ S ⟶ F ∈ S ∧ G ∈ S)
∧ (∀F G. F ∨ G ∈ S ⟶ F ∈ S ∨ G ∈ S)
∧ (∀F G. F → G ∈ S ⟶ ¬F ∈ S ∨ G ∈ S)
∧ (∀F. ¬ (¬F) ∈ S ⟶ F ∈ S)
∧ (∀F G. ¬(F ∧ G) ∈ S ⟶ ¬ F ∈ S ∨ ¬ G ∈ S)
∧ (∀F G. ¬(F ∨ G) ∈ S ⟶ ¬ F ∈ S ∧ ¬ G ∈ S)
∧ (∀F G. ¬(F → G) ∈ S ⟶ F ∈ S ∧ ¬ G ∈ S)
)

lemma Hintikka {Atom 0 ∧ ((¬ (Atom 1)) → Atom 2), ((¬ (Atom 1)) → Atom
2), Atom 0, ¬(¬ (Atom 1)), Atom 1}
  unfolding Hintikka-def by simp

theorem Hintikkas-lemma:
  assumes H: Hintikka S
  shows sat S
proof −
  from H[unfolded Hintikka-def]
  have H': ⊥ ∉ S
    Atom k ∈ S ⟹ ¬ (Atom k) ∈ S ⟹ False
    F ∧ G ∈ S ⟹ F ∈ S ∧ G ∈ S
    F ∨ G ∈ S ⟹ F ∈ S ∨ G ∈ S
    F → G ∈ S ⟹ ¬F ∈ S ∨ G ∈ S
    ¬ (¬ F) ∈ S ⟹ F ∈ S
    ¬ (F ∧ G) ∈ S ⟹ ¬ F ∈ S ∨ ¬ G ∈ S
    ¬ (F ∨ G) ∈ S ⟹ ¬ F ∈ S ∧ ¬ G ∈ S
    ¬ (F → G) ∈ S ⟹ F ∈ S ∧ ¬ G ∈ S
    for k F G by blast+
  let ?M = λk. Atom k ∈ S
  have (F ∈ S ⟶ (?M ⊨ F)) ∧ (¬ F ∈ S ⟶ (¬(?M ⊨ F))) for F
    by(induction F) (auto simp: H'(1) dest!: H'(2−))
  thus ?thesis unfolding sat-def by blast
qed

definition pcp C ≡ (∀S ∈ C.
  ⊥ ∉ S
∧ (∀k. Atom k ∈ S ⟶ ¬ (Atom k) ∈ S ⟶ False)
∧ (∀F G. F ∧ G ∈ S ⟶ F ▷ G ▷ S ∈ C)
∧ (∀F G. F ∨ G ∈ S ⟶ F ▷ S ∈ C ∨ G ▷ S ∈ C)
∧ (∀F G. F → G ∈ S ⟶ ¬F ▷ S ∈ C ∨ G ▷ S ∈ C)
∧ (∀F. ¬ (¬F) ∈ S ⟶ F ▷ S ∈ C)
∧ (∀F G. ¬(F ∧ G) ∈ S ⟶ ¬ F ▷ S ∈ C ∨ ¬ G ▷ S ∈ C)
∧ (∀F G. ¬(F ∨ G) ∈ S ⟶ ¬ F ▷ ¬ G ▷ S ∈ C)
∧ (∀F G. ¬(F → G) ∈ S ⟶ F ▷ ¬ G ▷ S ∈ C)
)
```

lemma *pcp* {} *pcp* {{}} *pcp* {{*Atom 0*}} **by** (*simp add*: *pcp-def*)+
lemma *pcp* {{(¬ (*Atom 1*)) → *Atom 2*},
{((¬ (*Atom 1*)) → *Atom 2*), ¬(¬ (*Atom 1*))},
{((¬ (*Atom 1*)) → *Atom 2*), ¬(¬ (*Atom 1*)), *Atom 1*}} **by** (*auto simp add*: *pcp-def*)

Fitting uses uniform notation [10] for the definition of *pcp*. We try to mimic this, more to see whether it works than because it is ultimately necessary.

inductive *Con* :: *'a formula => 'a formula => 'a formula => bool* **where**
Con (*And F G*) *F G* |
Con (*Not* (*Or F G*)) (*Not F*) (*Not G*) |
Con (*Not* (*Imp F G*)) *F* (*Not G*) |
Con (*Not* (*Not F*)) *F F*

inductive *Dis* :: *'a formula => 'a formula => 'a formula => bool* **where**
Dis (*Or F G*) *F G* |
Dis (*Imp F G*) (*Not F*) *G* |
Dis (*Not* (*And F G*)) (*Not F*) (*Not G*) |
Dis (*Not* (*Not F*)) *F F*

lemma *Con* (*Not* (*Not F*)) *F F Dis* (*Not* (*Not F*)) *F F* **by**(*intro Con.intros Dis.intros*)+

lemma *con-dis-simps*:
Con a1 a2 a3 = (*a1* = *a2* ∧ *a3* ∨ (∃ *F G*. *a1* = ¬ (*F* ∨ *G*) ∧ *a2* = ¬ *F* ∧ *a3* = ¬ *G*) ∨ (∃ *G*. *a1* = ¬ (*a2* → *G*) ∧ *a3* = ¬ *G*) ∨ *a1* = ¬ (¬ *a2*) ∧ *a3* = *a2*)
Dis a1 a2 a3 = (*a1* = *a2* ∨ *a3* ∨ (∃ *F G*. *a1* = *F* → *G* ∧ *a2* = ¬ *F* ∧ *a3* = *G*) ∨ (∃ *F G*. *a1* = ¬ (*F* ∧ *G*) ∧ *a2* = ¬ *F* ∧ *a3* = ¬ *G*) ∨ *a1* = ¬ (¬ *a2*) ∧ *a3* = *a2*)
by(*simp-all add*: *Con.simps Dis.simps*)

lemma *Hintikka-alt*: *Hintikka S* = (
⊥ ∉ *S*
∧ (∀ *k*. *Atom k* ∈ *S* ⟶ ¬ (*Atom k*) ∈ *S* ⟶ *False*)
∧ (∀ *F G H*. *Con F G H* ⟶ *F* ∈ *S* ⟶ *G* ∈ *S* ∧ *H* ∈ *S*)
∧ (∀ *F G H*. *Dis F G H* ⟶ *F* ∈ *S* ⟶ *G* ∈ *S* ∨ *H* ∈ *S*)
)
apply(*simp add*: *Hintikka-def con-dis-simps*)
apply(*rule iffI*)

subgoal by *blast*
subgoal by *safe metis*+
done

lemma *pcp-alt*: *pcp C* = (∀ *S* ∈ *C*.
⊥ ∉ *S*

$\wedge$ ($\forall$ *k*. *Atom k* $\in$ *S* $\longrightarrow$ $\neg$ (*Atom k*) $\in$ *S* $\longrightarrow$ *False*)
$\wedge$ ($\forall$ *F G H*. *Con F G H* $\longrightarrow$ *F* $\in$ *S* $\longrightarrow$ *G* $\triangleright$ *H* $\triangleright$ *S* $\in$ *C*)
$\wedge$ ($\forall$ *F G H*. *Dis F G H* $\longrightarrow$ *F* $\in$ *S* $\longrightarrow$ *G* $\triangleright$ *S* $\in$ *C* $\vee$ *H* $\triangleright$ *S* $\in$ *C*)
)
  apply(*simp add*: *pcp-def con-dis-simps*)
  apply(*rule iffI*; *unfold Ball-def*; *elim all-forward*)
  by (*auto simp add*: *insert-absorb split*: *formula.splits*)

definition *subset-closed C* $\equiv$ ($\forall$ *S* $\in$ *C*. $\forall$ *s*$\subseteq$*S*. *s* $\in$ *C*)
definition *finite-character C* $\equiv$ ($\forall$ *S*. *S* $\in$ *C* $\longleftrightarrow$ ($\forall$ *s* $\subseteq$ *S*. *finite s* $\longrightarrow$ *s* $\in$ *C*))

lemma *ex1*: *pcp C* $\Longrightarrow$ $\exists$ *C'*. *C* $\subseteq$ *C'* $\wedge$ *pcp C'* $\wedge$ *subset-closed C'*
proof(*intro exI*[*of* - {*s* . $\exists$ *S* $\in$ *C*. *s* $\subseteq$ *S*}] *conjI*)
  let *?E* = {*s*. $\exists$ *S*$\in$*C*. *s* $\subseteq$ *S*}
  show *C* $\subseteq$ *?E* **by** *blast*
  show *subset-closed ?E* **unfolding** *subset-closed-def* **by** *blast*
  assume *C*: ‹*pcp C*›
  show *pcp ?E* **using** *C* **unfolding** *pcp-alt*
    by (*intro ballI conjI*; *simp*; *meson insertI1 rev-subsetD subset-insertI subset-insertI2*)
qed

lemma *sallI*: ($\bigwedge$*s*. *s* $\subseteq$ *S* $\Longrightarrow$ *P s*) $\Longrightarrow$ $\forall$ *s* $\subseteq$ *S*. *P s* **by** *simp*

lemma *ex2*:
  assumes *fc*: *finite-character C*
  shows *subset-closed C*
  unfolding *subset-closed-def*
proof (*intro ballI sallI*)
  fix *s S*
  assume *e*: ‹*S* $\in$ *C*› **and** *s*: ‹*s* $\subseteq$ *S*›
  hence $*$: *t* $\subseteq$ *s* $\Longrightarrow$ *t* $\subseteq$ *S* **for** *t* **by** *simp*
  from *fc* **have** *t* $\subseteq$ *S* $\Longrightarrow$ *finite t* $\Longrightarrow$ *t* $\in$ *C* **for** *t* **unfolding** *finite-character-def* **using** *e* **by** *blast*
  hence *t* $\subseteq$ *s* $\Longrightarrow$ *finite t* $\Longrightarrow$ *t* $\in$ *C* **for** *t* **using** $*$ **by** *simp*
  with *fc* **show** ‹*s* $\in$ *C*› **unfolding** *finite-character-def* **by** *blast*
qed

lemma
  assumes *C*: *pcp C*
  assumes *S*: *subset-closed C*
  shows *ex3*: $\exists$ *C'*. *C* $\subseteq$ *C'* $\wedge$ *pcp C'* $\wedge$ *finite-character C'*
proof(*intro exI*[*of* - *C* $\cup$ {*S*. $\forall$ *s* $\subseteq$ *S*. *finite s* $\longrightarrow$ *s* $\in$ *C*}] *conjI*)
  let *?E* = {*S*. $\forall$ *s* $\subseteq$ *S*. *finite s* $\longrightarrow$ *s* $\in$ *C*}
  show *C* $\subseteq$ *C* $\cup$ *?E* **by** *blast*
  from *S* **show** *finite-character* (*C* $\cup$ *?E*) **unfolding** *finite-character-def subset-closed-def* **by** *blast*
  note *C''* = *C*[*unfolded pcp-alt*, *THEN bspec*]

have *CON*: *G* ▹ *H* ▹ *S* ∈ *C* ∪ {*S*. ∀ *s*⊆*S*. *finite s* ⟶ *s* ∈ *C*} **if** *si*: ⋀*s*. ⟦*s*⊆*S*; *finite s*⟧ ⟹ *s* ∈ *C* **and**
un: *Con F G H* **and** *el*: *F* ∈ *S* **for** *F G H S* **proof** −
have *k*: ∀ *s* ⊆ *S*. *finite s* ⟶ *F* ∈ *s* ⟶ *G* ▹ *H* ▹ *s* ∈ *C*
using *si un C''* **by** *simp*
have *G* ▹ *H* ▹ *S* ∈ *?E*
unfolding *mem-Collect-eq Un-iff* **proof** *safe*
fix *s*
assume *s* ⊆ *G* ▹ *H* ▹ *S* **and** *f*: *finite s*
hence *F* ▹ (*s* − {*G*,*H*}) ⊆ *S* **using** *el* **by** *blast*
with *k f* **have** *G* ▹ *H* ▹ *F* ▹ (*s* − {*G*,*H*}) ∈ *C* **by** *simp*
hence *F* ▹ *G* ▹ *H* ▹ *s* ∈ *C* **using** *insert-absorb* **by** *fastforce*
thus *s* ∈ *C* **using** *S* **unfolding** *subset-closed-def* **by** *fast*
qed
thus *G* ▹ *H* ▹ *S* ∈ *C* ∪ *?E* **by** *simp*
qed
have *DIS*: *G* ▹ *S* ∈ *C* ∪ {*S*. ∀ *s*⊆*S*. *finite s* ⟶ *s* ∈ *C*} ∨ *H* ▹ *S* ∈ *C* ∪ {*S*. ∀ *s*⊆*S*. *finite s* ⟶ *s* ∈ *C*}
if *si*: ⋀*s*. *s*⊆*S* ⟹ *finite s* ⟹ *s* ∈ *C* **and** *un*: *Dis F G H* **and** *el*: *F* ∈ *S*
for *F G H S* **proof** −
have *l*: ∃ *I*∈{*G*, *H*}. *I* ▹ *s1* ∈ *C* ∧ *I* ▹ *s2* ∈ *C*
if *s1* ⊆ *S finite s1 F* ∈ *s1*
s2 ⊆ *S finite s2 F* ∈ *s2* **for** *s1 s2*
proof −
let *?s* = *s1* ∪ *s2*
have *?s* ⊆ *S finite ?s* **using** *that* **by** *simp-all*
with *si* **have** *?s* ∈ *C* **by** *simp*
moreover have *F* ∈ *?s* **using** *that* **by** *simp*
ultimately have ∃ *I*∈{*G*,*H*}. *I* ▹ *?s* ∈ *C*
using *C'' un* **by** *simp*
thus ∃ *I*∈{*G*,*H*}. *I* ▹ *s1* ∈ *C* ∧ *I* ▹ *s2* ∈ *C*
by (*meson S*[*unfolded subset-closed-def*, *THEN bspec*] *insert-mono sup.cobounded2 sup-ge1*)
qed
have *m*: ⟦*s1* ⊆ *S*; *finite s1*; *F* ∈ *s1*; *G* ▹ *s1* ∉ *C*; *s2* ⊆ *S*; *finite s2*; *F* ∈ *s2*; *H* ▹ *s2* ∉ *C*⟧ ⟹ *False* **for** *s1 s2*
using *l* **by** *blast*
have *False* **if** *s1* ⊆ *S finite s1 G* ▹ *s1* ∉ *C s2* ⊆ *S finite s2 H* ▹ *s2* ∉ *C* **for** *s1 s2*
proof −
have ∗: *F* ▹ *s1* ⊆ *S finite* (*F* ▹ *s1*) *F* ∈ *F* ▹ *s1* **if** *s1* ⊆ *S finite s1* **for** *s1*
using *that el* **by** *simp-all*
have *G* ▹ *F* ▹ *s1* ∉ *C H* ▹ *F* ▹ *s2* ∉ *C*
by (*meson S insert-mono subset-closed-def subset-insertI that*(*3*,*6*))+
from *m*[*OF* ∗[*OF that*(*1*−*2*)] *this*(*1*) ∗[*OF that*(*4*−*5*)] *this*(*2*)]
show *False* **.**
qed
hence *G* ▹ *S* ∈ *?E* ∨ *H* ▹ *S* ∈ *?E*

unfolding *mem-Collect-eq Un-iff*
by (*metis* (*no-types*, *lifting*) *finite-Diff insert-Diff si subset-insert-iff*)
thus $G \triangleright S \in C \cup ?E \vee H \triangleright S \in C \cup ?E$ **by** *blast*
qed

have *CON′*: $\bigwedge f2\ g2\ h2\ F2\ G2\ S2.\ [\![\bigwedge s.\ [\![s \in C;\ h2\ F2\ G2 \in s]\!] \Longrightarrow f2\ F2 \triangleright s \in C \vee g2\ G2 \triangleright s \in C;$
$\forall s \subseteq S2.\ finite\ s \longrightarrow s \in C;\ h2\ F2\ G2 \in S2;\ False]\!]$
$\Longrightarrow f2\ F2 \triangleright S2 \in C \cup \{S.\ \forall s \subseteq S.\ finite\ s \longrightarrow s \in C\} \vee g2\ G2 \triangleright S2 \in C \cup \{S.\ \forall s \subseteq S.\ finite\ s \longrightarrow s \in C\}$
by *fast*

show *pcp* ($C \cup ?E$) **unfolding** *pcp-alt*
apply(*intro ballI conjI*; *elim UnE*; (*unfold mem-Collect-eq*)?)
subgoal using *C″* **by** *blast*
subgoal using *C″* **by** *blast*
subgoal using *C″* **by** (*simp*;*fail*)
subgoal by (*meson C″ empty-subsetI finite.emptyI finite-insert insert-subset subset-insertI*)
subgoal using *C″* **by** *simp*
subgoal using *CON* **by** *simp*
subgoal using *C″* **by** *blast*
subgoal using *DIS* **by** *simp*
done
qed

primrec *pcp-seq* **where**
pcp-seq C S 0 = S |
pcp-seq C S (*Suc n*) = (
let Sn = pcp-seq C S n; *Sn1 = from-nat n* ▷ *Sn in*
if Sn1 ∈ *C then Sn1 else Sn*
)

lemma *pcp-seq-in*: $pcp\ C \Longrightarrow S \in C \Longrightarrow pcp\text{-}seq\ C\ S\ n \in C$
proof(*induction n*)
case (*Suc n*)
hence *pcp-seq C S n* ∈ *C* **by** *simp*
thus *?case* **by**(*simp add*: *Let-def*)
qed *simp*

lemma *pcp-seq-mono*: $n \le m \Longrightarrow pcp\text{-}seq\ C\ S\ n \subseteq pcp\text{-}seq\ C\ S\ m$
proof(*induction m*)
case (*Suc m*)
thus *?case* **by**(*cases n = Suc m*; *simp add*: *Let-def*; *blast*)
qed *simp*

lemma *pcp-seq-UN*: $\bigcup\{pcp\text{-}seq\ C\ S\ n | n.\ n \le m\} = pcp\text{-}seq\ C\ S\ m$

proof(*induction m*)
  case (*Suc m*)
  have $\{f\ n\ |n.\ n \leq Suc\ m\} = f\ (Suc\ m) \triangleright \{f\ n\ |n.\ n \leq m\}$ **for** *f* **using** *le-Suc-eq* **by** *auto*
  hence $\{pcp\text{-}seq\ C\ S\ n\ |n.\ n \leq Suc\ m\} = pcp\text{-}seq\ C\ S\ (Suc\ m) \triangleright \{pcp\text{-}seq\ C\ S\ n\ |n.\ n \leq m\}$ **.**
  hence $\bigcup\{pcp\text{-}seq\ C\ S\ n\ |n.\ n \leq Suc\ m\} = \bigcup\{pcp\text{-}seq\ C\ S\ n\ |n.\ n \leq m\} \cup pcp\text{-}seq\ C\ S\ (Suc\ m)$ **by** *auto*
  thus *?case* **using** *Suc pcp-seq-mono* **by** *blast*
qed *simp*

lemma *wont-get-added*: $(F :: ('a :: countable)\ formula) \notin pcp\text{-}seq\ C\ S\ (Suc\ (to\text{-}nat\ F)) \Longrightarrow F \notin pcp\text{-}seq\ C\ S\ (Suc\ (to\text{-}nat\ F) + n)$

We don't necessarily have $n = to\text{-}nat\ (from\text{-}nat\ n)$, so this doesn't hold.

oops

definition $pcp\text{-}lim\ C\ S \equiv \bigcup\{pcp\text{-}seq\ C\ S\ n|n.\ True\}$

lemma *pcp-seq-sub*: $pcp\text{-}seq\ C\ S\ n \subseteq pcp\text{-}lim\ C\ S$
  unfolding *pcp-lim-def* **by**(*induction n*; *blast*)

lemma *pcp-lim-inserted-at-ex*: $x \in pcp\text{-}lim\ C\ S \Longrightarrow \exists k.\ x \in pcp\text{-}seq\ C\ S\ k$
  unfolding *pcp-lim-def* **by** *blast*

lemma *pcp-lim-in*:
  assumes *c*: *pcp C*
  and *el*: $S \in C$
  and *sc*: *subset-closed C*
  and *fc*: *finite-character C*
  shows $pcp\text{-}lim\ C\ S \in C$ (**is** $?cl \in C$)
proof −
  from *pcp-seq-in*[*OF c el*, *THEN allI*] **have** $\forall n.\ pcp\text{-}seq\ C\ S\ n \in C$ **.**
  hence $\forall m.\ \bigcup\{pcp\text{-}seq\ C\ S\ n|n.\ n \leq m\} \in C$ **unfolding** *pcp-seq-UN* **.**

  have $\forall s \subseteq ?cl.\ finite\ s \longrightarrow s \in C$
  proof *safe*
    fix *s* :: *'a formula set*
    have $pcp\text{-}seq\ C\ S\ (Suc\ (Max\ (to\text{-}nat\ `\ s))) \subseteq pcp\text{-}lim\ C\ S$ **using** *pcp-seq-sub* **by** *blast*
    assume ‹*finite s*› ‹$s \subseteq pcp\text{-}lim\ C\ S$›
    hence $\exists k.\ s \subseteq pcp\text{-}seq\ C\ S\ k$
    proof(*induction s rule*: *finite-induct*)
      case (*insert x s*)
      then obtain *k1* **where** $s \subseteq pcp\text{-}seq\ C\ S\ k1$ **by** *blast*
      moreover obtain *k2* **where** $x \in pcp\text{-}seq\ C\ S\ k2$
        by (*meson pcp-lim-inserted-at-ex insert.prems insert-subset*)
      ultimately have $x \triangleright s \subseteq pcp\text{-}seq\ C\ S\ (max\ k1\ k2)$
          by (*meson pcp-seq-mono dual-order.trans insert-subset max.bounded-iff*

order-refl subsetCE)
      thus *?case* **by** *blast*
    qed *simp*
    with *pcp-seq-in*[*OF c el*] *sc*
    show $s \in C$ **unfolding** *subset-closed-def* **by** *blast*
  qed
  thus $?cl \in C$ **using** *fc* **unfolding** *finite-character-def* **by** *blast*
qed

lemma *cl-max*:
  assumes *c*: *pcp C*
  assumes *sc*: *subset-closed C*
  assumes *el*: $K \in C$
  assumes *su*: *pcp-lim* $C\ S \subseteq K$
  shows *pcp-lim* $C\ S = K$ (**is** *?e*)
proof (*rule ccontr*)
  assume ‹¬*?e*›
  with *su* **have** *pcp-lim* $C\ S \subset K$ **by** *simp*
  then obtain F **where** *e*: $F \in K$ **and** *ne*: $F \notin$ *pcp-lim* $C\ S$ **by** *blast*
  from *ne* **have** $F \notin$ *pcp-seq* $C\ S$ (*Suc* (*to-nat* F)) **using** *pcp-seq-sub* **by** *fast*
  hence *1*: $F \triangleright$ *pcp-seq* $C\ S$ (*to-nat* F) $\notin C$ **by** (*simp add*: *Let-def split*: *if-splits*)
  have $F \triangleright$ *pcp-seq* $C\ S$ (*to-nat* F) $\subseteq K$ **using** *pcp-seq-sub e su* **by** *blast*
  hence $F \triangleright$ *pcp-seq* $C\ S$ (*to-nat* F) $\in C$ **using** *sc* **unfolding** *subset-closed-def*
using *el* **by** *blast*
  with *1* **show** *False* **..**
qed

lemma *cl-max'*:
  assumes *c*: *pcp C*
  assumes *sc*: *subset-closed C*
  shows $F \triangleright$ *pcp-lim* $C\ S \in C \Longrightarrow F \in$ *pcp-lim* $C\ S$
    $F \triangleright G \triangleright$ *pcp-lim* $C\ S \in C \Longrightarrow F \in$ *pcp-lim* $C\ S \wedge G \in$ *pcp-lim* $C\ S$
using *cl-max*[*OF assms*] **by** *blast+*

lemma *pcp-lim-Hintikka*:
  assumes *c*: *pcp C*
  assumes *sc*: *subset-closed C*
  assumes *fc*: *finite-character C*
  assumes *el*: $S \in C$
  shows *Hintikka* (*pcp-lim* $C\ S$)
proof −
  let *?cl* = *pcp-lim* $C\ S$
  have $?cl \in C$ **using** *pcp-lim-in*[*OF c el sc fc*] **.**
  from *c*[*unfolded pcp-alt*, *THEN bspec*, *OF this*]
  have *d*: $\bot \notin ?cl$
    Atom k $\in ?cl \Longrightarrow \neg$ (*Atom k*) $\in ?cl \Longrightarrow$ *False*
    Con F G H $\Longrightarrow F \in ?cl \Longrightarrow G \triangleright H \triangleright ?cl \in C$
    Dis F G H $\Longrightarrow F \in ?cl \Longrightarrow G \triangleright ?cl \in C \vee H \triangleright ?cl \in C$
  for *k F G H* **by** *blast+*

have
Con F G H ⟹ *F* ∈ *?cl* ⟹ *G* ∈ *?cl* ∧ *H* ∈ *?cl*
Dis F G H ⟹ *F* ∈ *?cl* ⟹ *G* ∈ *?cl* ∨ *H* ∈ *?cl*
for *F G H*
by(*auto dest*: *d*(*3*−) *cl-max′*[*OF c sc*])
with *d*(*1*,*2*) **show** *?thesis* **unfolding** *Hintikka-alt* **by** *fast*
qed

theorem *pcp-sat*: — model existence theorem
fixes *S* :: *′a* :: *countable formula set*
assumes *c*: *pcp C*
assumes *el*: *S* ∈ *C*
shows *sat S*
proof −
note [[*show-types*]]
from *c* **obtain** *Ce* **where** *C* ⊆ *Ce pcp Ce subset-closed Ce finite-character Ce*
using *ex1*[**where** *′a*=*′a*] *ex2*[**where** *′a*=*′a*] *ex3*[**where** *′a*=*′a*]
by (*meson dual-order.trans ex2*)
have *S* ∈ *Ce* **using** ‹*C* ⊆ *Ce*› *el* **..**
with *pcp-lim-Hintikka* ‹*pcp Ce*› ‹*subset-closed Ce*› ‹*finite-character Ce*›
have *Hintikka* (*pcp-lim Ce S*) **.**
with *Hintikkas-lemma* **have** *sat* (*pcp-lim Ce S*) **.**
moreover have *S* ⊆ *pcp-lim Ce S* **using** *pcp-seq.simps*(*1*) *pcp-seq-sub* **by** *fast*
ultimately show *?thesis* **unfolding** *sat-def* **by** *fast*
qed

end

1.7 Compactness

theory *Compactness*
imports *Sema*
begin

lemma *fin-sat-extend*: *fin-sat S* ⟹ *fin-sat* (*insert F S*) ∨ *fin-sat* (*insert* (¬*F*) *S*)
proof (*rule ccontr*)
assume *fs*: *fin-sat S*
assume *nfs*: ¬ (*fin-sat* (*insert F S*) ∨ *fin-sat* (*insert* (¬ *F*) *S*))
from *nfs* **obtain** *s1* **where** *s1*: *s1* ⊆ *insert F S* *finite s1* ¬*sat s1* **unfolding** *fin-sat-def* **by** *blast*
from *nfs* **obtain** *s2* **where** *s2*: *s2* ⊆ *insert* (*Not F*) *S* *finite s2* ¬*sat s2* **unfolding** *fin-sat-def* **by** *blast*
let *?u* = (*s1* − {*F*}) ∪ (*s2* − {*Not F*})
have *?u* ⊆ *S* *finite ?u* **using** *s1 s2* **by** *auto*
hence *sat ?u* **using** *fs* **unfolding** *fin-sat-def* **by** *blast*
then obtain *A* **where** *A*: ∀ *F* ∈ *?u*. *A* ⊨ *F* **unfolding** *sat-def* **by** *blast*
have *A* ⊨ *F* ∨ *A* ⊨ ¬*F* **by** *simp*
hence *sat s1* ∨ *sat s2* **using** *A* **unfolding** *sat-def* **by**(*fastforce intro*!: *exI*[**where**

```
x=A])
  thus False using s1(3) s2(3) by simp
qed

context
begin

lemma fin-sat-antimono: fin-sat F ⟹ G ⊆ F ⟹ fin-sat G unfolding fin-sat-def
by simp
lemmas fin-sat-insert = fin-sat-antimono[OF - subset-insertI]

primrec extender :: nat ⇒ ('a :: countable) formula set ⇒ 'a formula set where
extender 0 S = S |
extender (Suc n) S = (
  let r = extender n S;
  rt = insert (from-nat n) r;
  rf = insert (¬(from-nat n)) r
  in if fin-sat rf then rf else rt
)

private lemma extender-fin-sat: fin-sat S ⟹ fin-sat (extender n S)
proof(induction n arbitrary: S)
  case (Suc n)
  note mIH = Suc.IH[OF Suc.prems]
  show ?case proof(cases fin-sat (insert (Not (from-nat n)) (extender n S)))
    case True thus ?thesis by simp
  next
    case False
    hence fin-sat (insert ((from-nat n)) (extender n S)) using mIH fin-sat-extend
by auto
    thus ?thesis by(simp add: Let-def)
  qed
qed simp

definition extended S = ⋃{extender n S|n. True}

lemma extended-max: F ∈ extended S ∨ Not F ∈ extended S
proof -
  obtain n where [simp]: F = from-nat n by (metis from-nat-to-nat)
   have F ∈ extender (Suc n) S ∨ Not F ∈ extender (Suc n) S by(simp add:
Let-def)
  thus ?thesis unfolding extended-def by blast
qed

private lemma extender-Sucset: extender k S ⊆ extender (Suc k) S by(force simp
add: Let-def)
private lemma extender-deeper: F ∈ extender k S ⟹ k ≤ l ⟹ F ∈ extender l
S using extender-Sucset le-Suc-eq
  by(induction l) (auto simp del: extender.simps)
```

private lemma *extender-subset*: $S \subseteq$ *extender k S*
proof −
  from *extender-deeper*[*OF* - *le0*] **have** $F \in$ *extender 0 Sa* $\Longrightarrow$ $F \in$ *extender l Sa* **for** *Sa l F* **.**
  thus *?thesis* **by** *auto*
qed

lemma *extended-fin-sat*:
  assumes *fin-sat S*
  shows *fin-sat* (*extended S*)
proof −
  have *assm*: $[\![s \subseteq$ *extender n S*; *finite s*$]\!] \Longrightarrow$ *sat s* **for** *s n*
    using *extender-fin-sat*[*OF assms*] **unfolding** *fin-sat-def* **by** *presburger*
  hence *sat s* **if** *su*: $s \subseteq \bigcup\{$*extender n S* $|n.$ *True*$\}$ **and** *fin*: *finite s* **for** *s* **proof** −
    { fix *x* **assume** *e*: $x \in s$
      with *su* **have** $x \in \bigcup\{$*extender n S* $|n.$ *True*$\}$ **by** *blast*
      hence $\exists\, n.\ x \in$ *extender n S* **unfolding** *Union-eq* **by** *blast* **}**
    hence $\forall\, x \in s.\ \exists\, n.\ x \in$ *extender n S* **by** *blast*
    from *finite-set-choice*[*OF fin this*] **obtain** *f* **where** *cf*: $\forall\, x{\in}s.\ x \in$ *extender* (*f x*) *S* **..**
    have $\exists\, k.\ s \subseteq \bigcup\{$*extender n S* $|n.\ n \leq k\}$ **proof**(*intro exI subsetI*)
      fix *x* **assume** *e*: $x \in s$
      with *cf* **have** $x \in$ *extender* (*f x*) *S* **..**
      hence $x \in$ *extender* (*Max* (*f* ‘ *s*)) *S* **by**(*elim extender-deeper*; *simp add*: *e fin*)
      thus $x \in \bigcup\{$*extender n S* $|n.\ n \leq$ *Max* (*f* ‘ *s*)$\}$ **by** *blast*
    qed
    moreover have $\bigcup\{$*extender n S* $|n.\ n \leq k\} =$ *extender k S* **for** *k* **proof**(*induction k*)
      case (*Suc k*)
      moreover have $\bigcup\{$*extender n S* $|n.\ n \leq$ *Suc k*$\} = \bigcup\{$*extender n S* $|n.\ n \leq k\} \cup$ *extender* (*Suc k*) *S*
        unfolding *Union-eq le-Suc-eq*
        using *le-Suc-eq* **by**(*auto simp del*: *extender.simps*)
        ultimately show *?case* **using** *extender-Sucset* **by**(*force simp del*: *extender.simps*)
    qed *simp*
    ultimately show *sat s* **using** *assm fin* **by** *auto*
  qed
  thus *?thesis* **unfolding** *extended-def fin-sat-def* **by** *presburger*
qed

lemma *extended-superset*: $S \subseteq$ *extended S* **unfolding** *extended-def* **using** *extender.simps*(*1*) **by** *blast*

lemma *extended-complem*:
  assumes *fs*: *fin-sat S*
  shows ($F \in$ *extended S*) $\neq$ (*Not F* $\in$ *extended S*)
proof −
  note *fs* = *fs*[*THEN extended-fin-sat*]

show *?thesis* **proof**(*cases* $F \in$ *extended* S)
  case *False* **with** *extended-max* **show** *?thesis* **by** *blast*
next
  case *True* **have** *Not* $F \notin$ *extended* S **proof**
    assume *False*: *Not* $F \in$ *extended* S
    with *True* **have** $\{F,$ *Not* $F\} \subseteq$ *extended* S **by** *blast*
    moreover have *finite* $\{F,$ *Not* $F\}$ **by** *simp*
    ultimately have *sat* $\{F,$ *Not* $F\}$ **using** *fs* **unfolding** *fin-sat-def* **by** *blast*
    thus *False* **unfolding** *sat-def* **by** *simp*
  qed
  with *True* **show** *?thesis* **by** *blast*
qed
qed

lemma *not-fin-sat-extended-UNIV*: **fixes** S :: *'a* :: *countable formula set* **assumes** $\neg$*fin-sat* S **shows** *extended* S = *UNIV*

Note that this crucially depends on the fact that we check *first* whether adding ¬ F makes the set not satisfiable, and add F otherwise *without any further checks*. The proof of compactness does (to the best of my knowledge) depend on neither of these two facts.

proof −
  from *assms*[*unfolded fin-sat-def*, *simplified*] **obtain** *s* :: *'a* :: *countable formula set*
    where *finite s* ¬ *sat s* **by** *clarify*
  from *this*(*2*)[*unfolded sat-def*, *simplified*] **have** $\exists x \in s.\ \neg\ A \models x$ **for** A **..**
  have *nfs*: ¬*fin-sat* (*insert x* (*extender n S*)) **for** *n x*
    apply(*rule notI*)
    apply(*drule fin-sat-insert*)
    apply(*drule fin-sat-antimono*)
    apply(*rule extender-subset*)
    apply(*erule notE*[*rotated*])
    apply(*fact assms*)
  done
  have $x \in \bigcup\{$*extender n S* $|n.$ *True*$\}$ **for** *x* **proof** *cases*
    assume $x \in S$ **thus** *?thesis* **by** (*metis extended-def extended-superset insert-absorb insert-subset*)
  next
    assume $x \notin S$
    have $x \in$ *extender* (*Suc* (*to-nat x*)) S
      unfolding *extender.simps Let-def*
      unfolding *if-not-P*[*OF nfs*] **by** *simp*
    thus *?thesis* **by** *blast*
  qed
  thus *?thesis* **unfolding** *extended-def* **by** *auto*
qed

lemma *extended-tran*: $S \subseteq T \Longrightarrow$ *extended* $S \subseteq$ *extended* T

This lemma doesn't hold: think of making S empty and inserting a formula into T s.t. it can never be satisfied simultaneously with the first non-tautological formula in the extension S. Showing that this is possible is not worth the effort, since we can't influence the ordering of formulae. But we showed it anyway.

oops
lemma *extended-not-increasing*: ∃ *S T*. *fin-sat S* ∧ *fin-sat T* ∧ ¬ (*S* ⊆ *T* ⟶ *extended S* ⊆ *extended* (*T* :: ′*a* :: *countable formula set*))
proof −
 have *ex-then-min*: ∃ *x* :: *nat*. *P x* ⟹ *P* (*LEAST x*. *P x*) **for** *P* **using** *LeastI2-wellorder* **by** *auto*
 define *P* **where** *P x* = (*let F* = (*from-nat x* :: ′*a formula*) *in* (∃ *A*. ¬ *A* ⊨ *F*) ∧ (∃ *A*. *A* ⊨ *F*)) **for** *x*
 define *x* **where** *x* = (*LEAST n*. *P n*)
 hence ∃ *n*. *P n* **unfolding** *P-def Let-def* **by**(*auto intro*!: *exI*[**where** *x*=*to-nat* (*Atom undefined* :: ′*a formula*)])
 from *ex-then-min*[*OF this*] **have** *Px*: *P x* **unfolding** *x-def* **.**
 have *lessx*: *n* < *x* ⟹ ¬ *P n* **for** *n* **unfolding** *x-def* **using** *not-less-Least* **by** *blast*
 let *?S* = {} :: ′*a formula set* **let** *?T* = {*from-nat x* :: ′*a formula*}
 have *s*: *fin-sat ?S fin-sat ?T* **using** *Px* **unfolding** *P-def fin-sat-def sat-def Let-def* **by** *fastforce*+
 have *reject*: *Q A* ⟹ ∀ *A*. ¬ *Q A* ⟹ *False* **for** *A Q* **by** *simp*
 have *y* ≤ *x* ⟹ *F* ∈ *extender y ?S* ⟹ ⊨ *F* **for** *F y*
 proof(*induction y arbitrary*: *F*)
  case (*Suc y*)
   have ∗: *F* ∈ *extender y* {} ⟹ ⊨ *F* **for** *F* :: ′*a formula* **using** *Suc.IH Suc.prems*(*1*) **by** *auto*
  let *?Y* = *from-nat y* :: ′*a formula*
   have *ex*: (∀ *A*. ¬ *A* ⊨ *?Y*) ∨ ⊨ *?Y* **unfolding** *formula-semantics.simps* **by** (*meson P-def Suc.prems*(*1*) *Suc-le-lessD lessx*)
  have *1*: ∀ *A*. ¬ *A* ⊨ *?Y* **if** *fin-sat* (*Not ?Y* ▹ *extender y ?S*)
  proof −
   note[[*show-types*]]
    from *that* **have** ∃ *A*. *A* ⊨ *Not ?Y* **unfolding** *fin-sat-def sat-def* **by**(*elim allE*[**where** *x*={*Not ?Y*}]) *simp*
   hence ¬⊨ *?Y* **by** *simp*
   hence ∀ *A*. ¬ *A* ⊨ *?Y* **using** *ex* **by** *argo*
   thus *?thesis* **by** *simp*
  qed
  have *2*: ¬ *fin-sat* (*Not ?Y* ▹ *extender y ?S*) ⟹ ⊨ *?Y*
  proof(*erule contrapos-np*)
   assume ¬ ⊨ *?Y*
   hence ∀ *A*. ¬ *A* ⊨ *?Y* **using** *ex* **by** *argo*
   hence ⊨ ¬ *?Y* **by** *simp*
   thus *fin-sat* (¬ *?Y* ▹ *extender y ?S*) **unfolding** *fin-sat-def sat-def*
    by(*auto intro*!: *exI*[**where** *x*=λ- :: ′*a*. *False*] *dest*!: *rev-subsetD*[*rotated*] ∗)
  qed

show *?case* **using** *Suc.prems*(*2*) **by**(*simp add*: *Let-def split*: *if-splits*; *elim disjE*; *simp add*: ∗ *1 2*)
qed *simp*
hence *fin-sat* (¬ (*from-nat x*) ▷ *extender x ?S*) **using** *Px* **unfolding** *P-def Let-def*
by (*clarsimp simp*: *fin-sat-def sat-def*) (*insert formula-semantics.simps*(*3*), *blast*)
hence *Not* (*from-nat x*) ∈ *extender* (*Suc x*) *?S* **by**(*simp*)
hence *Not* (*from-nat x*) ∈ *extended ?S* **unfolding** *extended-def* **by** *blast*
moreover have *Not* (*from-nat x*) ∉ *extended ?T* **using** *extended-complem extended-superset s*(*2*) **by** *blast*
ultimately show *?thesis* **using** *s* **by** *blast*
qed

private lemma *not-in-extended-FE*: *fin-sat S* ⟹ (¬*sat* (*insert* (*Not F*) *G*)) ⟹ *F* ∉ *extended S* ⟹ *G* ⊆ *extended S* ⟹ *finite G* ⟹ *False*
proof(*goal-cases*)
case *1*
hence *Not F* ∈ *extended S* **using** *extended-max* **by** *blast*
thus *False* **using** *1 extended-fin-sat fin-sat-def*
by (*metis Diff-eq-empty-iff finite.insertI insert-Diff-if*)
qed

lemma *extended-id*: *extended* (*extended S*) = *extended S*
using *extended-complem extended-fin-sat extended-max extended-superset not-fin-sat-extended-UNIV*

by(*intro equalityI*[*rotated*] *extended-superset*) *blast*

lemma *ext-model*:
assumes *r*: *fin-sat S*
shows (λ*k*. *Atom k* ∈ *extended S*) ⊨ *F* ⟷ *F* ∈ *extended S*
proof −
note *fs* = *r*[*THEN extended-fin-sat*]
have *Elim*: *F* ∈ *S* ∧ *G* ∈ *S* ⟹ {*F*,*G*} ⊆ *S F* ∈ *S* ⟹ {*F*} ⊆ *S* **for** *F G S* **by** *simp*+
show *?thesis*
proof(*induction F*)
case *Atom* **thus** *?case* **by**(*simp*)
next
case *Bot*
have *False* **if** ⊥ ∈ *extended S* **proof** −
have *finite* {⊥} **by** *simp*
moreover from *that* **have** {⊥} ⊆ *extended S* **by** *simp*
ultimately have ∃ *A*. *A* ⊨ ⊥ **using** *fs* **unfolding** *fin-sat-def sat-def*
by(*elim allE*[*of* - {⊥}]) *simp*
thus *False* **by** *simp*
qed
thus *?case* **by** *auto*

next
case (*Not F*)
moreover have $A \models F \neq A \models \neg F$ **for** *A F* **by** *simp*
ultimately show *?case* **using** *extended-complem*[*OF r*] **by** *blast*
next
case (*And F G*)
have ($F \in$ *extended S* $\wedge$ $G \in$ *extended S*) = ($F \wedge G \in$ *extended S*) **proof** −
have ∗: ¬*sat* {¬ ($F \wedge G$), *F*, *G*} ¬*sat* {¬ *F*, ($F \wedge G$)} ¬*sat* {¬ *G*, ($F \wedge$ G)} **unfolding** *sat-def* **by** *auto*
show *?thesis* **by**(*intro iffI*; *rule ccontr*) (*auto intro*: ∗[*THEN not-in-extended-FE*[*OF r*]])
qed
thus *?case* **by**(*simp add*: *And*)
next
case (*Or F G*)
have ($F \in$ *extended S* $\vee$ $G \in$ *extended S*) = ($F \vee G \in$ *extended S*) **proof** −
have ¬*sat* {¬($F \vee G$), *F*} ¬*sat* {¬($F \vee G$), *G*} **unfolding** *sat-def* **by** *auto*
from *this*[*THEN not-in-extended-FE*[*OF r*]] **have** *1*: ⟦$F \in$ *extended S* $\vee$ $G \in$ *extended S*; $F \vee G \notin$ *extended S*⟧ $\Longrightarrow$ *False* **by** *auto*
have ¬*sat* {¬*F*, ¬*G*, $F \vee G$} **unfolding** *sat-def* **by** *auto*
hence *2*: ⟦$F \vee G \in$ *extended S*; $F \notin$ *extended S*; $G \notin$ *extended S*⟧ $\Longrightarrow$ *False* **using** *extended-max not-in-extended-FE*[*OF r*] **by** *fastforce*
show *?thesis* **by**(*intro iffI*; *rule ccontr*) (*auto intro*: *1 2*)
qed
thus *?case* **by**(*simp add*: *Or*)
next
case (*Imp F G*)
have ($F \in$ *extended S* $\longrightarrow$ $G \in$ *extended S*) = ($F \rightarrow G \in$ *extended S*) **proof** −
have ¬*sat* {¬*G*, *F*, $F \rightarrow G$} **unfolding** *sat-def* **by** *auto*
hence *1*: ⟦$F \rightarrow G \in$ *extended S*; $F \in$ *extended S*; $G \notin$ *extended S*⟧ $\Longrightarrow$ *False* **using** *extended-max not-in-extended-FE*[*OF r*] **by** *blast*
have ¬*sat* {¬*F*,¬($F \rightarrow G$)} **unfolding** *sat-def* **by** *auto*
hence *2*: ⟦$F \rightarrow G \notin$ *extended S*; $F \notin$ *extended S*⟧ $\Longrightarrow$ *False* **using** *extended-max not-in-extended-FE*[*OF r*] **by** *blast*
have ¬*sat* {¬($F \rightarrow G$),*G*} **unfolding** *sat-def* **by** *auto*
hence *3*: ⟦$F \rightarrow G \notin$ *extended S*; $G \in$ *extended S*⟧ $\Longrightarrow$ *False* **using** *extended-max not-in-extended-FE*[*OF r*] **by** *blast*
show *?thesis* **by**(*intro iffI*; *rule ccontr*) (*auto intro*: *1 2 3*)
qed
thus *?case* **by**(*simp add*: *Imp*)
qed
qed

theorem *compactness*:
fixes *S* :: *'a* :: *countable formula set*
shows *sat S* $\longleftrightarrow$ *fin-sat S* (**is** *?l* = *?r*)
proof
assume *?l* **thus** *?r* **unfolding** *sat-def fin-sat-def* **by** *blast*

next
  assume *r*: *?r*
  note *ext-model*[*OF r*, *THEN iffD2*]
  hence $\forall F{\in}S.\ (\lambda k.\ Atom\ k \in extended\ S) \models F$ **using** *extended-superset* **by** *blast*
  thus *?l* **unfolding** *sat-def* **by** *blast*
qed

corollary *compact-entailment*:
  fixes $F :: 'a :: countable\ formula$
  assumes *fent*: $\Gamma \Vdash F$
  shows $\exists \Gamma'.\ finite\ \Gamma' \wedge \Gamma' \subseteq \Gamma \wedge \Gamma' \Vdash F$
proof −
  have *ND-sem*: $\Gamma \Vdash F \longleftrightarrow \neg sat\ (insert\ (\neg F)\ \Gamma)$
    for Γ *F* **unfolding** *sat-def entailment-def* **by** *auto*
  obtain Γ′ **where** *0*: *finite* Γ′ Γ′ ⊫ *F* Γ′ ⊆ Γ **proof**(*goal-cases*)
    from *fent*[*unfolded ND-sem compactness*] **have** ¬ *fin-sat* (*insert* (¬ *F*) Γ) **.**
    from *this*[*unfolded fin-sat-def*] **obtain** *s* **where** *s*: *s* ⊆ *insert*(¬*F*) Γ *finite s* ¬*sat s* **by** *blast*
    have *2*: *finite* (*s* − {¬ *F*}) **using** *s* **by** *simp*
    have *3*: *s* − {¬ *F*} ⊫ *F* **unfolding** *ND-sem* **using** *s*(*3*) **unfolding** *sat-def* **by** *blast*
    have *4*: *s* − {¬ *F*} ⊆ Γ **using** *s* **by** *blast*
    case *1* **from** *2 3 4* **show** *?case* **by**(*intro 1*[*of s* − {*Not F*}])
  qed
  thus *?thesis* **by** *blast*
qed

corollary *compact-to-formula*:
  fixes $F :: 'a :: countable\ formula$
  assumes *fent*: $\Gamma \Vdash F$
  obtains Γ′ **where** $set\ \Gamma' \subseteq \Gamma \models (\bigwedge\Gamma') \to F$
proof *goal-cases*
  case *1*
  from *compact-entailment*[*OF assms*]
  obtain Γ′ **where** Γ′: $finite\ \Gamma' \wedge \Gamma' \subseteq \Gamma \wedge \Gamma' \Vdash F$ **..**
  then obtain Γ′′ **where** Γ′ = *set* Γ′′ **using** *finite-list* **by** *auto*
  with Γ′ **show** *thesis* **by**(*intro 1*) (*blast*, *simp add*: *entailment-def*)
qed

end

end
theory *Compactness-Consistency*
imports *Consistency*
begin

theorem *sat S* ⟷ *fin-sat* (*S* :: ′*a* :: *countable formula set*) (**is** *?l* = *?r*)
proof
  assume *0*: *?r*

let *?C* = {*W* :: *'a formula set*. *fin-sat W*}
have *1*: *S* ∈ *?C* **using** *0* **unfolding** *mem-Collect-eq* **.**
have *2*: *pcp ?C* **proof** −
{ fix *S* :: *'a formula set*
assume *S* ∈ *?C*
hence *a*: ∀ *s*⊆*S*. *finite s* ⟶ (∃ 𝒜. ∀ *F*∈*s*. 𝒜 ⊨ *F*) **by** (*simp add*: *fin-sat-def sat-def*)
have *conj*: ⟦*h F G* ∈ *S*; *s* ⊆ *f F* ▹ *g G* ▹ *S*; *finite s*;
⋀*A*. *A* ⊨ *h F G* ⟹ *A* ⊨ *f F* ∧ *A* ⊨ *g G*⟧ ⟹ ∃ 𝒜. ∀ *F*∈*s*. 𝒜 ⊨ *F*
for *F G s* **and** *f g* :: *'a formula* ⇒ *'a formula* **and** *h* :: *'a formula* ⇒ *'a formula* ⇒ *'a formula*
proof *goal-cases*
case *1*
have *h F G* ▹ *s* − {*f F*,*g G*}⊆*S finite* (*h F G* ▹ *s*−{*f F*,*g G*}) **using** *1* **by** *auto*
then obtain *A* **where** *2*: ∀ *H*∈*h F G* ▹ *s*−{*f F*,*g G*}. *A* ⊨ *H* **using** *a* **by** *presburger*
hence *A* ⊨ *f F A* ⊨ *g G* **using** *1*(*4*) **by** *simp-all*
with *2* **have** ∀ *H*∈*h F G* ▹ *s*. *A* ⊨ *H* **by** *blast*
thus *?case* **by** *blast*
qed
have *disj*: ⟦*h F G* ∈ *S*; *s1* ⊆ *f F* ▹ *S*; *s2* ⊆ *g G* ▹ *S*; *finite s1*; ∀ 𝒜. ∃ *x*∈*s1*. ¬ 𝒜 ⊨ *x*; *finite s2*; ∀ 𝒜. ∃ *x*∈*s2*. ¬ 𝒜 ⊨ *x*;
⋀*A*. *A* ⊨ *h F G* ⟹ *A* ⊨ *f F* ∨ *A* ⊨ *g G*⟧ ⟹ *False*
for *F G s1 s2* **and** *f g* :: *'a formula* ⇒ *'a formula* **and** *h* :: *'a formula* ⇒ *'a formula* ⇒ *'a formula*
proof *goal-cases*
case *1*
let *?U* = *h F G* ▹ (*s1* − {*f F*}) ∪ (*s2* − {*g G*})
have *?U* ⊆ *S finite ?U* **using** *1* **by** *auto*
with *a* **obtain** *A* **where** *2*: *H* ∈ *?U* ⟹ *A* ⊨ *H* **for** *H* **by** *meson*
with *1*(*1*) *1*(*8*) **have** *A* ⊨ *f F* ∨ *A* ⊨ *g G* **by** *force*
hence (∀ *H* ∈ *s1*. *A* ⊨ *H*) ∨ (∀ *H* ∈ *s1*. *A* ⊨ *H*) **using** *1*(*7*) *2*
by (*metis DiffI Diff-empty Diff-iff UnCI insert-iff*)
thus *?case* **using** *1* **by** *fast*
qed
have *1*: ⊥ ∉ *S* **using** *a* **by** (*meson empty-subsetI finite.emptyI finite.insertI formula-semantics.simps*(*2*) *insertI1 insert-subset*)
have *2*: *Atom k* ∈ *S* ⟶ ¬ (*Atom k*) ∈ *S* ⟶ *False* **for** *k* **using** *a*[*THEN spec*[*of* - {*Atom k*, ¬(*Atom k*)}]] **by** *auto*
have *3*: *F* ∧ *G* ∈ *S* ⟶ *F* ▹ *G* ▹ *S* ∈ *Collect fin-sat* **for** *F G* **unfolding** *fin-sat-def sat-def mem-Collect-eq* **using** *conj*[] **by** *fastforce*
have *4*: *F* ∨ *G* ∈ *S* ⟶ *F* ▹ *S* ∈ *Collect fin-sat* ∨ *G* ▹ *S* ∈ *Collect fin-sat* **for** *F G*
unfolding *fin-sat-def sat-def mem-Collect-eq* **using** *disj*[*of Or F G* - λ*k*. *k* - λ*k*. *k*] **by** (*metis formula-semantics.simps*(*5*))
have *5*: *F* → *G* ∈ *S* ⟶ ¬ *F* ▹ *S* ∈ *Collect fin-sat* ∨ *G* ▹ *S* ∈ *Collect fin-sat* **for** *F G*
unfolding *fin-sat-def sat-def mem-Collect-eq* **using** *disj*[*of Imp F G* - *Not* -

λk. k] **by** (*metis formula-semantics.simps*(*3*,*6*))
have *6*: ¬ (¬ *F*) ∈ *S* ⟶ *F* ▷ *S* ∈ *Collect fin-sat* **for** *F* **unfolding** *fin-sat-def sat-def mem-Collect-eq* **using** *conj*[*of λF G. Not* (*Not F*) *F F* - *λk. k λk. k*] **by** *simp*
have *7*: ¬ (*F* ∧ *G*) ∈ *S* ⟶ ¬ *F* ▷ *S* ∈ *Collect fin-sat* ∨ ¬ *G* ▷ *S* ∈ *Collect fin-sat* **for** *F G*
unfolding *fin-sat-def sat-def mem-Collect-eq* **using** *disj*[*of λF G. Not* (*And F G*) *F G* - *Not* - *Not*] **by** (*metis formula-semantics.simps*(*3*,*4*))
have *8*: ∀ *F G*. ¬ (*F* ∨ *G*) ∈ *S* ⟶ ¬ *F* ▷ ¬ *G* ▷ *S* ∈ *Collect fin-sat* **unfolding** *fin-sat-def sat-def mem-Collect-eq* **using** *conj*[] **by** *fastforce*
have *9*: ∀ *F G*. ¬ (*F* → *G*) ∈ *S* ⟶ *F* ▷ ¬ *G* ▷ *S* ∈ *Collect fin-sat* **unfolding** *fin-sat-def sat-def mem-Collect-eq* **using** *conj*[] **by** *fastforce*
note *1 2 3 4 5 6 7 8 9*
}
thus *pcp ?C* **unfolding** *pcp-def* **by** *auto*
qed
from *pcp-sat 2 1* **show** *?l* **.**
next
assume *?l* **thus** *?r* **unfolding** *sat-def fin-sat-def* **by** *blast*
qed

end

1.8 Craig Interpolation using Semantics

theory *Sema-Craig*
imports *Substitution-Sema*
begin

Semantic proof of Craig interpolation following Harrison [5].

lemma *subst-true-false*:
assumes 𝒜 ⊨ *F*
shows 𝒜 ⊨ ((*F*[⊤ / *n*]) ∨ (*F*[⊥ / *n*]))
using *assms* **by**(*cases* 𝒜 *n*; *simp add*: *substitution-lemma fun-upd-idem*)

theorem *interpolation*:
assumes ⊨ Γ → Δ
obtains ϱ **where**
⊨ Γ → ϱ ⊨ ϱ → Δ
atoms ϱ ⊆ *atoms* Γ
atoms ϱ ⊆ *atoms* Δ
proof(*goal-cases*)
let *?as* = *atoms* Γ − *atoms* Δ
have *fas*: *finite ?as* **by** *simp*
from *fas assms* **have** ∃ ϱ. ((⊨ Γ → ϱ) ∧ (⊨ ϱ → Δ) ∧ (*atoms* ϱ ⊆ *atoms* Γ) ∧ (*atoms* ϱ ⊆ *atoms* Δ))
proof(*induction ?as arbitrary*: Γ *rule*: *finite-induct*)
case *empty*
from ‹{} = *atoms* Γ − *atoms* Δ› **have** *atoms* Γ ⊆ *atoms* Δ **by** *blast*

with ‹$\models \Gamma \to \Delta$› **show** *?case* **by**(*intro exI*[**where** x=Γ]) *simp*
next
case (*insert a A*)
hence *e*: $a \in$ *atoms* Γ $a \notin$ *atoms* Δ **by** *auto*
define Γ' **where** $\Gamma' = (\Gamma[\top\ /\ a]) \vee (\Gamma[\bot\ /\ a])$
have *su*: *atoms* $\Gamma' \subseteq$ *atoms* Γ **unfolding** Γ'*-def* **by**(*cases* $a \in$ *atoms* Γ; *simp add*: *subst-atoms*)
from ‹$\models \Gamma \to \Delta$› *e* **have** $\models \Gamma' \to \Delta$ **by** (*auto simp add*: *substitution-lemma* Γ'*-def*)
from ‹$a \triangleright A$ = *atoms* Γ − *atoms* Δ› ‹$a \notin A$› *e* **have** A = *atoms* Γ' − *atoms* Δ **by**(*simp add*: *subst-atoms* Γ'*-def*) *blast*
from *insert.hyps*(*3*)[*OF this* ‹$\models \Gamma' \to \Delta$›] **obtain** ϱ **where** ϱ: $\models \Gamma' \to \varrho$ $\models \varrho \to \Delta$ *atoms* $\varrho \subseteq$ *atoms* Γ' *atoms* $\varrho \subseteq$ *atoms* Δ **by** *clarify*
have $\models \Gamma \to \varrho$ **using** ϱ(*1*) *subst-true-false* **unfolding** Γ'*-def* **by** *fastforce*
with ϱ *su* **show** *?case* **by**(*intro exI*[**where** x=ϱ]) *simp*
qed
moreover case *1*
ultimately show *thesis* **by** *blast*
qed

The above proof is constructive, and it is actually very easy to write a procedure down.

function *interpolate* **where**
interpolate F H = (
let K = *atoms F* − *atoms H in*
if K = {}
then F
else (
let k = *Min K*
in interpolate ((*F*[$\top$ / *k*]) $\vee$ (*F*[$\bot$ / *k*])) *H*
)
) **by** *pat-completeness simp*

Showing termination is slightly technical...

termination *interpolate*
apply(*relation measure* (λ(*F*,*H*). *card* (*atoms F* − *atoms H*)))

subgoal by *simp*
apply (*simp add*: *subst-atoms-simp*)
apply(*intro conjI impI*)
apply(*intro psubset-card-mono*)
subgoal by *simp*
apply(*subgoal-tac Min* (*atoms F* − *atoms H*) $\notin$ *atoms H*)
subgoal by *blast*
apply (*meson atoms-finite Diff-eq-empty-iff Diff-iff Min-in finite-Diff*)+
done

Surprisingly, *interpolate* is even executable, despite all the set operations involving *atoms*

lemma *interpolate* (*And* (*Atom* (*0*::*nat*)) (*Atom 1*)) (*Or* (*Atom 1*) (*Atom 2*)) =
(⊤ ∧ *Atom 1*) ∨ (⊥ ∧ *Atom 1*) **by** *simp*
value[*code*] *simplify-consts* (*interpolate* (*And* (*Atom* (*0*::*nat*)) (*Atom 1*)) (*Or* (*Atom 1*) (*Atom 2*)))

lemma *let P* = *Atom* (*0* :: *nat*); *Q* = *Atom 1*; *R* = *Atom 2*; *T* = *Atom 3*;
φ = (¬(*P* ∧ *Q*)) → (¬*R* ∧ *Q*);
ψ = (*T* → *P*) ∨ (*T* → (¬*R*));
I = *interpolate* φ ψ *in*
(*size I*) = *23* ∧ *simplify-consts I* = *Atom 2* → *Atom 0*
by *code-simp*

theorem *nonexistential-interpolation*:
assumes ⊨ *F* → *H*
shows
⊨ *F* → *interpolate F H* (**is** *?t1*) ⊨ *interpolate F H* → *H* (**is** *?t2*)
atoms (*interpolate F H*) ⊆ *atoms F* ∩ *atoms H* (**is** *?s*)
proof −
let *?as* = *atoms F* − *atoms H*
have *fas*: *finite ?as* **by** *simp*
hence *?t1* ∧ *?t2* ∧ *?s* **using** *assms*
proof(*induction card ?as arbitrary*: *F H*)
case (*Suc n*)
let *?inf* = *Min* (*atoms F* − *atoms H*)
define *G* **where** *G* = (*F*[⊤ / *?inf*]) ∨ (*F*[⊥ / *?inf*])
have *e*: *Min* (*atoms F* − *atoms H*) ∈ *atoms F* − *atoms H* **by** (*metis Min-in Suc.hyps*(*2*) *Suc.prems*(*1*) *card.empty nat.simps*(*3*))
with *Suc*(*2*) **have** *n* = *card* (*atoms G* − *atoms H*) **unfolding** *G-def subst-atoms-simp*
proof −
assume *a1*: *Suc n* = *card* (*atoms F* − *atoms H*)
assume *Min* (*atoms F* − *atoms H*) ∈ *atoms F* − *atoms H*
hence *a2*: *Min* (*atoms F* − *atoms H*) ∈ *atoms F* ∧ *Min* (*atoms F* − *atoms H*) ∉ *atoms H* **by** *simp*
have *n* = *card* (*atoms F* − *atoms H*) − *1*
using *a1* **by** *presburger*
hence *n* = *card* (*atoms* (*F*[⊤ / *Min* (*atoms F* − *atoms H*)]) ∪ *atoms* (*F*[⊥ / *Min* (*atoms F* − *atoms H*)]) − *atoms H*)
using *a2* **by** (*metis* (*full-types*) *formula.set*(*2*) *Diff-insert Diff-insert2 Suc.prems*(*1*) *Un-absorb Un-empty-right card-Diff-singleton e subst-atoms top-atoms-simp*)

thus *n* = *card* (*atoms* ((*F*[⊤ / *Min* (*atoms F* − *atoms H*)]) ∨ (*F*[⊥ / *Min* (*atoms F* − *atoms H*)])) − *atoms H*) **by** *simp*
qed
moreover have *finite* (*atoms G* − *atoms H*) ⊨ *G* → *H* **using** *Suc*(*3*−) *e*
by(*auto simp*: *G-def substitution-lemma*)
ultimately have *IH*: ⊨ *G* → *interpolate G H* ⊨ *interpolate G H* → *H*
atoms (*interpolate G H*) ⊆ *atoms G* ∩ *atoms H* **using** *Suc* **by** *blast+*
moreover have ⊨ *F* → *G* **unfolding** *G-def*
using *subst-true-false* **by** *fastforce*

moreover {
assume *a1*: *atoms* (*interpolate* ((*F*[⊤/*Min* (*atoms F* − *atoms H*)]) ∨ (*F*[⊥/*Min* (*atoms F* − *atoms H*)])) *H*) ⊆ *atoms* (*F*[⊤/*Min* (*atoms F* − *atoms H*)]) ∪ *atoms* (*F*[⊥/*Min* (*atoms F* − *atoms H*)]) ∧ *atoms* (*interpolate* ((*F*[⊤/*Min* (*atoms F* − *atoms H*)]) ∨ (*F*[⊥/*Min* (*atoms F* − *atoms H*)])) *H*) ⊆ *atoms H*
have *f2*: *atoms* ((⊥::'*a formula*) → ⊥) = *atoms* ⊥
by *simp*
then have *f3*: *atoms F* − {*Min* (*atoms F* − *atoms H*)} = *atoms* (*F*[⊤/*Min* (*atoms F* − *atoms H*)])
by (*metis* (*no-types*) *DiffD1 Top-def Un-empty-right e formula.simps*(*91*) *subst-atoms*)
have *atoms* (*F*[⊥/*Min* (*atoms F* − *atoms H*)]) = *atoms* (*F*[⊤/*Min* (*atoms F* − *atoms H*)])
using *f2* **by** (*metis* (*no-types*) *DiffD1 Top-def e subst-atoms*)
then have ¬ *atoms F* ⊆ *atoms H* ⟶ *atoms* (*interpolate* ((*F*[⊤/*Min* (*atoms F* − *atoms H*)]) ∨ (*F*[⊥/*Min* (*atoms F* − *atoms H*)])) *H*) ⊆ *atoms F*
using *f3 a1* **by** *blast*
} **ultimately show** *?case*
by (*intro conjI*; *subst interpolate.simps*; *simp del*: *interpolate.simps add*: *Let-def G-def*; *blast?*)
qed *auto*
thus *?t1 ?t2 ?s* **by** *simp-all*
qed

So no, the proof is by no means easier this way. Admittedly, part of the fuzz is due to *Min*, but replacing atoms with something that returns lists doesn't make it better.

end

2 Proof Systems

2.1 Sequent Calculus

theory *SC*
imports *Formulas HOL−Library.Multiset*
begin

abbreviation *msins* (‹-, -› [*56*,*56*] *56*) **where** *x*,*M* == *add-mset x M*

We do not formalize the concept of sequents, only that of sequent calculus derivations.

inductive *SCp* :: '*a formula multiset* ⇒ '*a formula multiset* ⇒ *bool* (‹(- ⇒/ -)› [*53*] *53*) **where**
BotL: ⊥ ∈# Γ ⟹ Γ⇒Δ |
Ax: *Atom k* ∈# Γ ⟹ *Atom k* ∈# Δ ⟹ Γ⇒Δ |
NotL: Γ ⇒ *F*,Δ ⟹ *Not F*, Γ ⇒ Δ |
NotR: *F*,Γ ⇒ Δ ⟹ Γ ⇒ *Not F*, Δ |
AndL: *F*,*G*,Γ ⇒ Δ ⟹ *And F G*, Γ ⇒ Δ |

AndR: ⟦ Γ ⇒ *F*,Δ; Γ ⇒ *G*,Δ ⟧ ⟹ Γ ⇒ *And F G*, Δ |
OrL: ⟦ *F*,Γ ⇒ Δ; *G*,Γ ⇒ Δ ⟧ ⟹ *Or F G*, Γ ⇒ Δ |
OrR: Γ ⇒ *F*,*G*,Δ ⟹ Γ ⇒ *Or F G*, Δ |
ImpL: ⟦ Γ ⇒ *F*,Δ; *G*,Γ ⇒ Δ ⟧ ⟹ *Imp F G*, Γ ⇒ Δ |
ImpR: *F*,Γ ⇒ *G*,Δ ⟹ Γ ⇒ *Imp F G*, Δ

Many of the proofs here are inspired Troelstra and Schwichtenberg [11].

lemma
 shows *BotL-canonical*[*intro*!]: ⊥,Γ⇒Δ
  and *Ax-canonical*[*intro*!]: *Atom k*,Γ ⇒ *Atom k*,Δ
 by (*meson SCp.intros union-single-eq-member*)+

lemma *lem1*: $x \neq y \Longrightarrow x,\ M = y{,}N \longleftrightarrow x \in\#\ N \wedge M = y{,}(N - \{\#x\#\})$
 by (*metis* (*no-types*, *lifting*) *add-eq-conv-ex add-mset-remove-trivial add-mset-remove-trivial-eq*)

lemma *lem2*: $x \neq y \Longrightarrow x,\ M = y,\ N \longleftrightarrow y \in\#\ M \wedge N = x,\ (M - \{\#y\#\})$
 using *lem1* **by** *fastforce*

This is here to deal with a technical problem: the way the simplifier uses *?x*, *?y*, *?M* = *?y*, *?x*, *?M* is really suboptimal for the unification of SC rules.

lemma *sc-insertion-ordering*[*simp*]:
 NO-MATCH (*I*→*J*) *H* ⟹ *H*,*F*→*G*,*S* = *F*→*G*,*H*,*S*
 NO-MATCH (*I*→*J*) *H* ⟹ *NO-MATCH* (*I*∨*J*) *H* ⟹ *H*,*F*∨*G*,*S* = *F*∨*G*,*H*,*S*
 NO-MATCH (*I*→*J*) *H* ⟹ *NO-MATCH* (*I*∨*J*) *H* ⟹ *NO-MATCH* (*I*∧*J*) *H* ⟹ *H*,*F*∧*G*,*S* = *F*∧*G*,*H*,*S*
 NO-MATCH (*I*→*J*) *H* ⟹ *NO-MATCH* (*I*∨*J*) *H* ⟹ *NO-MATCH* (*I*∧*J*) *H* ⟹ *NO-MATCH* (¬*J*) *H* ⟹ *H*,¬*G*,*S* = ¬*G*,*H*,*S*
 NO-MATCH (*I*→*J*) *H* ⟹ *NO-MATCH* (*I*∨*J*) *H* ⟹ *NO-MATCH* (*I*∧*J*) *H* ⟹ *NO-MATCH* (¬*J*) *H* ⟹ *NO-MATCH* (⊥) *H* ⟹ *H*,⊥,*S* = ⊥,*H*,*S*
 NO-MATCH (*I*→*J*) *H* ⟹ *NO-MATCH* (*I*∨*J*) *H* ⟹ *NO-MATCH* (*I*∧*J*) *H* ⟹ *NO-MATCH* (¬*J*) *H* ⟹ *NO-MATCH* (⊥) *H* ⟹ *NO-MATCH* (*Atom k*) *H* ⟹ *H*,*Atom l*,*S* = *Atom l*,*H*,*S*

 by *simp-all*

lemma shows
  inR1: Γ ⇒ *G*, *H*, Δ ⟹ Γ ⇒ *H*, *G*, Δ
 and *inL1*: *G*, *H*, Γ ⇒ Δ ⟹ *H*, *G*, Γ ⇒ Δ
 and *inR2*: Γ ⇒ *F*, *G*, *H*, Δ ⟹ Γ ⇒ *G*, *H*, *F*, Δ
 and *inL2*: *F*, *G*, *H*, Γ ⇒ Δ ⟹ *G*, *H*, *F*, Γ ⇒ Δ **by**(*simp-all add*: *add-mset-commute*)
lemmas *SCp-swap-applies*[*elim*!,*intro*] = *inL1 inL2 inR1 inR2*

lemma *NotL-inv*: *Not F*, Γ ⇒ Δ ⟹ Γ ⇒ *F*,Δ
proof(*induction Not F*, Γ Δ *arbitrary*: Γ *rule*: *SCp.induct*)
 case (*NotL* Γ′ *G* Δ) **thus** *?case* **by**(*cases Not F* = *Not G*) (*auto intro*!: *SCp.intros*(*3*−) *dest*!: *multi-member-split simp*: *lem1 lem2*)
qed (*auto intro*!: *SCp.intros*(*3*−) *dest*!: *multi-member-split simp*: *SCp.intros lem1 lem2*)

lemma *AndL-inv*: *And F G*, $\Gamma \Rightarrow \Delta \Longrightarrow F,G,\Gamma \Rightarrow \Delta$
proof(*induction And F G*, Γ Δ *arbitrary*: Γ *rule*: *SCp.induct*)
  case (*AndL* F' G' Γ' Δ) **thus** *?case*
    by(*cases And F G* = *And* F' G';
      auto intro!: *SCp.intros*(*3*−) *dest*!: *multi-member-split simp*: *lem1 lem2*;
      metis add-mset-commute)
qed (*auto intro*!: *SCp.intros*(*3*−) *dest*!: *multi-member-split simp*: *SCp.intros lem1 lem2 inL2*)

lemma *OrL-inv*:
  assumes *Or F G*, $\Gamma \Rightarrow \Delta$
  shows $F,\Gamma \Rightarrow \Delta \wedge G,\Gamma \Rightarrow \Delta$
proof(*insert assms*, *induction Or F G*, Γ Δ *arbitrary*: Γ *rule*: *SCp.induct*)
  case (*OrL* F' Γ' Δ G') **thus** *?case*
    by(*cases Or F G* = *Or* F' G';
      auto intro!: *SCp.intros*(*3*−) *dest*!: *multi-member-split simp*: *lem1 lem2*;
      blast)
qed (*fastforce intro*!: *SCp.intros*(*3*−) *dest*!: *multi-member-split simp*: *SCp.intros lem1 lem2*)+

lemma *ImpL-inv*:
  assumes *Imp F G*, $\Gamma \Rightarrow \Delta$
  shows $\Gamma \Rightarrow F,\Delta \wedge G,\Gamma \Rightarrow \Delta$
proof(*insert assms*, *induction Imp F G*, Γ Δ *arbitrary*: Γ *rule*: *SCp.induct*)
  case (*ImpL* Γ' F' Δ G') **thus** *?case*
    by(*cases Or F G* = *Or* F' G';
      auto intro!: *SCp.intros*(*3*−) *dest*!: *multi-member-split simp*: *lem1 lem2*;
      blast)
qed (*fastforce intro*!: *SCp.intros*(*3*−) *dest*!: *multi-member-split simp*: *SCp.intros lem1 lem2*)+

lemma *ImpR-inv*:
  assumes $\Gamma \Rightarrow$ *Imp F G*,Δ
  shows $F,\Gamma \Rightarrow G,\Delta$
proof(*insert assms*, *induction* Γ *Imp F G*, Δ *arbitrary*: Δ *rule*: *SCp.induct*)
  case (*ImpR* F' Γ G' Δ') **thus** *?case*
    by(*cases Or F G* = *Or* F' G';
      auto intro!: *SCp.intros*(*3*−) *dest*!: *multi-member-split simp*: *lem1 lem2*;
      blast)
qed (*fastforce intro*!: *SCp.intros*(*3*−) *dest*!: *multi-member-split simp*: *SCp.intros lem1 lem2*)+

lemma *AndR-inv*:
shows $\Gamma \Rightarrow$ *And F G*, $\Delta \Longrightarrow \Gamma \Rightarrow F,\ \Delta \wedge \Gamma \Rightarrow G,\ \Delta$
proof(*induction* Γ *And F G*, Δ *arbitrary*: Δ *rule*: *SCp.induct*)
  case (*AndR* Γ F' Δ' G') **thus** *?case*
    by(*cases Or F G* = *Or* F' G';
      auto intro!: *SCp.intros*(*3*−) *dest*!: *multi-member-split simp*: *lem1 lem2*;

blast)
qed (*fastforce intro*!: *SCp.intros*(*3*−) *dest*!: *multi-member-split simp*: *SCp.intros lem1 lem2*)+

lemma *OrR-inv*: Γ ⇒ *Or F G*, Δ ⟹ Γ ⇒ *F*,*G*,Δ
proof(*induction* Γ *Or F G*, Δ *arbitrary*: Δ *rule*: *SCp.induct*)
  case (*OrR* Γ *F′ G′* Δ′) **thus** *?case*
    by(*cases Or F G* = *Or F′ G′*;
      auto intro!: *SCp.intros*(*3*−) *dest*!: *multi-member-split simp*: *lem1 lem2*;
      metis add-mset-commute)
qed (*fastforce intro*!: *SCp.intros*(*3*−) *dest*!: *multi-member-split simp*: *SCp.intros lem1 lem2*)+

lemma *NotR-inv*: Γ ⇒ *Not F*, Δ ⟹ *F*,Γ ⇒ Δ
proof(*induction* Γ *Not F*, Δ *arbitrary*: Δ *rule*: *SCp.induct*)
  case (*NotR* *G* Γ Δ′) **thus** *?case*
    by(*cases Not F* = *Not G*;
      auto intro!: *SCp.intros*(*3*−) *dest*!: *multi-member-split simp*: *lem1 lem2*;
      metis add-mset-commute)
qed (*fastforce intro*!: *SCp.intros*(*3*−) *dest*!: *multi-member-split simp*: *SCp.intros lem1 lem2*)+

lemma *weakenL*: Γ ⇒ Δ ⟹ *F*,Γ ⇒ Δ
  by(*induction rule*: *SCp.induct*)
    (*auto intro*!: *SCp.intros*(*3*−) *intro*: *SCp.intros*(*1*,*2*))

lemma *weakenR*: Γ ⇒ Δ ⟹ Γ ⇒ *F*,Δ
  by(*induction rule*: *SCp.induct*)
    (*auto intro*!: *SCp.intros*(*3*−) *intro*: *SCp.intros*(*1*,*2*))

lemma *weakenL-set*: Γ ⇒ Δ ⟹ *F* + Γ ⇒ Δ
  by(*induction F*; *simp add*: *weakenL*)
lemma *weakenR-set*: Γ ⇒ Δ ⟹ Γ ⇒ *F* + Δ
  by(*induction F*; *simp add*: *weakenR*)

lemma *extended-Ax*: Γ ∩# Δ ≠ {#} ⟹ Γ ⇒ Δ
proof −
  assume Γ ∩# Δ ≠ {#}
  then obtain *W* **where** *W* ∈# Γ *W* ∈# Δ **by** *force*
  then show *?thesis* **proof**(*induction W arbitrary*: Γ Δ)
    case (*Not W*)
    from *Not.prems* **obtain** Γ′ Δ′ **where** [*simp*]: Γ = *Not W*,Γ′ Δ = *Not W*,Δ′ **by** (*metis insert-DiffM*)
    have *W* ∈# *W*,Γ′ *W* ∈# *W*,Δ′ **by** *simp-all*
    from *Not.IH*[*OF this*] **obtain** *n* **where** *W*, Γ′ ⇒ *W*, Δ′ **by** *presburger*
    hence *Not W*, Γ′ ⇒ *Not W*, Δ′ **using** *SCp.intros*(*3*,*4*) *add-mset-commute* **by** *metis*
    thus Γ ⇒ Δ **by** *auto*
  next

case (*And G H*)
from *And.prems* **obtain** Γ' Δ' **where** [*simp*]: Γ = *And G H*,Γ' Δ = *And G H*,Δ' **by** (*metis insert-DiffM*)
have $G \in\# G, H, \Gamma'$ $G \in\# G, \Delta'$ **by** *simp-all*
with *And.IH*(*1*) **have** *IH1*: $G, H, \Gamma' \Rightarrow G, \Delta'$ **.**
have $H \in\# G, H, \Gamma'$ $H \in\# H, \Delta'$ **by** *simp-all*
with *And.IH*(*2*) **have** *IH2*: $G, H, \Gamma' \Rightarrow H, \Delta'$ **.**
from *IH1 IH2* **have** $G, H, \Gamma' \Rightarrow G, \Delta'$ $G, H, \Gamma' \Rightarrow H, \Delta'$ **by** *fast+*
thus $\Gamma \Rightarrow \Delta$ **using** *SCp.intros*(*5*,*6*) **by** *fastforce*
next
case (*Or G H*)
case (*Imp G H*)

analogously

qed (*auto intro*: *SCp.intros*)
qed

lemma *Bot-delR*: $\Gamma \Rightarrow \Delta \Longrightarrow \Gamma \Rightarrow \Delta - \{\#\bot\#\}$
proof(*induction rule*: *SCp.induct*)
case *BotL*
thus *?case* **by** (*simp add*: *SCp.BotL*)
next
case *Ax*
thus *?case*
by (*metis SCp.Ax diff-union-swap formula.distinct*(*1*) *insert-DiffM union-single-eq-member*)
next
case *NotL*
thus *?case*
by (*metis SCp.NotL diff-single-trivial diff-union-swap2*)
next
case *NotR*
thus *?case* **by**(*simp add*: *SCp.NotR*)
next
case *AndL*
thus *?case* **by** (*simp add*: *SCp.AndL*)
next
case *AndR*
thus *?case*
by (*metis SCp.AndR diff-single-trivial diff-union-swap diff-union-swap2 formula.distinct*(*13*))
next
case *OrL*
thus *?case* **by**(*simp add*: *SCp.OrL*)
next
case *OrR*
thus *?case*
by (*metis SCp.OrR diff-single-trivial diff-union-swap2 formula.distinct*(*15*) *insert-iff set-mset-add-mset-insert*)
next

case *ImpL*
thus *?case* **by** (*metis SCp.ImpL diff-single-trivial diff-union-swap2*)
next
case *ImpR*
thus *?case*
by (*metis SCp.ImpR diff-single-trivial diff-union-swap diff-union-swap2 formula.distinct*(*17*))
qed
corollary *Bot-delR-simp*: $\Gamma \Rightarrow \bot,\Delta = \Gamma \Rightarrow \Delta$
using *Bot-delR weakenR* **by** *fastforce*

end
theory *SC-Cut*
imports *SC*
begin

2.1.1 Contraction

First, we need contraction:

lemma *contract*:
$(F,F,\Gamma \Rightarrow \Delta \longrightarrow F,\Gamma \Rightarrow \Delta) \wedge (\Gamma \Rightarrow F,F,\Delta \longrightarrow \Gamma \Rightarrow F,\Delta)$
proof(*induction F arbitrary*: Γ Δ)
case (*Atom k*)
have *Atom k*, *Atom k*, $\Gamma \Rightarrow \Delta \Longrightarrow$ *Atom k*, $\Gamma \Rightarrow \Delta$
by(*induction Atom k*, *Atom k*, Γ Δ *arbitrary*: Γ *rule*: *SCp.induct*)
(*auto dest*!: *multi-member-split intro*!: *SCp.intros*(*3*−) *simp add*: *lem2 lem1 SCp.intros*)
moreover have $\Gamma \Rightarrow$ *Atom k*, *Atom k*, $\Delta \Longrightarrow \Gamma \Rightarrow$ *Atom k*, Δ
by(*induction* Γ *Atom k*, *Atom k*, Δ *arbitrary*: Δ *rule*: *SCp.induct*)
(*auto dest*!: *multi-member-split intro*!: *SCp.intros*(*3*−) *simp add*: *lem2 lem1 SCp.intros*)
ultimately show *?case* **by** *blast*
next
case *Bot*
have $\bot, \bot, \Gamma \Rightarrow \Delta \Longrightarrow \bot, \Gamma \Rightarrow \Delta$ **by**(*blast*)
moreover have $(\Gamma \Rightarrow \bot, \bot, \Delta \Longrightarrow \Gamma \Rightarrow \bot, \Delta)$
using *Bot-delR* **by** *fastforce*
ultimately show *?case* **by** *blast*
next
case (*Not F*)
then show *?case* **by** (*meson NotL-inv NotR-inv SCp.NotL SCp.NotR*)
next
case (*And F1 F2*) **then show** *?case* **by**(*auto intro*!: *SCp.intros*(*3*−) *dest*!: *AndR-inv AndL-inv*) (*metis add-mset-commute*)
next
case (*Or F1 F2*) **then show** *?case* **by**(*auto intro*!: *SCp.intros*(*3*−) *dest*!: *OrR-inv OrL-inv*) (*metis add-mset-commute*)
next

case (*Imp F1 F2*) **show** *?case* **by**(*auto dest*!: *ImpR-inv ImpL-inv intro*!: *SCp.intros*(*3*−))
(*insert Imp.IH*; *blast*)+
qed
corollary
shows *contractL*: $F, F, \Gamma \Rightarrow \Delta \Longrightarrow F, \Gamma \Rightarrow \Delta$
and *contractR*: $\Gamma \Rightarrow F, F, \Delta \Longrightarrow \Gamma \Rightarrow F, \Delta$
using *contract* **by** *blast*+

2.1.2 Cut

Which cut rule we show is up to us:

lemma *cut-cs-cf*:
assumes *context-sharing-Cut*: $\bigwedge(A{::}'a\ formula)\ \Gamma\ \Delta.\ \Gamma \Rightarrow A{,}\Delta \Longrightarrow A{,}\Gamma \Rightarrow \Delta \Longrightarrow \Gamma \Rightarrow \Delta$
shows *context-free-Cut*: $\Gamma \Rightarrow (A{::}'a\ formula){,}\Delta \Longrightarrow A{,}\Gamma' \Rightarrow \Delta' \Longrightarrow \Gamma + \Gamma' \Rightarrow \Delta + \Delta'$
by(*intro context-sharing-Cut*[*of* $\Gamma + \Gamma'$ *A* $\Delta + \Delta'$])
(*metis add.commute union-mset-add-mset-left weakenL-set weakenR-set*)+
lemma *cut-cf-cs*:
assumes *context-free-Cut*: $\bigwedge(A{::}'a\ formula)\ \Gamma\ \Gamma'\ \Delta\ \Delta'.\ \Gamma \Rightarrow A{,}\Delta \Longrightarrow A{,}\Gamma' \Rightarrow \Delta' \Longrightarrow \Gamma + \Gamma' \Rightarrow \Delta + \Delta'$
shows *context-sharing-Cut*: $\Gamma \Rightarrow (A{::}'a\ formula){,}\Delta \Longrightarrow A{,}\Gamma \Rightarrow \Delta \Longrightarrow \Gamma \Rightarrow \Delta$
proof −
have *contract*ΓΓ: $\Gamma{+}\Gamma' \Rightarrow \Delta \Longrightarrow \Gamma' \subseteq\# \Gamma \Longrightarrow \Gamma \Rightarrow \Delta$ **for** $\Gamma\ \Gamma'\ \Delta$
proof(*induction* Γ' *arbitrary*: Γ)
case *empty* **thus** *?case* **using** *weakenL-set* **by** *force*
next
case (*add x* Γ' Γ)
from *add.prems*(*2*) **have** $x \in\# \Gamma$ **by** (*simp add*: *insert-subset-eq-iff*)
then obtain Γ'' **where** Γ[*simp*]: $\Gamma = x{,}\Gamma''$ **by** (*meson multi-member-split*)
from *add.prems*(*1*) **have** $x{,}x{,}\Gamma'' + \Gamma' \Rightarrow \Delta$ **by** *simp*
with *contractL* **have** $x,\ \Gamma'' + \Gamma' \Rightarrow \Delta$ **.**
with *add.IH*[*of* Γ] **show** *?case* **using** Γ *add.prems*(*2*) *subset-mset.trans* **by** *force*
qed
have *contract*ΔΔ: $\Gamma \Rightarrow \Delta{+}\Delta' \Longrightarrow \Delta' \subseteq\# \Delta \Longrightarrow \Gamma \Rightarrow \Delta$ **for** $\Gamma\ \Delta\ \Delta'$
proof(*induction* Δ' *arbitrary*: Δ)
case *empty* **thus** *?case* **using** *weakenL-set* **by** *force*
next
case (*add x* Δ' Δ)
from *add.prems*(*2*) **have** $x \in\# \Delta$ **by** (*simp add*: *insert-subset-eq-iff*)
then obtain Δ'' **where** Δ[*simp*]: $\Delta = x{,}\Delta''$ **by** (*metis multi-member-split*)
from *add.prems*(*1*) **have** $\Gamma \Rightarrow x{,}x{,}\Delta'' + \Delta'$ **by** *simp*
with *contractR* **have** $\Gamma \Rightarrow x,\ \Delta'' + \Delta'$ **.**
with *add.IH*[*of* Δ] **show** *?case* **using** Δ *add.prems*(*2*) *subset-mset.trans* **by** *force*
qed
show $\Gamma \Rightarrow A{,}\Delta \Longrightarrow A{,}\Gamma \Rightarrow \Delta \Longrightarrow \Gamma \Rightarrow \Delta$
using *context-free-Cut*[*of* Γ *A* Δ Γ Δ] *contract*ΓΓ *contract*ΔΔ

```
  by blast
qed
```

According to Troelstra and Schwichtenberg [11], the sharing variant is the one we want to prove.

```
lemma Cut-Atom-pre: Atom k,Γ ⇒ Δ ⟹ Γ ⇒ Atom k,Δ ⟹ Γ ⇒ Δ
proof(induction Atom k,Γ Δ arbitrary: Γ rule: SCp.induct)
  case BotL
  hence ⊥ ∈# Γ by simp
  thus ?case using SCp.BotL by blast
next
  case (Ax l Δ)
  show ?case proof cases
    assume l = k
    with ‹Atom l ∈# Δ› obtain Δ' where Δ = Atom k, Δ' by (meson multi-member-split)
    with ‹Γ ⇒ Atom k, Δ› have Γ ⇒ Δ using contractR by blast
    thus ?thesis by auto
  next
    assume l ≠ k
    with ‹Atom l ∈# Atom k, Γ› have Atom l ∈# Γ by simp
    with ‹Atom l ∈# Δ› show ?thesis using SCp.Ax[of l] by blast
  qed
next
  case NotL
  thus ?case by(auto simp: add-eq-conv-ex intro!: SCp.NotL dest!: NotL-inv)
next
  case NotR
  then show ?case by(auto intro!: SCp.NotR dest!: NotR-inv)
next
  case AndL
  thus ?case by(auto simp: add-eq-conv-ex intro!: SCp.AndL dest!: AndL-inv)
next
  case AndR
  then show ?case by(auto intro!: SCp.AndR dest!: AndR-inv)
next
  case OrL
  thus ?case by(auto simp: add-eq-conv-ex intro!: SCp.OrL dest!: OrL-inv)
next
  case OrR
  thus ?case by(auto intro!: SCp.OrR dest!: OrR-inv)
next
  case ImpL
  thus ?case by(auto simp: add-eq-conv-ex intro!: SCp.ImpL dest!: ImpL-inv)
next
  case ImpR
  then show ?case by (auto dest!: ImpR-inv intro!: SCp.ImpR)
qed
```

We can show the admissibility of the cut rule by induction on the cut formula (or, if you will, as a procedure that splits the cut into smaller formulas that get cut). The only mildly complicated case is that of cutting in an *Atom k*. It is, contrary to the general case, only mildly complicated, since the cut formula can only appear principal in the axiom rules.

theorem *cut*: $\Gamma \Rightarrow F,\Delta \Longrightarrow F,\Gamma \Rightarrow \Delta \Longrightarrow \Gamma \Rightarrow \Delta$
proof(*induction F arbitrary*: Γ Δ)
  case *Atom* **thus** *?case* **using** *Cut-Atom-pre* **by** *metis*
next
  case *Bot* **thus** *?case* **using** *Bot-delR* **by** *fastforce*
next
  case *Not* **with** *NotL-inv NotR-inv* **show** *?case* **by** *blast*
next
  case *And* **thus** *?case* **by** (*meson AndL-inv AndR-inv weakenL*)
next
  case *Or* **thus** *?case* **by** (*metis OrL-inv OrR-inv weakenR*)
next
  case (*Imp F G*)

  from *ImpR-inv* ‹$\Gamma \Rightarrow F \rightarrow G, \Delta$› **have** *R*: $F, \Gamma \Rightarrow G, \Delta$ **by** *blast*
  from *ImpL-inv* ‹$F \rightarrow G, \Gamma \Rightarrow \Delta$› **have** *L*: $\Gamma \Rightarrow F, \Delta$ $G, \Gamma \Rightarrow \Delta$ **by** *blast+*
  from *L*(*1*) **have** $\Gamma \Rightarrow F, G, \Delta$ **using** *weakenR add-ac*(*3*) **by** *blast*
  with *R* **have** $\Gamma \Rightarrow G, \Delta$ **using** *Imp.IH*(*1*) **by** *simp*
  with *L*(*2*) **show** $\Gamma \Rightarrow \Delta$ **using** *Imp.IH*(*2*) **by** *clarsimp*
qed

corollary *cut-cf*: $[\![\Gamma \Rightarrow A, \Delta; A, \Gamma' \Rightarrow \Delta']\!] \Longrightarrow \Gamma + \Gamma' \Rightarrow \Delta + \Delta'$
  using *cut-cs-cf cut* **by** *metis*

lemma assumes *cut*: $\bigwedge \Gamma' \Delta'$ (*A*::*'a formula*). $[\![\Gamma' \Rightarrow A, \Delta'; A, \Gamma' \Rightarrow \Delta']\!] \Longrightarrow \Gamma' \Rightarrow \Delta'$
  shows *contraction-admissibility*: $F,F,\Gamma \Rightarrow \Delta \Longrightarrow$ (*F*::*'a formula*),$\Gamma \Rightarrow \Delta$
  by(*rule cut*[*of F*,Γ *F* Δ, *OF extended-Ax*]) *simp-all*

end
theory *SC-Depth*
imports *SC*
begin

Many textbook arguments about SC use the depth of the derivation tree as basis for inductions. We had originally thought that this is mandatory for the proof of contraction, but found out it is not. It remains unclear to us whether there is any proof on SC that requires an argument using depth.

We keep our formalization of SC with depth for didactic reasons: we think that arguments about depth do not need much meta-explanation, but structural induction and rule induction usually need extra explanation for students unfamiliar with Isabelle/HOL. They are also a lot harder to execute. We dare the reader to write down (a few of) the cases for, e.g. *AndL-inv′*, by hand.

inductive *SCc* :: *′a formula multiset* ⇒ *′a formula multiset* ⇒ *nat* ⇒ *bool* (‹((-⇒/ -) ↓ -)› [*53*,*53*] *53*) **where**
BotL: ⊥ ∈# Γ ⟹ Γ⇒Δ ↓ *Suc n* |
Ax: *Atom k* ∈# Γ ⟹ *Atom k* ∈# Δ ⟹ Γ⇒Δ ↓ *Suc n* |
NotL: Γ ⇒ *F*,Δ ↓ *n* ⟹ *Not F*, Γ ⇒ Δ ↓ *Suc n* |
NotR: *F*,Γ ⇒ Δ ↓ *n* ⟹ Γ ⇒ *Not F*, Δ ↓ *Suc n* |
AndL: *F*,*G*,Γ ⇒ Δ ↓ *n* ⟹ *And F G*, Γ ⇒ Δ ↓ *Suc n* |
AndR: ⟦ Γ ⇒ *F*,Δ ↓ *n*; Γ ⇒ *G*,Δ ↓ *n* ⟧ ⟹ Γ ⇒ *And F G*, Δ ↓ *Suc n* |
OrL: ⟦ *F*,Γ ⇒ Δ ↓ *n*; *G*,Γ ⇒ Δ ↓ *n* ⟧ ⟹ *Or F G*, Γ ⇒ Δ ↓ *Suc n* |
OrR: Γ ⇒ *F*,*G*,Δ ↓ *n* ⟹ Γ ⇒ *Or F G*, Δ ↓ *Suc n* |
ImpL: ⟦ Γ ⇒ *F*,Δ ↓ *n*; *G*,Γ ⇒ Δ ↓ *n* ⟧ ⟹ *Imp F G*, Γ ⇒ Δ ↓ *Suc n* |
ImpR: *F*,Γ ⇒ *G*,Δ ↓ *n* ⟹ Γ ⇒ *Imp F G*, Δ ↓ *Suc n*

lemma
  shows *BotL-canonical′*[*intro*!]: ⊥,Γ⇒Δ ↓ *Suc n*
    and *Ax-canonical′*[*intro*!]: *Atom k*,Γ ⇒ *Atom k*,Δ ↓ *Suc n*
  by (*meson SCc.intros union-single-eq-member*)+

lemma *deeper*: Γ ⇒ Δ ↓ *n* ⟹ Γ ⇒ Δ ↓ *n* + *m*
  by(*induction rule*: *SCc.induct*; *insert SCc.intros*; *auto*)
lemma *deeper-suc*: Γ ⇒ Δ ↓ *n* ⟹ Γ ⇒ Δ ↓ *Suc n*

  thm *deeper*[*unfolded Suc-eq-plus1*[*symmetric*]]
  by(*drule deeper*[**where** *m*=*1*]) *simp*

The equivalence is obvious.

theorem *SC-SCp-eq*:
  fixes Γ Δ :: *′a formula multiset*
  shows (∃ *n*. Γ ⇒ Δ ↓ *n*) ⟷ Γ ⇒ Δ (**is** *?c* ⟷ *?p*)
proof
  assume *?c*
  then obtain *n* **where** Γ ⇒ Δ ↓ *n* **..**
  thus *?p* **by**(*induction rule*: *SCc.induct*; *simp add*: *SCp.intros*)
next
  have *deeper-max*: Γ ⇒ Δ ↓ *max m n* Γ ⇒ Δ ↓ *max n m* **if** Γ ⇒ Δ ↓ *n* **for** *n m* **and** Γ Δ :: *′a formula multiset*
  proof −
    have *n* ≤ *m* ⟹ ∃ *k*. *m* = *n* + *k* **by** *presburger*
    with *that*[*THEN deeper*] *that*
    show Γ ⇒ Δ ↓ *max n m* **unfolding** *max-def* **by** *clarsimp*
    thus Γ ⇒ Δ ↓ *max m n* **by** (*simp add*: *max.commute*)
  qed
  assume *?p* **thus** *?c* **by**(*induction rule*: *SCp.induct*)
    (*insert SCc.intros*[**where** *′a*=*′a*] *deeper-max*; *metis*)+
qed

lemma *no-0-SC*[*dest*!]: $\Gamma \Rightarrow \Delta \downarrow 0 \Longrightarrow$ *False*
  by(*cases rule*: *SCc.cases*[*of* Γ Δ *0*]) *assumption*

lemma *inR1'*: $\Gamma \Rightarrow G, H, \Delta \downarrow n \Longrightarrow \Gamma \Rightarrow H, G, \Delta \downarrow n$ **by**(*simp add*: *add-mset-commute*)
lemma *inL1'*: $G, H, \Gamma \Rightarrow \Delta \downarrow n \Longrightarrow H, G, \Gamma \Rightarrow \Delta \downarrow n$ **by**(*simp add*: *add-mset-commute*)
lemma *inR2'*: $\Gamma \Rightarrow F, G, H, \Delta \downarrow n \Longrightarrow \Gamma \Rightarrow G, H, F, \Delta \downarrow n$ **by**(*simp add*: *add-mset-commute*)
lemma *inL2'*: $F, G, H, \Gamma \Rightarrow \Delta \downarrow n \Longrightarrow G, H, F, \Gamma \Rightarrow \Delta \downarrow n$ **by**(*simp add*: *add-mset-commute*)
lemma *inR3'*: $\Gamma \Rightarrow F, G, H, \Delta \downarrow n \Longrightarrow \Gamma \Rightarrow H, F, G, \Delta \downarrow n$ **by**(*simp add*: *add-mset-commute*)
lemma *inR4'*: $\Gamma \Rightarrow F, G, H, I, \Delta \downarrow n \Longrightarrow \Gamma \Rightarrow H, I, F, G, \Delta \downarrow n$ **by**(*simp add*: *add-mset-commute*)
lemma *inL3'*: $F, G, H, \Gamma \Rightarrow \Delta \downarrow n \Longrightarrow H, F, G, \Gamma \Rightarrow \Delta \downarrow n$ **by**(*simp add*: *add-mset-commute*)
lemma *inL4'*: $F, G, H, I, \Gamma \Rightarrow \Delta \downarrow n \Longrightarrow H, I, F, G, \Gamma \Rightarrow \Delta \downarrow n$ **by**(*simp add*: *add-mset-commute*)
lemmas *SC-swap-applies*[*intro*,*elim*!] = *inL1' inL2' inL3' inL4' inR1' inR2' inR3' inR4'*

lemma *Atom C* $\rightarrow$ *Atom D* $\rightarrow$ *Atom E*,
    Atom k $\rightarrow$ *Atom C* $\wedge$ *Atom D*,
    Atom k, {#}
$\Rightarrow$ {# *Atom E* #} $\downarrow$ *Suc* (*Suc* (*Suc* (*Suc* (*Suc 0*))))
  by(*auto intro*!: *SCc.intros*(*3*−) *intro*: *SCc.intros*(*1*,*2*))

lemma *Bot-delR'*: $\Gamma \Rightarrow \Delta \downarrow n \Longrightarrow \Gamma \Rightarrow \Delta -$ {#⊥#} $\downarrow n$
proof(*induction rule*: *SCc.induct*)
  case *BotL* **thus** *?case* **by**(*rule SCc.BotL*; *simp*)
next case (*Ax k*) **thus** *?case* **by**(*intro SCc.Ax*[*of k*]; *simp*; *metis diff-single-trivial formula.distinct*(*1*) *insert-DiffM lem1*)
next case *NotL* **thus** *?case* **using** *SCc.NotL* **by** (*metis add-mset-remove-trivial diff-single-trivial diff-union-swap insert-DiffM*)
next case *NotR* **thus** *?case* **using** *SCc.NotR* **by** (*metis diff-union-swap formula.distinct*(*11*))
next case *AndR* **thus** *?case* **using** *SCc.AndR* **by** (*metis diff-single-trivial diff-union-swap diff-union-swap2 formula.distinct*(*13*))
next case *OrR* **thus** *?case* **using** *SCc.OrR* **by** (*metis diff-single-trivial diff-union-swap2 formula.distinct*(*15*) *insert-iff set-mset-add-mset-insert*)
next case *ImpL* **thus** *?case* **using** *SCc.ImpL* **by** (*metis diff-single-trivial diff-union-swap2*)
next case *ImpR* **thus** *?case* **using** *SCc.ImpR* **by** (*metis diff-single-trivial diff-union-swap diff-union-swap2 formula.distinct*(*17*))
qed (*simp-all add*: *SCc.intros*)

lemma *NotL-inv'*: *Not F*, $\Gamma \Rightarrow \Delta \downarrow n \Longrightarrow \Gamma \Rightarrow F, \Delta \ \downarrow n$
proof(*induction Not F*, Γ Δ *n arbitrary*: Γ *rule*: *SCc.induct*)
  case (*NotL* Γ' *G* Δ *n*) **thus** *?case* **by**(*cases Not F* = *Not G*)

(*auto intro*!: *SCc.intros*(*3−*) *dest*!: *multi-member-split simp*: *lem1 lem2 deeper-suc*)
qed (*auto intro*!: *SCc.intros*(*3−*) *dest*!: *multi-member-split simp*: *SCp.intros lem1 lem2*)

lemma *AndL-inv*′: *And F G*, Γ ⇒ Δ ↓ *n* ⟹ *F*,*G*,Γ ⇒ Δ ↓ *n*
proof(*induction And F G*, Γ Δ *n arbitrary*: Γ *rule*: *SCc.induct*)
  case (*AndL F*′ *G*′ Γ′ Δ) **thus** *?case*
    by(*cases And F G* = *And F*′ *G*′;
      auto intro!: *SCc.intros*(*3−*) *dest*!: *multi-member-split simp*: *lem1 lem2 deeper-suc*;
        metis add-mset-commute)
qed (*auto intro*!: *SCc.intros*(*3−*) *dest*!: *multi-member-split simp*: *SCc.intros lem1 lem2 inL2*′)

lemma *OrL-inv*′:
  assumes *Or F G*, Γ ⇒ Δ ↓ *n*
  shows *F*,Γ ⇒ Δ ↓ *n* ∧ *G*,Γ ⇒ Δ ↓ *n*
proof(*insert assms*, *induction Or F G*, Γ Δ *n arbitrary*: Γ *rule*: *SCc.induct*)
  case (*OrL F*′ Γ′ Δ *n G*′) **thus** *?case*
    by(*cases Or F G* = *Or F*′ *G*′;
      auto intro!: *SCc.intros*(*3−*) *dest*!: *multi-member-split simp*: *lem1 lem2 deeper-suc*;
        blast)
qed (*fastforce intro*!: *SCc.intros*(*3−*) *dest*!: *multi-member-split simp*: *SCc.intros lem1 lem2*)+

lemma *ImpL-inv*′:
  assumes *Imp F G*, Γ ⇒ Δ ↓ *n*
  shows Γ ⇒ *F*,Δ ↓ *n* ∧ *G*,Γ ⇒ Δ ↓ *n*
proof(*insert assms*, *induction Imp F G*, Γ Δ *n arbitrary*: Γ *rule*: *SCc.induct*)
  case (*ImpL* Γ′ *F*′ Δ *n G*′) **thus** *?case*
    by(*cases Or F G* = *Or F*′ *G*′;
      auto intro!: *SCc.intros*(*3−*) *dest*!: *multi-member-split simp*: *lem1 lem2 deeper-suc*;
        blast)
qed (*fastforce intro*!: *SCc.intros*(*3−*) *dest*!: *multi-member-split simp*: *SCc.intros lem1 lem2*)+

lemma *ImpR-inv*′:
  assumes Γ ⇒ *Imp F G*,Δ ↓ *n*
  shows *F*,Γ ⇒ *G*,Δ ↓ *n*
proof(*insert assms*, *induction* Γ *Imp F G*, Δ *n arbitrary*: Δ *rule*: *SCc.induct*)
  case (*ImpR F*′ Γ *G*′ Δ′) **thus** *?case*
    by(*cases Or F G* = *Or F*′ *G*′;
      auto intro!: *SCc.intros*(*3−*) *dest*!: *multi-member-split simp*: *lem1 lem2 deeper-suc*;
        blast)
qed (*fastforce intro*!: *SCc.intros*(*3−*) *dest*!: *multi-member-split simp*: *SCc.intros lem1 lem2*)+

lemma *AndR-inv*′:
shows Γ ⇒ *And F G*, Δ ↓ *n* ⟹ Γ ⇒ *F*, Δ ↓ *n* ∧ Γ ⇒ *G*, Δ ↓ *n*
proof(*induction* Γ *And F G*, Δ *n arbitrary*: Δ *rule*: *SCc.induct*)

case (*AndR* Γ F' Δ' *n* G') **thus** *?case*
by(*cases Or F G* = *Or F' G'*;
auto intro!: *SCc.intros*(*3*−) *dest*!: *multi-member-split simp*: *lem1 lem2 deeper-suc*;
blast)
qed (*fastforce intro*!: *SCc.intros*(*3*−) *dest*!: *multi-member-split simp*: *SCc.intros lem1 lem2*)+

lemma *OrR-inv'*: $\Gamma \Rightarrow Or\ F\ G,\ \Delta \downarrow n \Longrightarrow \Gamma \Rightarrow F,G,\Delta \downarrow n$
proof(*induction* Γ *Or F G*, Δ *n arbitrary*: Δ *rule*: *SCc.induct*)
case (*OrR* Γ F' G' Δ') **thus** *?case*
by(*cases Or F G* = *Or F' G'*;
auto intro!: *SCc.intros*(*3*−) *dest*!: *multi-member-split simp*: *lem1 lem2 deeper-suc*;
metis add-mset-commute)
qed (*fastforce intro*!: *SCc.intros*(*3*−) *dest*!: *multi-member-split simp*: *SCc.intros lem1 lem2*)+

lemma *NotR-inv'*: $\Gamma \Rightarrow Not\ F,\ \Delta \downarrow n \Longrightarrow F,\Gamma \Rightarrow \Delta \downarrow n$
proof(*induction* Γ *Not F*, Δ *n arbitrary*: Δ *rule*: *SCc.induct*)
case (*NotR* *G* Γ Δ') **thus** *?case*
by(*cases Not F* = *Not G*;
auto intro!: *SCc.intros*(*3*−) *dest*!: *multi-member-split simp*: *lem1 lem2 deeper-suc*;
metis add-mset-commute)
qed (*fastforce intro*!: *SCc.intros*(*3*−) *dest*!: *multi-member-split simp*: *SCc.intros lem1 lem2*)+

lemma *weakenL'*: $\Gamma \Rightarrow \Delta \downarrow n \Longrightarrow F,\Gamma \Rightarrow \Delta \downarrow n$
by(*induction rule*: *SCc.induct*)
(*auto intro*!: *SCc.intros*(*3*−) *intro*: *SCc.intros*(*1*,*2*))

lemma *weakenR'*: $\Gamma \Rightarrow \Delta \downarrow n \Longrightarrow \Gamma \Rightarrow F,\Delta \downarrow n$
by(*induction rule*: *SCc.induct*)
(*auto intro*!: *SCc.intros*(*3*−) *intro*: *SCc.intros*(*1*,*2*))

lemma *contract'*:
$(F,F,\Gamma \Rightarrow \Delta \downarrow n \longrightarrow F,\Gamma \Rightarrow \Delta \downarrow n) \wedge (\Gamma \Rightarrow F,F,\Delta \downarrow n \longrightarrow \Gamma \Rightarrow F,\Delta \downarrow n)$
proof(*induction n arbitrary*: *F* Γ Δ)
case (*Suc n*)
hence *IH*: $F,\ F,\ \Gamma \Rightarrow \Delta \downarrow n \Longrightarrow F,\ \Gamma \Rightarrow \Delta \downarrow n$ $\Gamma \Rightarrow F,\ F,\ \Delta \downarrow n \Longrightarrow \Gamma \Rightarrow F,\ \Delta \downarrow n$ **for** *F* :: *'a formula* **and** Γ Δ **by** *blast*+
show *?case* **proof**(*intro conjI allI impI*, *goal-cases*)
case *1*
let *?ffs* = $\lambda\Gamma.\ \Gamma - \{\#\ F\ \#\} - \{\#\ F\ \#\}$
from *1* **show** *?case* **proof**(*insert 1*; *cases rule*: *SCc.cases*[*of F*,*F*,Γ Δ *Suc n*])
case (*NotL* Γ' *G*)
show *?thesis*
proof(*cases*)
assume *e*: $F = \neg G$
with *NotL* **have** Γ': $\Gamma' = \neg G,\ \Gamma$ **by** *simp*
from *NotL-inv' NotL*(*2*) **have** $\Gamma \Rightarrow G,\ G,\ \Delta \downarrow n$ **unfolding** Γ' **.**

with *IH*(*2*) **have** $\Gamma \Rightarrow G, \Delta \downarrow n$ **.**
with *SCc.NotL* **show** *?thesis* **unfolding** *e* **.**
next
assume *e*: $F \neq \neg G$
have *?thesis*
using *NotL*(*2*) *IH*(*1*)[*of F ?ffs* Γ' *G*, Δ] *SCc.NotL*[*of F*, Γ' − {# *F* #} − {# *F* #} *G* Δ *n*]
using *e NotL*(*1*) **by** (*metis* (*no-types*, *lifting*) *insert-DiffM lem2*)
from *e NotL*(*1*) **have** Γ: $\Gamma = \neg G$, *?ffs* Γ' **by** (*meson lem1*)
with *NotL*(*1*) **have** Γ′: *F*,*F*,*?ffs* $\Gamma' = \Gamma'$ **by** *simp*
show *?thesis* **using** *NotL*(*2*) *IH*(*1*)[*of F ?ffs* Γ' *G*, Δ] *SCc.NotL*[*of F*, *?ffs* Γ' *G* Δ *n*] ‹*F*, $\Gamma \Rightarrow \Delta \downarrow$ *Suc n*› **by** *blast*
qed
next
case (*AndL G H* Γ') **show** *?thesis* **proof** *cases*
assume *e*: F = *And G H*
with *AndL*(*1*) **have** Γ′: Γ' = *And G H*, Γ **by** *simp*
have $G \wedge H, G, H, \Gamma \Rightarrow \Delta \downarrow n$ **using** *AndL*(*2*) **unfolding** Γ′ **by** *auto*
hence $G, H, G, H, \Gamma \Rightarrow \Delta \downarrow n$ **by**(*rule AndL-inv′*)
hence $G, H, \Gamma \Rightarrow \Delta \downarrow n$ **using** *IH*(*1*) **by** (*metis inL1′ inL3′*)
thus $F, \Gamma \Rightarrow \Delta \downarrow$ *Suc n* **using** *e SCc.AndL* **by** *simp*
next
assume *ne*: $F \neq$ *And G H*
with *AndL*(*1*) **have** Γ: Γ = *And G H*, *?ffs* Γ' **by** (*metis* (*no-types*, *lifting*) *diff-diff-add lem2*)
have *F*, *F*, *G*, *H*, *?ffs* $\Gamma' \Rightarrow \Delta \downarrow n$ **using** *AndL*(*2*) **using** Γ *inL4′ local.AndL*(*1*) **by** *auto*
hence *G*, *H*, *F*, *?ffs* $\Gamma' \Rightarrow \Delta \downarrow n$ **using** *IH*(*1*) *inL2* **by** *blast*
thus *?thesis* **using** *SCc.AndL* **unfolding** Γ **using** *inL1* **by** *blast*
qed
next
case (*OrL G* Γ' *H*) **show** *?thesis* **proof** *cases*
assume *e*: F = *Or G H*
with *OrL*(*1*) **have** Γ′: Γ' = *Or G H*, Γ **by** *simp*
have *Or G H*, *G*, $\Gamma \Rightarrow \Delta \downarrow n$ *Or G H*, *H*, $\Gamma \Rightarrow \Delta \downarrow n$ **using** *OrL*(*2*,*3*) **unfolding** Γ′ **by** *simp-all*
hence $G, G, \Gamma \Rightarrow \Delta \downarrow n$ $H, H, \Gamma \Rightarrow \Delta \downarrow n$ **using** *OrL-inv′* **by** *blast+*
hence $G, \Gamma \Rightarrow \Delta \downarrow n$ $H, \Gamma \Rightarrow \Delta \downarrow n$ **using** *IH*(*1*) **by** *presburger+*
thus $F, \Gamma \Rightarrow \Delta \downarrow$ *Suc n* **unfolding** *e* **using** *SCc.OrL* **by** *blast*
next
assume *ne*: $F \neq$ *Or G H*
with *OrL*(*1*) **have** Γ: Γ = *Or G H*, *?ffs* Γ' **by** (*metis* (*no-types*, *lifting*) *diff-diff-add lem2*)
have *F*, *F*, *G*, *?ffs* $\Gamma' \Rightarrow \Delta \downarrow n$ *F*, *F*, *H*,*?ffs* $\Gamma' \Rightarrow \Delta \downarrow n$ **using** *OrL* Γ **by** *auto*
hence *G*, *F*, *?ffs* $\Gamma' \Rightarrow \Delta \downarrow n$ *H*, *F*, *?ffs* $\Gamma' \Rightarrow \Delta \downarrow n$ **using** *IH*(*1*) **by**(*metis add-mset-commute*)+
thus *?thesis* **using** *SCc.OrL* **unfolding** Γ **by** *auto*
qed

next
case (*ImpL* Γ′ *G H*) **show** *?thesis* **proof** *cases*
assume *e*: *F* = *Imp G H*
with *ImpL*(*1*) **have** Γ′: Γ′ = *Imp G H*, Γ **by** *simp*
have *H*, Γ ⇒ Δ ↓ *n* Γ ⇒ *G*,Δ ↓ *n* **using** *IH ImpL-inv′ ImpL*(*2*,*3*) **unfolding** Γ′
by (*metis add-mset-commute*)+
thus *?thesis* **unfolding** *e* **using** *SCc.ImpL*[**where** ′*a*=′*a*] **by** *simp*
next
assume *ne*: *F* ≠ *Imp G H*
with *ImpL*(*1*) **have** Γ: Γ = *Imp G H*, *?ffs* Γ′ **by** (*metis* (*no-types*, *lifting*) *diff-diff-add lem2*)
have *F*, *F*, *?ffs* Γ′ ⇒ *G*, Δ ↓ *n F*, *F*, *H*, *?ffs* Γ′ ⇒ Δ ↓ *n* **using** *ImpL* Γ **by** *auto*
thus *?thesis* **using** *SCc.ImpL IH* **unfolding** Γ **by** (*metis inL1′*)
qed
next
case *ImpR* **thus** *?thesis* **by** (*simp add*: *IH*(*1*) *SCc.intros*(*10*) *add-mset-commute*)
next
case (*NotR G* Δ′) **thus** *?thesis* **using** *IH*(*1*) **by** (*simp add*: *SCc.NotR add-mset-commute*)
qed (*auto intro*: *IH SCc.intros*(*1*,*2*) *intro*!: *SCc.intros*(*3*−*10*))
next
case *2*
let *?ffs* = λΓ. Γ − {# *F* #} − {# *F* #}
have *not-principal*[*dest*]:
⟦*F* ≠ *f G H*; *F*, *F*, Δ = *f G H*, Δ′⟧ ⟹ Δ = *f G H*, *?ffs* Δ′ ∧ *F*, *F*, *?ffs* Δ′ = Δ′ **for** *f G H* Δ′ **proof**(*intro conjI*, *goal-cases*)
case *2*
from *2* **have** *F* ∈# Δ′ **by**(*blast dest*: *lem1*[*THEN iffD1*])
then obtain Δ″ **where** Δ′: Δ′ = *F*,Δ″ **by** (*metis insert-DiffM*)
with *2*(*2*) **have** *F*, Δ = *f G H*, Δ″ **by**(*simp add*: *add-mset-commute*)
hence *F* ∈# Δ″ **using** *2*(*1*) **by**(*blast dest*: *lem1*[*THEN iffD1*])
then obtain Δ‴ **where** Δ″: Δ″ = *F*,Δ‴ **by** (*metis insert-DiffM*)
show *?case* **unfolding** Δ′ Δ″ **by** *simp*
case *1* **show** *?case* **using** *1*(*2*) **unfolding** Δ′ Δ″ **by**(*simp add*: *add-mset-commute*)
qed
have *principal*[*dest*]: *F*, *F*, Δ = *f G H*, Δ′ ⟹ *F* = *f G H* ⟹ Δ′ = *f G H*, Δ **for** *f G H* Δ′ **by** *simp*
from *2* **show** *?case* **proof**(*cases rule*: *SCc.cases*[*of* Γ *F*,*F*,Δ *Suc n*])
case (*ImpR G H* Δ′) **thus** *?thesis* **proof** *cases*
assume *e*[*simp*]: *F* = *Imp G H*
with *ImpR*(*1*) **have** Δ′[*simp*]: Δ′ = *Imp G H*, Δ **by** *simp*
have *G*, Γ ⇒ *Imp G H*, *H*, Δ ↓ *n* **using** *ImpR*(*2*) **by** *simp*
hence *G*, *G*, Γ ⇒ *H*, *H*, Δ ↓ *n* **by**(*rule ImpR-inv′*)
hence *G*, Γ ⇒ *H*, Δ ↓ *n* **using** *IH* **by** *fast*
thus Γ ⇒ *F*, Δ ↓ *Suc n* **using** *SCc.ImpR* **by** *simp*
next
assume *a*: *F* ≠ *Imp G H*

with *ImpR*(*1*) **have** Δ: $\Delta = Imp\ G\ H,\ ?ffs\ \Delta'$ **by** (*metis* (*no-types*, *lifting*) *diff-diff-add lem2*)

have $G,\Gamma \Rightarrow F,\ F,\ H,\ ?ffs\ \Delta' \downarrow n$ **using** *ImpR* Δ **by** *fastforce*

thus *?thesis* **using** *SCc.ImpR IH* **unfolding** Δ **by** (*metis inR1′*)

qed

next

case (*AndR G* Δ′ *H*) **thus** *?thesis* **proof**(*cases* $F = And\ G\ H$)

case *True* **thus** *?thesis* **using** *AndR* **by**(*auto intro*!: *SCc.intros*(*3*−) *dest*!: *AndR-inv′ intro*: *IH*)

next

case *False*

hence *e*: $\Delta = And\ G\ H,\ ?ffs\ \Delta'$ **using** *AndR*(*1*) **using** *diff-diff-add lem2* **by** *blast*

hence $G \wedge H,\ F,\ F,\ ?ffs\ \Delta' = G \wedge H,\ \Delta'$ **using** *AndR*(*1*) **by** *simp*

hence $\Gamma \Rightarrow F,\ F,\ G,\ ?ffs\ \Delta' \downarrow n\ \Gamma \Rightarrow F,\ F,\ H,\ ?ffs\ \Delta' \downarrow n$ **using** *AndR*(*2*,*3*) **using** *add-left-imp-eq inR2* **by** *fastforce*+

hence $\Gamma \Rightarrow G,\ F,\ ?ffs\ \Delta' \downarrow n\ \Gamma \Rightarrow H,\ F,\ ?ffs\ \Delta' \downarrow n$ **using** *IH*(*2*) **by** *blast*+

thus *?thesis* **unfolding** *e* **by**(*intro SCc.AndR*[*THEN inR1′*])

qed

next

case (*OrR G H* Δ′) **thus** *?thesis* **proof** *cases*

assume *a*: $F = Or\ G\ H$

hence Δ′: $\Delta' = G \vee H,\ \Delta$ **using** *OrR*(*1*) **by**(*intro principal*)

hence $\Gamma \Rightarrow G,\ H,\ G,\ H,\ \Delta \downarrow n$ **using** *inR3′*[*THEN OrR-inv′*] *OrR*(*2*) **by** *auto*

hence $\Gamma \Rightarrow H,\ G,\ \Delta \downarrow n$ **using** *IH*(*2*)[*of* Γ *G H*,*H*,Δ] *IH*(*2*)[*of* Γ *H G*,Δ]

unfolding *add-ac*(*3*)[*of* {#*H*#} {#*G*#}] **using** *inR2* **by** *blast*

hence $\Gamma \Rightarrow G,\ H,\ \Delta \downarrow n$ **by**(*elim SC-swap-applies*)

thus *?thesis* **unfolding** *a* **by** (*simp add*: *SCc.OrR*)

next

assume *a*: $F \neq Or\ G\ H$

with *not-principal* **have** *np*: $\Delta = G \vee H,\ ?ffs\ \Delta' \wedge F,\ F,\ ?ffs\ \Delta' = \Delta'$ **using** *OrR*(*1*) **.**

with *OrR*(*2*) **have** $\Gamma \Rightarrow G,\ H,\ F,\ ?ffs\ \Delta' \downarrow n$ **using** *IH*(*2*) **by** (*metis inR2′ inR4′*)

hence $\Gamma \Rightarrow F,\ G \vee H,\ ?ffs\ \Delta' \downarrow Suc\ n$ **by**(*intro SCc.OrR*[*THEN inR1′*])

thus *?thesis* **using** *np* **by** *simp*

qed

next

case (*NotR G* Δ′) **thus** *?thesis* **proof**(*cases* $F = Not\ G$)

case *True*

with *principal NotR*(*1*) **have** $\Delta' = \neg\ G,\ \Delta$ **.**

with *NotR-inv′ NotR*(*2*) **have** $G,\ G,\ \Gamma \Rightarrow \Delta \downarrow n$ **by** *blast*

with *IH*(*1*) **have** $G,\ \Gamma \Rightarrow \Delta \downarrow n$ **.**

thus $\Gamma \Rightarrow F,\ \Delta \downarrow Suc\ n$ **unfolding** *True* **by**(*intro SCc.NotR*)

next

case *False*

with *not-principal* **have** *np*: $\Delta = \neg\ G,\ \Delta' - (F,\ \{\#F\#\}) \wedge F,\ F,\ \Delta' -$

$(F, \{\#F\#\}) = \Delta'$ **using** *NotR*(*1*) **by** *auto*
    hence $G, \Gamma \Rightarrow F, F,$ *?ffs* $\Delta' \downarrow n$ **using** *NotR*(*2*) **by** *simp*
    hence $G, \Gamma \Rightarrow F,$ *?ffs* $\Delta' \downarrow n$ **by**(*elim IH*(*2*))
    thus *?thesis* **using** *np SCc.NotR inR1* **by** *auto*
  qed
 next
  case *BotL* **thus** *?thesis* **by**(*elim SCc.BotL*)
 next
  case (*Ax k*) **thus** *?thesis* **by**(*intro SCc.Ax*[**where** *k=k*]) *simp-all*
 next
  case *NotL* **thus** *?thesis* **by** (*simp add*: *SCc.NotL Suc.IH add-mset-commute*)
 next
  case *AndL* **thus** *?thesis* **using** *SCc.AndL Suc.IH* **by** *blast*
 next
  case *OrL* **thus** *?thesis* **using** *SCc.OrL Suc.IH* **by** *blast*
 next
  case *ImpL* **thus** *?thesis* **by** (*metis SCc.ImpL Suc.IH add-mset-commute*)
 qed
 qed
qed *blast*

lemma *Cut-Atom-depth*: *Atom* $k,\Gamma \Rightarrow \Delta \downarrow n \Longrightarrow \Gamma \Rightarrow$ *Atom* $k,\Delta \downarrow m \Longrightarrow \Gamma \Rightarrow \Delta \downarrow n + m$
proof(*induction Atom k*,Γ Δ *n arbitrary*: Γ *m rule*: *SCc.induct*)
 case (*BotL* Δ)
 hence $\bot \in\# \Gamma$ **by** *simp*
 thus *?case* **using** *SCc.BotL* **by** *auto*
next
 case (*Ax l* Δ)
 show *?case* **proof** *cases*
  assume $l = k$
  with ‹*Atom* $l \in\# \Delta$› **obtain** Δ' **where** $\Delta =$ *Atom* k, Δ' **by** (*meson multi-member-split*)
  with ‹$\Gamma \Rightarrow$ *Atom* $k, \Delta \downarrow m$› **have** $\Gamma \Rightarrow \Delta \downarrow m$ **using** *contract'* **by** *blast*
  thus *?thesis* **by** (*metis add.commute deeper*)
 next
  assume $l \neq k$
  with ‹*Atom* $l \in\#$ *Atom* k, Γ› **have** *Atom* $l \in\# \Gamma$ **by** *simp*
  with ‹*Atom* $l \in\# \Delta$› **show** *?thesis* **using** *SCc.Ax*[*of l*] **by** *simp*
 qed
next
 case (*NotL* Γ *F* Δ)
 obtain Γ' **where** Γ: $\Gamma =$ *Not F*, Γ' **by** (*meson NotL.hyps*(*3*) *add-eq-conv-ex formula.simps*(*9*))
 show *?case* **unfolding** Γ
  apply(*unfold plus-nat.add-Suc*)
  apply(*intro SCc.NotL*)
  apply(*intro NotL.hyps*)
   subgoal using *NotL* Γ **by** (*simp add*: *lem2*)

```
    subgoal using Γ NotL.prems NotL-inv′ by blast
  done
next
  case (NotR F Δ)
  then show ?case by(auto intro!: SCc.NotR dest!: NotR-inv′)
next
  case (AndL F G Γ Δ)
  obtain Γ′ where Γ: Γ = And F G, Γ′ by (metis AndL.hyps(3) add-eq-conv-diff
formula.distinct(5))
  show ?case unfolding Γ
    apply(unfold plus-nat.add-Suc)
    apply(intro SCc.AndL)
    apply(intro AndL.hyps )
     subgoal using AndL Γ by (simp add: lem2)
    subgoal using Γ AndL.prems AndL-inv′ by blast
  done
next
  case (AndR F Δ G)
  then show ?case
    using AndR-inv′ SCc.AndR by (metis add-Suc inR1′)
next
  case (OrL F Γ′ Δ n G)
  obtain Γ″ where Γ: Γ = Or F G, Γ″ by (meson OrL.hyps(5) add-eq-conv-ex
formula.simps(13))
  have ihm: F, Γ′ = Atom k, F, Γ″ G, Γ′ = Atom k, G, Γ″ using OrL Γ  by
(simp-all add: lem2)
  show ?case unfolding Γ
    apply(unfold plus-nat.add-Suc)
    apply(intro SCc.OrL OrL.hyps(2)[OF ihm(1)] OrL.hyps(4)[OF ihm(2)])
     subgoal using Γ OrL.prems OrL-inv′ by blast
    subgoal using Γ OrL.prems OrL-inv′ by blast
  done
next
  case (OrR F G Δ)
  then show ?case by(auto intro!: SCc.intros(3−) dest!: OrR-inv′)
next
  case (ImpL Γ′ F Δ n G)
  obtain Γ″ where Γ: Γ = Imp F G, Γ″ by (metis ImpL.hyps(5) add-eq-conv-ex
formula.simps)
  show ?case unfolding Γ
    apply(unfold plus-nat.add-Suc)
    apply(intro SCc.ImpL ImpL.hyps(2) ImpL.hyps(4))
       subgoal using ImpL Γ by (simp add: lem2)
      subgoal using Γ ImpL.prems by(auto dest!: ImpL-inv′)
     subgoal using ImpL Γ by (simp add: lem2)
    subgoal using Γ ImpL.prems ImpL-inv′ by blast
  done
next
  case (ImpR F G Δ)
```

then show *?case* **by** (*auto dest*!: *ImpR-inv′ intro*!: *SCc.ImpR*)
qed
primrec *cut-bound* :: *nat* ⇒ *nat* ⇒ *′a formula* ⇒ *nat* **where**
cut-bound n m (*Atom* -) = *m* + *n* |
cut-bound n m Bot = *n* |
cut-bound n m (*Not F*) = *cut-bound m n F* |
cut-bound n m (*And F G*) = *cut-bound n* (*cut-bound n m F*) *G* |
cut-bound n m (*Or F G*) = *cut-bound* (*cut-bound n m F*) *m G* |
cut-bound n m (*Imp F G*) = *cut-bound* (*cut-bound m n F*) *m G*
theorem *cut-bound*: Γ ⇒ *F*,Δ ↓ *n* ⟹ *F*,Γ ⇒ Δ ↓ *m* ⟹ Γ ⇒ Δ ↓ *cut-bound n m F*
proof(*induction F arbitrary*: Γ Δ *n m*)
case (*Atom k*) **thus** *?case* **using** *Cut-Atom-depth* **by** *simp fast*
next
case *Bot* **thus** *?case* **using** *Bot-delR′* **by** *fastforce*
next
case *Not* **from** *Not.prems* **show** *?case* **by**(*auto dest*!: *NotL-inv′ NotR-inv′ intro*!: *Not.IH elim*!: *weakenL*)
next
case (*And F G*) **from** *And.prems* **show** *?case* **by**(*auto dest*!: *AndL-inv′ AndR-inv′ intro*!: *And.IH elim*!: *weakenR′ weakenL′*)
next
case (*Or F G*) **from** *Or.prems* **show** *?case* **by**(*auto dest*: *OrL-inv′ OrR-inv′ intro*!: *Or.IH elim*!: *weakenR′ weakenL′*)
next
case (*Imp F G*)
from *ImpR-inv′* ‹Γ ⇒ *F* → *G*, Δ ↓ *n*› **have** *R*: *F*, Γ ⇒ *G*, Δ ↓ *n* **by** *blast*
from *ImpL-inv′* ‹*F* → *G*, Γ ⇒ Δ ↓ *m*› **have** *L*: Γ ⇒ *F*, Δ ↓ *m G*, Γ ⇒ Δ ↓ *m* **by** *blast+*
from *L*(*1*) **have** Γ ⇒ *F*, *G*, Δ ↓ *m* **using** *weakenR′* **by** *blast*
from *Imp.IH*(*1*)[*OF this R*] **have** Γ ⇒ *G*, Δ ↓ *cut-bound m n F* **.**
from *Imp.IH*(*2*)[*OF this L*(*2*)] **have** Γ ⇒ Δ ↓ *cut-bound* (*cut-bound m n F*) *m G* **.**
thus Γ ⇒ Δ ↓ *cut-bound n m* (*F* → *G*) **by** *simp*
qed

context begin
private primrec *cut-bound′* :: *nat* ⇒ *′a formula* ⇒ *nat* **where**
cut-bound′ n (*Atom* -) = *2*∗*n* |
cut-bound′ n Bot = *n* |
cut-bound′ n (*Not F*) = *cut-bound′ n F* |
cut-bound′ n (*And F G*) = *cut-bound′* (*cut-bound′ n F*) *G* |
cut-bound′ n (*Or F G*) = *cut-bound′* (*cut-bound′ n F*) *G* |
cut-bound′ n (*Imp F G*) = *cut-bound′* (*cut-bound′ n F*) *G*

private lemma *cut-bound′-mono*: *a* ≤ *b* ⟹ *cut-bound′ a F* ≤ *cut-bound′ b F*
by(*induction F arbitrary*: *a b*; *simp*)

private lemma *cut-bound-mono*: *a* ≤ *c* ⟹ *b* ≤ *d* ⟹ *cut-bound a b F* ≤

cut-bound c d F
  by(*induction F arbitrary*: *a b c d*; *simp*)

private lemma *cut-bound-max*: *max n* (*cut-bound′* (*max n m*) *F*) = *cut-bound′* (*max n m*) *F*
  by(*induction F arbitrary*: *n m*; *simp*; *metis*)
private lemma *cut-bound-max′*: *max n* (*cut-bound′ n F*) = *cut-bound′ n F*
  by(*induction F arbitrary*: *n* ; *simp*; *metis max.assoc*)

private lemma *cut-bound-′*: *cut-bound n m F* ≤ *cut-bound′* (*max n m*) *F*
proof(*induction F arbitrary*: *n m*)
  case (*Not F*)
  then show *?case* **by** *simp* (*metis max.commute*)
next
  case (*And F1 F2*)
  from *And.IH*(*1*) **have** *1*: *cut-bound n* (*cut-bound n m F1*) *F2* ≤ *cut-bound n* (*cut-bound′* (*max n m*) *F1*) *F2*
    by(*rule cut-bound-mono*[*OF order.refl*])
  also from *And.IH*(*2*) **have** ... ≤ *cut-bound′* (*max n* (*cut-bound′* (*max n m*) *F1*)) *F2* **by** *simp*
  also have ... = *cut-bound′* (*cut-bound′* (*max n m*) *F1*) *F2* **by** (*simp add*: *cut-bound-max*)
  finally show *?case* **by** *simp*
next
  case (*Or F1 F2*)
  from *Or.IH*(*1*) **have** *1*: *cut-bound* (*cut-bound n m F1*) *m F2* ≤ *cut-bound* (*cut-bound′* (*max n m*) *F1*) *m F2*
    by(*rule cut-bound-mono*[*OF - order.refl*])
  also from *Or.IH*(*2*)[*of cut-bound′* (*max n m*) *F1*] **have** ... ≤ *cut-bound′* (*max* (*cut-bound′* (*max n m*) *F1*) *m*) *F2* **by** *simp*
  also have ... = *cut-bound′* (*cut-bound′* (*max n m*) *F1*) *F2* **by** (*simp add*: *cut-bound-max max.commute*)
  finally show *?case* **by** *simp*
next
  case (*Imp F1 F2*)
  from *Imp.IH*(*1*) **have** *1*: *cut-bound* (*cut-bound m n F1*) *m F2* ≤ *cut-bound* (*cut-bound′* (*max m n*) *F1*) *m F2*
    by(*rule cut-bound-mono*[*OF - order.refl*])
  also from *Imp.IH*(*2*)[*of cut-bound′* (*max m n*) *F1*] **have** ... ≤ *cut-bound′* (*max* (*cut-bound′* (*max m n*) *F1*) *m*) *F2* **by** *simp*
  also have ... = *cut-bound′* (*cut-bound′* (*max n m*) *F1*) *F2* **by** (*simp add*: *cut-bound-max max.commute*)
  finally show *?case* **by** *simp*
qed *simp-all*

primrec *depth* :: *′a formula* ⇒ *nat* **where**
  depth (*Atom* -) = *0* |
  depth Bot = *0* |
  depth (*Not F*) = *Suc* (*depth F*) |

```
  depth (And F G) = Suc (max (depth F) (depth G)) |
  depth (Or F G) = Suc (max (depth F) (depth G)) |
  depth (Imp F G) = Suc (max (depth F) (depth G))

private primrec cbnd where
  cbnd k 0 = 2*k |
  cbnd k (Suc n) = cbnd (cbnd k n) n

private lemma cbnd-grow: (k :: nat) ≤ cbnd k d
  by(induction d arbitrary: k; simp) (insert le-trans, blast)

private lemma cbnd-mono: assumes b ≤ d shows cbnd (a::nat) b ≤ cbnd a d
proof −
  have cbnd (a::nat) b ≤ cbnd a (b + d) for b d
    by(induction d arbitrary: a b; simp) (insert le-trans cbnd-grow, blast)
  thus ?thesis using assms using le-Suc-ex by blast
qed

private lemma cut-bound'-cbnd: cut-bound' n F ≤ cbnd n (depth F)
proof(induction F arbitrary: n)
next
  case (Not F)
  then show ?case using cbnd-grow dual-order.trans by fastforce
next
  case (And F1 F2)
  let ?md = max (depth F1) (depth F2)
  have cut-bound' (cut-bound' n F1) F2 ≤ cut-bound' (cbnd n (depth F1)) F2 by
(simp add: And.IH(1) cut-bound'-mono)
 also have ... ≤ cut-bound' (cbnd n ?md) F2 by (simp add: cbnd-mono cut-bound'-mono)
  also have ... ≤ cbnd (cbnd n ?md) (depth F2) using And.IH(2) by blast
  also have ... ≤ cbnd (cbnd n ?md) ?md by (simp add: cbnd-mono)
  finally show ?case by simp
next
  case (Imp F1 F2)
  case (Or F1 F2)
```

analogous

```
qed simp-all

value map (cbnd (0::int)) [0,1,2,3,4]
value map (cbnd (1::int)) [0,1,2,3,4]
value map (cbnd (2::int)) [0,1,2,3,4]
value map (cbnd (3::int)) [0,1,2,3,4]
value [nbe] map (int ∘ (λn. n div 3) ∘ cut-bound 3 3 ∘ (λn. ((λF. And F F) ^^
n) (Atom 0))) [0,1,2,3,4,5,6,7]
value [nbe] map (int ∘ (λn. n div 3) ∘ cut-bound' 3 ∘ (λn. ((λF. And F F) ^^ n)
(Atom 0))) [0,1,2,3,4]

value [nbe] map (int ∘ (λn. n div 3) ∘ cut-bound 3 3 ∘ (λn. ((λF. Imp (Or F F)
```

(*And F F*)) ^^ *n*) (*Atom 0*))) [*0*,*1*,*2*]
value [*nbe*] *map* (*int* ∘ (λ*n*. *n div 3*) ∘ *cut-bound′ 3* ∘ (λ*n*. ((λ*F*. *Imp* (*Or F F*) (*And F F*)) ^^ *n*) (*Atom 0*))) [*0*,*1*,*2*]

value [*nbe*] (λ*F*. *And* (*Or F F*) (*Or F F*)) ^^ *2*

lemma *n* + ((*n* + *m*) ∗ *2* ^ (*size F* − *Suc 0*) +
(*n* + (*n* + *m* + (*n* + *m*) ∗ *2* ^ (*size F* − *Suc 0*))) ∗ *2* ^ (*size G* − *Suc 0*))
≤ (*n* + (*m* :: *nat*)) ∗ *2* ^ (*size F* + *size G*)
oops

lemma *cut-bound* (*n* :: *nat*) *m F* ≤ (*n* + *m*) ∗ (*2* ^ (*size F* − *1*) + *1*)
proof(*induction F arbitrary*: *n m*)
next
case (*Not F*)
show *?case* **unfolding** *cut-bound.simps* **by**(*rule le-trans*[*OF Not*]) (*simp add*: *add.commute*)
next
have *1* ≤ *size F* **for** *F* :: ′*a formula* **by**(*cases F*; *simp*)
case (*And F G*)
from *And*(*2*) **have** *cut-bound n* (*cut-bound n m F*) *G* ≤ (*n* + (*cut-bound n m F*)) ∗ (*2* ^ (*size G* − *1*) + *1*) **by** *simp*
also from *And*(*1*) **have** ... ≤ (*n* + (*n* + *m*) ∗ (*2* ^ (*size F* − *1*) + *1*)) ∗ (*2* ^ (*size G* − *1*) + *1*)
by (*meson add-le-cancel-left mult-le-mono1*)
also have ... ≤ (*n* + *m*) ∗ (*2* ^ (*size* (*F* ∧ *G*) − *1*) + *1*)
apply *simp*
oops

private lemma *cbnd-comm*: *cbnd* (*l* ∗ *k*::*nat*) *n* = *l* ∗ *cbnd* (*k*::*nat*) *n*
by(*induction n arbitrary*: *k*; *simp*)

private lemma *cbnd-closed*: *cbnd* (*k*::*nat*) *n* = *k* ∗ *2* ^ (*2* ^ *n*)
by(*induction n arbitrary*: *k*;*simp add*: *semiring-normalization-rules*(*26*))

theorem *cut′*: **assumes** Γ ⇒ *F*,Δ ↓ *n F*,Γ ⇒ Δ ↓ *n* **shows** Γ ⇒ Δ ↓ *n* ∗ *2* ^ (*2* ^ *depth F*)
proof −
from *cut-bound*[*OF assms*] **have** *c*: Γ ⇒ Δ ↓ *cut-bound n n F* **.**
have *d*: *cut-bound n n F* ≤ *max n n* ∗ *2* ^ (*2* ^ *depth F*)
using *cut-bound-′ cut-bound′-cbnd cbnd-closed* **by** (*metis order-trans*)
show *?thesis* **using** *c d le-Suc-ex deeper* **unfolding** *max.idem* **by** *metis*
qed

end

end

2.1.3 Mimicking the original

theory *SC-Gentzen*
imports *SC-Depth SC-Cut*
begin

This system attempts to mimic the original sequent calculus ("Reihen von Formeln, durch Kommata getrennt", translates roughly to sequences of formulas, separated by a comma) [4].

inductive *SCg* :: *'a formula list* ⇒ *'a formula list* ⇒ *bool* (**infix** ‹⇒› *30*) **where**
Anfang: $[\mathfrak{D}] \Rightarrow [\mathfrak{D}]$ |
FalschA: $[\bot] \Rightarrow []$ |
VerduennungA: $\Gamma \Rightarrow \Theta \Longrightarrow \mathfrak{D}\#\Gamma \Rightarrow \Theta$ |
VerduennungS: $\Gamma \Rightarrow \Theta \Longrightarrow \Gamma \Rightarrow \mathfrak{D}\#\Theta$ |
ZusammenziehungA: $\mathfrak{D}\#\mathfrak{D}\#\Gamma \Rightarrow \Theta \Longrightarrow \mathfrak{D}\#\Gamma \Rightarrow \Theta$ |
ZusammenziehungS: $\Gamma \Rightarrow \mathfrak{D}\#\mathfrak{D}\#\Theta \Longrightarrow \Gamma \Rightarrow \mathfrak{D}\#\Theta$ |
VertauschungA: $\Delta @ \mathfrak{D}\#\mathfrak{E}\#\Gamma \Rightarrow \Theta \Longrightarrow \Delta @ \mathfrak{E}\#\mathfrak{D}\#\Gamma \Rightarrow \Theta$ |
VertauschungS: $\Gamma \Rightarrow \Theta @ \mathfrak{E}\#\mathfrak{D}\#\Lambda \Longrightarrow \Gamma \Rightarrow \Theta @ \mathfrak{D}\#\mathfrak{E}\#\Lambda$ |
Schnitt: $[\![\Gamma \Rightarrow \mathfrak{D}\#\Theta; \mathfrak{D}\#\Delta \Rightarrow \Lambda]\!] \Longrightarrow \Gamma @ \Delta \Rightarrow \Theta @ \Lambda$ |
UES: $[\![\Gamma \Rightarrow \mathfrak{A}\#\Theta; \Gamma \Rightarrow \mathfrak{B}\#\Theta]\!] \Longrightarrow \Gamma \Rightarrow \mathfrak{A}\wedge\mathfrak{B}\#\Theta$ |
UEA1: $\mathfrak{A}\#\Gamma \Rightarrow \Theta \Longrightarrow \mathfrak{A}\wedge\mathfrak{B}\#\Gamma \Rightarrow \Theta$ | *UEA2*: $\mathfrak{B}\#\Gamma \Rightarrow \Theta \Longrightarrow \mathfrak{A}\wedge\mathfrak{B}\#\Gamma \Rightarrow \Theta$ |
OEA: $[\![\mathfrak{A}\#\Gamma \Rightarrow \Theta; \mathfrak{B}\#\Gamma \Rightarrow \Theta]\!] \Longrightarrow \mathfrak{A}\vee\mathfrak{B}\#\Gamma \Rightarrow \Theta$ |
OES1: $\Gamma \Rightarrow \mathfrak{A}\#\Theta \Longrightarrow \Gamma \Rightarrow \mathfrak{A}\vee\mathfrak{B}\#\Theta$ | *OES2*: $\Gamma \Rightarrow \mathfrak{B}\#\Theta \Longrightarrow \Gamma \Rightarrow \mathfrak{A}\vee\mathfrak{B}\#\Theta$ |
FES: $\mathfrak{A}\#\Gamma \Rightarrow \mathfrak{B}\#\Theta \Longrightarrow \Gamma \Rightarrow \mathfrak{A}\rightarrow\mathfrak{B}\#\Theta$ |
FEA: $[\![\Gamma \Rightarrow \mathfrak{A}\#\Theta; \mathfrak{B}\#\Delta \Rightarrow \Lambda]\!] \Longrightarrow \mathfrak{A}\rightarrow\mathfrak{B}\#\Gamma @ \Delta \Rightarrow \Theta @ \Lambda$ |
NES: $\mathfrak{A}\#\Gamma \Rightarrow \Theta \Longrightarrow \Gamma \Rightarrow \neg\mathfrak{A}\#\Theta$ |
NEA: $\Gamma \Rightarrow \mathfrak{A}\#\Theta \Longrightarrow \neg\mathfrak{A}\#\Gamma \Rightarrow \Theta$

Nota bene: E here stands for "Einführung", which is introduction and not elemination.

The rule for ⊥ is not part of the original calculus. Its addition is necessary to show equivalence to our *SCp*.

Note that we purposefully did not recreate the fact that Gentzen sometimes puts his principal formulas on end and sometimes on the beginning of the list.

lemma *AnfangTauschA*: $\mathfrak{D}\#\Delta @ \Gamma \Rightarrow \Theta \Longrightarrow \Delta @ \mathfrak{D}\#\Gamma \Rightarrow \Theta$
  by(*induction* Δ *arbitrary*: Γ *rule*: *List.rev-induct*) (*simp-all add*: *VertauschungA*)
lemma *AnfangTauschS*: $\Gamma \Rightarrow \mathfrak{D}\#\Delta @ \Theta \Longrightarrow \Gamma \Rightarrow \Delta @ \mathfrak{D}\#\Theta$
  by(*induction* Δ *arbitrary*: Θ *rule*: *List.rev-induct*) (*simp-all add*: *VertauschungS*)
lemma *MittenTauschA*: $\Delta @ \mathfrak{D}\#\Gamma \Rightarrow \Theta \Longrightarrow \mathfrak{D}\#\Delta @ \Gamma \Rightarrow \Theta$
  by(*induction* Δ *arbitrary*: Γ *rule*: *List.rev-induct*) (*simp-all add*: *VertauschungA*)
lemma *MittenTauschS*: $\Gamma \Rightarrow \Delta @ \mathfrak{D}\#\Theta \Longrightarrow \Gamma \Rightarrow \mathfrak{D}\#\Delta @ \Theta$
  by(*induction* Δ *arbitrary*: Θ *rule*: *List.rev-induct*) (*simp-all add*: *VertauschungS*)

lemma *BotLe*: ⊥∈*set* Γ ⟹ Γ⇒Δ
proof −
  have *A*: ⊥#Γ⇒[] **for** Γ **by**(*induction* Γ) (*simp-all add*: *FalschA VerduennungA VertauschungA*[**where** Δ=*Nil*, *simplified*])

```
  have ∗: ⊥#Γ⇒Δ for Γ by(induction Δ) (simp-all add: A VerduennungS)
  assume ⊥∈set Γ then obtain Γ1 Γ2 where Γ: Γ = Γ1@⊥#Γ2 by (meson
split-list)
  show ?thesis unfolding Γ using AnfangTauschA ∗ by blast
qed

lemma Axe: A ∈ set Γ ⟹ A ∈ set Δ ⟹ Γ ⇒ Δ
proof −
  have A: A#Γ ⇒ [A] for Γ by(induction Γ) (simp-all add: Anfang VertauschungA[where
Δ=Nil, simplified] VerduennungA)
  have S: A#Γ ⇒ A#Δ for Γ Δ by(induction Δ) (simp-all add: A Anfang Ver-
tauschungS[where Θ=Nil, simplified] VerduennungS)
  assume A ∈ set Γ A ∈ set Δ thus ?thesis
    apply(−)
    apply(drule split-list)+
    apply(clarify)
    apply(intro AnfangTauschA AnfangTauschS)
    apply(rule S)
  done
qed

lemma VerduennungListeA: Γ ⇒ Θ ⟹ Γ@Γ ⇒ Θ
proof −
  have Γ ⇒ Θ ⟹ ∃Γ''. Γ=Γ''@Γ' ⟹ Γ'@Γ ⇒ Θ for Γ'
  proof(induction Γ')
    case (Cons a as)
    then obtain Γ'' where Γ = Γ'' @ a # as by blast
    hence ∃Γ''. Γ = Γ'' @ as by(intro exI[where x=Γ'' @ [a]]) simp
    from Cons.IH[OF Cons.prems(1) this] have as @ Γ ⇒ Θ .
    thus ?case using VerduennungA by simp
  qed simp
  thus Γ ⇒ Θ ⟹ Γ@Γ ⇒ Θ by simp
qed
lemma VerduennungListeS: Γ ⇒ Θ ⟹ Γ ⇒ Θ@Θ
proof −
  have Γ ⇒ Θ ⟹ ∃Θ''. Θ=Θ''@Θ' ⟹ Γ ⇒ Θ'@Θ for Θ'
  proof(induction Θ')
    case (Cons a as)
    then obtain Θ'' where Θ = Θ'' @ a # as by blast
    hence ∃Θ''. Θ = Θ'' @ as by(intro exI[where x=Θ'' @ [a]]) simp
    from Cons.IH[OF Cons.prems(1) this] have Γ ⇒ as @ Θ .
    thus ?case using VerduennungS by simp
  qed simp
  thus Γ ⇒ Θ ⟹ Γ ⇒ Θ@Θ by simp
qed

lemma ZusammenziehungListeA: Γ@Γ ⇒ Θ ⟹ Γ ⇒ Θ
proof −
  have Γ'@Γ ⇒ Θ ⟹ ∃Γ''. Γ=Γ''@Γ' ⟹ Γ ⇒ Θ for Γ'
```

proof(*induction* Γ')
case (*Cons a* Γ')
then obtain Γ'' **where** Γ'': $\Gamma = \Gamma''$ @ a # Γ' **by** *blast*
then obtain $\Gamma 1$ $\Gamma 2$ **where** Γ: $\Gamma = \Gamma 1$ @ a # $\Gamma 2$ **by** *blast*
from Γ'' **have** $**$: $\exists \Gamma''. \Gamma = \Gamma''$ @ Γ' **by**(*intro exI*[**where** x=Γ'' @ [a]]) *simp*
from *Cons.prems*(*1*) **have** a # (a # Γ') @ $\Gamma 1$ @ $\Gamma 2 \Rightarrow \Theta$ **unfolding** Γ **using** *MittenTauschA* **by** (*metis append-assoc*)
hence (a # Γ') @ $\Gamma 1$ @ $\Gamma 2 \Rightarrow \Theta$ **using** *ZusammenziehungA* **by** *auto*
hence Γ' @ $\Gamma \Rightarrow \Theta$ **unfolding** Γ **using** *AnfangTauschA* **by** (*metis append-Cons append-assoc*)
from *Cons.IH*[*OF this* $**$] **show** $\Gamma \Rightarrow \Theta$ **.**
qed *simp*
thus Γ@$\Gamma \Rightarrow \Theta \Longrightarrow \Gamma \Rightarrow \Theta$ **by** *simp*
qed
lemma *ZusammenziehungListeS*: $\Gamma \Rightarrow \Theta$@$\Theta \Longrightarrow \Gamma \Rightarrow \Theta$
proof −
have $\Gamma \Rightarrow \Theta'$@$\Theta \Longrightarrow \exists \Theta''. \Theta$=$\Theta''$@$\Theta' \Longrightarrow \Gamma \Rightarrow \Theta$ **for** Θ'
proof(*induction* Θ')
case (*Cons a* Θ')
then obtain Θ'' **where** Θ'': $\Theta = \Theta''$ @ a # Θ' **by** *blast*
then obtain $\Theta 1$ $\Theta 2$ **where** Θ: $\Theta = \Theta 1$ @ a # $\Theta 2$ **by** *blast*
from Θ'' **have** $**$: $\exists \Theta''. \Theta = \Theta''$ @ Θ' **by**(*intro exI*[**where** x=Θ'' @ [a]]) *simp*
from *Cons.prems*(*1*) **have** $\Gamma \Rightarrow a$ # (a # Θ') @ $\Theta 1$ @ $\Theta 2$ **unfolding** Θ **using** *MittenTauschS* **by** (*metis append-assoc*)
hence $\Gamma \Rightarrow$ (a # Θ') @ $\Theta 1$ @ $\Theta 2$ **using** *ZusammenziehungS* **by** *auto*
hence $\Gamma \Rightarrow \Theta'$ @ Θ **unfolding** Θ **using** *AnfangTauschS* **by** (*metis append-Cons append-assoc*)
from *Cons.IH*[*OF this* $**$] **show** $\Gamma \Rightarrow \Theta$ **.**
qed *simp*
thus $\Gamma \Rightarrow \Theta$@$\Theta \Longrightarrow \Gamma \Rightarrow \Theta$ **by** *simp*
qed

theorem *gentzen-sc-eq*: *mset* $\Gamma \Rightarrow$ *mset* $\Delta \longleftrightarrow \Gamma \Rightarrow \Delta$ **proof**
assume *mset* $\Gamma \Rightarrow$ *mset* Δ
then obtain n **where** *mset* $\Gamma \Rightarrow$ *mset* $\Delta \downarrow n$ **unfolding** *SC-SCp-eq*[*symmetric*] **..**
thus $\Gamma \Rightarrow \Delta$

proof(*induction n arbitrary*: Γ Δ *rule*: *nat.induct*)
case (*Suc n*)
have *sr*: $\exists \Gamma 1$ $\Gamma 2$. $\Gamma = \Gamma 1$ @ F # $\Gamma 2 \wedge \Gamma' =$ *mset* ($\Gamma 1$@$\Gamma 2$) (**is** *?s*) **if** *mset* Γ = F, Γ' **for** Γ Γ' F **proof** −
from *that* **obtain** $\Gamma 1$ $\Gamma 2$ **where** Γ: $\Gamma = \Gamma 1$ @ F # $\Gamma 2$ **by** (*metis split-list add.commute ex-mset list.set-intros*(*1*) *mset.simps*(*2*) *set-mset-mset*)
hence Γ': $\Gamma' =$ *mset* ($\Gamma 1$@$\Gamma 2$) **using** *that* **by** *auto*
show *?s* **using** Γ Γ' **by** *blast*
qed
from *Suc.prems* **show** *?case* **proof**(*cases rule*: *SCc.cases*)
case *BotL* **thus** *?thesis* **using** *BotLe* **by** *simp*

next
case *Ax* **thus** *?thesis* **using** *Axe* **by** *simp*
next
case (*NotL* Γ′ *F*)
from ‹*mset* Γ = ¬ *F*, Γ′› **obtain** Γ*1* Γ*2* **where** Γ: Γ = Γ*1* @ ¬ *F* # Γ*2*
by (*metis split-list add.commute ex-mset list.set-intros*(*1*) *mset.simps*(*2*) *set-mset-mset*)
hence Γ′: Γ′ = *mset* (Γ*1*@Γ*2*) **using** *NotL*(*1*) **by** *simp*
from ‹Γ′ ⇒ *F*, *mset* Δ ↓ *n*› **have** *mset* (Γ*1*@Γ*2*) ⇒ *mset* (*F*#Δ) ↓ *n* **unfolding** Γ′ **by** (*simp add*: *add.commute*)
from *Suc.IH*[*OF this*] **show** *?thesis* **unfolding** Γ **using** *AnfangTauschA NEA* **by** *blast*
next
case (*NotR F* Δ′)
from *sr*[*OF NotR*(*1*)] **obtain** Δ*1* Δ*2* **where** Δ: Δ = Δ*1* @ ¬ *F* # Δ*2* ∧ Δ′ = *mset* (Δ*1* @ Δ*2*)
by *blast*
with *NotR* **have** *mset* (*F*#Γ) ⇒ *mset* (Δ*1*@Δ*2*) ↓ *n* **by** (*simp add*: *add.commute*)
from *Suc.IH*[*OF this*] **show** *?thesis* **using** Δ **using** *AnfangTauschS NES* **by** *blast*
next
case (*AndR F* Δ′ *G*)
from *sr*[*OF AndR*(*1*)] **obtain** Δ*1* Δ*2* **where** Δ: Δ = Δ*1* @ *F* ∧ *G* # Δ*2* ∧ Δ′ = *mset* (Δ*1* @ Δ*2*)
by *blast*
with *AndR* **have** *mset* Γ ⇒ *mset* (*F* # Δ*1*@Δ*2*) ↓ *n* *mset* Γ ⇒ *mset* (*G* # Δ*1*@Δ*2*) ↓ *n* **by** (*simp add*: *add.commute*)+
from *this*[*THEN Suc.IH*] **show** *?thesis* **using** Δ **using** *AnfangTauschS UES* **by** *blast*
next
case (*OrR F G* Δ′)
from *sr*[*OF OrR*(*1*)] **obtain** Δ*1* Δ*2* **where** Δ: Δ = Δ*1* @ *F* ∨ *G* # Δ*2* ∧ Δ′ = *mset* (Δ*1* @ Δ*2*)
by *blast*
with *OrR* **have** *mset* Γ ⇒ *mset* (*G* # *F* # Δ*1*@Δ*2*) ↓ *n* **by** (*simp add*: *add.commute add.left-commute add-mset-commute*)
from *this*[*THEN Suc.IH*] **have** Γ ⇒ *G* # *F* # Δ*1* @ Δ*2* **.**
with *OES2* **have** Γ ⇒ *F* ∨ *G* # *F* # Δ*1* @ Δ*2* **.**
with *VertauschungS*[**where** Θ=*Nil*, *simplified*] **have** Γ ⇒ *F* # *F* ∨ *G* # Δ*1* @ Δ*2* **.**
with *OES1* **have** Γ ⇒ *F* ∨ *G* # *F* ∨ *G* # Δ*1* @ Δ*2* **.**
hence Γ ⇒ *F* ∨ *G* # Δ*1* @ Δ*2* **using** *ZusammenziehungS* **by** *fast*
thus *?thesis* **unfolding** Δ[*THEN conjunct1*] **using** *AnfangTauschS* **by** *blast*
next
case (*ImpR F G* Δ′)
from *sr*[*OF ImpR*(*1*)] **obtain** Δ*1* Δ*2* **where** Δ: Δ = Δ*1* @ *F* → *G* # Δ*2* ∧ Δ′ = *mset* (Δ*1* @ Δ*2*)
by *blast*

with *ImpR* **have** *mset* (*F*#Γ) ⇒ *mset* (*G* # Δ*1*@Δ*2*) ↓ *n* **by** (*simp add*: *add.commute*)
from *this*[*THEN Suc.IH*] **show** *?thesis* **using** Δ **using** *AnfangTauschS FES* **by** *blast*
next
case (*AndL F G* Γ′)
from *sr*[*OF this*(*1*)] **obtain** Γ*1* Γ*2* **where** Γ: Γ = Γ*1* @ *F* ∧ *G* # Γ*2* ∧ Γ′ = *mset* (Γ*1* @ Γ*2*)
by *blast*
with *AndL* **have** *mset* (*G* # *F* # Γ*1*@Γ*2*) ⇒ *mset* Δ ↓ *n* **by** (*simp add*: *add.commute add.left-commute add-mset-commute*)
from *this*[*THEN Suc.IH*] **have** *G* # *F* # Γ*1* @ Γ*2* ⇒ Δ **.**
with *UEA2* **have** *F* ∧ *G* # *F* # Γ*1* @ Γ*2* ⇒ Δ **.**
with *VertauschungA*[**where** Δ=*Nil*, *simplified*] **have** *F* # *F* ∧ *G* # Γ*1* @ Γ*2* ⇒ Δ **.**
with *UEA1* **have** *F* ∧ *G* # *F* ∧ *G* # Γ*1* @ Γ*2* ⇒ Δ **.**
hence *F* ∧ *G* # Γ*1* @ Γ*2* ⇒ Δ **using** *ZusammenziehungA* **by** *fast*
thus *?thesis* **unfolding** Γ[*THEN conjunct1*] **using** *AnfangTauschA* **by** *blast*
next
case (*OrL F* Δ′ *G*)
from *sr*[*OF this*(*1*)] **obtain** Γ*1* Γ*2* **where** Γ: Γ = Γ*1* @ *F* ∨ *G* # Γ*2* ∧ Δ′ = *mset* (Γ*1* @ Γ*2*)
by *blast*
with *OrL* **have** *mset* (*F* # Γ*1*@Γ*2*) ⇒ *mset* Δ ↓ *n mset* (*G* # Γ*1*@Γ*2*) ⇒ *mset* Δ ↓ *n* **by** (*simp add*: *add.commute*)+
from *this*[*THEN Suc.IH*] **show** *?thesis* **using** Γ **using** *AnfangTauschA OEA* **by** *blast*
next
case (*ImpL* Γ′ *F G*)
from *sr*[*OF this*(*1*)] **obtain** Γ*1* Γ*2* **where** Γ: Γ = Γ*1* @ *F* → *G* # Γ*2* ∧ Γ′ = *mset* (Γ*1* @ Γ*2*)
by *blast*
with *ImpL* **have** *mset* (Γ*1*@Γ*2*) ⇒ *mset* (*F*#Δ) ↓ *n mset* (*G* # Γ*1*@Γ*2*) ⇒ *mset* Δ ↓ *n* **by** (*simp add*: *add.commute*)+
from *this*[*THEN Suc.IH*] **have** Γ*1* @ Γ*2* ⇒ *F* # Δ *G* # Γ*1* @ Γ*2* ⇒ Δ **.**
from *FEA*[*OF this*] **have** *F* → *G* # (Γ*1* @ Γ*2*) @ (Γ*1* @ Γ*2*) ⇒ Δ @ Δ **.**
hence *F* → *G* # (Γ*1* @ Γ*2*) @ (*F* → *G* # Γ*1* @ Γ*2*) ⇒ Δ @ Δ **using** *AnfangTauschA VerduennungA* **by** *blast*
hence *F* → *G* # (Γ*1* @ Γ*2*) ⇒ Δ @ Δ **using** *ZusammenziehungListeA*[**where** Γ=*F* → *G* # (Γ*1* @ Γ*2*)] **by** *simp*
thus *?thesis* **unfolding** Γ[*THEN conjunct1*] **by**(*intro AnfangTauschA*; *elim ZusammenziehungListeS*)
qed
qed *blast*
next
have *mset-Cons*[*simp*]: *mset* (*A* # *S*) = *A*, *mset S* **for** *A*::′*a formula* **and** *S* **by** (*simp add*: *add.commute*)
note *mset.simps*(*2*)[*simp del*]
show Γ ⇒ Δ ⟹ *mset* Γ ⇒ *mset* Δ **proof**(*induction rule*: *SCg.induct*)

case (*Anfang* 𝔇) **thus** *?case* **using** *extended-Ax SC-SCp-eq* **by** *force*
next
case (*FalschA*) **thus** *?case* **using** *SCp.BotL* **by** *force*
next
case (*VerduennungA* Γ Θ 𝔇) **thus** *?case* **by** (*simp add*: *SC.weakenL*)
next
case (*VerduennungS* Γ Θ 𝔇) **thus** *?case* **by** (*simp add*: *SC.weakenR*)
next
case (*ZusammenziehungA* 𝔇 Γ Θ) **thus** *?case* **using** *contractL* **by** *force*
next
case (*ZusammenziehungS* Γ 𝔇 Θ) **thus** *?case* **using** *contract* **by** *force*
next
case (*VertauschungA* Δ 𝔇 𝔈 Γ Θ) **thus** *?case* **by** *fastforce*
next
case (*VertauschungS* Γ Θ 𝔈 𝔇 Λ) **thus** *?case* **by** *fastforce*
next
case (*Schnitt* Γ 𝔇 Θ Δ Λ)
hence *mset* Γ ⇒ 𝔇,*mset* Θ 𝔇,*mset* Δ ⇒ *mset* Λ **using** *SC-SCp-eq* **by** *auto*
from *cut-cf*[*OF this*] **show** *?case* **unfolding** *SC-SCp-eq* **by** *simp*
next
case (*UES* Γ 𝔄 Θ 𝔅) **thus** *?case* **using** *SCp.AndR* **by** (*simp add*: *SC-SCp-eq*)
next
case (*UEA1* 𝔄 Γ Θ 𝔅)
from ‹*mset* (𝔄 # Γ) ⇒ *mset* Θ› **have** 𝔄,𝔅,*mset* Γ ⇒ *mset* Θ **using** *SC.weakenL* **by** *auto*
thus *?case* **using** *SCp.AndL* **by** *force*
next
case (*UEA2* 𝔅 Γ Θ 𝔄)
from ‹*mset* (𝔅 # Γ) ⇒ *mset* Θ› **have** 𝔄,𝔅,*mset* Γ ⇒ *mset* Θ **using** *SC.weakenL* **by** *auto*
thus *?case* **using** *SCp.AndL* **by** *force*
next
case (*OEA* 𝔄 Γ Θ 𝔅) **thus** *?case* **unfolding** *SC-SCp-eq* **by** (*simp add*: *SCp.OrL*)
next
case (*OES1* Γ 𝔄 Θ 𝔅) **thus** *?case* **using** *SC.weakenR*[**where** ′*a*=′*a*] **by**(*auto intro*!: *SCp.intros*(*3*−))
next
case (*OES2* Γ 𝔅 Θ 𝔄) **thus** *?case* **by** (*simp add*: *SC.weakenR SCp.OrR*)
next
case (*FES* 𝔄 Γ 𝔅 Θ) **thus** *?case* **using** *weakenR* **unfolding** *SC-SCp-eq* **by** (*simp add*: *SCp.ImpR*)
next
case (*FEA* Γ 𝔄 Θ 𝔅 Δ Λ)
from ‹*mset* Γ ⇒ *mset* (𝔄 # Θ)›[*THEN weakenL-set*, *THEN weakenR-set*, *of mset* Δ *mset* Λ]
have *S*: *mset* (Γ@Δ) ⇒ 𝔄,*mset* (Θ@Λ) **unfolding** *mset-append mset-Cons* **by** (*simp add*: *add-ac*)

from *FEA* **obtain** *m* **where** *mset* (𝔅 # Δ) ⇒ *mset* Λ **by** *blast*
hence *mset* Γ + *mset* (𝔅 # Δ) ⇒ *mset* Θ + *mset* Λ **using** *weakenL-set weakenR-set* **by** *fast*
hence *A*: 𝔅,*mset* (Γ@Δ) ⇒ *mset* (Θ@Λ) **by** (*simp add*: *add.left-commute*)
show *?case* **using** *S A SC-SCp-eq SCp.ImpL* **unfolding** *mset-Cons* **by** *blast*
next
case (*NES* 𝔄 Γ Θ) **thus** *?case* **using** *SCp.NotR* **by**(*simp add*: *SC-SCp-eq*)
next
case (*NEA* Γ 𝔄 Θ) **thus** *?case* **using** *SCp.NotL* **by**(*simp add*: *SC-SCp-eq*)
qed
qed

end

2.1.4 Soundness, Completeness

theory *SC-Sema*
imports *SC Sema*
begin

definition *sequent-semantics* :: *'a valuation* ⇒ *'a formula multiset* ⇒ *'a formula multiset* ⇒ *bool* (‹(- ⊨ (- ⇒/ -))› [*53*, *53*,*53*] *53*) **where**
𝒜 ⊨ Γ ⇒ Δ ≡ (∀γ ∈# Γ. 𝒜 ⊨ γ) ⟶ (∃δ ∈# Δ. 𝒜 ⊨ δ)
abbreviation *sequent-valid* :: *'a formula multiset* ⇒ *'a formula multiset* ⇒ *bool* (‹(⊨ (- ⇒/ -))› [*53*,*53*] *53*) **where**
⊨ Γ ⇒ Δ ≡ ∀*A*. *A* ⊨ Γ ⇒ Δ
abbreviation *sequent-nonvalid* :: *'a valuation* ⇒ *'a formula multiset* ⇒ *'a formula multiset* ⇒ *bool* (‹(- ¬⊨ (- ⇒/ -))› [*53*, *53*,*53*] *53*) **where**
𝒜 ¬⊨ Γ ⇒ Δ ≡ ¬𝒜⊨ Γ ⇒ Δ

lemma *sequent-intuitonistic-semantics*: ⊨ Γ ⇒ {#δ#} ⟷ *set-mset* Γ ⊫ δ
unfolding *sequent-semantics-def entailment-def* **by** *simp*

lemma *SC-soundness*: Γ ⇒ Δ ⟹ ⊨ Γ ⇒ Δ
by(*induction rule*: *SCp.induct*) (*auto simp add*: *sequent-semantics-def*)

definition *sequent-cost* Γ Δ = *Suc* (*sum-list* (*sorted-list-of-multiset* (*image-mset size* (Γ + Δ))))

function(*sequential*)
sc :: *'a formula list* ⇒ *'a list* ⇒ *'a formula list* ⇒ *'a list* ⇒ (*'a list* × *'a list*) *set*
where
sc (⊥ # Γ) *A* Δ *B* = {} |
sc [] *A* [] *B* = (*if set A* ∩ *set B* = {} *then* {(*remdups A*,*remdups B*)} *else* {}) |
sc (*Atom k* # Γ) *A* Δ *B* = *sc* Γ (*k*#*A*) Δ *B* |
sc (*Not F* # Γ) *A* Δ *B* = *sc* Γ *A* (*F*#Δ) *B* |
sc (*And F G* # Γ) *A* Δ *B* = *sc* (*F*#*G*#Γ) *A* Δ *B* |
sc (*Or F G* # Γ) *A* Δ *B* = *sc* (*F*#Γ) *A* Δ *B* ∪ *sc* (*G*#Γ) *A* Δ *B* |

sc (*Imp F G* # Γ) *A* Δ *B* = *sc* Γ *A* (*F*#Δ) *B* ∪ *sc* (*G*#Γ) *A* Δ *B* |
sc Γ *A* (⊥#Δ) *B* = *sc* Γ *A* Δ *B* |
sc Γ *A* (*Atom k* # Δ) *B* = *sc* Γ *A* Δ (*k*#*B*) |
sc Γ *A* (*Not F* # Δ) *B* = *sc* (*F*#Γ) *A* Δ *B* |
sc Γ *A* (*And F G* # Δ) *B* = *sc* Γ *A* (*F*#Δ) *B* ∪ *sc* Γ *A* (*G*#Δ) *B* |
sc Γ *A* (*Or F G* # Δ) *B* = *sc* Γ *A* (*F*#*G*#Δ) *B* |
sc Γ *A* (*Imp F G* # Δ) *B* = *sc* (*F*#Γ) *A* (*G*#Δ) *B*
  by *pat-completeness auto*

definition *list-sequent-cost* Γ Δ = *2*∗*sum-list* (*map size* (Γ@Δ)) + *length* (Γ@Δ)
termination *sc* **by** (*relation measure* (λ(Γ,*A*,Δ,*B*). *list-sequent-cost* Γ Δ)) (*simp-all add*: *list-sequent-cost-def*)

lemma *sc* [] [] [((*Atom 0* → *Atom 1*) → *Atom 0*) → *Atom 1*] [] = {([*0*], [*1* :: *nat*])}

by *code-simp*

lemma *sc-sim*:
  fixes Γ Δ :: *'a formula list* **and** *G D* :: *'a list*
  assumes *sc* Γ *A* Δ *B* = {}
  shows *image-mset Atom* (*mset A*) + *mset* Γ ⇒ *image-mset Atom* (*mset B*) + *mset* Δ
proof −
  have ∗[*simp*]: *image-mset Atom* (*mset A*) ⇒ *image-mset Atom* (*mset B*) (**is** *?k*) **if** *k* ∈ *set A k* ∈ *set B* **for** *A B* :: *'a list* **and** *k*
  proof −
    from *that* **obtain** *a* **where** *a* ∈ *set A a* ∈ *set B* **by** *blast*
    thus *?k* **by**(*force simp*: *in-image-mset intro*: *SCp.Ax*[**where** *k*=*a*])
  qed
  from *assms* **show** *?thesis*
    by(*induction rule*: *sc.induct*[**where** *'a*=*'a*]) (*auto*
        simp add: *list-sequent-cost-def add.assoc Bot-delR-simp*
        split: *if-splits option.splits*
        intro: *SCp.intros*(*3*−))
qed

lemma *scc-ce-distinct*:
  (*C*,*E*) ∈ *sc* Γ *G* Δ *D* ⟹ *set C* ∩ *set E* = {}
by(*induction* Γ *G* Δ *D arbitrary*: *C E rule*: *sc.induct*)
  (*fastforce split*: *if-splits*)+

Completeness set aside, this is an interesting fact on the side: Sequent Calculus can provide counterexamples.

theorem *SC-counterexample*:
  (*C*,*D*) ∈ *sc* Γ *A* Δ *B* ⟹
  (λ*a*. *a* ∈ *set C*) ¬⊨ *image-mset Atom* (*mset A*) + *mset* Γ ⇒ *image-mset Atom* (*mset B*) + *mset* Δ

```
by(induction rule: sc.induct[where 'a='a];
   simp add: sequent-semantics-def split: if-splits;
    blast)

corollary SC-counterexample':
  assumes (C,D) ∈ sc Γ [] Δ []
  shows (λk. k ∈ set C) ¬⊨ mset Γ ⇒ mset Δ
using SC-counterexample[OF assms] by simp

theorem SC-sound-complete: Γ ⇒ Δ ⟷ ⊨ Γ ⇒ Δ
proof
  assume Γ ⇒ Δ thus ⊨ Γ ⇒ Δ using SC-soundness by blast
next
  obtain Γ' Δ' where [simp]: Γ = mset  Γ' Δ = mset Δ' by (metis ex-mset)
  assume ⊨ Γ ⇒ Δ
  hence sc Γ' [] Δ' [] = {}
  proof(rule contrapos-pp)
    assume sc Γ' [] Δ' [] ≠ {}
    then obtain C E where (C,E) ∈ sc Γ' [] Δ' [] by fast
    thus ¬ ⊨ Γ ⇒ Δ using SC-counterexample' by fastforce
  qed
  from sc-sim[OF this] show Γ ⇒ Δ by auto
qed

theorem ⊨ Γ ⇒ Δ ⟹ Γ ⇒ Δ
proof –
  assume s: ⊨ Γ ⇒ Δ
  obtain Γ' Δ' where p: Γ = mset Γ' Δ = mset Δ' by (metis ex-mset)
  have mset Γ' ⇒ mset Δ'
   proof cases — just to show that we didn't need to show the lemma above by
contraposition. It's just quicker to do so.
    assume sc Γ' [] Δ' [] = {}
    from sc-sim[OF this] show mset Γ' ⇒ mset Δ' by auto
  next
    assume sc Γ' [] Δ' [] ≠ {}
    with SC-counterexample have ¬ ⊨ mset Γ' ⇒ mset Δ' by fastforce
    moreover note s[unfolded p]
    ultimately have False ..
    thus mset Γ' ⇒ mset Δ' ..
  qed
  thus ?thesis unfolding p .
qed



end
theory SC-Depth-Limit
imports SC-Sema SC-Depth
begin
```

lemma *SC-completeness*: $\models \Gamma \Rightarrow \Delta \Longrightarrow \Gamma \Rightarrow \Delta \downarrow$ *sequent-cost* Γ Δ
proof(*induction sequent-cost* Γ Δ *arbitrary*: Γ Δ)
  case *0* **hence** *False* **by**(*simp add*: *sequent-cost-def*) **thus** *?case* **by** *clarify*
next
  case (*Suc n*)
  from *Suc*(*3*) **show** *?case*
    using *SCc.cases*[*OF Suc.hyps*(*1*)]
oops

Making this proof of completeness go through should be possible, but finding the right way to split the cases could get verbose. The variant with the search procedure is a lot more elegant.

lemma *sc-sim-depth*:
  assumes *sc* Γ *A* Δ *B* = {}
  shows *image-mset Atom* (*mset A*) + *mset* $\Gamma \Rightarrow$ *image-mset Atom* (*mset B*) + *mset* $\Delta \downarrow$ *sum-list* (*map size* (Γ@Δ)) + (*if set A* $\cap$ *set B* = {} *then 0 else 1*)
proof −
  have [*simp*]: *image-mset Atom* (*mset A*) $\Rightarrow$ *image-mset Atom* (*mset B*) $\downarrow$ *Suc 0* (**is** *?k*) **if** *set A* $\cap$ *set B* $\neq$ {} **for** *A B*
  proof −
    from *that* **obtain** *a* **where** *a* $\in$ *set A* *a* $\in$ *set B* **by** *blast*
    thus *?k* **by**(*force simp*: *in-image-mset intro*: *SCc.Ax*[**where** *k*=*a*])
  qed
  note *SCc.intros*(*3*−)[*intro*]
  have [*elim*!]: $\Gamma \Rightarrow \Delta \downarrow n \Longrightarrow n \leq m \Longrightarrow \Gamma \Rightarrow \Delta \downarrow m$ **for** Γ Δ *n m* **using** *dec-induct* **by**(*fastforce elim*!: *deeper-suc*)
  from *assms* **show** *?thesis*
    by(*induction* Γ *A* Δ *B rule*: *sc.induct*)
      (*auto*
      simp add: *list-sequent-cost-def add.assoc deeper-suc weakenR*′
      split: *if-splits option.splits*)
qed

corollary *sc-depth-complete*:
  assumes *s*: $\models \Gamma \Rightarrow \Delta$
  shows $\Gamma \Rightarrow \Delta \downarrow$ *sum-mset* (*image-mset size* (Γ+Δ))
proof −
  obtain Γ' Δ' **where** *p*: Γ = *mset* Γ' Δ = *mset* Δ' **by** (*metis ex-mset*)
  with *s* **have** *sl*: $\models$ *mset* $\Gamma' \Rightarrow$ *mset* Δ' **by** *simp*
  let *?d* = *sum-mset* (*image-mset size* (Γ+Δ))
  have *d*: *?d* = *sum-list* (*map size* (Γ'@Δ'))
    unfolding *p* **by** (*metis mset-append mset-map sum-mset-sum-list*)
  have *mset* $\Gamma' \Rightarrow$ *mset* $\Delta' \downarrow$ *?d*
  proof *cases*
    assume *sc* Γ' [] Δ' [] = {}
    from *sc-sim-depth*[*OF this*] **show** *mset* $\Gamma' \Rightarrow$ *mset* $\Delta' \downarrow$ *?d* **unfolding** *d* **by** *auto*
  next

assume *sc* Γ' [] Δ' [] $\neq$ {}
with *SC-counterexample* **have** $\neg \models$ *mset* $\Gamma' \Rightarrow$ *mset* Δ' **by** *fastforce*
moreover note *s*[*unfolded p*]
ultimately have *False* **..**
thus *mset* $\Gamma' \Rightarrow$ *mset* $\Delta' \downarrow$ *?d* **..**
qed
thus *?thesis* **unfolding** *p* **.**
qed

end
theory *SC-Compl-Consistency*
imports *Consistency SC-Cut SC-Sema*
begin

context begin
private lemma *reasonable*:
$\forall \Gamma'. F \quad \triangleright$ *set-mset* $\Gamma =$ *set-mset* $\Gamma' \longrightarrow P\ \Gamma' \Longrightarrow P\ (F, \Gamma)$
$\forall \Gamma'. F \triangleright G \triangleright$ *set-mset* $\Gamma =$ *set-mset* $\Gamma' \longrightarrow P\ \Gamma' \Longrightarrow P\ (F, G, \Gamma)$ **by** *simp-all*

lemma *SC-consistent*: *pcp* {*set-mset* Γ| Γ. $\neg(\Gamma \Rightarrow$ {#})}
unfolding *pcp-def*
apply(*intro ballI conjI*; *erule contrapos-pp*; *clarsimp*; ((*drule reasonable*)+)*?*)
apply(*auto dest*!: *NotL-inv AndL-inv OrL-inv ImpL-inv NotR-inv AndR-inv OrR-inv ImpR-inv multi-member-split contractL contractR intro*!: *SCp.intros*(*3*−) *intro*: *contractR contractL*)
apply (*metis add-mset-commute contract*)

done

end

lemma
fixes Γ Δ :: *'a* :: *countable formula multiset*
shows $\models \Gamma \Rightarrow \Delta \Longrightarrow \Gamma \Rightarrow \Delta$
proof(*erule contrapos-pp*)
have *NotInv*: Γ + *image-mset Not* $\Delta \Rightarrow$ {#} $\Longrightarrow \Gamma \Rightarrow \Delta$
by (*induction* Δ *arbitrary*: Γ; *simp add*: *NotL-inv*)
assume ‹$\neg$ $\Gamma \Rightarrow \Delta$›
hence ‹$\neg$ Γ + *image-mset Not* $\Delta \Rightarrow$ {#}› **using** *NotInv* **by** *blast*
with *pcp-sat*[*OF SC-consistent*]
have *sat* (*set-mset* (Γ + *image-mset* $\neg$ Δ)) **by** *blast*
thus ‹$\neg$ ($\models \Gamma \Rightarrow \Delta$)› **unfolding** *sat-def sequent-semantics-def not-all* **by** (*force elim*!: *ex-forward*)
qed

end

2.2 Natural Deduction

theory *ND*
imports *Formulas*
begin

inductive *ND* :: *'a formula set* ⇒ *'a formula* ⇒ *bool* (**infix** ‹⊢› *30*) **where**
Ax: $F \in \Gamma \Longrightarrow \Gamma \vdash F$ |
NotE: $[\![\Gamma \vdash Not\ F;\ \Gamma \vdash F\ ]\!] \Longrightarrow \Gamma \vdash \bot$ |
NotI: $F \triangleright \Gamma \vdash \bot \Longrightarrow \Gamma \vdash Not\ F$ |
CC: $Not\ F \triangleright \Gamma \vdash \bot \Longrightarrow \Gamma \vdash F$ |
AndE1: $\Gamma \vdash And\ F\ G \Longrightarrow \Gamma \vdash F$ |
AndE2: $\Gamma \vdash And\ F\ G \Longrightarrow \Gamma \vdash G$ |
AndI: $[\![\ \Gamma \vdash F;\ \Gamma \vdash G\ ]\!] \Longrightarrow \Gamma \vdash And\ F\ G$ |
OrI1: $\Gamma \vdash F \Longrightarrow \Gamma \vdash Or\ F\ G$ |
OrI2: $\Gamma \vdash G \Longrightarrow \Gamma \vdash Or\ F\ G$ |
OrE: $[\![\ \Gamma \vdash Or\ F\ G;\ F \triangleright \Gamma \vdash H;\ G \triangleright \Gamma \vdash H\ ]\!] \Longrightarrow \Gamma \vdash H$ |
ImpI: $F \triangleright \Gamma \vdash G \Longrightarrow \Gamma \vdash Imp\ F\ G$ |
ImpE: $[\![\ \Gamma \vdash Imp\ F\ G;\ \Gamma \vdash F\ ]\!] \Longrightarrow \Gamma \vdash G$

lemma *Weaken*: $[\![\ \Gamma \vdash F;\ \Gamma \subseteq \Gamma'\ ]\!] \Longrightarrow \Gamma' \vdash F$
proof(*induct arbitrary*: Γ′ *rule*: *ND.induct*)
  case (*NotI F* Γ) **thus** *?case* **using** *ND.NotI* **by** *auto*
next
  case *Ax* **thus** *?case* **by**(*blast intro*: *ND.Ax*)
next
  case *NotE* **thus** *?case* **by**(*blast intro*: *ND.NotE*)
next
  case *CC* **thus** *?case* **using** *ND.CC* **by** *blast*
next
  case *AndE1* **thus** *?case* **using** *ND.AndE1* **by** *metis*
next
  case *AndE2* **thus** *?case* **using** *ND.AndE2* **by** *metis*
next
  case *AndI* **thus** *?case* **by** (*simp add*: *ND.AndI*)
next
  case *OrI1* **thus** *?case* **using** *ND.OrI1* **by** *blast*
next
  case *OrI2* **thus** *?case* **using** *ND.OrI2* **by** *blast*
next
  case (*OrE* Γ *F G H*) **show** *?case* **apply**(*insert OrE.prems*)
    apply(*rule ND.OrE*[*of* Γ′ *F G*])
      apply(*rule OrE.hyps*(*2*)[*OF OrE.prems*])
     apply(*rule OrE.hyps*(*4*); *blast*)
    apply(*rule OrE.hyps*(*6*); *blast*)
  done
next
  case *ImpI* **thus** *?case* **using** *ND.ImpI* **by** *blast*

next
  case *ImpE* **thus** *?case* **using** *ND.ImpE* **by** *metis*
qed

lemma *BotE* : $\Gamma \vdash \bot \Longrightarrow \Gamma \vdash F$
  by (*meson CC subset-insertI Weaken*)

lemma *Not2E*: $Not(Not\ F) \triangleright \Gamma \vdash F$
  by (*metis CC ND.Ax NotE insertI1 insert-commute*)

lemma *Not2I*: $F \triangleright \Gamma \vdash Not(Not\ F)$
  by (*metis CC ND.Ax NotE insertI1 insert-commute*)

lemma *Not2IE*: $F \triangleright \Gamma \vdash G \Longrightarrow Not\ (Not\ F) \triangleright \Gamma \vdash G$
  by (*meson ImpE ImpI Not2E Weaken subset-insertI*)

lemma *NDtrans*: $\Gamma \vdash F \Longrightarrow F \triangleright \Gamma \vdash G \Longrightarrow \Gamma \vdash G$
  using *ImpE ImpI* **by** *blast*

lemma *AndL-sim*: $F \triangleright G \triangleright \Gamma \vdash H \Longrightarrow And\ F\ G \triangleright \Gamma \vdash H$
  apply(*drule Weaken*[**where** $\Gamma' = And\ F\ G \triangleright F \triangleright G \triangleright \Gamma$])
   apply *blast*
by (*metis AndE1 AndE2 ND.Ax NDtrans insertI1 insert-commute*)

lemma *NotSwap*: $Not\ F \triangleright \Gamma \vdash G \Longrightarrow Not\ G \triangleright \Gamma \vdash F$
  using *CC NotE insert-commute subset-insertI Weaken* **by** (*metis Ax insertI1*)
lemma *AndR-sim*: $[\![\ Not\ F \triangleright \Gamma \vdash H;\ Not\ G \triangleright \Gamma \vdash H\ ]\!] \Longrightarrow Not(And\ F\ G) \triangleright \Gamma \vdash H$
  using *AndI NotSwap* **by** *blast*

lemma *OrL-sim*: $[\![\ F \triangleright \Gamma \vdash H;\ G \triangleright \Gamma \vdash H\ ]\!] \Longrightarrow F \lor G \triangleright \Gamma \vdash H$
  using *Weaken*[**where** $\Gamma' = F \triangleright Or\ F\ G \triangleright \Gamma$] *Weaken*[**where** $\Gamma' = G \triangleright Or\ F\ G \triangleright \Gamma$]
  by (*meson ND.Ax OrE insertI1 insert-mono subset-insertI*)

lemma *OrR-sim*: $[\![\ \neg\ F \triangleright \neg\ G \triangleright \Gamma \vdash \bot]\!] \Longrightarrow \neg\ (G \lor F) \triangleright \Gamma \vdash \bot$
proof –
  assume $\neg\ F \triangleright \neg\ G \triangleright \Gamma \vdash \bot$
  then have $\bigwedge f.\ f \triangleright \neg\ F \triangleright \neg\ G \triangleright \Gamma \vdash \bot$ **by** (*meson Weaken subset-insertI*)
  then have $\bigwedge f.\ \neg\ G \triangleright \neg\ (f \lor F) \triangleright \Gamma \vdash \bot$ **by** (*metis NDtrans Not2E NotSwap OrI2 insert-commute*)
  then show *?thesis* **by** (*meson NDtrans Not2I NotSwap OrI1*)
qed

lemma *ImpL-sim*: $[\![\ \neg\ F \triangleright \Gamma \vdash \bot;\ G \triangleright \Gamma \vdash \bot]\!] \Longrightarrow F \rightarrow G \triangleright \Gamma \vdash \bot$
  by (*meson CC ImpE ImpI ND.Ax Weaken insertI1 subset-insertI*)

lemma *ImpR-sim*: $[\![\ \neg\ G \triangleright F \triangleright \Gamma \vdash \bot]\!] \Longrightarrow \neg\ (F \rightarrow G) \triangleright \Gamma \vdash \bot$
  by (*metis* (*full-types*) *ImpI NotSwap insert-commute*)

lemma *ND-lem*: $\{\} \vdash Not\ F \lor F$

apply(*rule CC*)
apply(*rule OrE*[*of - Not F F*])
apply(*rule OrI1*)
apply(*rule NotI*)
apply(*rule NotE*[*of -* (¬ *F* ∨ *F*)]; *blast intro*: *OrI1 OrI2 Ax*)+
done

lemma *ND-caseDistinction*: ⟦ *F*▷Γ ⊢ *H*; *Not F*▷Γ ⊢ *H* ⟧ ⟹ Γ ⊢ *H*
by (*meson ND-lem OrE empty-subsetI Weaken*)

lemma ⟦¬ *F*▷ Γ ⊢ *H*; *G*▷ Γ ⊢ *H*⟧ ⟹ *F* → *G*▷ Γ ⊢ *H*
apply(*rule ND-caseDistinction*[*of F*])
apply (*meson ImpE ImpI ND.intros*(*1*) *Weaken insertI1 subset-insertI*)
apply (*metis Weaken insert-commute subset-insertI*)
done

lemma *ND-deMorganAnd*: {¬ (*F* ∧ *G*)} ⊢ ¬ *F* ∨ ¬ *G*
apply(*rule CC*)
apply(*rule NotE*[*of - F* ∧ *G*])
apply(*simp add*: *Ax*; *fail*)
apply(*rule AndI*)
apply(*rule CC*)
apply(*rule NotE*[*of -* ¬ *F* ∨ ¬ *G*])
apply(*simp add*: *Ax*; *fail*)
apply(*rule OrI1*)
apply(*simp add*: *Ax*; *fail*)
apply(*rule CC*)
apply(*rule NotE*[*of -* ¬ *F* ∨ ¬ *G*])
apply(*simp add*: *Ax*; *fail*)
apply(*rule OrI2*)
apply(*simp add*: *Ax*; *fail*)
done

lemma *ND-deMorganOr*: {¬ (*F* ∨ *G*)} ⊢ ¬ *F* ∧ ¬ *G*
apply(*rule ND-caseDistinction*[*of F*]; *rule ND-caseDistinction*[*of G*])
apply(*rule CC*; *rule NotE*[*of - F* ∨ *G*]; *simp add*: *Ax OrI2 OrI1*; *fail*)+
apply(*rule AndI*; *simp add*: *Ax*; *fail*)
done

lemma *sim-sim*: *F*▷ Γ ⊢ *H* ⟹ *G*▷Γ ⊢ *F* ⟹ *G*▷ Γ ⊢ *H*
by (*meson ImpE ImpI Weaken subset-insertI*)
thm *sim-sim*[**where** Γ={}, *rotated*, *no-vars*]

lemma *Top-provable*[*simp*,*intro*!]: Γ ⊢ ⊤ **unfolding** *Top-def* **by** (*intro ND.ImpI ND.Ax*) *simp*

lemma *NotBot-provable*[*simp*,*intro*!]: Γ ⊢ ¬ ⊥ **using** *NotI BotE Ax* **by** *blast*

lemma *Top-useless*: Γ ⊢ *F* ⟹ Γ − {⊤} ⊢ *F* by (*metis NDtrans Top-provable Weaken insert-Diff-single subset-insertI*)

lemma *AssmBigAnd*: *set G* ⊢ *F* ⟷ {} ⊢ (⋀*G* → *F*)
proof(*induction G arbitrary*: *F*)
case *Nil* **thus** *?case* **by**(*fastforce intro*: *ND.ImpI elim*: *Weaken ImpE*[*OF* - *NotBot-provable*])
next
case (*Cons G GS*) **show** *?case* **proof**
assume *set* (*G* # *GS*) ⊢ *F*
hence *set GS* ⊢ *G* → *F* **by**(*intro ND.ImpI*) *simp*
with *Cons.IH* **have** ∗: {} ⊢ ⋀ *GS* → *G* → *F* **..**
hence {*G*,⋀*GS*} ⊢ *F* **proof** −
have ∗: {⋀*GS*} ⊢ *G* → *F*
using *Weaken*[*OF* ∗ *empty-subsetI*] *ImpE*[**where** Γ={⋀ *GS*} **and** *F*=⋀ *GS*] **by** (*simp add*: *ND.Ax*)
show {*G*,⋀*GS*} ⊢ *F* **using** *Weaken*[*OF* ∗] *ImpE*[**where** Γ={*G*,⋀ *GS*} **and** *F*=*G*] *ND.Ax* **by** (*simp add*: *ND.Ax*)
qed
thus {} ⊢ ⋀ (*G* # *GS*) → *F* **by**(*intro ND.ImpI*; *simp add*: *AndL-sim*)
next
assume {} ⊢ ⋀ (*G* # *GS*) → *F*
hence {*G* ∧ ⋀*GS*} ⊢ *F* **using** *ImpE*[*OF* - *Ax*[*OF singletonI*]] *Weaken* **by** *fastforce*
hence {*G*,⋀*GS*} ⊢ *F* **by** (*meson AndI ImpE ImpI ND.intros*(*1*) *Weaken insertI1 subset-insertI*)
hence {⋀*GS*} ⊢ *G* → *F* **using** *ImpI* **by** *blast*
hence {} ⊢ ⋀ *GS* → *G* → *F* **using** *ImpI* **by** *blast*
with *Cons.IH* **have** *set GS* ⊢ *G* → *F* **..**
thus *set* (*G* # *GS*) ⊢ *F* **using** *ImpE Weaken* **by** (*metis Ax list.set-intros*(*1*) *set-subset-Cons*)
qed
qed

end
theory *ND-Sound*
imports *ND Sema*
begin

lemma *BigAndImp*: *A* ⊨ (⋀*P* → *G*) ⟷ ((∀ *F* ∈ *set P*. *A* ⊨ *F*) ⟶ *A* ⊨ *G*)
by(*induction P*; *simp add*: *entailment-def*)

lemma *ND-sound*: Γ ⊢ *F* ⟹ Γ ⊫ *F*
by(*induction rule*: *ND.induct*; *simp add*: *entailment-def*; *blast*)

end
theory *ND-Compl-Truthtable*
imports *ND-Sound*
begin

This proof is inspired by Huth and Ryan [7].

definition *turn-true* $\mathcal{A}$ $F \equiv$ *if* $\mathcal{A} \models F$ *then* F *else* (*Not* F)
lemma *lemma0*[*simp*,*intro*!]: $\mathcal{A} \models$ *turn-true* $\mathcal{A}$ F **unfolding** *turn-true-def* **by** *simp*

lemma *turn-true-simps*[*simp*]:
  $\mathcal{A} \models F \Longrightarrow$ *turn-true* $\mathcal{A}$ $F = F$
  $\neg\ \mathcal{A} \models F \Longrightarrow$ *turn-true* $\mathcal{A}$ $F = \neg\ F$
unfolding *turn-true-def* **by** *simp-all*

definition *line-assm* :: *'a valuation* $\Rightarrow$ *'a set* $\Rightarrow$ *'a formula set* **where**
line-assm $\mathcal{A} \equiv$ (') ($\lambda k.$ *turn-true* $\mathcal{A}$ (*Atom* k))
definition *line-suitable* :: *'a set* $\Rightarrow$ *'a formula* $\Rightarrow$ *bool* **where**
line-suitable Z $F \equiv$ (*atoms* $F \subseteq Z$)
lemma *line-suitable-junctors*[*simp*]:
  line-suitable $\mathcal{A}$ (*Not* F) = *line-suitable* $\mathcal{A}$ F
  line-suitable $\mathcal{A}$ (*And* F G) = (*line-suitable* $\mathcal{A}$ F $\wedge$ *line-suitable* $\mathcal{A}$ G)
  line-suitable $\mathcal{A}$ (*Or* F G) = (*line-suitable* $\mathcal{A}$ F $\wedge$ *line-suitable* $\mathcal{A}$ G)
  line-suitable $\mathcal{A}$ (*Imp* F G) = (*line-suitable* $\mathcal{A}$ F $\wedge$ *line-suitable* $\mathcal{A}$ G)
unfolding *line-suitable-def* **by**(*clarsimp*; *linarith*)+

lemma *line-assm-Cons*[*simp*]: *line-assm* $\mathcal{A}$ ($k \triangleright ks$) = (*if* $\mathcal{A}$ k *then Atom* k *else Not* (*Atom* k)) $\triangleright$ *line-assm* $\mathcal{A}$ *ks*
unfolding *line-assm-def* **by** *simp*

lemma *NotD*: $\Gamma \vdash \neg\ F \Longrightarrow F \triangleright \Gamma \vdash \bot$ **by** (*meson Not2I NotE Weaken subset-insertI*)

lemma *truthline-ND-proof*:
  fixes F :: *'a formula*
  assumes *line-suitable* Z F
  shows *line-assm* $\mathcal{A}$ $Z \vdash$ *turn-true* $\mathcal{A}$ F
using *assms* **proof**(*induction* F)
 case (*Atom* k) **thus** *?case* **using** *Ax*[**where** *'a='a*] **by** (*simp add*: *line-suitable-def line-assm-def*)
next
  case *Bot*
  have *turn-true* $\mathcal{A}$ $\bot$ = *Not Bot* **unfolding** *turn-true-def* **by** *simp*
  thus *?case* **by** (*simp add*: *Ax NotI*)
next
  have [*simp*]: $\Gamma \vdash \neg\ (\neg\ F) \longleftrightarrow \Gamma \vdash F$ **for** F :: *'a formula* **and** Γ **by** (*metis NDtrans Not2E Not2I*)
  case (*Not* F)
  hence *line-assm* $\mathcal{A}$ $Z \vdash$ *turn-true* $\mathcal{A}$ F **by** *simp*
  thus *?case* **by**(*cases* $\mathcal{A} \models F$; *simp*)
next
  have [*simp*]: $[\![$*line-assm* $\mathcal{A}$ $Z \vdash \neg\ F$; $\neg\ \mathcal{A} \models F]\!] \Longrightarrow F \wedge G \triangleright$ *line-assm* $\mathcal{A}$ $Z \vdash \bot$ **for** F G **by**(*blast intro*!: *NotE*[**where** F=F] *intro*: *AndE1*[*OF Ax*] *Weaken*[*OF - subset-insertI*])

have [*simp*]: ⟦*line-assm* $\mathcal{A}$ *Z* ⊢ ¬ *G*; ¬ $\mathcal{A}$ ⊨ *G*⟧ ⟹ *F* ∧ *G*▷ *line-assm* $\mathcal{A}$ *Z* ⊢ ⊥ **for** *F G* **by**(*blast intro*!: *NotE*[**where** *F*=*G*] *intro*: *AndE2*[*OF Ax*] *Weaken*[*OF* - *subset-insertI*])

case (*And F G*)
thus *?case* **by**(*cases* $\mathcal{A}$ ⊨ *F*; *cases* $\mathcal{A}$ ⊨ *G*; *simp*; *intro ND.NotI AndI*; *simp*)
next
case (*Or F G*)
thus *?case* **by**(*cases* $\mathcal{A}$ ⊨ *F*; *cases* $\mathcal{A}$ ⊨ *G*; *simp*; (*elim ND.OrI1 ND.OrI2*)*?*) (*force intro*!: *NotI dest*!: *NotD dest*: *OrL-sim*)
next
case (*Imp F G*)
hence *mIH*: *line-assm* $\mathcal{A}$ *Z* ⊢ *turn-true* $\mathcal{A}$ *F line-assm* $\mathcal{A}$ *Z* ⊢ *turn-true* $\mathcal{A}$ *G* **by** *simp*+
thus *?case* **by**(*cases* $\mathcal{A}$ ⊨ *F*; *cases* $\mathcal{A}$ ⊨ *G*; *simp*; *intro ImpI NotI ImpL-sim*; *simp add*: *Weaken*[*OF* - *subset-insertI*] *NotSwap NotD NotD*[*THEN BotE*])
qed
thm *NotD*[*THEN BotE*]

lemma *deconstruct-assm-set*:
assumes *IH*: ⋀$\mathcal{A}$. *line-assm* $\mathcal{A}$ (*k*▷*Z*) ⊢ *F*
shows ⋀$\mathcal{A}$. *line-assm* $\mathcal{A}$ *Z* ⊢ *F*
proof *cases*
assume *k* ∈ *Z* **with** *IH* **show** *?thesis* $\mathcal{A}$ **for** $\mathcal{A}$ **by** (*simp add*: *insert-absorb*)
next
assume *k* ∉ *Z*
fix $\mathcal{A}$

Since we require the IH for arbitrary $\mathcal{A}$, we use a modified $\mathcal{A}$ from the conclusion like this:

from *IH* **have** *av*: *line-assm* ($\mathcal{A}$(*k* := *v*)) (*k*▷*Z*) ⊢ *F* **for** *v* **by** *blast*

However, that modification is only relevant for *k* ▷ *Z*, nothing from *Z* gets touched.

from ‹*k* ∉ *Z*› **have** *line-assm* ($\mathcal{A}$(*k* := *v*)) *Z* = *line-assm* $\mathcal{A}$ *Z* **for** *v* **unfolding** *line-assm-def turn-true-def* **by** *force*

That means we can rewrite the modified *line-assm* like this:

hence *line-assm* ($\mathcal{A}$(*k* := *v*)) (*k*▷*Z*) =
(*if v then Atom k else Not* (*Atom k*)) ▷ *line-assm* $\mathcal{A}$ *Z* **for** *v* **by** *simp*

Inserting *True* and *False* for *v* yields the two alternatives.

with *av* **have** *Atom k* ▷ *line-assm* $\mathcal{A}$ *Z* ⊢ *F Not* (*Atom k*) ▷ *line-assm* $\mathcal{A}$ *Z* ⊢ *F*
by(*metis* (*full-types*))+
with *ND-caseDistinction* **show** *line-assm* $\mathcal{A}$ *Z* ⊢ *F* **.**
qed

theorem *ND-complete*:

```
  assumes taut: ⊨ F
  shows {} ⊢ F
proof −
  have [simp]: turn-true Z F = F for Z using taut by simp

  have line-assm 𝒜 {} ⊢ F for 𝒜
  proof(induction arbitrary: 𝒜 rule: finite-empty-induct)
    show fat: finite (atoms F) by (fact atoms-finite)
  next
    have su: line-suitable (atoms F) F unfolding line-suitable-def by simp
    with truthline-ND-proof[OF su] show base: line-assm 𝒜 (atoms F) ⊢ F for 𝒜
by simp
  next
    case (3 k Z)
    from ‹k ∈ Z› have ∗: ‹k ▷ Z − {k} = Z› by blast
    from ‹⋀𝒜. line-assm 𝒜 Z ⊢ F›
    show ‹line-assm 𝒜 (Z − {k}) ⊢ F›
      using deconstruct-assm-set[of k Z − {k} F 𝒜]
      unfolding ∗ by argo
  qed

  thus ?thesis unfolding line-assm-def by simp
qed

corollary ND-sound-complete: {} ⊢ F ⟷ ⊨ F
  using ND-sound[of {} F] ND-complete[of F] unfolding entailment-def by blast

end
theory ND-Compl-Truthtable-Compact
imports ND-Compl-Truthtable Compactness
begin

theorem
  fixes Γ :: 'a :: countable formula set
  shows Γ ⊫ F ⟹ Γ ⊢ F
proof −
  assume ‹Γ ⊫ F›
  then obtain G where set G ⊆ Γ ⊨ ⋀G → F by (rule compact-to-formula)
  from ND-complete ‹⊨ ⋀G → F› have ‹{} ⊢ ⋀G → F› .
  with AssmBigAnd have ‹set G ⊢ F› ..
  with Weaken show ?thesis using ‹set G ⊆ Γ› .
qed


end
```

2.3 Hilbert Calculus

theory *HC*

imports *Formulas*
begin

We can define Hilbert Calculus as the modus ponens closure over a set of axioms:

inductive *HC* :: *'a formula set* ⇒ *'a formula* ⇒ *bool* (**infix** ‹$\vdash_H$› *30*) **for** Γ :: *'a formula set* **where**
Ax: $F \in \Gamma \Longrightarrow \Gamma \vdash_H F$ |
MP: $\Gamma \vdash_H F \Longrightarrow \Gamma \vdash_H F \to G \Longrightarrow \Gamma \vdash_H G$

.

context begin

The problem with that is defining the axioms. Normally, we just write that $F \to G \to F$ is an axiom, and mean that anything can be substituted for F and G. Now, we can't just write down a set $\{F \to (G \to F)$, *dots.* Instead, defining it as an inductive set with no induction is a good idea.

inductive-set *AX0* **where**
$F \to (G \to F) \in AX0$ |
$(F \to (G \to H)) \to ((F \to G) \to (F \to H)) \in AX0$
inductive-set *AX10* **where**
$F \in AX0 \Longrightarrow F \in AX10$ |
$F \to (F \vee G) \in AX10$ |
$G \to (F \vee G) \in AX10$ |
$(F \to H) \to ((G \to H) \to ((F \vee G) \to H)) \in AX10$ |
$(F \wedge G) \to F \in AX10$ |
$(F \wedge G) \to G \in AX10$ |
$F \to (G \to (F \wedge G)) \in AX10$ |
$(F \to \bot) \to \neg F \in AX10$ |
$\neg F \to (F \to \bot) \in AX10$ |
$(\neg F \to \bot) \to F \in AX10$
lemmas *HC-intros*[*intro*!] =
  AX0.intros[*THEN HC.intros*(*1*)]
  AX0.intros[*THEN AX10.intros*(*1*), *THEN HC.intros*(*1*)]
  AX10.intros(*2*−)[*THEN HC.intros*(*1*)]

The first four axioms, as originally formulated by Hilbert [6].

inductive-set *AXH* **where**
  $(F \to (G \to F)) \in AXH$ |
  $(F \to (F \to G)) \to (F \to G) \in AXH$ |
  $(F \to (G \to H)) \to (G \to (F \to H)) \in AXH$ |
  $(G \to H) \to ((F \to G) \to (F \to H)) \in AXH$

lemma *HC-mono*: $S \vdash_H F \Longrightarrow S \subseteq T \Longrightarrow T \vdash_H F$
  by(*induction rule*: *HC.induct*) (*auto intro*: *HC.intros*)
lemma *AX010*: $AX0 \subseteq AX10$
  apply(*rule*)
  apply(*cases rule*: *AX0.cases*, *assumption*)

```
  apply(auto intro: AX10.intros)
done
lemma AX100[simp]: AX0 ∪ AX10 = AX10 using AX010 by blast
```

Hilbert's first four axioms and *AX0* are syntactically quite different. Derivability does not change:

```
lemma hilbert-folgeaxiome-as-strong-as-AX0: AX0 ∪ Γ ⊢H F ⟷ AXH ∪ Γ ⊢H
F
proof −
  have 0:
  AX0 ⊢H (F → (F → G)) → (F → G)
  AX0 ⊢H (F → (G → H)) → (G → (F → H))
  AX0 ⊢H (G → H) → ((F → G) → (F → H))
    for F G H using HC-intros(1,2) MP by metis+
  have H:
    AXH ⊢H (F → (G → H)) → ((F → G) → (F → H))
    for F G H
  proof −
    note AXH.intros[THEN HC.Ax]
    thus ?thesis using MP by metis
  qed
  note ∗ = H 0
  note ∗ = ∗[THEN HC-mono, OF Un-upper1]
  show ?thesis (is ?Z ⟷ ?H)
  proof
    assume ?Z thus ?H proof induction
      case MP thus ?case using HC.MP by blast
    next
      case (Ax F) thus ?case proof
       assume F ∈ AX0 thus ?thesis by induction (simp-all add: AXH.intros(1)
HC.Ax ∗)
      next
        assume F ∈ Γ thus ?case using HC.Ax[of F] by simp
      qed
    qed
  next
    assume ?H thus ?Z proof induction
      case MP thus ?case using HC.MP by blast
    next
      case (Ax F) thus ?case proof
       assume F ∈ AXH thus ?thesis by induction (simp-all add: AX0.intros(1)
HC.Ax ∗)
      next
        assume F ∈ Γ thus ?case using HC.Ax[of F] by simp
      qed
    qed
  qed
qed
```

lemma $AX0 \vdash_H F \to F$ **by** (*meson HC-intros*(*1*,*2*) *HC.MP*)

lemma *imp-self*: $AX0 \vdash_H F \to F$ **proof** −
let $?d = \lambda f.\ AX0 \vdash_H f$
note *MP*
moreover have $?d\ (F \to (G \to F))$ **for** G **using** *HC-intros*(*1*)[**where** $G{=}G$ **and** $F{=}F$] **.**
moreover {
note *MP*
moreover have $?d\ (F \to ((G \to F) \to F))$ **for** G **using** *HC-intros*(*1*)[**where** $G{=}G \to F$ **and** $F{=}F$] **.**
moreover have $?d\ ((F \to ((G \to F) \to F)) \to ((F \to (G \to F)) \to (F \to F)))$ **for** G **using** *HC-intros*(*2*)[**where** $G{=}G \to F$ **and** $F{=}F$ **and** $H{=}F$] **.**
ultimately have $?d\ ((F \to (G \to F)) \to (F \to F))$ **for** G **. }**
ultimately show $?d\ (F \to F)$ **.**
qed

theorem *Deduction-theorem*: $AX0 \cup insert\ F\ \Gamma \vdash_H G \Longrightarrow AX0 \cup \Gamma \vdash_H F \to G$
proof(*induction rule*: *HC.induct*)
case (*Ax G*)
show *?case* **proof** *cases*
assume $F = G$
from *imp-self* **have** $AX0 \vdash_H G \to G$ **.**
with *HC-mono* **show** *?case* **unfolding** ‹$F = G$› **using** *sup-ge1* **.**
next
assume $F \neq G$
note *HC.MP*
moreover {
from ‹$F \neq G$› ‹$G \in AX0 \cup insert\ F\ \Gamma$› **have** $G \in AX0 \cup \Gamma$ **by** *simp*
with *HC.Ax* **have** $AX0 \cup \Gamma \vdash_H G$ **.**
}
moreover from *HC-mono*[*OF HC-intros*(*1*) *sup-ge1*] **have** $AX0 \cup \Gamma \vdash_H G \to (F \to G)$ **.**
ultimately show *?case* **.**
qed
next
case (*MP G H*)
have $AX0 \cup \Gamma \vdash_H (F \to (G \to H)) \to ((F \to G) \to (F \to H))$ **using** *HC-mono* **by** *blast*
with *HC.MP* ‹$AX0 \cup \Gamma \vdash_H F \to (G \to H)$› **have** $AX0 \cup \Gamma \vdash_H (F \to G) \to (F \to H)$ **.**
with *HC.MP* ‹$AX0 \cup \Gamma \vdash_H F \to G$› **show** $AX0 \cup \Gamma \vdash_H F \to H$ **.**
qed

end

end

theory *HC-Compl-Consistency*
imports *Consistency HC*
begin

context begin
private lemma *dt*: $F \triangleright \Gamma \cup AX10 \vdash_H G \Longrightarrow \Gamma \cup AX10 \vdash_H F \to G$
by (*metis AX100 Deduction-theorem Un-insert-right sup-left-commute*)
lemma *sim*: $\Gamma \cup AX10 \vdash_H F \Longrightarrow F \triangleright \Gamma \cup AX10 \vdash_H G \Longrightarrow \Gamma \cup AX10 \vdash_H G$
using *MP dt* **by** *blast*
lemma *sim-conj*: $F \triangleright G \triangleright \Gamma \cup AX10 \vdash_H H \Longrightarrow \Gamma \cup AX10 \vdash_H F \Longrightarrow \Gamma \cup AX10 \vdash_H G \Longrightarrow \Gamma \cup AX10 \vdash_H H$
using *MP dt* **by** (*metis Un-insert-left*)
lemma *sim-disj*: $[\![F \triangleright \Gamma \cup AX10 \vdash_H H; G \triangleright \Gamma \cup AX10 \vdash_H H; \Gamma \cup AX10 \vdash_H F \lor G]\!] \Longrightarrow \Gamma \cup AX10 \vdash_H H$
proof *goal-cases*
case *1*
have *2*: $\Gamma \cup AX10 \vdash_H F \to H$ **by** (*simp add*: *1 dt*)
have *3*: $\Gamma \cup AX10 \vdash_H G \to H$ **by** (*simp add*: *1 dt*)
have *4*: $\Gamma \cup AX10 \vdash_H (F \lor G) \to H$ **by** (*meson 2 3 HC.simps HC-intros*(*7*) *HC-mono sup-ge2*)
thus *?case* **using** *1*(*3*) *MP* **by** *blast*
qed

private lemma *someax*: $\Gamma \cup AX10 \vdash_H F \to \neg F \to \bot$
proof −
have $F \triangleright \Gamma \cup AX10 \vdash_H \neg F \to F \to \bot$
by (*meson HC-intros*(*12*) *HC-mono subset-insertI sup-ge2*)
then have $\neg F \triangleright F \triangleright \Gamma \cup AX10 \vdash_H \bot$
by (*meson HC.simps HC-mono insertI1 subset-insertI*)
then show *?thesis*
by (*metis* (*no-types*) *Un-insert-left dt*)
qed

lemma *lem*: $\Gamma \cup AX10 \vdash_H \neg F \lor F$
proof −
thm *HC-intros*(*7*)[*of F* ⊥ *Not F*]
have $F \triangleright \Gamma \cup AX10 \vdash_H (\neg F \lor F)$
by (*metis AX10.intros*(*3*) *Ax HC-mono MP Un-commute Un-insert-left insertI1 sup-ge1*)
hence $F \triangleright \Gamma \cup AX10 \vdash_H \neg (\neg F \lor F) \to \bot$ **using** *someax* **by** (*metis HC.simps Un-insert-left*)
hence $\neg (\neg F \lor F) \triangleright F \triangleright \Gamma \cup AX10 \vdash_H \bot$ **by** (*meson Ax HC-mono MP insertI1 subset-insertI*)
hence $\neg (\neg F \lor F) \triangleright \Gamma \cup AX10 \vdash_H F \to \bot$
by (*metis Un-insert-left dt insert-commute*)

have $\neg F \triangleright \Gamma \cup AX10 \vdash_H (\neg F \lor F)$
by (*metis HC.simps HC-intros*(*5*) *HC-mono inf-sup-ord*(*4*) *insertI1 insert-is-Un*)
hence $\neg F \triangleright \Gamma \cup AX10 \vdash_H \neg (\neg F \lor F) \to \bot$ **using** *someax* **by** (*metis*

HC.simps Un-insert-left)
hence $\neg\ (\ \neg\ F \vee F) \triangleright \neg F \triangleright \Gamma \cup AX10 \vdash_H \bot$ **by** (*meson Ax HC-mono MP insertI1 subset-insertI*)
hence $\neg\ (\ \neg\ F \vee F) \triangleright \Gamma \cup AX10 \vdash_H \neg F \to \bot$
by (*metis Un-insert-left dt insert-commute*)

hence $\Gamma \cup AX10 \vdash_H \neg\ (\ \neg\ F \vee F) \to \bot$
by (*meson HC-intros*(*13*) *HC-mono MP* ‹$\neg\ (\neg\ F \vee F) \triangleright \Gamma \cup AX10 \vdash_H F \to \bot$› *dt subset-insertI sup-ge2*)
thus *?thesis* **by** (*meson HC.simps HC-intros*(*13*) *HC-mono sup-ge2*)

qed

lemma *exchg*: $\Gamma \cup AX10 \vdash_H F \vee G \Longrightarrow \Gamma \cup AX10 \vdash_H G \vee F$
by (*meson AX10.intros*(*3*) *HC.simps HC-intros*(*5*) *HC-intros*(*7*) *HC-mono sup-ge2*)

lemma *lem2*: $\Gamma \cup AX10 \vdash_H F \vee \neg\ F$ **by** (*simp add*: *exchg lem*)

lemma *imp-sim*: $\Gamma \cup AX10 \vdash_H F \to G \Longrightarrow \Gamma \cup AX10 \vdash_H \neg\ F \vee G$
proof *goal-cases* **case** *1*
have $\Gamma \cup AX10 \vdash_H F \to \neg\ F \vee G$
proof −
have *f1*: $\forall F\ f\ Fa.\ \neg\ (F \vdash_H f) \vee \neg\ F \subseteq Fa \vee Fa \vdash_H f$
using *HC-mono* **by** *blast*
then have *f2*: $F \triangleright \Gamma \cup AX10 \vdash_H F \to G$
by (*metis 1 subset-insertI*)
have $\Gamma \cup AX10 \vdash_H G \to \neg\ F \vee G$
using *f1* **by** *blast*
then show *?thesis*
using *f2 f1* **by** (*metis* (*no-types*) *HC.simps dt insertI1 subset-insertI*)
qed
moreover have $\Gamma \cup AX10 \vdash_H \neg F \to \neg\ F \vee G$ **by** (*simp add*: *AX10.intros*(*2*) *Ax*)
ultimately show *?case*
proof −
have $\bigwedge F\ f\ fa\ fb.\ \neg\ (F \vdash_H f \to fa) \vee \neg\ (fb \triangleright F \vdash_H f) \vee fb \triangleright F \vdash_H fa$
by (*meson HC-mono MP subset-insertI*)
then show *?thesis*
by (*metis Ax* ‹$\Gamma \cup AX10 \vdash_H F \to \neg\ F \vee G$› ‹$\Gamma \cup AX10 \vdash_H \neg\ F \to \neg\ F \vee G$› *insertI1 lem sim-disj*)
qed
qed

lemma *inpcp*: $\Gamma \cup AX10 \vdash_H \bot \Longrightarrow \Gamma \cup AX10 \vdash_H F$
by (*meson HC-intros*(*13*) *HC-mono MP dt subset-insertI sup-ge2*)

lemma *HC-case-distinction*: $\Gamma \cup AX10 \vdash_H F \to G \Longrightarrow \Gamma \cup AX10 \vdash_H \neg\ F \to G \Longrightarrow \Gamma \cup AX10 \vdash_H G$

using *HC-intros*(*7*)[*of F G Not F*] *lem2*
by (*metis* (*no-types*, *opaque-lifting*) *HC.simps HC-mono insertI1 sim-disj subset-insertI*)

lemma *nand-sim*: $\Gamma \cup AX10 \vdash_H \neg (F \wedge G) \Longrightarrow \Gamma \cup AX10 \vdash_H \neg F \vee \neg G$
proof *goal-cases* **case** *1*
have $\Gamma \cup AX10 \vdash_H F \to G \to F \wedge G$ **by** (*simp add*: *AX10.intros*(*7*) *Ax*)
hence *2*: $F \triangleright G \triangleright \Gamma \cup AX10 \vdash_H F \wedge G$
by (*meson Ax HC-mono MP insertI1 subset-insertI*)
hence *3*: $F \triangleright G \triangleright \Gamma \cup AX10 \vdash_H \bot$ **using** *1* **by** (*meson HC-intros*(*12*) *HC-mono MP subset-insertI sup-ge2*)
from *2* **have** $\Gamma \cup AX10 \vdash_H G \to F \to F \wedge G$ **by** (*metis Un-insert-left dt*)
have *4*: $\Gamma \cup AX10 \vdash_H \neg F \to \neg F \vee \neg G$ **by** (*simp add*: *AX10.intros*(*2*) *Ax*)
have *5*: $\Gamma \cup AX10 \vdash_H \neg G \to F \to \neg F \vee \neg G$
by (*metis* (*full-types*) *AX10.intros*(*3*) *AX100 Ax HC-mono MP Un-assoc Un-insert-left dt inf-sup-ord*(*4*) *insertI1 subset-insertI sup-ge2*)
have *6*: $\Gamma \cup AX10 \vdash_H G \to F \to \neg F \vee \neg G$ **using** *3 inpcp* **by** (*metis Un-insert-left dt*)
have *7*: $\Gamma \cup AX10 \vdash_H F \to \neg F \vee \neg G$ **using** *5 6 HC-case-distinction* **by** *blast*
show *?case* **using** *4 7 HC-case-distinction* **by** *blast*
qed

lemma *HC-contrapos-nn*:
$[\![\Gamma \cup AX10 \vdash_H \neg F; \Gamma \cup AX10 \vdash_H G \to F]\!] \Longrightarrow \Gamma \cup AX10 \vdash_H \neg G$
proof *goal-cases* **case** *1*
from *1*(*1*) **have** $\Gamma \cup AX10 \vdash_H F \to \bot$ **using** *HC-intros*(*12*) **using** *HC-mono MP* **by** *blast*
hence $\Gamma \cup AX10 \vdash_H G \to \bot$ **by** (*meson 1*(*2*) *HC.intros*(*1*) *HC-mono MP dt insertI1 subset-insertI*)
thus *?case* **by**(*meson HC-intros*(*11*) *HC-intros*(*3*) *HC-mono MP sup-ge2*)
qed

lemma *nor-sim*:
assumes $\Gamma \cup AX10 \vdash_H \neg (F \vee G)$
shows $\Gamma \cup AX10 \vdash_H \neg F$ $\Gamma \cup AX10 \vdash_H \neg G$
using *HC-contrapos-nn assms* **by** (*metis HC-intros*(*5*,*6*) *HC-mono sup-ge2*)+

lemma *HC-contrapos-np*:
$[\![\Gamma \cup AX10 \vdash_H \neg F; \Gamma \cup AX10 \vdash_H \neg G \to F]\!] \Longrightarrow \Gamma \cup AX10 \vdash_H G$
by (*meson HC-intros*(*12*) *HC-intros*(*13*) *HC-mono MP sup-ge2 HC-contrapos-nn*[*of* Γ *F Not G*])

lemma *not-imp*: $\Gamma \cup AX10 \vdash_H \neg F \to F \to G$
proof *goal-cases* **case** *1*
have $\Gamma \cup AX10 \vdash_H \neg F \to F \to \bot$ **by** (*simp add*: *AX10.intros*(*9*) *Ax*)
hence $\neg F \triangleright F \triangleright \Gamma \cup AX10 \vdash_H \bot$ **by** (*meson HC.simps HC-mono insertI1 subset-insertI*)
hence $\neg F \triangleright F \triangleright \Gamma \cup AX10 \vdash_H G$ **by** (*metis* (*no-types*, *opaque-lifting*) *Un-commute*

Un-insert-right inpcp)
thus *?case* **by** (*metis Un-insert-left dt insert-commute*)
qed

lemma *HC-consistent*: *pcp* $\{\Gamma|\ \Gamma.\ \neg(\Gamma \cup AX10 \vdash_H \bot)\}$
unfolding *pcp-def*
apply(*intro ballI conjI*; *unfold mem-Collect-eq*; *elim exE conjE*; *erule contra-pos-np*; *clarsimp*)
subgoal by (*simp add*: *HC.Ax*)
subgoal by (*meson Ax HC-intros*(*12*) *HC-mono MP Un-upper1 sup-ge2*)
subgoal using *sim-conj* **by** (*metis* (*no-types*, *lifting*) *Ax HC-intros*(*8*) *HC-intros*(*9*) *HC-mono MP sup-ge1 sup-ge2*)
subgoal using *sim-disj* **using** *Ax* **by** *blast*
subgoal by (*erule* (*1*) *sim-disj*) (*simp add*: *Ax imp-sim*)
subgoal by (*metis Ax HC-contrapos-nn MP Un-iff Un-insert-left dt inpcp someax*)
subgoal by(*erule* (*1*) *sim-disj*) (*simp add*: *Ax nand-sim*)
subgoal by(*erule sim-conj*) (*meson Ax Un-iff nor-sim*)+
subgoal for Γ *F G* **apply**(*erule sim-conj*)
subgoal by (*meson Ax HC-Compl-Consistency.not-imp HC-contrapos-np Un-iff*)
subgoal by (*metis Ax HC-contrapos-nn HC-intros*(*3*) *HC-mono sup-ge1 sup-ge2*)
done
done

end

corollary *HC-complete*:
fixes F :: *'a* :: *countable formula*
shows $\models F \Longrightarrow AX10 \vdash_H F$
proof(*erule contrapos-pp*)
let $?W = \{\Gamma|\ \Gamma.\ \neg((\Gamma$ :: (*'a* :: *countable*) *formula set*) $\cup\ AX10 \vdash_H \bot)\}$
note [[*show-types*]]
assume ‹$\neg$ ($AX10 \vdash_H F$)›
hence $\neg$ ($\neg F \triangleright AX10 \vdash_H \bot$)
by (*metis AX100 Deduction-theorem HC-intros*(*13*) *MP Un-insert-right*)
hence $\{\neg F\} \in ?W$ **by** *simp*
with *pcp-sat HC-consistent* **have** *sat* $\{\neg F\}$ **.**
thus $\neg \models F$ **by** (*simp add*: *sat-def*)
qed

end

2.4 Resolution

theory *Resolution*
imports *CNF HOL−Library.While-Combinator*

begin

Resolution is different from the other proof systems: its derivations do not represent proofs in the way the other systems do. Rather, they represent invariant additions (under satisfiability) to set of clauses.

inductive *Resolution* :: *'a literal set set* ⇒ *'a literal set* ⇒ *bool* (‹- ⊢ -› [*30*] *28*)
where
Ass: $C \in S \Longrightarrow S \vdash C$ |
R: $S \vdash C \Longrightarrow S \vdash D \Longrightarrow k^{+} \in C \Longrightarrow k^{-1} \in D \Longrightarrow S \vdash (C - \{k^{+}\}) \cup (D - \{k^{-1}\})$

The problematic part of this formulation is that we can't talk about a "Resolution Refutation" in an inductive manner. In the places where Gallier's proofs [3] do that, we have to work around that.

lemma *Resolution-weaken*: $S \vdash D \Longrightarrow T \cup S \vdash D$
  by(*induction rule*: *Resolution.induct*; *auto intro*: *Resolution.intros*)

lemma *Resolution-unnecessary*:
  assumes *sd*: $\forall\, C \in T.\ S \vdash C$
  shows $T \cup S \vdash D \longleftrightarrow S \vdash D$ (**is** *?l* ⟷ *?r*)
proof
  assume *?l*
  from ‹*?l*› *sd* **show** *?r*
  proof(*induction* $T \cup S$ *D rule*: *Resolution.induct*)
    case (*Ass D*)
    show *?case* **proof** *cases*
      assume $D \in S$ **with** *Resolution.Ass* **show** *?thesis* **.**
    next
      assume $D \notin S$
      with *Ass.hyps* **have** $D \in T$ **by** *simp*
      with *Ass.prems* **show** *?thesis* **by** *blast*
    qed
  next
    case (*R D H k*) **thus** *?case* **by** (*simp add*: *Resolution.R*)
  qed
next
  assume *?r* **with** *Resolution-weaken* **show** *?l* **by** *blast*
qed

lemma *Resolution-taint-assumptions*: $S \cup T \vdash C \Longrightarrow \exists\, R \subseteq D.\ ((\cup)\ D\ `\ S) \cup T \vdash R \cup C$

proof(*induction* $S \cup T$ *C rule*: *Resolution.induct*)
 case (*Ass C*)
  show *?case* **proof**(*cases* $C \in S$)
    case *True*
    hence $D \cup C \in (\cup)\ D\ `\ S \cup T$ **by** *simp*
    with *Resolution.Ass* **have** $((\cup)\ D\ `\ S) \cup T \vdash D \cup C$ **.**
    thus *?thesis* **by** *blast*

next
  case *False*
  with *Ass* **have** $C \in T$ **by** *simp*
  hence $((\cup)\ D\ `\ S) \cup T \vdash C$ **by**(*simp add*: *Resolution.Ass*)
  thus *?thesis* **by**(*intro exI*[**where** x={}]) *simp*
qed
next
  case (*R C1 C2 k*)
  let $?m = ((\cup)\ D\ `\ S) \cup T$
  from *R* **obtain** *R1* **where** *IH1*: $R1 \subseteq D$ $?m \vdash R1 \cup C1$ **by** *blast*
  from *R* **obtain** *R2* **where** *IH2*: $R2 \subseteq D$ $?m \vdash R2 \cup C2$ **by** *blast*
  from *R* **have** $k^+ \in R1 \cup C1$ $k^{-1} \in R2 \cup C2$ **by** *simp-all*
  note *Resolution.R*[*OF IH1*(*2*) *IH2*(*2*) *this*]
  hence $?m \vdash (R1 - \{k^+\}) \cup (R2 - \{k^{-1}\}) \cup (C1 - \{k^+\} \cup (C2 - \{k^{-1}\}))$
    by (*simp add*: *Un-Diff Un-left-commute sup.assoc*)
  moreover have $(R1 - \{k^+\}) \cup (R2 - \{k^{-1}\}) \subseteq D$
    using *IH1*(*1*) *IH2*(*1*) **by** *blast*
  ultimately show *?case* **by** *blast*
qed

Resolution is "strange": Given a set of clauses that is presumed to be satisfiable, it derives new clauses that can be added while preserving the satisfiability of the set of clauses. However, not every clause that could be added while keeping satisfiability can actually be added by resolution. Especially, the above lemma *Resolution-taint-assumptions* gives us the derivability of a clause $R \cup C$, where $R \subseteq D$. What we might actually want is the derivability of $D \cup C$. Any model that satisfies $R \cup C$ obviously satisfies $D \cup C$ (since they are disjunctions), but Resolution only allows to derive the former.

Nevertheless, *Resolution-taint-assumptions*, can still be a quite useful lemma: picking D to be a singleton set only leaves two possibilities for R.

lemma *Resolution-useless-infinite*:
assumes *R*: $S \vdash R$
assumes *finite R*
shows $\exists S' \subseteq S.$ *Ball* S' *finite* $\wedge$ *finite* S' $\wedge$ $(S' \vdash R)$
using *assms* **proof**(*induction rule*: *Resolution.induct*)
  case (*Ass C S*) **thus** *?case* **using** *Resolution.Ass* **by**(*intro exI*[**where** x={C}]) *auto*
next
  case (*R S C D k*)
  from *R.prems* **have** *finite C finite D* **by** *simp-all*
  with *R.IH* **obtain** *SC SD* **where** *IH*:
    $SC \subseteq S$ $(\forall C \in SC.$ *finite* $C)$ *finite SC* $SC \vdash C$
    $SD \subseteq S$ $(\forall D \in SD.$ *finite* $D)$ *finite SD* $SD \vdash D$
    by *blast*
  hence *IHw*: $SC \cup SD \vdash C$ $SC \cup SD \vdash D$ **using** *Resolution-weaken*
    by(*simp-all add*: *sup-commute Resolution-weaken*)
  with *IH*(*1*−*3*,*5*−*7*) **show** *?case*

by(*blast intro*!: *exI*[**where** *x*=*SC* ∪ *SD*] *Resolution.R*[*OF* - - ‹k^+ ∈ *C*› ‹k^{-1} ∈ *D*›])
qed

Now we define and verify a toy resolution prover. Function *res* computes the set of resolvents of a clause set:

context begin

definition *res* :: *'a clause set* ⇒ *'a clause set* **where**
res S =
(⋃ *C1* ∈ *S*. ⋃ *C2* ∈ *S*. ⋃ *L1* ∈ *C1*. ⋃ *L2* ∈ *C2*.
(*case* (*L1*,*L2*) *of* (*Pos i*,*Neg j*) ⇒ *if i*=*j then* {(*C1* − {*Pos i*}) ∪ (*C2* − {*Neg j*})} *else* {}
| - ⇒ {}))

private definition *ex1* ≡ {{*Neg* (*0*::*int*)}, {*Pos 0*, *Pos 1*, *Neg 2*}, {*Pos 0*, *Pos 1*, *Pos 2*}, {*Pos 0*, *Neg 1*}}
value *res ex1*

definition *Rwhile* :: *'a clause set* ⇒ *'a clause set option* **where**
Rwhile = *while-option* (λ*S*. □ ∉ *S* ∧ ¬ *res S* ⊆ *S*) (λ*S*. *res S* ∪ *S*)

value [*code*] *Rwhile ex1*
lemma □ ∈ *the* (*Rwhile ex1*) **by** *eval*

lemma *Rwhile-sound*: **assumes** *Rwhile S* = *Some S'*
shows ∀ *C* ∈ *S'*. *Resolution S C*
apply(*rule while-option-rule*[*OF* - *assms*[*unfolded Rwhile-def*]])
apply (*auto simp*: *Ass R res-def split*: *if-splits literal.splits*)
done

definition *all-clauses S* = {*s*. *s* ⊆ {*Pos k*|*k*. *k* ∈ *atoms-of-cnf S*} ∪ {*Neg k*|*k*. *k* ∈ *atoms-of-cnf S*}}
lemma *s-sub-all-clauses*: *S* ⊆ *all-clauses S*
unfolding *all-clauses-def*
apply(*rule*)
apply(*simp*)
apply(*rule*)
apply(*simp add*: *atoms-of-cnf-alt lit-atoms-cases*[*abs-def*])
by (*metis imageI literal.exhaust literal.simps*(*5*) *literal.simps*(*6*))
lemma *atoms-res*: *atoms-of-cnf* (*res S*) ⊆ *atoms-of-cnf S*
unfolding *res-def atoms-of-cnf-alt*
apply (*clarsimp simp*: *lit-atoms-cases* [*abs-def*] *split*: *literal.splits if-splits*)
apply (*clarsimp simp add*: *image-iff*)
apply *force*
done

lemma *exlitE*: $(\bigwedge x.\ xe = Pos\ x \Longrightarrow P\ x) \Longrightarrow (\bigwedge x.\ xe = Neg\ x \Longrightarrow False) \Longrightarrow \exists x.\ xe = Pos\ x \land P\ x$
by(*cases xe*) *auto*
lemma *res-in-all-clauses*: *res S* ⊆ *all-clauses S*
apply (*clarsimp simp*: *res-def all-clauses-def atoms-of-cnf-alt lit-atoms-cases split*: *literal.splits if-splits*)
apply (*clarsimp simp add*: *image-iff*)
apply (*metis atoms-of-lit.simps*(*1*) *atoms-of-lit.simps*(*2*) *lit-atoms-cases literal.exhaust*)
done

lemma *Res-in-all-clauses*: *res S* ∪ *S* ⊆ *all-clauses S*
by (*simp add*: *res-in-all-clauses s-sub-all-clauses*)
lemma *all-clauses-Res-inv*: *all-clauses* (*res S* ∪ *S*) = *all-clauses S*
unfolding *all-clauses-def atoms-of-cnf-Un*
using *atoms-res* **by** *fast*
lemma *all-clauses-finite*: *finite S* ∧ (∀ *C* ∈ *S*. *finite C*) ⟹ *finite* (*all-clauses S*)
unfolding *all-clauses-def atoms-of-cnf-def* **by** *simp*
lemma *finite-res*: ∀ *C* ∈ *S*. *finite C* ⟹ ∀ *C* ∈ *res S*. *finite C*
unfolding *res-def* **by**(*clarsimp split*: *literal.splits*)

lemma *finite T* ⟹ *S* ⊆ *T* ⟹ *card S* < *Suc* (*card T*)
by (*simp add*: *card-mono le-imp-less-Suc*)

lemma *finite S* ∧ (∀ *C* ∈ *S*. *finite C*) ⟹ ∃ *T*. *Rwhile S* = *Some T*
apply(*unfold Rwhile-def*)
apply(*rule measure-while-option-Some*[*rotated*, **where** *f*=λ*T*. *Suc* (*card* (*all-clauses S*)) − *card T*
and *P*=λ*T*. *finite T* ∧ (∀ *C* ∈ *T*. *finite C*) ∧ *all-clauses T* = *all-clauses S*])
apply(*simp*;*fail*)
apply(*intro conjI*)
subgoal by (*meson all-clauses-finite finite-UnI finite-subset res-in-all-clauses*)
subgoal using *finite-res* **by** *blast*
subgoal using *all-clauses-Res-inv* **by** *blast*
subgoal
apply(*rule diff-less-mono2*)
subgoal by (*metis Res-in-all-clauses all-clauses-finite card-seteq finite-subset not-le sup-commute sup-ge2*)
subgoal apply(*intro card-mono le-imp-less-Suc*)
subgoal using *all-clauses-finite* **by** *blast*
subgoal using *s-sub-all-clauses* **by** *blast*
done
done
done

partial-function(*option*) *Res* **where**
Res S = (*let R* = *res S* ∪ *S in if R* = *S then Some S else Res R*)
declare *Res.simps*[*code*]

value [*code*] *Res ex1*

```
lemma □ ∈ the (Res ex1) by code-simp

lemma res: C ∈ res S ⟹ S ⊢ C
  unfolding res-def by(auto split: literal.splits if-splits) (metis Resolution.simps literal.exhaust)

lemma Res-sound: Res S = Some S′ ⟹ (∀ C ∈ S′. S ⊢ C)
proof (induction arbitrary: S S′ rule: Res.fixp-induct)
  fix X S S′
  assume IH: ⋀S S′. X S = Some S′ ⟹ (∀ C∈S′. S ⊢ C)
  assume prem: (let R = res S ∪ S in if R = S then Some S else X R) = Some S′
  thus (∀ C∈S′. S ⊢ C)
  proof cases
    assume res S ∪ S = S
    with prem show ?thesis by (simp add: Resolution.Ass)
  next
    assume 1: res S ∪ S ≠ S
    with prem have X (res S ∪ S) = Some S′ by simp
    with IH have ∀ C∈S′. res S ∪ S ⊢ C by blast
    thus ?thesis using Resolution-unnecessary res by blast
  qed
qed (fast intro!: option-admissible)+

lemma Res-terminates: finite S ⟹ ∀ C ∈ S. finite C ⟹ ∃ T. Res S = Some T
proof(induction card (all-clauses S) − card S arbitrary: S rule: less-induct)
  case less
  let ?r = res S ∪ S
  show ?case proof(cases ?r = S)
    case False
     have b: finite (res S ∪ S) by (meson less Res-in-all-clauses all-clauses-finite infinite-super)
    have c: Ball (res S ∪ S) finite using less.prems(2) finite-res by auto
    have card S < card ?r by (metis False b psubsetI psubset-card-mono sup-ge2)
    moreover have card ?r ≤ (card (all-clauses S))
     by (meson less Res-in-all-clauses all-clauses-finite card-mono le-imp-less-Suc)
    ultimately have a: (card (all-clauses ?r)) − card ?r < (card (all-clauses S)) − card S
      using all-clauses-Res-inv[of S] by simp
    from less(1)[OF a b c] show ?thesis by (subst Res.simps) (simp add: Let-def)
  qed (simp add: Res.simps)
qed


code-pred Resolution .
print-theorems

end
end
theory Resolution-Sound
```

imports *Resolution CNF-Formulas-Sema*
begin

lemma *Resolution-insert*: $S \vdash R \Longrightarrow$ *cnf-semantics* $\mathcal{A}$ $S \Longrightarrow$ *cnf-semantics* $\mathcal{A}$ $\{R\}$
by(*induction rule*: *Resolution.induct*;
clarsimp simp add: *cnf-semantics-def clause-semantics-def lit-semantics-cases split*: *literal.splits*;
blast)

lemma $S \vdash R \Longrightarrow$ *cnf-semantics* $\mathcal{A}$ $S \longleftrightarrow$ *cnf-semantics* $\mathcal{A}$ $(R \triangleright S)$
using *Resolution-insert cnf-semantics-def* **by** (*metis insert-iff*)

corollary *Resolution-cnf-sound*: **assumes** $S \vdash \Box$ **shows** $\neg$ *cnf-semantics* $\mathcal{A}$ S
proof(*rule notI*)
assume *cnf-semantics* $\mathcal{A}$ S
with *Resolution-insert assms* **have** *cnf-semantics* $\mathcal{A}$ $\{\Box\}$ **.**
thus *False* **by**(*simp add*: *cnf-semantics-def clause-semantics-def*)
qed

corollary *Resolution-sound*:
assumes *rp*: *cnf* (*nnf* F) $\vdash \Box$
shows $\neg$ $\mathcal{A} \models F$
proof $-$
from *Resolution-cnf-sound rp* **have** $\neg$ *cnf-semantics* $\mathcal{A}$ (*cnf* (*nnf* F)) **.**
hence $\neg\mathcal{A} \models$ *nnf* F **unfolding** *cnf-semantics*[*OF is-nnf-nnf*] **.**
thus *?thesis* **unfolding** *nnf-semantics* **.**
qed

end

2.4.1 Completeness

theory *Resolution-Compl*
imports *Resolution CNF-Sema*
begin

Completeness proof following Schöning [9].

definition *make-lit v a* $\equiv$ *case v of True* $\Rightarrow$ *Pos a* | *False* $\Rightarrow$ *Neg a*

definition *restrict-cnf-atom a v C* $\equiv$ $\{c - \{$*make-lit* $(\neg v)$ $a\}$ | $c.$ $c \in C \wedge$ *make-lit v a* $\notin c\}$

lemma *restrict-cnf-remove*: *atoms-of-cnf* (*restrict-cnf-atom a v c*) $\subseteq$
atoms-of-cnf $c - \{a\}$
unfolding *restrict-cnf-atom-def atoms-of-cnf-alt lit-atoms-cases make-lit-def*

by (*force split*: *literal.splits bool.splits*)

lemma *cnf-substitution-lemma*:
cnf-semantics A (*restrict-cnf-atom a v S*) = *cnf-semantics* ($A(a := v)$) *S*
unfolding *restrict-cnf-atom-def cnf-semantics-def clause-semantics-def lit-semantics-cases make-lit-def*
apply (*clarsimp split*: *bool.splits literal.splits*)
apply *safe*
subgoal for *s* **by**(*fastforce elim*!: *allE*[*of - s* − {*Neg a*}])
subgoal by (*metis DiffI singletonD*)
subgoal for *s* **by**(*fastforce elim*!: *allE*[*of - s* − {*Pos a*}])
subgoal by (*metis DiffI singletonD*)
done

lemma *finite-restrict*: *finite S* ⟹ *finite* (*restrict-cnf-atom a v S*)
unfolding *restrict-cnf-atom-def* **by**(*simp add*: *image-iff*)

The next lemma describes what we have to (or can) do to a CNF after it has been mangled by *restrict-cnf-atom* to get back to (a subset of) the original CNF. The idea behind this will be clearer upon usage.

lemma *unrestrict-effects*:
(λc. *if* {*make-lit* ($\neg v$) *a*} ∪ *c* ∈ *S then* {*make-lit* ($\neg v$) *a*} ∪ *c else c*) ‘ *restrict-cnf-atom a v S* ⊆ *S*
proof −
have ⟦*xa* ∈ *restrict-cnf-atom a v S*; {*make-lit* (¬ *v*) *a*} ∪ *xa* ∉ *S*; *x* = *xa*⟧ ⟹ *xa* ∈ *S* **for** *x xa*
unfolding *restrict-cnf-atom-def* **using** *insert-Diff* **by** *fastforce*
hence *x* ∈ (λc. *if* {*make-lit* (¬ *v*) *a*} ∪ *c* ∈ *S then* {*make-lit* (¬ *v*) *a*} ∪ *c else c*) ‘ *restrict-cnf-atom a v S* ⟹ *x* ∈ *S* **for** *x*
unfolding *image-iff* **by**(*elim bexE*) *simp*
thus *?thesis* **..**
qed

lemma *can-cope-with-unrestrict-effects*:
assumes *pr*: *S* ⊢ □
assumes *su*: *S* ⊆ *T*
shows ∃ *R* ⊆ {*make-lit v a*}. (λc. *if c* ∈ *n then* {*make-lit v a*} ∪ *c else c*) ‘ *T* ⊢ *R*
proof −
from *Resolution-taint-assumptions*[**where** *D*={*make-lit v a*}]
have *taint*: Γ ∪ Λ ⊢ □ ⟹ ∃ *R*⊆{*make-lit v a*}. *insert* (*make-lit v a*) ‘ Γ ∪ Λ ⊢ *R*
for Γ Λ **by** (*metis image-cong insert-def sup-bot.right-neutral*)
have *S*: *S* = {*c* ∈ *S*. *c* ∈ *n*} ∪ {*c* ∈ *S*. *c* ∉ *n*} **by** *blast*
hence *SI*: (λc. *if c* ∈ *n then* {*make-lit v a*} ∪ *c else c*) ‘ *S* =
(*insert* (*make-lit v a*) ‘ {*c* ∈ *S*. *c* ∈ *n*}) ∪ {*c* ∈ *S*. *c* ∉ *n*}
by *auto*
from *pr* **have** ∃ *R* ⊆ {*make-lit v a*}.
(λc. *if c* ∈ *n then* {*make-lit v a*} ∪ *c else c*) ‘ *S* ⊢ *R*

apply(*subst SI*)
apply(*subst*(*asm*) *S*)
apply(*elim taint*)
done
thus *?thesis* **using** *Resolution-weaken su* **by** (*metis* (*no-types*, *lifting*) *image-Un sup.order-iff*)
qed

lemma *unrestrict'*:
fixes *R* :: *'a clause*
assumes *rp*: *restrict-cnf-atom a v S* ⊢ □
shows ∃ *R* ⊆ {*make-lit* (¬*v*) *a*}. *S* ⊢ *R*
proof −
fix *C* :: *'a clause* **fix** *k* :: *'a*
from *unrestrict-effects*[*of v a S*]

The idea is that the restricted set lost some clauses, and that some others were crippled. So, there must be a set of clauses to heal and a set of clauses to reinsert to get the original. (Mind you, this is not exactly what is happening, because e.g. both C and $\{k^{-1}\} \cup C$ might be in there and get reduced to one C. You then heal that C to $\{k^{-1}\} \cup C$ and insert the shadowed C... Details.)

obtain *n* **where** *S*:
(λ*c*. *if c* ∈ *n then* {*make-lit* (¬ *v*) *a*} ∪ *c else c*) ' *restrict-cnf-atom a v S* ⊆ *S*
using *exI*[**where** *x*={*c*. {*make-lit* (¬*v*) *a*} ∪ *c* ∈ *S*}] **by** *force*
note *finite-restrict S*
show *?thesis* **using** *can-cope-with-unrestrict-effects*[*OF rp*]
by (*metis* (*no-types*) *S Resolution-weaken subset-refl sup.order-iff*)
qed

lemma *Resolution-complete-standalone-finite*:
assumes *ns*: ∀ 𝒜. ¬*cnf-semantics* 𝒜 *S*
assumes *fin*: *finite* (*atoms-of-cnf S*)
shows *S* ⊢ □
using *fin ns*
proof(*induction atoms-of-cnf S arbitrary*: *S rule*: *finite-psubset-induct*)
case *psubset*
show *?case* **proof**(*cases*)
assume *e*: *atoms-of-cnf S* = {}
from ‹∀ 𝒜. ¬ *cnf-semantics* 𝒜 *S*› **have** *S* ≠ {} **unfolding** *cnf-semantics-def* **by** *blast*
with *e* **have** *S* = {□} **unfolding** *atoms-of-cnf-def* **by** *simp fast*
thus *?case* **using** *Resolution.Ass* **by** *blast*
next
have *unsat-restrict*: ∀ 𝒜. ¬ *cnf-semantics* 𝒜 (*restrict-cnf-atom a v S*) **for** *a v*
using ‹∀ 𝒜. ¬ *cnf-semantics* 𝒜 *S*› **by**(*simp add*: *cnf-substitution-lemma*)
assume *ne*: *atoms-of-cnf S* ≠ {}
then obtain *a* **where** *a* ∈ *atoms-of-cnf S* **by** *blast*
hence *atoms-of-cnf* (*restrict-cnf-atom a v S*) ⊂ *atoms-of-cnf S* **for** *v*

using *restrict-cnf-remove*[**where** *'a='a*] **by** *blast*
from *psubset(2)*[*OF this unsat-restrict*]
have *IH*: *restrict-cnf-atom a v S* ⊢ □ **for** *v* **.**
from *unrestrict'*[*OF IH*, *of* ¬ -] **have** *unr-IH*: ∃ *R*⊆{*make-lit v a*}. *S* ⊢ *R*
for *v* **by** *simp*
from *this*[*of False*] *this*[*of True*] **show** *?case* **using** *Resolution.R*[*OF - - singletonI singletonI*]
by (*simp add*: *make-lit-def*) (*fast dest*: *subset-singletonD*)
qed
qed

What you might actually want is ∀ 𝒜. ¬ *cnf-semantics* 𝒜 *S* ⟹ *S* ⊢ □. Unfortunately, applying compactness (to get a finite set with a finite number of atoms) here is problematic: You would need to convert all clauses to disjunction-formulas, but there might be clauses with an infinite number of atoms. Removing those has to be done before applying compactness, we would possibly have to remove an infinite number of infinite clauses. Since the notion of a formula with an infinite number of atoms is not exactly standard, it is probably better to just skip this.

end
theory *Resolution-Compl-Consistency*
imports *Resolution Consistency CNF-Formulas CNF-Formulas-Sema*
begin

lemma *OrI2'*: (¬*P* ⟹ *Q*) ⟹ *P* ∨ *Q* **by** *auto*

lemma *atomD*: *Atom k* ∈ *S* ⟹ {*Pos k*} ∈ ⋃(*cnf* ' *S*) *Not* (*Atom k*) ∈ *S* ⟹ {*Neg k*} ∈ ⋃(*cnf* ' *S*) **by** *force+*

lemma *pcp-disj*:
⟦*F* ∨ *G* ∈ Γ; (∀ *xa*. (*xa* = *F* ∨ *xa* ∈ Γ) ⟶ *is-cnf xa*) ⟶ (*cnf F* ∪ (⋃*x*∈Γ. *cnf x*) ⊢ □); (∀ *xa*. (*xa* = *G* ∨ *xa* ∈ Γ) ⟶ *is-cnf xa*) ⟶ (*cnf G* ∪ (⋃*x*∈Γ. *cnf x*) ⊢ □); ∀ *x*∈Γ. *is-cnf x*⟧
⟹ (⋃*x*∈Γ. *cnf x*) ⊢ □
proof *goal-cases*
case *1*
from *1*(*1*,*4*) **have** *is-cnf* (*F* ∨ *G*) **by** *blast*
hence *db*: *is-disj F is-lit-plus F is-disj G* **by**(*cases F*; *simp*)+
hence *is-cnf F* ∧ *is-cnf G* **by**(*cases F*; *cases G*; *simp*)
with *1* **have** *IH*: (⋃(*cnf* ' (*F* ▷ Γ))) ⊢ □ (⋃(*cnf* ' (*G* ▷ Γ))) ⊢ □ **by** *simp-all*
let *?*Γ = (⋃(*cnf* ' Γ))
from *IH* **have** *IH-readable*: *cnf F* ∪ *?*Γ ⊢ □ *cnf G* ∪ *?*Γ ⊢ □ **by** *auto*
show *?case* **proof**(*cases cnf F* = {} ∨ *cnf G* = {})
case *True*
hence *cnf* (*F* ∨ *G*) = {} **by** *auto*
thus *?thesis* **using** *True IH* **by** *auto*
next
case *False*

then obtain *S T* **where** *ST*: *cnf F* = {*S*} *cnf G* = {*T*}
using *cnf-disj-ex db*(*1*,*3*) **by** *metis*

hence *R*: *cnf* (*F* ∨ *G*) = { *S* ∪ *T* } **by** *simp*
have ⟦*S*▷*?Γ* ⊢ □; *T*▷*?Γ* ⊢ □⟧ ⟹ *S* ∪ *T*▷ *?Γ* ⊢ □ **proof** −
assume *s*: *S*▷*?Γ* ⊢ □ **and** *t*: *T*▷*?Γ* ⊢ □
hence *s-w*: *S* ▷ *S* ∪ *T* ▷ *?Γ* ⊢ □ **using** *Resolution-weaken* **by** (*metis insert-commute insert-is-Un*)
note *Resolution-taint-assumptions*[*of* {*T*} *?Γ* □ *S*] *t*
then obtain *R* **where** *R*: *S* ∪ *T* ▷ *?Γ* ⊢ *R R*⊆*S* **by** (*auto simp*: *Un-commute*)
have *literal-subset-sandwich*: *R* = □ ∨ *R* = *S* **if** *is-lit-plus F cnf F* = {*S*} *R* ⊆ *S*
using *that* **by**(*cases F rule*: *is-lit-plus.cases*; *simp*) *blast*+
show *?thesis* **using** *literal-subset-sandwich*[*OF db*(*2*) *ST*(*1*) *R*(*2*)] **proof**
assume *R* = □ **thus** *?thesis* **using** *R*(*1*) **by** *blast*
next
from *Resolution-unnecessary*[**where** *T*={-}, *simplified*] *R*(*1*)
have (*R* ▷ *S* ∪ *T* ▷ *?Γ* ⊢ □) = (*S* ∪ *T* ▷ *?Γ* ⊢ □) **.**
moreover assume *R* = *S*
ultimately show *?thesis* **using** *s-w* **by** *simp*
qed
qed
thus *?thesis* **using** *IH ST R 1*(*1*) **by** (*metis UN-insert insert-absorb insert-is-Un*)
qed
qed

lemma *R-consistent*: *pcp* {Γ|Γ. ¬((∀ γ ∈ Γ. *is-cnf* γ) ⟶ ((⋃ (*cnf* ‘ Γ)) ⊢ □))}
unfolding *pcp-def*
unfolding *Ball-def*
unfolding *mem-Collect-eq*
apply(*intro allI impI*)
apply(*erule contrapos-pp*)
apply(*unfold not-ex de-Morgan-conj de-Morgan-disj not-not not-all not-imp disj-not1*)
apply(*intro impI allI*)
apply(*elim disjE exE conjE*; *intro OrI2′*)
apply(*unfold not-ex de-Morgan-conj de-Morgan-disj not-not not-all not-imp disj-not1 Ball-def*[*symmetric*])
apply *safe*
apply (*metis Ass Pow-bottom Pow-empty UN-I cnf.simps*(*3*))
apply (*metis Diff-insert-absorb Resolution.simps insert-absorb singletonI sup-bot.right-neutral atomD*)
apply (*simp*; *metis* (*no-types*, *opaque-lifting*) *UN-insert cnf.simps*(*5*) *insert-absorb is-cnf.simps*(*1*) *sup-assoc*)
apply (*auto intro*: *pcp-disj*)
done

theorem *Resolution-complete*:
fixes *F* :: ′*a* :: *countable formula*

shows $\models F \Longrightarrow cnf\ (nnf\ (\neg F)) \vdash \Box$
proof(*erule contrapos-pp*)
assume c: $\neg\ (cnf\ (nnf\ (\neg\ F)) \vdash \Box)$
have $\{cnf\text{-}form\text{-}of\ (nnf\ (\neg F))\} \in \{\Gamma\ |\Gamma.\ \neg\ ((\forall\gamma{\in}\Gamma.\ is\text{-}cnf\ \gamma) \longrightarrow \bigcup\ (cnf\ `\ \Gamma) \vdash \Box)\}$
by(*simp add*: *cnf-cnf*[*OF is-nnf-nnf*] *c cnf-form-of-is*[*OF is-nnf-nnf*])
from *pcp-sat*[*OF R-consistent this*] **have** *sat* $\{cnf\text{-}form\text{-}of\ (nnf\ (\neg\ F))\}$ **.**
thus $\neg \models F$ **by**(*simp add*: *sat-def cnf-form-semantics*[*OF is-nnf-nnf*] *nnf-semantics*)
qed

end

3 Proof Transformation

This is organized as a ring closure

3.1 HC to SC

theory *HCSC*
imports *HC SC-Cut*
begin

lemma *extended-AxE*[*intro*!]: $F,\ \Gamma \Rightarrow F,\ \Delta$ **by** (*intro extended-Ax*) (*simp add*: *add.commute inter-add-right2*)

theorem *HCSC*: $AX10 \cup set\text{-}mset\ \Gamma \vdash_H F \Longrightarrow \Gamma \Rightarrow \{\#F\#\}$
proof(*induction rule*: *HC.induct*)
case (*Ax F*) **thus** *?case* **proof**
note *SCp.intros*(*3*−)[*intro*!]

Essentially, we need to prove all the axioms of Hibert Calculus in Sequent Calculus.

have A: $\Gamma \Rightarrow \{\#F \to (F \vee G)\#\}$ **for** $F\ G$ **by** *blast*
have B: $\Gamma \Rightarrow \{\#G \to (F \vee G)\#\}$ **for** $G\ F$ **by** *blast*
have C: $\Gamma \Rightarrow \{\#(F \to H) \to ((G \to H) \to ((F \vee G) \to H))\#\}$ **for** $F\ H\ G$ **by** *blast*
have D: $\Gamma \Rightarrow \{\#(F \wedge G) \to F\#\}$ **for** $F\ G$ **by** *blast*
have E: $\Gamma \Rightarrow \{\#(F \wedge G) \to G\#\}$ **for** $F\ G$ **by** *blast*
have F: $\Gamma \Rightarrow \{\#F \to (G \to (F \wedge G))\#\}$ **for** $F\ G$ **by** *blast*
have G: $\Gamma \Rightarrow \{\#(F \to \bot) \to \neg\ F\#\}$ **for** F **by** *blast*
have H: $\Gamma \Rightarrow \{\#\neg\ F \to (F \to \bot)\#\}$ **for** F **by** *blast*
have I: $\Gamma \Rightarrow \{\#(\neg\ F \to \bot) \to F\#\}$ **for** F **by** *blast*
have K: $\Gamma \Rightarrow \{\#F \to (G \to F)\#\}$ **for** $F\ G$ **by** *blast*
have L: $\Gamma \Rightarrow \{\#(F \to (G \to H)) \to ((F \to G) \to (F \to H))\#\}$ **for** $F\ G\ H$ **by** *blast*
have J: $F \in AX0 \Longrightarrow \Gamma \Rightarrow \{\#F\#\}$ **for** F **by**(*induction rule*: *AX0.induct*; *intro K L*)

assume *F* ∈ *AX10* **thus** *?thesis* **by**(*induction rule*: *AX10.induct*; *intro A B C D E F G H I J*)
next
assume *F* ∈ *set-mset* Γ **thus** *?thesis* **by**(*intro extended-Ax*) *simp*
qed
next
case (*MP F G*)
from *MP* **have** *IH*: Γ ⇒ {#*F*#} Γ ⇒ {#*F* → *G*#} **by** *blast+*
with *ImpR-inv*[**where** Δ=‹{#}›, *simplified*] **have** *F*,Γ ⇒ {#*G*#} **by** *auto*
moreover from *IH*(*1*) *weakenR* **have** Γ ⇒ *F*, {#*G*#} **by** *blast*
ultimately show Γ ⇒ {#*G*#} **using** *cut*[**where** *F*=*F*] **by** *simp*
qed

end

3.2 SC to ND

theory *SCND*
imports *SC ND*
begin

lemma *SCND*: Γ ⇒ Δ ⟹ (*set-mset* Γ) ∪ *Not* ‘ (*set-mset* Δ) ⊢ ⊥
proof(*induction* Γ Δ *rule*: *SCp.induct*)
case *BotL* **thus** *?case* **by** (*simp add*: *ND.Ax*)
next
case *Ax* **thus** *?case* **by** (*meson ND.Ax NotE UnCI image-iff*)
next
case *NotL* **thus** *?case* **by** (*simp add*: *NotI*)
next
case (*NotR F* Γ Δ) **thus** *?case* **by** (*simp add*: *Not2IE*)
next
case (*AndL F G* Γ Δ) **thus** *?case* **by** (*simp add*: *AndL-sim*)
next
case (*AndR* Γ *F* Δ *G*) **thus** *?case* **by**(*simp add*: *AndR-sim*)
next
case *OrL* **thus** *?case* **by** (*simp add*: *OrL-sim*)
next
case *OrR* **thus** *?case* **using** *OrR-sim*[**where** ′*a*=′*a*] **by** (*simp add*: *insert-commute*)
next
case (*ImpL* Γ *F* Δ *G*) **from** *ImpL.IH* **show** *?case* **by** (*simp add*: *ImpL-sim*)
next
case *ImpR* **from** *ImpR.IH* **show** *?case* **by** (*simp add*: *ImpR-sim*)
qed

end

3.3 ND to HC

theory *NDHC*
imports *ND HC*

begin

The fundamental difference between the two is that Natural Deduction updates its set of assumptions while Hilbert Calculus does not. The Deduction Theorem $AX0 \cup (?F \triangleright ?\Gamma) \vdash_H ?G \Longrightarrow AX0 \cup ?\Gamma \vdash_H ?F \rightarrow ?G$ helps with this.

theorem *NDHC*: $\Gamma \vdash F \Longrightarrow AX10 \cup \Gamma \vdash_H F$
proof(*induction rule*: *ND.induct*)
  case *Ax* **thus** *?case* **by**(*auto intro*: *HC.Ax*)
next
  case *NotE* **thus** *?case* **by** (*meson AX10.intros*(*9*) *HC.simps subsetCE sup-ge1*)
next
  case (*NotI F* Γ)
  from *HC-intros*(*11*) **have** *HC-NotI*: $AX10 \cup \Gamma \vdash_H A \rightarrow \bot \Longrightarrow AX10 \cup \Gamma \vdash_H \neg A$ **for** *A*
    using *MP HC-mono* **by** (*metis sup-ge1*)
  from *NotI* **show** *?case* **using** *Deduction-theorem*[**where** $\Gamma = AX10 \cup \Gamma$] *HC-NotI* **by** (*metis AX100 Un-assoc Un-insert-right*)
next
  case (*CC F* Γ)
  hence $AX10 \cup \Gamma \vdash_H \neg F \rightarrow \bot$ **using** *Deduction-theorem*[**where** $\Gamma = AX10 \cup \Gamma$] **by** (*metis AX100 Un-assoc Un-insert-right*)
  thus $AX10 \cup \Gamma \vdash_H F$ **using** *AX10.intros*(*10*) **by** (*metis HC.simps UnCI*)
next
  case (*AndE1* Γ *F G*) **thus** *?case* **by** (*meson AX10.intros*(*5*) *HC.simps UnCI*)
next
  case (*AndE2* Γ *F G*) **thus** *?case* **by** (*meson AX10.intros*(*6*) *HC.simps UnCI*)
next
  case (*AndI* Γ *F G*) **thus** *?case* **by** (*meson HC-intros*(*10*) *HC-mono HC.simps sup-ge1*)
next
  case (*OrE* Γ *F G H*)
  from ‹$AX10 \cup (F \triangleright \Gamma) \vdash_H H$› ‹$AX10 \cup (G \triangleright \Gamma) \vdash_H H$› **have**
      $AX10 \cup \Gamma \vdash_H F \rightarrow H$ $AX10 \cup \Gamma \vdash_H G \rightarrow H$
    using *Deduction-theorem*[**where** $\Gamma = AX10 \cup \Gamma$] **by** (*metis AX100 Un-assoc Un-insert-right*)+
  with *HC-intros*(*7*)[*THEN HC-mono*[*OF - sup-ge1*]] *MP*
  have $AX10 \cup \Gamma \vdash_H (F \vee G) \rightarrow H$ **by** *metis*
  with *MP* ‹$AX10 \cup \Gamma \vdash_H F \vee G$› **show** *?case* **.**
next
  case (*OrI1* Γ *F G*) **thus** *?case* **by** (*meson AX10.intros*(*2*) *HC.simps UnCI*)
next
  case (*OrI2* Γ *F G*) **thus** *?case* **by** (*meson AX10.intros*(*3*) *HC.simps UnCI*)
next
  case (*ImpE* Γ *F G*)
  from *MP* ‹$AX10 \cup \Gamma \vdash_H F$› ‹$AX10 \cup \Gamma \vdash_H F \rightarrow G$› **show** *?case* **.**
next
  case (*ImpI F* Γ *G*) **thus** *?case* **using** *Deduction-theorem*[**where** $\Gamma = AX10 \cup \Gamma$] **by** (*metis AX100 Un-assoc Un-insert-right*)

qed

end

3.4 HC, SC, and ND

theory *HCSCND*
imports *HCSC SCND NDHC*
begin

theorem *HCSCND*:
  defines *hc F* ≡ *AX10* $\vdash_H$ *F*
  defines *nd F* ≡ {} ⊢ *F*
  defines *sc F* ≡ {#} ⇒ {# *F* #}
  shows *hc F* ⟷ *nd F* **and** *nd F* ⟷ *sc F* **and** *sc F* ⟷ *hc F*
using *HCSC*[**where** *F*=*F* **and** Γ=‹{#}›, *simplified*]
    SCND[**where** Γ=‹{#}› **and** Δ={#*F*#}] *ND.ND.CC*[**where** *F*=*F* **and** Γ={}]
      NDHC[**where** Γ={} **and** *F*=*F*]
by(*simp-all add*: *assms*) *blast*+

end

3.5 Transforming SC proofs into proofs of CNFs

theory *LSC*
imports *CNF-Formulas SC-Cut*
begin

Left handed SC with NNF transformation:

inductive *LSC* (‹(- $\Rightarrow_n$)› [*53*]) **where**
— logic:
Ax: ¬(*Atom k*),*Atom k*,Γ $\Rightarrow_n$ |
BotL: ⊥,Γ $\Rightarrow_n$ |
AndL: *F*,*G*,Γ $\Rightarrow_n$ ⟹ *F*∧*G*,Γ $\Rightarrow_n$ |
OrL: *F*,Γ $\Rightarrow_n$ ⟹ *G*,Γ $\Rightarrow_n$ ⟹ *F*∨*G*,Γ $\Rightarrow_n$ |
— nnf rules:
NotOrNNF: ¬*F*,¬*G*,Γ $\Rightarrow_n$ ⟹ ¬(*F*∨*G*),Γ $\Rightarrow_n$ |
NotAndNNF: ¬*F*,Γ $\Rightarrow_n$ ⟹ ¬*G*,Γ $\Rightarrow_n$ ⟹ ¬(*F*∧*G*),Γ $\Rightarrow_n$ |
ImpNNF: ¬*F*,Γ $\Rightarrow_n$ ⟹ *G*,Γ $\Rightarrow_n$ ⟹ *F*→*G*,Γ $\Rightarrow_n$ |
NotImpNNF: *F*,¬*G*,Γ $\Rightarrow_n$ ⟹ ¬(*F*→*G*),Γ $\Rightarrow_n$ |
NotNotNNF: *F*,Γ $\Rightarrow_n$ ⟹ ¬(¬*F*),Γ $\Rightarrow_n$
lemmas *LSC.intros*[*intro*!]

You can prove that derivability in *SCp* is invariant to *nnf*, and then transform *SCp* to *LSC* while assuming NNF. However, the transformation introduces the trouble of handling the right side of *SCp*. The idea behind this is that handling the transformation is easier when not requiring NNF.

One downside of the whole approach is that we often need everything to be in NNF. To shorten:

abbreviation *is-nnf-mset* $\Gamma \equiv \forall\, x \in$ *set-mset* Γ. *is-nnf x*

lemma $\Gamma \Rightarrow \{\#\} \Longrightarrow$ *is-nnf-mset* $\Gamma \Longrightarrow \Gamma \Rightarrow_n$
proof(*induction* Γ $\{\#\}$::*'a formula multiset rule*: *SCp.induct*)
  case (*BotL* Γ)
  then obtain Γ' **where** $\Gamma = \bot,\Gamma'$ **by**(*meson multi-member-split*)
  thus *?case* **by** *auto*
next
  case (*Ax A* Γ) **hence** *False* **by** *simp* **thus** *?case* **..**

next
  case (*AndL* Γ *F G*)
  hence *IH*: Γ, *F*, *G* $\Rightarrow_n$ **by** *force*
  thus *?case* **by** *auto*
next
  case (*NotL*) **thus** *?case*

oops

lemma *LSC-to-SC*:
  shows $\Gamma \Rightarrow_n \Longrightarrow \Gamma \Rightarrow \{\#\}$
proof(*induction rule*: *LSC.induct*)
qed (*auto dest*!: *NotL-inv intro*!: *SCp.intros*(*3*−) *intro*: *extended-Ax*)

lemma *SC-to-LSC*:
  assumes $\Gamma \Rightarrow \Delta$
  shows Γ + (*image-mset Not* Δ) $\Rightarrow_n$
proof −
  have *GO*[*simp*]:
    NO-MATCH {# *B* #} *F* $\Longrightarrow$ (*F*,*S*) + *T* = *F*, (*S*+*T*)
    NO-MATCH {# *B* #} *S* $\Longrightarrow$ *S* + (*F*,*T*) = *F*, (*S*+*T*)
    NO-MATCH ($\neg$*H*) *F* $\Longrightarrow$ *F*,$\neg$*G*,*S* = $\neg$*G*,*F*,*S*
    for *B S F G H T* **by** *simp-all*
  from *assms* **show** *?thesis*
  proof(*induction rule*: *SCp.induct*)
    case (*BotL* Γ)
    then obtain Γ' **where** $\Gamma = \bot,\Gamma'$ **by**(*meson multi-member-split*)
    thus *?case* **by** *auto*
  next
    case (*Ax k* Γ Δ)
     then obtain Γ' Δ' **where** Γ = *Atom k*, Γ' Δ = *Atom k*, Δ' **by**(*meson multi-member-split*)
    thus *?case* **using** *LSC.Ax* **by** *simp*
  qed *auto*
qed

corollary *SC-LSC*: $\Gamma \Rightarrow \{\#\} \longleftrightarrow \Gamma \Rightarrow_n$ **using** *SC-to-LSC LSC-to-SC* **by** *fastforce*

The nice thing: The NNF-Transformation is even easier to show on the onesided variant.

lemma *LSC-NNF*: $\Gamma \Rightarrow_n \Longrightarrow$ *image-mset nnf* $\Gamma \Rightarrow_n$
proof(*induction rule*: *LSC.induct*)

case (*NotOrNNF F G* Γ)
from *NotOrNNF.IH* **have** *nnf* ($\neg$ *F*), *nnf* ($\neg$ *G*), *image-mset nnf* $\Gamma \Rightarrow_n$ **by** *simp*
with *LSC.AndL* **have** *nnf* ($\neg$ *F*) $\wedge$ *nnf* ($\neg$ *G*), *image-mset nnf* $\Gamma \Rightarrow_n$ **.**
thus *?case* **by** *simp*
next
case (*AndL F G* Γ)
from *AndL.IH* **have** *nnf F*, *nnf G*, *image-mset nnf* $\Gamma \Rightarrow_n$ **by** *simp*
with *LSC.AndL*[**where** *'a='a*] **have** *nnf F* $\wedge$ *nnf G*, *image-mset nnf* $\Gamma \Rightarrow_n$ **by** *simp*
thus *?case* **by** *simp*
next
qed (*auto*, *metis add-mset-commute*)

lemma *LSC-NNF-back*: *image-mset nnf* $\Gamma \Rightarrow_n \Longrightarrow \Gamma \Rightarrow_n$
proof(*induction image-mset nnf* Γ *rule*: *LSC.induct*)
oops

If we got rid of the rules for NNF, we could call it Gentzen-Schütte-calculus. But it turned out that not doing that works quite fine.

If you stare at left-handed Sequent calcului for too long, and they start staring back: Try imagining that there is a $\bot$ on the right hand side. Also, bear in mind that provability of $\Gamma \Rightarrow_n$ and satisfiability of Γ are opposites here.

lemma *LHCut*:
assumes *F*,$\Gamma \Rightarrow_n$ $\neg F$, $\Gamma \Rightarrow_n$
shows $\Gamma \Rightarrow_n$
using *assms*
unfolding *SC-LSC*[*symmetric*]
using *NotL-inv cut* **by** *blast*

lemma
shows *LSC-AndL-inv*: $F \wedge G$,$\Gamma \Rightarrow_n \Longrightarrow F$,$G$,$\Gamma \Rightarrow_n$
and *LSC-OrL-inv*: $F \vee G$,$\Gamma \Rightarrow_n \Longrightarrow F$,$\Gamma \Rightarrow_n \wedge G$,$\Gamma \Rightarrow_n$
using *SC-LSC AndL-inv OrL-inv* **by** *blast+*
lemmas *LSC-invs* = *LSC-AndL-inv LSC-OrL-inv*

lemma *LSC-weaken-set*: $\Gamma \Rightarrow_n \Longrightarrow \Gamma + \Theta \Rightarrow_n$
by(*induction rule*: *LSC.induct*) (*auto simp*: *add.assoc*)
lemma *LSC-weaken*: $\Gamma \Rightarrow_n \Longrightarrow F$,$\Gamma \Rightarrow_n$
using *LSC-weaken-set* **by** (*metis add-mset-add-single*)

lemma *LSC-Contract*:
assumes *sfp*: $F,\ F,\ \Gamma \Rightarrow_n$
shows $F,\ \Gamma \Rightarrow_n$
using *SC-LSC contractL sfp* **by** *blast*

lemma *cnf*:
shows
$F \vee (G \wedge H),\ \Gamma \Rightarrow_n \longleftrightarrow (F \vee G) \wedge (F \vee H),\ \Gamma \Rightarrow_n$ (**is** *?l1* $\longleftrightarrow$ *?r1*)
$(G \wedge H) \vee F,\ \Gamma \Rightarrow_n \longleftrightarrow (G \vee F) \wedge (H \vee F),\ \Gamma \Rightarrow_n$ (**is** *?l2* $\longleftrightarrow$ *?r2*)
proof −
have *GO*[*simp*]:
$F,G,S \Rightarrow_n \Longrightarrow G,F,S \Rightarrow_n$
for $F\ G$ :: *'a formula* **and** S **by** (*simp add*: *add-mset-commute*)
have *?r1* **if** *?l1* **proof** −
from ‹*?l1*›[*THEN LSC-invs*(*2*)] **have** *f*: $F,\ \Gamma \Rightarrow_n\ G \wedge H,\ \Gamma \Rightarrow_n$ **by** *simp-all*
from *this*(*2*)[*THEN LSC-invs*(*1*)] **have** *gh*: $G,\ H,\ \Gamma \Rightarrow_n$ **by** *simp*
show *?r1* **using** *f gh* **by**(*auto dest*!: *LSC-invs simp*: *LSC-weaken*)
qed
moreover have *?r2* **if** *?l2* **proof** −
from ‹*?l2*› **have** *f*: $F,\ \Gamma \Rightarrow_n\ G,\ H,\ \Gamma \Rightarrow_n$ **by**(*auto dest*!: *LSC-invs*)
thus *?r2* **by**(*auto dest*!: *LSC-invs simp*: *LSC-weaken*)
qed
moreover have *?l1* **if** *?r1* **proof** −
from ‹*?r1*›[*THEN LSC-invs*(*1*)] **have** ∗: $F \vee G,\ F \vee H,\ \Gamma \Rightarrow_n$ **by** *simp*
hence $F,\ \Gamma \Rightarrow_n\ G,\ H,\ \Gamma \Rightarrow_n$ **by** (*auto elim*!: *LSC-Contract dest*!: *LSC-invs*)
thus *?l1* **by**(*intro LSC.intros*)
qed
moreover have *?l2* **if** *?r2* **proof** −
from ‹*?r2*›[*THEN LSC-invs*(*1*)] **have** ∗: $G \vee F,\ H \vee F,\ \Gamma \Rightarrow_n$ **by** *simp*
hence $F,\ \Gamma \Rightarrow_n\ G,\ H,\ \Gamma \Rightarrow_n$ **by**(*auto elim*!: *LSC-Contract dest*!: *LSC-invs*)
thus *?l2* **by**(*intro LSC.intros*)
qed
ultimately show *?l1* $\longleftrightarrow$ *?r1 ?l2* $\longleftrightarrow$ *?r2* **by** *argo+*
qed

Interestingly, the DNF congruences are a lot easier to show, requiring neither weakening nor contraction. The reasons are to be sought in the asymmetries between the rules for (∧) and (∨).

lemma *dnf*:
shows
$F \wedge (G \vee H),\ \Gamma \Rightarrow_n \longleftrightarrow (F \wedge G) \vee (F \wedge H),\ \Gamma \Rightarrow_n$ (**is** *?t1*)
$(G \vee H) \wedge F,\ \Gamma \Rightarrow_n \longleftrightarrow (G \wedge F) \vee (H \wedge F),\ \Gamma \Rightarrow_n$ (**is** *?t2*)
proof −
have *GO*[*simp*]:
$F,G,S \Rightarrow_n \Longrightarrow G,F,S \Rightarrow_n$
for $F\ G\ H\ I\ J\ S$ **by**(*simp-all add*: *add-mset-commute*)
show *?t1 ?t2* **by**(*auto dest*!: *LSC-invs*)
qed

lemma *LSC-sim-Resolution1*:
  assumes *R*: $S \vee T, \Gamma \Rightarrow_n$
  shows $Atom\ k \vee S, (\neg(Atom\ k)) \vee T, \Gamma \Rightarrow_n$
proof –
  from *R* **have** *r*: $T, \Gamma \Rightarrow_n S, \Gamma \Rightarrow_n$ **by**(*auto dest*: *LSC-invs*)
  show *?thesis* **proof**(*rule LHCut*[**where** *F=Atom k*])
    have *2*: $T, Atom\ k, Atom\ k \vee S, \Gamma \Rightarrow_n$ **using** *LSC-weaken r*(*1*) **by** *auto*
    hence $\neg (Atom\ k) \vee T, Atom\ k, Atom\ k \vee S, \Gamma \Rightarrow_n$ **by** *auto*
    thus $Atom\ k, Atom\ k \vee S, \neg (Atom\ k) \vee T, \Gamma \Rightarrow_n$
      by(*auto dest*!: *LSC-invs*) (*metis add-mset-commute*)
  next

analogously

    show $\neg (Atom\ k), Atom\ k \vee S, \neg (Atom\ k) \vee T, \Gamma \Rightarrow_n$ **by** *simp*
  qed
qed

lemma
  LSC-need-it-once-have-many:
  assumes *el*: $A \in set\ F$
  assumes *once*: *form-of-lit* $A \vee$ *disj-of-clause* (*removeAll* A F),$\Gamma \Rightarrow_n$
  shows *disj-of-clause* F,$\Gamma \Rightarrow_n$
using *assms*
proof(*induction F*)
  case *Nil* **hence** *False* **by** *simp* **thus** *?case* **..**
next
  case (*Cons f F*)
  thus *?case* **proof**(*cases* $A = f$)
    case *True*
    with *Cons.prems* **have** *ihm*: *form-of-lit* $A \vee$ *disj-of-clause* (*removeAll A F*), Γ $\Rightarrow_n$ **by** *simp*
    with *True* **have** *split*: *form-of-lit f*, $\Gamma \Rightarrow_n$ *disj-of-clause* (*removeAll A F*), Γ $\Rightarrow_n$
      by (*auto dest*!: *LSC-invs*(*2*))
    from *Cons.IH*[*OF - ihm*] **have** $A \in set\ F \Longrightarrow$ *disj-of-clause* F, $\Gamma \Rightarrow_n$ **.**
    with *split*(*2*) **have** *disj-of-clause* F, $\Gamma \Rightarrow_n$ **by**(*cases* $A \in set\ F$) *simp-all*
    with *split*(*1*) **show** *?thesis* **by** *auto*
  next
    case *False*
    with *Cons.prems*(*2*) **have** *prem*: *form-of-lit A*, $\Gamma \Rightarrow_n$ *form-of-lit f*, $\Gamma \Rightarrow_n$ *disj-of-clause* (*removeAll A F*), $\Gamma \Rightarrow_n$
      by (*auto dest*!: *LSC-invs*(*2*))
    hence *d*: *form-of-lit* $A \vee$ *disj-of-clause* (*removeAll A F*), $\Gamma \Rightarrow_n$ **by** *blast*
    from *False Cons.prems* **have** *el*: $A \in set\ F$ **by** *simp*
    from *Cons.IH*[*OF el d*] **have** *disj-of-clause* F, $\Gamma \Rightarrow_n$ **.**
    with *prem*(*2*) **show** *?thesis* **by** *auto*
  qed
qed

lemma *LSC-Sim-resolution-la*:
  fixes $k :: 'a$
  assumes *R*: *disj-of-clause* (*removeAll* (k^+) *F* @ *removeAll* (k^{-1}) *G*), $\Gamma \Rightarrow_n$
  assumes *el*: $k^+ \in$ *set F* $k^{-1} \in$ *set G*
  shows *disj-of-clause F*, *disj-of-clause G*, $\Gamma \Rightarrow_n$
proof −
  have *LSC-or-assoc*: $(F \vee G) \vee H, \Gamma \Rightarrow_n \longleftrightarrow F \vee (G \vee H), \Gamma \Rightarrow_n$ **if** *is-nnf F is-nnf G is-nnf H* **for** *F G H*
    using *that* **by**(*auto dest*!: *LSC-invs*(*2*))
  have *dd*: *disj-of-clause* (*F* @ *G*), $\Gamma \Rightarrow_n \Longrightarrow$ *disj-of-clause F* $\vee$ *disj-of-clause G*, $\Gamma \Rightarrow_n$ **for** *F G*
    by(*induction F*) (*auto dest*!: *LSC-invs*(*2*) *simp add*: *LSC-or-assoc*)
  from *LSC-sim-Resolution1*[*OF dd*[*OF R*]]
  have *unord*: *Atom k* $\vee$ *disj-of-clause* (*removeAll* (k^+) *F*), ¬ (*Atom k*) $\vee$ *disj-of-clause* (*removeAll* (k^{-1}) *G*), $\Gamma \Rightarrow_n$.
  show *?thesis*
    using *LSC-need-it-once-have-many*[*OF el*(*1*)] *LSC-need-it-once-have-many*[*OF el*(*2*)] *unord*
    by(*simp add*: *add-mset-commute del*: *sc-insertion-ordering*)
qed

lemma *two-list-induct*[*case-names Nil Cons*]: *P* [] [] $\Longrightarrow$ ($\bigwedge a\ S\ T.\ P\ S\ T \Longrightarrow P$ (*a* # *S*) *T* &&& *P S* (*a* # *T*)) $\Longrightarrow$ *P S T*
  apply(*induction S*)
   apply(*induction T*)
    apply(*simp-all*)
done

lemma *distrib1*: ⟦*F*, $\Gamma \Rightarrow_n$; *image-mset disj-of-clause* (*mset G*) + $\Gamma \Rightarrow_n$⟧
    $\Longrightarrow$ *mset* (*map* ($\lambda d.$ *F* $\vee$ *disj-of-clause d*) *G*) + $\Gamma \Rightarrow_n$

proof(*induction G arbitrary*: Γ)
  have *GO*[*simp*]:
   NO-MATCH ({#*I*∨*J*#}) *H* $\Longrightarrow$ *H*+{#*F*∨*G*#}+*S* = *F*∨*G*,*H*+*S*
   for *F G H S I J* **by**(*simp-all add*: *add-mset-commute*)
  case (*Cons g G*)
   from ‹*F*, $\Gamma \Rightarrow_n$› **have** *1*: *F*, *disj-of-clause g*, $\Gamma \Rightarrow_n$ **by** (*metis LSC-weaken add-mset-commute*)
  from ‹*image-mset disj-of-clause* (*mset* (*g* # *G*)) + $\Gamma \Rightarrow_n$›
  have *2*: *image-mset disj-of-clause* (*mset G*) + (*disj-of-clause g*, Γ) $\Rightarrow_n$ **by**(*simp add*: *add-mset-commute*)
 from *Cons.IH*[*OF 1 2*] **have** *IH*: *disj-of-clause g*, *mset* (*map* ($\lambda d.$ *F* $\vee$ *disj-of-clause d*) *G*) + $\Gamma \Rightarrow_n$
    by(*simp add*: *add-mset-commute*)
  from ‹*F*, $\Gamma \Rightarrow_n$› **have** *3*: *F*, *mset* (*map* ($\lambda d.$ *F* $\vee$ *disj-of-clause d*) *G*) + $\Gamma \Rightarrow_n$
    using *LSC-weaken-set* **by** (*metis add.assoc add.commute add-mset-add-single*)
  from *IH 3* **show** *?case* **by** *auto*
qed *simp*

lemma *mset-concat-map-cons*:
mset (*concat* (*map* (λc. *F c* # *G c*) *S*)) = *mset* (*map F S*) + *mset* (*concat* (*map* *G S*))
by(*induction S*; *simp add*: *add-mset-commute*)

lemma *distrib*:
image-mset disj-of-clause (*mset F*) + $\Gamma \Rightarrow_n \Longrightarrow$
image-mset disj-of-clause (*mset G*) + $\Gamma \Rightarrow_n \Longrightarrow$
mset [*disj-of-clause c* $\vee$ *disj-of-clause d*. *c* $\leftarrow$ *F*, *d* $\leftarrow$ *G*] + $\Gamma \Rightarrow_n$
proof(*induction F G arbitrary*: Γ *rule*: *two-list-induct*)
case (*Cons a F G*)
case *1*
from ‹*image-mset disj-of-clause* (*mset* (*a* # *F*)) + $\Gamma \Rightarrow_n$›
have *a*: *disj-of-clause a*, *image-mset disj-of-clause* (*mset F*) + $\Gamma \Rightarrow_n$ **by**(*simp add*: *add-mset-commute*)
from ‹*image-mset disj-of-clause* (*mset G*) + $\Gamma \Rightarrow_n$›
have *b*: *image-mset disj-of-clause* (*mset G*) + (*image-mset disj-of-clause* (*mset F*) + Γ) $\Rightarrow_n$
and *c*: *image-mset disj-of-clause* (*mset G*) + (*mset* (*map* (λd. *disj-of-clause a* $\vee$ *disj-of-clause d*) *G*) + Γ) $\Rightarrow_n$
using *LSC-weaken-set* **by** (*metis add.commute union-assoc*)+
from *distrib1*[*OF a b*]
have *image-mset disj-of-clause* (*mset F*) + (*mset* (*map* (λd. *disj-of-clause a* $\vee$ *disj-of-clause d*) *G*) + Γ) $\Rightarrow_n$
by (*simp add*: *union-lcomm*)
from *Cons*[*OF this c*]
have *mset* (*concat* (*map* (λc. *map* (λd. *disj-of-clause c* $\vee$ *disj-of-clause d*) *G*) *F*)) +
(*mset* (*map* (λd. *disj-of-clause a* $\vee$ *disj-of-clause d*) *G*) + Γ) $\Rightarrow_n$ **.**
thus *?case* **by**(*simp add*: *add.commute union-assoc*)
next
case (*Cons a F G*) **case** *2*

Just the whole thing again, with slightly more mset magic and swapping things around.

from ‹*image-mset disj-of-clause* (*mset* (*a* # *G*)) + $\Gamma \Rightarrow_n$›
have *a*: *disj-of-clause a*, *image-mset disj-of-clause* (*mset G*) + $\Gamma \Rightarrow_n$ **by**(*simp add*: *add-mset-commute*)
from ‹*image-mset disj-of-clause* (*mset F*) + $\Gamma \Rightarrow_n$›
have *b*: *image-mset disj-of-clause* (*mset F*) + (*image-mset disj-of-clause* (*mset G*) + Γ) $\Rightarrow_n$
and *c*: *image-mset disj-of-clause* (*mset F*) + (*mset* (*map* (λd. *disj-of-clause a* $\vee$ *disj-of-clause d*) *F*) + Γ) $\Rightarrow_n$
using *LSC-weaken-set* **by** (*metis add.commute union-assoc*)+
have *list-commute*: (*mset* (*map* (λd. *disj-of-clause a* $\vee$ *disj-of-clause d*) *F*) + Γ) $\Rightarrow_n$ =
(*mset* (*map* (λd. *disj-of-clause d* $\vee$ *disj-of-clause a*) *F*) + Γ) $\Rightarrow_n$ **for** Γ
proof(*induction F arbitrary*: Γ)

case (*Cons f F*)
have *mset* (*map* (λ*d*. *disj-of-clause a* $\vee$ *disj-of-clause d*) (*f* # *F*)) + $\Gamma \Rightarrow_n$ = *disj-of-clause a* $\vee$ *disj-of-clause f*, *mset* (*map* (λ*d*. *disj-of-clause a* $\vee$ *disj-of-clause d*) *F*) + $\Gamma \Rightarrow_n$ **by**(*simp add*: *add-mset-commute*)
also have ... = *disj-of-clause f* $\vee$ *disj-of-clause a*, *mset* (*map* (λ*d*. *disj-of-clause a* $\vee$ *disj-of-clause d*) *F*) + $\Gamma \Rightarrow_n$
by(*auto dest*!: *LSC-invs*)
also have ... = *mset* (*map* (λ*d*. *disj-of-clause a* $\vee$ *disj-of-clause d*) *F*) + (*disj-of-clause f* $\vee$ *disj-of-clause a*, Γ) $\Rightarrow_n$ **by** (*simp add*: *add-mset-commute*)
also have ... = *mset* (*map* (λ*d*. *disj-of-clause d* $\vee$ *disj-of-clause a*) *F*) + (*disj-of-clause f* $\vee$ *disj-of-clause a*, Γ) $\Rightarrow_n$ **using** *Cons.IH*
by (*metis disj-of-clause-is-nnf insert-iff is-nnf.simps*(*3*) *set-mset-add-mset-insert*)

finally show *?case* **by** *simp*
qed *simp*
from *distrib1*[*OF a b*]
have *image-mset disj-of-clause* (*mset G*) + (*mset* (*map* (λ*d*. *disj-of-clause a* $\vee$ *disj-of-clause d*) *F*) + Γ) $\Rightarrow_n$
by(*auto simp add*: *add.left-commute*)
from *Cons*[*OF c this*]
have *mset* (*concat* (*map* (λ*c*. *map* (λ*d*. *disj-of-clause c* $\vee$ *disj-of-clause d*) *G*) *F*)) +
(*mset* (*map* (λ*d*. *disj-of-clause a* $\vee$ *disj-of-clause d*) *F*) + Γ) $\Rightarrow_n$ **.**
thus *?case* **using** *list-commute* **by** (*simp add*: *mset-concat-map-cons add.assoc add.left-commute*)
qed *simp*

lemma *LSC-BigAndL*: *mset F* + $\Gamma \Rightarrow_n \Longrightarrow \bigwedge F, \Gamma \Rightarrow_n$
by(*induction F arbitrary*: Γ; *simp add*: *LSC-weaken*) (*metis LSC.AndL add-mset-commute union-mset-add-mset-right*)
lemma *LSC-Top-unused*: $[\![\Gamma \Rightarrow_n$; *is-nnf-mset* $\Gamma]\!] \Longrightarrow \Gamma - \{\#\neg \bot\#\} \Rightarrow_n$
proof(*induction rule*: *LSC.induct*)
case *Ax* **thus** *?case* **by** (*metis LSC.Ax add.commute diff-union-swap formula.distinct*(*1*,*3*) *formula.inject*(*2*))
next
case *BotL* **thus** *?case* **by** (*metis LSC.BotL add.commute diff-union-swap formula.distinct*(*11*))
next
case (*AndL F G* Γ)
hence (*F*, *G*, Γ) − $\{\#\neg \bot\#\} \Rightarrow_n$ **by** *simp-all*
hence *F* $\wedge$ *G*, $\Gamma - \{\#\neg \bot\#\} \Rightarrow_n$
by (*metis AndL.hyps LSC.AndL diff-single-trivial diff-union-swap2*)
thus *?case* **by** (*metis add.commute diff-union-swap formula.distinct*(*19*))
next
case (*OrL F* Γ *G*)
hence (*F*, Γ) − $\{\#\neg \bot\#\} \Rightarrow_n$ (*G*, Γ) − $\{\#\neg \bot\#\} \Rightarrow_n$ **by** *simp-all*
hence *F* $\vee$ *G*, $\Gamma - \{\#\neg \bot\#\} \Rightarrow_n$ **by** (*metis LSC.OrL OrL.hyps*(*1*) *OrL.hyps*(*2*) *diff-single-trivial diff-union-swap2*)
thus *?case* **by** (*metis diff-union-swap formula.distinct*(*21*))

qed *auto*

lemma *LSC-BigAndL-inv*: $\bigwedge F$, $\Gamma \Rightarrow_n \Longrightarrow \forall f \in set\ F.\ is\text{-}nnf\ f \Longrightarrow is\text{-}nnf\text{-}mset\ \Gamma$ $\Longrightarrow mset\ F + \Gamma \Rightarrow_n$
proof(*induction F arbitrary*: Γ)
  case *Nil*
  then show *?case* **using** *LSC-Top-unused* **by** *fastforce*
next
  case (*Cons a F*)
  hence $\bigwedge F$, *a*, $\Gamma \Rightarrow_n$ **by**(*auto dest*: *LSC-invs simp*: *add-mset-commute*)
  with *Cons* **have** *mset F* + (*a*, Γ) $\Rightarrow_n$ **by** *fastforce*
  then show *?case* **by** *simp*
qed

lemma *LSC-reassociate-Ands*: {#*disj-of-clause c* ∨ *disj-of-clause d*. (*c*, *d*) ∈# *C*#} + Γ $\Rightarrow_n \Longrightarrow$
   is-nnf-mset Γ ⟹
  {#*disj-of-clause* (*c* @ *d*). (*c*, *d*) ∈# *C*#} + Γ $\Rightarrow_n$
proof(*induction C arbitrary*: Γ)
  case (*add x C*)
  obtain *a b* **where** [*simp*]: *x* = (*a*, *b*) **by**(*cases x*)
  from *add.prems* **have** *a*: (*disj-of-clause a* ∨ *disj-of-clause b*), {#*disj-of-clause c* ∨ *disj-of-clause d*. (*c*, *d*) ∈# *C*#} + Γ $\Rightarrow_n$ **by**(*simp add*: *add-mset-commute*)
   hence (*disj-of-clause* (*a*@*b*)), {#*disj-of-clause c* ∨ *disj-of-clause d*. (*c*, *d*) ∈# *C*#} + Γ $\Rightarrow_n$ **proof** −
    have *pn*: *is-nnf-mset* ({#*disj-of-clause c* ∨ *disj-of-clause d*. (*c*, *d*) ∈# *C*#} + Γ)
      using ‹*is-nnf-mset* Γ› **by** *auto*
   have *disj-of-clause a* ∨ *disj-of-clause b*, Γ $\Rightarrow_n \Longrightarrow$ *is-nnf-mset* Γ ⟹ *disj-of-clause* (*a* @ *b*), Γ $\Rightarrow_n$ **for** Γ
      by(*induction a*) (*auto dest*!: *LSC-invs*)
    from *this*[*OF* - *pn*] *a* **show** *?thesis* **.**
  qed
  hence {#*disj-of-clause c* ∨ *disj-of-clause d*. (*c*, *d*) ∈# *C*#} + ((*disj-of-clause* (*a*@*b*)),Γ) $\Rightarrow_n$ **by**(*simp add*: *add-mset-commute*)
  with *add.IH* **have** {#*disj-of-clause* (*c* @ *d*). (*c*, *d*) ∈# *C*#} + (*disj-of-clause* (*a* @ *b*), Γ) $\Rightarrow_n$
    using ‹*is-nnf-mset* Γ› **by** *fastforce*
  thus *?case* **by**(*simp add*: *add-mset-commute*)
qed *simp*

lemma *LSC-cnf*: $\Gamma \Rightarrow_n \Longrightarrow$ *is-nnf-mset* Γ ⟹ *image-mset cnf-form-of* Γ $\Rightarrow_n$
proof(*induction* Γ *rule*: *LSC.induct*)
  have [*simp*]: *NO-MATCH* (*And I J*) *F* ⟹ *NO-MATCH* (¬⊥) *F* ⟹ *F*,¬⊥,Γ = ¬⊥,*F*,Γ **for** *F I J* Γ **by** *simp*
  have [*intro*!]: $\Gamma \Rightarrow_n \Longrightarrow \neg\bot, \Gamma \Rightarrow_n$ **for** Γ **by** (*simp add*: *LSC-weaken*)
  case *Ax* **thus** *?case* **by**(*auto simp*: *cnf-form-of-defs*)
next
  case *BotL* **show** *?case* **by**(*auto simp*: *cnf-form-of-defs*)

next
  have *GO*[*simp*]:
    NO-MATCH ({#⋀*I*#}) *F* ⟹ *F* + (⋀*G*, *S*) = ⋀*G*, (*F* + *S*)
    for *F G H S I J a b* **by**(*simp-all add*: *add-mset-commute*)
  case (*AndL F G* Γ) **thus** *?case*
  by(*auto dest*!: *LSC-BigAndL-inv intro*!: *LSC-BigAndL simp add*: *cnf-form-of-defs*) (*simp add*: *add-ac*)
next
  case (*OrL F* Γ *G*)
  have *2*: *image-mset disj-of-clause* (*mset* (*concat* (*map* (λ*f*. *map* ((@) *f*) (*cnf-lists G*)) (*cnf-lists F*)))) + Γ ⇒ₙ
    if *pig*: *is-nnf-mset* Γ **and** *a*:
      mset (*concat* (*map* (λ*c*. *map* (λ*d*. *disj-of-clause c* ∨ *disj-of-clause d*) (*cnf-lists G*)) (*cnf-lists F*))) + Γ ⇒ₙ
     for Γ
  proof −
    note *cms*[*simp*] = *mset-map*[*symmetric*] *map-concat comp-def*
    from *a* **have** *image-mset* (λ(*c*,*d*). *disj-of-clause c* ∨ *disj-of-clause d*) (
      mset (*concat* (*map* (λ*c*. *map* (λ*d*. (*c*,*d*)) (*cnf-lists G*)) (*cnf-lists F*)))) + Γ ⇒ₙ **by** *simp*
    hence *image-mset* (λ(*c*,*d*). *disj-of-clause* (*c*@*d*)) (
      mset (*concat* (*map* (λ*c*. *map* (λ*d*. (*c*,*d*)) (*cnf-lists G*)) (*cnf-lists F*)))) + Γ ⇒ₙ

      using *LSC-reassociate-Ands pig* **by** *blast*
    thus *?thesis* **by** *simp*
  qed
  have *1*: ⟦⋀ (*map disj-of-clause* (*cnf-lists F*)), Γ ⇒ₙ; ⋀ (*map disj-of-clause* (*cnf-lists G*)), Γ ⇒ₙ⟧
    ⟹ *is-nnf-mset* Γ
   ⟹ ⋀ (*map disj-of-clause* (*concat* (*map* (λ*f*. *map* ((@) *f*) (*cnf-lists G*)) (*cnf-lists F*)))), Γ ⇒ₙ

      for Γ **using** *distrib*[**where** ′*a*=′*a*] *2* **by**(*auto dest*!: *LSC-BigAndL-inv intro*!: *LSC-BigAndL*)
  from *OrL* **show** *?case*
    by(*auto elim*!: *1 simp add*: *cnf-form-of-def form-of-cnf-def*)
qed *auto*

end

3.6 Converting between Resolution and SC proofs

theory *LSC-Resolution*
imports *LSC Resolution*
begin

lemma *literal-subset-sandwich*:
  assumes *is-lit-plus F cnf F* = {*C*} *R* ⊆ *C*
  shows *R* = □ ∨ *R* = *C*

using *assms* **by**(*cases F rule*: *is-lit-plus.cases*; *simp*) *blast*+

Proof following Gallier [3].

theorem *CSC-Resolution-pre*: $\Gamma \Rightarrow_n \Longrightarrow \forall \gamma \in$ *set-mset* Γ. *is-cnf* $\gamma \Longrightarrow (\bigcup (cnf$ ‘ *set-mset* $\Gamma)) \vdash \Box$
proof(*induction rule*: *LSC.induct*)
  case (*Ax k* Γ)
  let *?s* = $\bigcup$(*cnf* ‘ *set-mset* (¬ (*Atom k*), *Atom k*, Γ))
  have *?s* $\vdash \{k^+\}$ *?s* $\vdash \{k^{-1}\}$ **using** *Resolution.Ass*[**where** ′*a*=′*a*] **by** *simp-all*
  from *Resolution.R*[*OF this*, *of k*]
  have *?s* ⊢ □ **by** *simp*
  thus *?case* **by** *simp*
next
  case (*BotL* Γ) **thus** *?case* **by**(*simp add*: *Ass*)
next
  case (*AndL F G* Γ)
  hence $\bigcup$(*cnf* ‘ *set-mset* (*F*, *G*, Γ)) ⊢ □ **by** *simp*
  thus *?case* **by**(*simp add*: *Un-left-commute sup.assoc*)
next

  case (*OrL F* Γ *G*)
  hence *is-cnf* (*F* ∨ *G*) **by** *simp*
  hence *d*: *is-disj* (*F* ∨ *G*) **by** *simp*
  hence *db*: *is-disj F is-lit-plus F is-disj G* **by** (−, *cases F*) *simp-all*
  hence *is-cnf F* ∧ *is-cnf G* **by**(*cases F*; *cases G*; *simp*)
  with *OrL* **have** *IH*: ($\bigcup$(*cnf* ‘ *set-mset* (*F*, Γ))) ⊢ □ ($\bigcup$(*cnf* ‘ *set-mset* (*G*, Γ))) ⊢ □ **by** *simp-all*
  let *?*Γ = ($\bigcup$(*cnf* ‘ *set-mset* Γ))
  from *IH* **have** *IH-readable*: *cnf F* ∪ *?*Γ ⊢ □ *cnf G* ∪ *?*Γ ⊢ □ **by** *auto*
  show *?case* **proof**(*cases cnf F* = {} ∨ *cnf G* = {})
    case *True*
    hence *cnf* (*F* ∨ *G*) = {} **by** *auto*
    thus *?thesis* **using** *True IH* **by** *auto*
  next
    case *False*
    then obtain *S T* **where** *ST*: *cnf F* = {*S*} *cnf G* = {*T*}
      using *cnf-disj-ex db*(*1*,*3*) **by** *metis*

    hence *R*: *cnf* (*F* ∨ *G*) = { *S* ∪ *T* } **by** *simp*
    have ⟦*S*▷*?*Γ ⊢ □; *T*▷*?*Γ ⊢ □⟧ ⟹ *S* ∪ *T*▷ *?*Γ ⊢ □ **proof** −
      assume *s*: *S*▷*?*Γ ⊢ □ **and** *t*: *T*▷*?*Γ ⊢ □
       hence *s-w*: *S* ▷ *S* ∪ *T* ▷ *?*Γ ⊢ □ **using** *Resolution-weaken* **by** (*metis insert-commute insert-is-Un*)
      note *Resolution-taint-assumptions*[*of* {*T*} *?*Γ □ *S*] *t*
      then obtain *R* **where** *R*: *S* ∪ *T* ▷ $\bigcup$(*cnf* ‘ *set-mset* Γ) ⊢ *R R*⊆*S* **by** (*auto simp*: *Un-commute*)
      show *?thesis* **using** *literal-subset-sandwich*[*OF db*(*2*) *ST*(*1*) *R*(*2*)] **proof**
        assume *R* = □ **thus** *?thesis* **using** *R*(*1*) **by** *blast*
      next

from *Resolution-unnecessary*[**where** *T*={-}, *simplified*] *R*(*1*)
have (*R* ▷ *S* ∪ *T* ▷ *?*Γ ⊢ □) = (*S* ∪ *T* ▷ *?*Γ ⊢ □) **.**
moreover assume *R* = *S*
ultimately show *?thesis* **using** *s-w* **by** *simp*
qed
qed
thus *?thesis* **using** *IH ST R* **by** *simp*
qed
hence *case-readable*: *cnf* (*F* ∨ *G*) ∪ *?*Γ ⊢ □ **by** *auto*
qed *auto*

corollary *LSC-Resolution*:
assumes Γ $\Rightarrow_n$
shows (⋃(*cnf* ‘ *nnf* ‘ *set-mset* Γ)) ⊢ □
proof −
from *assms*
have *image-mset nnf* Γ $\Rightarrow_n$ **by** (*simp add*: *LSC-NNF*)
from *LSC-cnf*[*OF this*]
have *image-mset* (*cnf-form-of* ∘ *nnf*) Γ $\Rightarrow_n$ **by**(*simp add*: *image-mset.compositionality is-nnf-nnf*)
moreover have ∀ γ ∈ *set-mset* (*image-mset* (*cnf-form-of* ∘ *nnf*) Γ). *is-cnf* γ
using *cnf-form-of-is*[**where** ′*a*=′*a*, *OF is-nnf-nnf*] **by** *simp*
moreover note *CSC-Resolution-pre*
ultimately have ⋃(*cnf* ‘ *set-mset* (*image-mset* (*cnf-form-of* ∘ *nnf*) Γ)) ⊢ □ **by** *blast*
hence ⋃((λ*F*. *cnf* (*cnf-form-of* (*nnf F*))) ‘ *set-mset* Γ) ⊢ □ **by** *simp*
thus *?thesis* **unfolding** *cnf-cnf*[*OF is-nnf-nnf*] **by** *simp*
qed

corollary *SC-Resolution*:
assumes Γ ⇒ {#}
shows (⋃(*cnf* ‘ *nnf* ‘ *set-mset* Γ)) ⊢ □
proof −
from *assms* **have** *image-mset nnf* Γ $\Rightarrow_n$ **by** (*simp add*: *LSC-NNF SC-LSC*)
hence ⋃(*cnf*‘ *nnf* ‘ *set-mset* (*image-mset nnf* Γ)) ⊢ □ **using** *LSC-Resolution* **by** *blast*
thus *?thesis* **using** *is-nnf-nnf-id*[**where** ′*a*=′*a*] *is-nnf-nnf*[**where** ′*a*=′*a*] **by** *auto*
qed

lemma *Resolution-LSC-pre*:
assumes *S* ⊢ *R*
assumes *finite R*
assumes *finite S Ball S finite*
shows ∃ *S*′ *R*′. ∀Γ. *set R*′ = *R* ∧ *set* (*map set S*′) = *S* ∧
(*disj-of-clause R*′, {#*disj-of-clause c*. *c* ∈# *mset S*′#} + Γ $\Rightarrow_n$ ⟶ {#*disj-of-clause c*. *c* ∈# *mset S*′#} + Γ $\Rightarrow_n$)

using *assms* **proof**(*induction S R rule*: *Resolution.induct*)
  case (*Ass F S*)

  define *Sm* **where** $Sm = S - \{F\}$
  hence *Sm*: $S = F \triangleright Sm$ $F \notin Sm$ **using** *Ass* **by** *fast+*
  with *Ass* **have** *fsm*: *finite Sm Ball Sm finite* **by** *auto*
  then obtain Sm' **where** $Sm = set\ (map\ set\ Sm')$ **by** (*metis* (*full-types*) *ex-map-conv finite-list*)
  moreover obtain R' **where** [*simp*]: $F = set\ R'$ **using** *Ass finite-list* **by** *blast*
  ultimately have *S*: $S = set\ (map\ set\ (R'\#Sm'))$ **unfolding** *Sm* **by** *simp*
  show *?case*
    using *LSC-Contract*[**where** $'a='a$]
    by(*intro exI*[**where** $x=R'\#Sm'$] *exI*[**where** $x=R'$]) (*simp add*: *S add-ac*)

next
  case (*R S F G k*)
  from *R.prems* **have** *fin*: *finite F finite G* **by** *simp-all*
  from *R.IH*(*1*)[*OF fin*(*1*) *R.prems*(*2*,*3*)] **obtain** *FR FS* **where** *IHF*:
    $set\ FR = F$ $set\ (map\ set\ FS) = S$
    $\bigwedge\Gamma\ GS.$ (*disj-of-clause FR*, *image-mset disj-of-clause* (*mset* (*FS@GS*)) + $\Gamma \Rightarrow_n$

      $\Longrightarrow$ *image-mset disj-of-clause* (*mset* (*FS@GS*)) + $\Gamma \Rightarrow_n$)
      by *simp* (*metis add.assoc*)
  from *R.IH*(*2*)[*OF fin*(*2*) *R.prems*(*2*,*3*)] **obtain** *GR GS* **where** *IHG*:
    $set\ GR = G$ $set\ (map\ set\ GS) = S$
    $\bigwedge\Gamma\ HS.$ (*disj-of-clause GR*, *image-mset disj-of-clause* (*mset* (*GS@HS*)) + Γ $\Rightarrow_n$
      $\Longrightarrow$ *image-mset disj-of-clause* (*mset* (*GS@HS*)) + $\Gamma \Rightarrow_n$)
    by *simp* (*metis add.assoc*)
  have *IH*: *image-mset disj-of-clause* (*mset* (*FS* @ *GS*)) + $\Gamma \Rightarrow_n$
    if *disj-of-clause FR*, *disj-of-clause GR*, *image-mset disj-of-clause* (*mset* (*FS* @ *GS*)) + $\Gamma \Rightarrow_n$
    for Γ **using** *IHF*(*3*)[*of GS* Γ] *IHG*(*3*)[*of FS disj-of-clause FR*, Γ] *that*
    by(*simp add*: *add-mset-commute add-ac*)
  show *?case*
     apply(*intro exI*[**where** *x*=*FS* @ *GS*] *exI*[**where** *x*=*removeAll* (k^+) *FR* @ *removeAll* (k^{-1}) *GR*] *allI impI conjI*)
      apply(*simp add*: *IHF IHG*;*fail*)
     apply(*insert IHF IHG*; *simp*;*fail*)
    apply(*intro IH*)
    apply(*auto dest*!: *LSC-Sim-resolution-la simp add*: *IHF IHG R.hyps*)
  done
qed

lemma *Resolution-LSC-pre-nodisj*:
  assumes $S \vdash R$
  assumes *finite R*
  assumes *finite S Ball S finite*
  shows $\exists S'\ R'.\ \forall\Gamma.$ *is-nnf-mset* $\Gamma \longrightarrow$ *is-disj* $R' \wedge$ *is-nnf* $S' \wedge$ *cnf* $R' = \{R\} \wedge$

cnf $S' \subseteq S \wedge$
  (R', S', $\Gamma \Rightarrow_n \longrightarrow S'$, $\Gamma \Rightarrow_n$)
proof −
  have *mehorder*: F, $\bigwedge G$, Γ = $\bigwedge G$, F, Γ **for** F G Γ **by**(*simp add*: *add-ac*)
  from *Resolution-LSC-pre*[**where** *'a='a*,*OF assms*]
  obtain S' R' **where** *o*: $\bigwedge\Gamma$. *is-nnf-mset* $\Gamma \Longrightarrow$ *set* $R' = R \wedge$ *set* (*map set* S') = $S \wedge$
  (*disj-of-clause* R', *image-mset disj-of-clause* (*mset* S') + $\Gamma \Rightarrow_n \longrightarrow$ *image-mset disj-of-clause* (*mset* S') + $\Gamma \Rightarrow_n$)
    by *blast*
  hence *p*: *is-nnf-mset* $\Gamma \Longrightarrow$ (*disj-of-clause* R', *image-mset disj-of-clause* (*mset* S') + $\Gamma \Rightarrow_n \Longrightarrow$ *image-mset disj-of-clause* (*mset* S') + $\Gamma \Rightarrow_n$)
    for Γ **by** *blast*
  show *?thesis*
    apply(*rule exI*[**where** x=$\bigwedge$*map disj-of-clause* S'])
    apply(*rule exI*[**where** x=*disj-of-clause* R'])
    apply *safe*
        apply(*intro disj-of-clause-is*;*fail*)
       apply(*simp add*: *cnf-disj o*; *fail*)+
     subgoal using *o* **by**(*fastforce simp add*: *cnf-BigAnd cnf-disj*)
    subgoal for Γ
      apply(*frule p*)
       apply(*unfold mehorder*)
       apply(*drule LSC-BigAndL-inv*)
         apply(*simp*;*fail*)+
      by (*simp add*: *LSC-BigAndL*)
    done
qed

corollary *Resolution-LSC1*:
  assumes $S \vdash \Box$
  shows $\exists F$. *is-nnf* $F \wedge$ *cnf* $F \subseteq S \wedge \{\#F\#\} \Rightarrow_n$
proof −
  have $*$: $\{f \cup g \mid f\ g.\ f \in F \wedge g \in G\} = \{\Box\} \Longrightarrow F = \{\Box\}$ **for** F G
  proof (*rule ccontr*)
    assume *m*: $\{f \cup g \mid f\ g.\ f \in F \wedge g \in G\} = \{\Box\}$
    assume $F \neq \{\Box\}$
    hence $F = \{\} \vee (\exists E.\ E \in F \wedge E \neq \Box)$ **by** *blast*
    thus *False* **proof**
      assume $F = \{\}$
      with *m* **show** *False* **by** *simp*
    next
      assume $\exists E.\ E \in F \wedge E \neq \Box$
      then obtain E **where** E: $E \in F \wedge E \neq \Box$ **..**
      show *False* **proof** *cases*
        assume $G = \{\}$ **with** *m* **show** *False* **by** *simp*
      next
        assume $G \neq \{\}$
        then obtain D **where** $D \in G$ **by** *blast*

```
      with E have E ∪ D ∈ {f ∪ g |f g. f ∈ F ∧ g ∈ G} by blast
      with m E show False by simp
    qed
  qed
 qed
 have ∗: F = {□} ∧ G = {□} if {f ∪ g |f g. f ∈ F ∧ g ∈ G} = {□} for F G
 proof (intro conjI)
  show G = {□}
   apply(rule ∗[of G F])
   apply(subst that[symmetric])
  by blast
 qed (rule ∗[OF that])
 have ∗: is-nnf F =⇒ is-nnf-mset Γ =⇒ cnf F = {□} =⇒ F,Γ ⇒n for F Γ
  apply(induction F rule: cnf.induct; simp)
   apply blast
  apply (metis LSC.LSC.AndL LSC-weaken add-mset-commute singleton-Un-iff)
  apply(drule ∗; simp add: LSC.LSC.OrL)
 done
 from Resolution-useless-infinite[OF assms]
 obtain S′ where su: S′⊆S and fin: finite S′ Ball S′ finite and pr: (S′ ⊢ □) by
blast
 from Resolution-LSC-pre-nodisj[OF pr finite.emptyI fin]
 obtain S′′ where is-nnf S′′ cnf S′′ ⊆ S′ {# S′′ #} ⇒n
  using ∗ [OF disj-is-nnf, of - ‹{#}›]
  by (metis LSC-weaken add-mset-commute empty-iff set-mset-empty)
 with su show ?thesis by blast
qed

corollary Resolution-SC1:
 assumes S ⊢ □
 shows ∃F. cnf (nnf F) ⊆ S ∧ {#F#} ⇒ {#}
 apply(insert Resolution-LSC1[OF assms])
 apply(elim ex-forward)
 apply(elim conjE; intro conjI)
  subgoal by(subst is-nnf-nnf-id; assumption)
 apply(unfold SC-LSC)
 subgoal by (simp;fail)
done




end
theory ND-FiniteAssms
imports ND
begin

lemma ND-finite-assms: Γ ⊢ F =⇒ ∃Γ′. Γ′ ⊆ Γ ∧ finite Γ′ ∧ (Γ′ ⊢ F)
proof(induction rule: ND.induct)
 case (Ax F Γ) thus ?case by(intro exI[of - {F}]) (simp add: ND.Ax)
```

next
case (*AndI* Γ F G)
from *AndI.IH* **obtain** $\Gamma 1$ $\Gamma 2$
where $\Gamma 1 \subseteq \Gamma \wedge$ *finite* $\Gamma 1 \wedge (\Gamma 1 \vdash F)$
and $\Gamma 2 \subseteq \Gamma \wedge$ *finite* $\Gamma 2 \wedge (\Gamma 2 \vdash G)$
by *blast*
then show *?case* **by**(*intro exI*[**where** $x{=}\Gamma 1 \cup \Gamma 2$]) (*force elim*: *Weaken intro*!: *ND.AndI*)
next
case (*CC* F Γ)
from *CC.IH* **obtain** Γ' **where** Γ': $\Gamma' \subseteq \neg F \triangleright \Gamma \wedge$ *finite* $\Gamma' \wedge (\Gamma' \vdash \bot)$ **..**
thus *?case* **proof**(*cases Not* $F \in \Gamma'$)

case distinction: Did we actually use $\neg F$?

case *False* **hence** $\Gamma' \subseteq \Gamma$ **using** Γ' **by** *blast*
with Γ' **show** *?thesis* **using** *BotE* **by**(*intro exI*[**where** $x{=}\Gamma'$]) *fast*
next
case *True*
then obtain Γ'' **where** $\Gamma' = \neg F \triangleright \Gamma''$ $\neg F \notin \Gamma''$ **by** (*meson Set.set-insert*)
hence $\Gamma'' \subseteq \Gamma$ *finite* Γ'' $\neg F \triangleright \Gamma'' \vdash \bot$ **using** Γ' **by** *auto*
thus *?thesis* **using** *ND.CC* **by** *auto*
qed
next
case *AndE1* **thus** *?case* **by**(*blast dest*: *ND.AndE1*) **next**
case *AndE2* **thus** *?case* **by**(*blast dest*: *ND.AndE2*)
next
case *OrI1* **thus** *?case* **by**(*blast dest*: *ND.OrI1*) **next**
case *OrI2* **thus** *?case* **by**(*blast dest*: *ND.OrI2*)
next
case (*OrE* Γ F G H)
from *OrE.IH* **obtain** $\Gamma 1$ $\Gamma 2$ $\Gamma 3$
where *IH*:
$\Gamma 1 \subseteq \Gamma \wedge$ *finite* $\Gamma 1 \wedge (\Gamma 1 \vdash F \vee G)$
$\Gamma 2 \subseteq F \triangleright \Gamma \wedge$ *finite* $\Gamma 2 \wedge (\Gamma 2 \vdash H)$
$\Gamma 3 \subseteq G \triangleright \Gamma \wedge$ *finite* $\Gamma 3 \wedge (\Gamma 3 \vdash H)$
by *blast*
let $?w = \Gamma 1 \cup (\Gamma 2 - \{F\}) \cup (\Gamma 3 - \{G\})$
from *IH* **have** $?w \vdash F \vee G$ **using** *Weaken*[*OF - sup-ge1*] **by** *metis* **moreover**
from *IH* **have** $F \triangleright ?w \vdash H$ $G \triangleright ?w \vdash H$ **using** *Weaken* **by** (*metis Un-commute Un-insert-right Un-upper1 Weaken insert-Diff-single*)+ **ultimately**
have $?w \vdash H$ **using** *ND.OrE* **by** *blast*
thus *?case* **using** *IH* **by**(*intro exI*[**where** $x{=}?w$]) *auto*

Clever evasion of the case distinction made for CC.

next
case (*ImpI* F Γ G)
from *ImpI.IH* **obtain** Γ' **where** $\Gamma' \subseteq F \triangleright \Gamma \wedge$ *finite* $\Gamma' \wedge (\Gamma' \vdash G)$ **..**
thus *?case* **by** (*intro exI*[**where** $x{=}\Gamma' - \{F\}$]) (*force elim*: *Weaken intro*!: *ND.ImpI*)

next
  case (*ImpE* Γ *F* *G*)
  from *ImpE.IH* **obtain** Γ*1* Γ*2* **where**
    $\Gamma 1 \subseteq \Gamma \wedge$ *finite* $\Gamma 1 \wedge (\Gamma 1 \vdash F \rightarrow G)$
    $\Gamma 2 \subseteq \Gamma \wedge$ *finite* $\Gamma 2 \wedge (\Gamma 2 \vdash F)$
   by *blast*
  then show *?case* **by**(*intro exI*[**where** *x*=Γ*1* ∪ Γ*2*]) (*force elim*: *Weaken intro*: *ND.ImpE*[**where** *F*=*F*])
next
  case (*NotE* Γ *F*)
  from *NotE.IH* **obtain** Γ*1* Γ*2* **where**
    $\Gamma 1 \subseteq \Gamma \wedge$ *finite* $\Gamma 1 \wedge (\Gamma 1 \vdash \neg F)$
    $\Gamma 2 \subseteq \Gamma \wedge$ *finite* $\Gamma 2 \wedge (\Gamma 2 \vdash F)$
   by *blast*
  then show *?case* **by**(*intro exI*[**where** *x*=Γ*1* ∪ Γ*2*]) (*force elim*: *Weaken intro*: *ND.NotE*[**where** *F*=*F*])
next
  case (*NotI* *F* Γ)
  from *NotI.IH* **obtain** Γ′ **where** $\Gamma' \subseteq F \triangleright \Gamma \wedge$ *finite* $\Gamma' \wedge (\Gamma' \vdash \bot)$ **..**
  thus *?case* **by**(*intro exI*[**where** *x*=Γ′ − {*F*}]) (*force elim*: *Weaken intro*: *ND.NotI*[**where** *F*=*F*])
qed

We thought that a lemma like this would be necessary for the ND completeness by SC completeness proof (this lemma shows that if we made an ND proof, we can always limit ourselves to a finite set of assumptions – and thus put all the assumptions into one formula). That is not the case, since in the completeness proof, we assume a valid entailment and have to show (the existence of) a derivation. The author hopes that his misunderstanding can help the reader's understanding.

corollary *ND-no-assms*:
  assumes Γ ⊢ *F*
  obtains Γ′ **where** *set* $\Gamma' \subseteq \Gamma \wedge (\{\} \vdash \bigwedge\Gamma' \rightarrow F)$
proof(*goal-cases*)
  case *1*
  from *ND-finite-assms*[*OF assms*] **obtain** Γ′ **where** Γ′⊆Γ *finite* Γ′ Γ′ ⊢ *F* **by** *blast*
  from ‹*finite* Γ′› **obtain** *G* **where** Γ′[*simp*]: Γ′ = *set G* **using** *finite-list* **by** *blast*
  with ‹Γ′⊆Γ› **have** *set G* ⊆ Γ **by** *clarify*
  moreover from ‹Γ′ ⊢ *F*› **have** $\{\} \vdash \bigwedge G \rightarrow F$ **unfolding** Γ′ *AssmBigAnd* **.**
  ultimately show *?case* **by**(*intro 1*[**where** Γ′=*G*] *conjI*)
qed

end

3.7 An alternate proof of ND completeness

theory *ND-Compl-SC*
imports *SC-Sema ND-Sound SCND Compactness*

begin

lemma *ND-sound-complete-countable*:
  fixes Γ :: *'a* :: *countable formula set*
  shows $\Gamma \vdash F \longleftrightarrow \Gamma \models F$ (**is** *?n* $\longleftrightarrow$ *?s*)
proof
  assume *?n* **thus** *?s* **by** (*fact ND-sound*)
next
  assume *s*: *?s*
  with *compact-entailment* **obtain** Γ' **where** *0*: *finite* Γ' $\Gamma' \models F$ $\Gamma' \subseteq \Gamma$
    unfolding *entailment-def* **by** *metis*
  then obtain Γ'' **where** Γ'': $\Gamma' =$ *set-mset* Γ'' **using** *finite-set-mset-mset-set* **by** *blast*
  have *su*: *set-mset* $\Gamma'' \subseteq \Gamma$ **using** *0* Γ'' **by** *fast*
  from *0* **have** $\models \Gamma'' \Rightarrow \{\#F\#\}$ **unfolding** *sequent-semantics-def entailment-def* Γ'' **by** *simp*
  with *SC-sound-complete* **have** $\Gamma'' \Rightarrow \{\#F\#\}$ **by** *blast*
  with *SCND* **have** *set-mset* $\Gamma'' \cup \neg$ ‘ *set-mset* $\{\#F\#\} \vdash \bot$ **.**
  thus *?n* **using** *ND.CC Weaken*[*OF - su*[*THEN insert-mono*]] **by** *force*
qed

If you do not like the requirement that our atoms are countable, you can also restrict yourself to a finite set of assumptions.

lemma *ND-sound-complete-finite*:
  assumes *finite* Γ
  shows $\Gamma \vdash F \longleftrightarrow \Gamma \models F$ (**is** *?n* $\longleftrightarrow$ *?s*)
proof
  assume *?n* **thus** *?s* **by** (*fact ND-sound*)
next
  assume *s*: *?s*
  then obtain Γ'' **where** Γ'': $\Gamma =$ *set-mset* Γ'' **using** *finite-set-mset-mset-set* *assms* **by** *blast*
  have *su*: *set-mset* $\Gamma'' \subseteq \Gamma$ **using** Γ'' **by** *fast*
  have $\models \Gamma'' \Rightarrow \{\#F\#\}$ **using** *s* **unfolding** *sequent-semantics-def entailment-def* Γ'' **by** *auto*
  with *SC-sound-complete* **have** $\Gamma'' \Rightarrow \{\#F\#\}$ **by** *blast*
  with *SCND* **have** *set-mset* $\Gamma'' \cup \neg$ ‘ *set-mset* $\{\#F\#\} \vdash \bot$ **.**
  thus *?n* **using** *ND.CC Weaken*[*OF - su*[*THEN insert-mono*]] **by** *force*
qed

end
theory *Resolution-Compl-SC-Small*
imports *LSC-Resolution Resolution SC-Sema CNF-Formulas-Sema*
begin

lemma *Resolution-complete'*:
  assumes *fin*: *finite S*
  assumes *val*: $S \models F$

shows $\bigcup((cnf \circ nnf)$ ‘ $(\{\neg F\} \cup S)) \vdash \Box$
proof −
from *fin* **obtain** S' **where** S: $S = set\text{-}mset\ S'$ **using** *finite-set-mset-mset-set* **by** *blast*
have *cnf*: $\forall F \in set\text{-}mset\ (image\text{-}mset\ (cnf\text{-}form\text{-}of \circ nnf)\ (\neg\ F,\ S')).\ is\text{-}cnf\ F$ **by**(*simp add*: *cnf-form-of-is is-nnf-nnf*)
note *entailment-def*[*simp*]
from *val*
have $S \models \neg(\neg F)$ **by** *simp*
hence $S \models \neg(nnf\ (\neg F))$ **by** (*simp add*: *nnf-semantics*)
hence $S \models \neg(cnf\text{-}form\text{-}of\ (nnf\ (\neg F)))$ **by** (*simp add*: *cnf-form-semantics*[*OF is-nnf-nnf*])
hence $set\text{-}mset\ (image\text{-}mset\ nnf\ S') \models \neg(cnf\text{-}form\text{-}of\ (nnf\ (\neg F)))$ **using** S **by** (*simp add*: *nnf-semantics*)
hence $set\text{-}mset\ (image\text{-}mset\ (cnf\text{-}form\text{-}of \circ nnf)\ S') \models \neg(cnf\text{-}form\text{-}of\ (nnf\ (\neg F)))$ **by** (*simp add*: *cnf-form-semantics*[*OF is-nnf-nnf*])
hence $image\text{-}mset\ (cnf\text{-}form\text{-}of \circ nnf)\ S' \Rightarrow \{\#\neg(cnf\text{-}form\text{-}of\ (nnf\ (\neg F)))\#\}$
unfolding *SC-sound-complete sequent-intuitonistic-semantics* **.**
hence $image\text{-}mset\ (cnf\text{-}form\text{-}of \circ nnf)\ (\neg F,\ S') \Rightarrow \{\#\}$ **using** *NotR-inv* **by** *simp*
hence $image\text{-}mset\ (cnf\text{-}form\text{-}of \circ nnf)\ (\neg F,\ S') \Rightarrow_n$ **by** (*simp add*: *SC-LSC is-nnf-nnf nnf-cnf-form*)
with *CSC-Resolution-pre* **have** $\bigcup(cnf$ ‘ $set\text{-}mset\ (image\text{-}mset\ (cnf\text{-}form\text{-}of \circ nnf)\ (\neg\ F,\ S'))) \vdash \Box$ **using** *cnf* **.**
thus *?thesis* **using** *cnf-cnf*[**where** *'a='a*, *OF is-nnf-nnf*]
unfolding *set-image-mset image-comp comp-def S* **by** *simp*
qed

corollary *Resolution-complete-single*:
assumes $\models F$
shows $cnf\ (nnf\ (\neg F)) \vdash \Box$
using *assms Resolution-complete'*[*OF finite.emptyI*, *of F*]
unfolding *entailment-def comp-def* **by** *simp*

end
theory *Resolution-Compl-SC-Full*
imports *LSC-Resolution Resolution SC-Sema Compactness*
begin

theorem *Resolution-complete*:
fixes S :: *'a* :: *countable formula set*
assumes *val*: $S \models F$
shows $\bigcup((cnf \circ nnf)$ ‘ $(\{\neg F\} \cup S)) \vdash \Box$

proof −
let $?mun = \lambda s. \bigcup((cnf \circ nnf)$ ‘ $s)$

from *compact-entailment*[*OF val*] **obtain** S'' **where** *fin*: *finite* S'' **and** *su*: S''

$\subseteq S$ **and** *val'*: $S'' \models F$ **by** *blast*
from *fin* **obtain** S' **where** *S*: $S'' = \textit{set-mset}\ S'$ **using** *finite-set-mset-mset-set* **by** *blast*
have *cnf*: $\forall F \in \textit{set-mset}\ (\textit{image-mset}\ (\textit{cnf-form-of} \circ \textit{nnf})\ (\neg\ F,\ S'))$. *is-cnf F* **by**(*simp add*: *cnf-form-of-is is-nnf-nnf*)
note *entailment-def*[*simp*]
from *val'* **have** $S'' \models \neg(\neg F)$ **by** *simp*
hence $S' \Rightarrow \{\#\neg(\neg F)\#\}$
unfolding *SC-sound-complete sequent-intuitonistic-semantics S* **.**
hence $\neg F,\ S' \Rightarrow \{\#\}$ **by** (*simp add*: *NotR-inv*)
hence *image-mset nnf* $(\neg F,\ S') \Rightarrow \{\#\}$ **using** *LSC-NNF SC-LSC* **by** *blast*
hence *image-mset nnf* $(\neg F,\ S') \Rightarrow_n$ **by** (*simp add*: *SC-LSC is-nnf-nnf*)
with *LSC-Resolution* **have** $\bigcup(\textit{cnf}$ ‘ *nnf* ‘ *set-mset* (*image-mset nnf* $(\neg\ F,\ S')))$ $\vdash \Box$ **.**
hence *?mun* $(\{\neg\ F\} \cup S'') \vdash \Box$
unfolding *set-image-mset image-comp comp-def S is-nnf-nnf-id*[*OF is-nnf-nnf*] **by** *simp*
from *Resolution-weaken*[*OF this*, *of ?mun S*] **show** *?thesis* **using** *su* **by** (*metis UN-Un Un-left-commute sup.order-iff*)
qed

end

3.8 SC and Implication-only formulas

theory *MiniSC*
imports *MiniFormulas SC*
begin

abbreviation *is-mini-mset* $\Gamma \equiv \forall F \in \textit{set-mset}\ \Gamma$. *is-mini-formula F*
lemma *to-mini-mset-is*: *is-mini-mset* (*image-mset to-mini-formula* Γ) **by** *simp*

lemma *SC-full-to-mini*:
defines $\textit{tms} \equiv \textit{image-mset to-mini-formula}$
assumes *asm*: $\Gamma \Rightarrow \Delta$
shows $\textit{tms}\ \Gamma \Rightarrow \textit{tms}\ \Delta$
proof −
have *tmsi*[*simp*]: *tms* $(F,S) =$ *to-mini-formula F*, *tms S* **for** *F S* **unfolding** *tms-def* **by** *simp*
from *asm* **show** *?thesis*
proof(*induction* $\Gamma\ \Delta$ *rule*: *SCp.induct*)
case (*BotL* Γ)
hence $\bot \in\#$ *tms* Γ **unfolding** *tms-def* **by** *force*
thus *?case* **using** *SCp.BotL* **by** *blast*
next
case (*Ax k* $\Gamma\ \Delta$)
hence *Atom k* $\in\#$ *tms* Γ *Atom k* $\in\#$ *tms* Δ **unfolding** *tms-def* **using** *image-iff* **by** *fastforce+*
thus *?case* **using** *SCp.Ax*[*of k*] **by** *blast*

```
  next
    case (NotR F Γ Δ) thus ?case
      unfolding tmsi to-mini-formula.simps
      using weakenR SCp.ImpR by blast
  next
    case (NotL Γ F Δ) from this(2) show ?case
      by(auto intro!: SCp.ImpL)
  next
    case ImpR thus ?case using SCp.ImpR by simp
  next
    case ImpL thus ?case using SCp.ImpL by simp
  next
    case AndR from AndR(3,4) show ?case
      using weakenR by(auto intro!: SCp.ImpR SCp.ImpL)
  next
    case AndL from AndL(2) show ?case
      using weakenR[where 'a='a] by(fastforce intro!: SCp.ImpR SCp.ImpL)
  next
    case OrR from OrR(2) show ?case
      using weakenR by(fastforce intro!: SCp.ImpR SCp.ImpL)
  next
    case (OrL F Γ Δ G)
    note SCp.ImpL
    moreover {
      have to-mini-formula F, tms Γ ⇒ tms Δ using  OrL(3)[unfolded tmsi] .
      with weakenR have to-mini-formula F, tms Γ ⇒ ⊥, tms Δ by blast
      with SCp.ImpR have tms Γ ⇒ to-mini-formula F → ⊥, tms Δ . }
   moreover have to-mini-formula G, tms Γ ⇒ tms Δ using ‹tms (G, Γ) ⇒ tms
Δ› unfolding tmsi .
    ultimately have (to-mini-formula F → ⊥) → to-mini-formula G, tms Γ ⇒
tms Δ .
    thus ?case unfolding tmsi to-mini-formula.simps .
  qed
qed

lemma SC-mini-to-full:
  defines tms ≡ image-mset to-mini-formula
  assumes asm: tms Γ ⇒ tms Δ
  shows Γ ⇒ Δ
proof −
  have tmsi[simp]: tms (F,S) = to-mini-formula F, tms S for F S unfolding
tms-def by simp
  note ImpL-inv ImpR-inv[dest]
  have no: f ≠ (λF G. Not F) f ≠ Or f ≠ And if f F G, S' = tms S for f F G S
S'
   by (metis that is-mini-formula.simps(4−6) msed-map-invR tms-def to-mini-is-mini
union-commute)+
  note dr = no(1)[where f=λF G. Not F, simplified, dest!]
           no(2)[where f=Or, simplified, dest!]
```

no(*3*)[**where** *f*=*And*, *simplified*, *dest*!]
have *whai*:
($\exists$ *S2 H J*. *S* = *H* $\rightarrow$ *J*, *S2* $\wedge$ *F* = *to-mini-formula H* $\wedge$ *G* = *to-mini-formula J*) $\vee$
($\exists$ *S2 H J*. *S* = *H* $\vee$ *J*, *S2* $\wedge$ *F* = (*to-mini-formula H* $\rightarrow$ $\bot$) $\wedge$ *G* = *to-mini-formula J*) $\vee$
($\exists$ *S2 H J*. *S* = *H* $\wedge$ *J*, *S2* $\wedge$ *F* = *to-mini-formula H* $\rightarrow$ *to-mini-formula J* $\rightarrow$ $\bot$ $\wedge$ *G* = $\bot$) $\vee$
($\exists$ *S2 H*. *S* = $\neg$ *H*, *S2* $\wedge$ *F* = *to-mini-formula H* $\wedge$ *G* = $\bot$)
if *F* $\rightarrow$ *G*, *S1* = *tms S* **for** *F G S1 S* **proof** −
note *that*[*unfolded tms-def*]
then obtain *S2 pim* **where** *S2*: *S1* = *image-mset to-mini-formula S2*
and *S*: *S* = *pim*, *S2*
and *pim*: *F* $\rightarrow$ *G* = *to-mini-formula pim*
by (*metis msed-map-invR union-commute*)
show *?thesis* **using** *pim* **unfolding** *S* **by**(*cases pim*; *simp*; *blast*)
qed
from *asm* **show** *?thesis*
proof(*induction tms* Γ *tms* Δ *arbitrary*: Γ Δ *rule*: *SCp.induct*)
have $*$: *to-mini-formula F* = $\bot$ $\Longrightarrow$ *F* = $\bot$ **for** *F* **by**(*cases F*; *simp*)
case *BotL* **thus** *?case* **unfolding** *tms-def* **using** $*$ *SCp.BotL* **by** (*metis image-iff multiset.set-map*)
next
have $*$: *Atom k* = *to-mini-formula F* $\Longrightarrow$ *F* = *Atom k* **for** *F k* **by**(*cases F*; *simp*)
case (*Ax - k*) **thus** *?case*
unfolding *tms-def* **unfolding** *in-image-mset Set.image-iff*
apply(*elim bexE*)
apply(*drule* $*$)+
apply(*intro SCp.Ax*)
by *simp-all*
next
case (*ImpL* Γ' *F G*)
note *whai*[*OF ImpL*(*5*)]
thus *?case* **proof**(*elim disjE exE conjE*)
fix *S2 H J*
assume $*$: Γ = *H* $\rightarrow$ *J*, *S2 F* = *to-mini-formula H G* = *to-mini-formula J*
hence Γ' = *tms S2 F*, *tms* Δ = *tms* (*H*,Δ) *G*, Γ' = *tms* (*J*, *S2*) *tms* Δ = *tms* Δ **using** *ImpL.hyps*(*5*) *add-left-imp-eq* **by** *auto*
from *ImpL*(*2*)[*OF this*(*1*,*2*)] *ImpL*(*4*)[*OF this*(*3*−)]
show *?thesis* **using** *SCp.ImpL* **by**(*simp add*: $*$)
next
fix *S2 H J*
assume $*$: Γ = *H* $\vee$ *J*, *S2 F* = *to-mini-formula H* $\rightarrow$ $\bot$ *G* = *to-mini-formula J*
hence Γ' = *tms S2 F*, *tms* Δ = *tms* (*H* $\rightarrow$ $\bot$,Δ) *G*, Γ' = *tms* (*J*, *S2*) *tms* Δ = *tms* Δ **using** *ImpL.hyps*(*5*) *add-left-imp-eq* **by** *auto*
from *ImpL*(*2*)[*OF this*(*1*,*2*)] *ImpL*(*4*)[*OF this*(*3*−)]
show *?thesis* **using** *Bot-delR* **by**(*force intro*!: *SCp.OrL dest*!: *ImpR-inv simp*:

∗)
next
fix *S2 H J*
assume ∗: Γ = *H* ∧ *J*, *S2 F* = *to-mini-formula H* → *to-mini-formula J* → ⊥ *G* = ⊥
hence Γ′ = *tms S2 F*, *tms* Δ = *tms* (*H* → *J* → ⊥,Δ) *G*, Γ′ = *tms* (⊥, *S2*) *tms* Δ = *tms* Δ **using** *ImpL.hyps*(*5*) *add-left-imp-eq* **by** *auto*
from *ImpL*(*2*)[*OF this*(*1*,*2*)]
show *?thesis* **using** *Bot-delR SCp.AndL ImpR-inv* ∗ **by** (*metis add-mset-remove-trivial inL1*)
next
fix *S2 H*
assume ∗: Γ = ¬ *H*, *S2 F* = *to-mini-formula H G* = ⊥
hence Γ′ = *tms S2 F*, *tms* Δ = *tms* (*H*,Δ) *G*, Γ′ = *tms* (⊥, *S2*) *tms* Δ = *tms* Δ **using** *ImpL.hyps*(*5*) *add-left-imp-eq* **by** *auto*
from *ImpL*(*2*)[*OF this*(*1*,*2*)]
show *?thesis* **by**(*force intro*!: *SCp.NotL dest*!: *ImpR-inv simp*: ∗)
qed
next
case (*ImpR F G* Δ′)
note *whai*[*OF ImpR*(*3*)]
thus *?case* **proof**(*elim disjE exE conjE*)
fix *S2 H J*
assume Δ = *H* → *J*, *S2 F* = *to-mini-formula H G* = *to-mini-formula J*
thus *?thesis* **using** *ImpR.hyps*(*2*,*3*) **by** (*auto intro*!: *SCp.ImpR*)
next
fix *S2 H J*
assume ∗: Δ = *H* ∨ *J*, *S2 F* = *to-mini-formula H* → ⊥ *G* = *to-mini-formula J*
hence Δ′ = *tms S2 F*, *tms* Δ = *tms* (*H* → ⊥,Δ) *G*, Δ′ = *tms* (*J*, *S2*) *tms* Δ = *tms* Δ **using** *ImpR.hyps*(*3*) *add-left-imp-eq* **by** *auto*
thus *?thesis* **using** *SCp.OrR*[**where** ′*a*=′*a*] ∗ *ImpR.hyps*(*2*) **by** (*simp add*: *NotL-inv*)
next
have *botoff*: Γ ⇒ *H*, ⊥, *S2* ⟹ Γ ⇒ *H*, *S2* **for** Γ *H S2* **using** *Bot-delR* **by** *fastforce*
fix *S2 H J*
assume ∗: Δ = *H* ∧ *J*, *S2 F* = *to-mini-formula H* → *to-mini-formula J* → ⊥ *G* = ⊥
hence *F*, *tms* Γ = *tms* (*H* → *J* → ⊥, Γ) *G*, Δ′ = *tms* (⊥, *S2*)
using *ImpR.hyps*(*3*) **by**(*auto*)
from *ImpR*(*2*)[*OF this*] **show** *?thesis* **by**(*auto simp add*: ∗ *intro*!: *SCp.intros*(*3*−) *dest*!: *ImpL-inv botoff*)
next

fix *S2 H*
assume Δ = ¬ *H*, *S2 F* = *to-mini-formula H G* = ⊥
then obtain *S2 H* **where** ∗: Δ = ¬ *H*, *S2 F* = *to-mini-formula H* ∧ *G* = ⊥ **by** *blast*

hence F, *tms* Γ = *tms* (H, Γ) G, Δ' = *tms* ($\bot$, *S2*) **using** *ImpR*(*3*) **by** *simp-all*
with *ImpR*(*2*) **have** $H, \Gamma \Rightarrow \bot$, *S2* **.**
hence $\Gamma \Rightarrow \neg H$, *S2* **using** *SCp.NotR Bot-delR* **by** *fastforce*
thus $\Gamma \Rightarrow \Delta$ **by**(*simp add*: $*$)
qed
qed *auto*
qed

theorem *MiniSC-eq*: *image-mset to-mini-formula* $\Gamma \Rightarrow$ *image-mset to-mini-formula* $\Delta \longleftrightarrow \Gamma \Rightarrow \Delta$
using *SC-mini-to-full SC-full-to-mini* **by** *blast*

end

3.8.1 SC to HC

theory *MiniSC-HC*
imports *MiniSC HC*
begin

inductive-set *AX1* **where**
$F \in$ *AX0* $\Longrightarrow F \in$ *AX1* |
$((F \to \bot) \to \bot) \to F \in$ *AX1*
lemma *AX01*: *AX0* $\subseteq$ *AX1* **by** (*simp add*: *AX1.intros*(*1*) *subsetI*)
lemma *AX1-away*: *AX1* $\cup\ \Gamma$ = *AX0* $\cup$ ($\Gamma\ \cup$ *AX1*) **using** *AX01* **by** *blast*

lemma *Deduction1*: $F \triangleright$ (*AX1* $\cup\ \Gamma$) $\vdash_H \bot \longleftrightarrow$ (*AX1* $\cup\ \Gamma$) $\vdash_H$ ($F \to \bot$) **unfolding** *AX1-away* **by** (*metis Deduction-theorem HC.simps HC-mono Un-commute Un-insert-left insertI1 subset-insertI*)
lemma *Deduction2*: ($F \to \bot$) $\triangleright$ (*AX1* $\cup\ \Gamma$) $\vdash_H \bot \longleftrightarrow$ (*AX1* $\cup\ \Gamma$) $\vdash_H F$ (**is** *?l* $\longleftrightarrow$ *?r*)
proof
assume *l*: *?l*
with *Deduction-theorem*[**where** Γ=*AX1* $\cup\ \Gamma$ **and** F=$F \to \bot$ **and** G=$\bot$]
have *AX1* $\cup\ \Gamma \vdash_H (F \to \bot) \to \bot$ **unfolding** *AX1-away* **by**(*simp add*: *Un-commute*)
moreover have *AX1* $\cup\ \Gamma \vdash_H ((F \to \bot) \to \bot) \to F$ **using** *AX1.intros*(*2*) *HC.Ax* **by** *blast*
ultimately show *?r* **using** *HC.MP* **by** *blast*
next
assume *r*: *?r* **thus** *?l* **by** (*meson HC.simps HC-mono insertI1 subset-insertI*)
qed

lemma
$\Gamma \Rightarrow \Delta \Longrightarrow$ *is-mini-mset* $\Gamma \Longrightarrow$ *is-mini-mset* $\Delta \Longrightarrow$
(*set-mset* $\Gamma \cup (\lambda F.\ F \to \bot)$ ‘ *set-mset* $\Delta\ \cup$ *AX1*) $\vdash_H \bot$
proof(*induction* $\Gamma\ \Delta$ *rule*: *SCp.induct*)
case (*ImpL* Γ F Δ G)
from *ImpL.prems* **have** *is-mini-mset* Γ *is-mini-mset* (F, Δ) **by** *simp-all*

with *ImpL.IH*(*1*) **have** *IH1*: *set-mset* Γ ∪ (λ*F*. *F* → ⊥) ' *set-mset* (*F*, Δ) ∪ *AX1* $\vdash_H$ ⊥ **.**
hence *IH1*: *F* → ⊥ ▷ *set-mset* Γ ∪ (λ*F*. *F* → ⊥) ' *set-mset* Δ ∪ *AX1* $\vdash_H$ ⊥ **by** *simp*
from *ImpL.prems* **have** *is-mini-mset* (*G*, Γ) *is-mini-mset* Δ **by** *simp-all*
with *ImpL.IH*(*2*) **have** *IH2*: *set-mset* (*G*, Γ) ∪ (λ*F*. *F* → ⊥) ' *set-mset* Δ ∪ *AX1* $\vdash_H$ ⊥ **.**
hence *IH2*: *G* ▷ (*set-mset* Γ ∪ (λ*F*. *F* → ⊥) ' *set-mset* Δ ∪ *AX1*) $\vdash_H$ ⊥ **by** *simp*
have *R*: *F* → *G* ▷ *AX1* ∪ Γ $\vdash_H$ ⊥ **if** *G* ▷ *AX1* ∪ Γ $\vdash_H$ ⊥ *F* → ⊥ ▷ *AX1* ∪ Γ $\vdash_H$ ⊥ **for** Γ
using *that* **by** (*metis Ax Deduction1 Deduction2 HC-mono MP insertI1 subset-insertI*)

from *R*[**where** Γ=*set-mset* Γ ∪ (λ*F*. *F* → ⊥) ' *set-mset* Δ]
have *F* → *G* ▷ (*set-mset* Γ ∪ (λ*F*. *F* → ⊥) ' *set-mset* Δ ∪ *AX1*) $\vdash_H$ ⊥ **using** *IH2 IH1* **by**(*simp add*: *Un-commute*)
thus *?case* **by** *simp*
next
case (*ImpR F* Γ *G* Δ)
hence *is-mini-mset* (*F*, Γ) *is-mini-mset* (*G*, Δ) **by** *simp-all*
with *ImpR.IH* **have** *IH*: *F* ▷ *G* → ⊥ ▷ *set-mset* Γ ∪ (λ*F*. *F* → ⊥) ' *set-mset* Δ ∪ *AX1* $\vdash_H$ ⊥ **by**(*simp add*: *insert-commute*)
have *R*: (*F* → *G*) → ⊥ ▷ Γ ∪ *AX1* $\vdash_H$ ⊥ **if** *F* ▷ *G* → ⊥ ▷ Γ ∪ *AX1* $\vdash_H$ ⊥ **for** Γ **using** *that*
by (*metis AX1-away Deduction2 Deduction-theorem Un-commute Un-insert-right*)
thus *?case* **using** *IH* **by** *simp*
next
case (*Ax k* Γ Δ)
have *R*: *F* ▷ *F* → ⊥ ▷ Γ ∪ *AX1* $\vdash_H$ ⊥ **for** *F* :: *'a formula* **and** Γ **by** (*meson HC.simps insert-iff*)
from *R*[**where** *F*=*Atom k* **and** Γ=*set-mset* Γ ∪ (λ*F*. *F* → ⊥) ' *set-mset* Δ] **show** *?case* **using** *Ax.hyps*
by (*simp add*: *insert-absorb*)
next
case *BotL* **thus** *?case* **by** (*simp add*: *HC.Ax*)
qed *simp-all*

end

3.8.2 Craig Interpolation

theory *MiniSC-Craig*
imports *MiniSC Formulas*
begin

abbreviation *atoms-mset* **where** *atoms-mset* Θ ≡ ⋃*F* ∈ *set-mset* Θ. *atoms F*

lemma *interpolation-equal-styles*:
(∀Γ Δ Γ′ Δ′. Γ + Γ′ ⇒ Δ + Δ′ ⟶ (∃*F* :: ′*a formula*. Γ ⇒ *F*,Δ ∧ *F*,Γ′ ⇒ Δ′ ∧ *atoms F* ⊆ *atoms-mset* (Γ + Δ) ∧ *atoms F* ⊆ *atoms-mset* (Γ′ + Δ′)))
⟷
(∀Γ Δ. Γ ⇒ Δ ⟶ (∃*F* :: ′*a formula*. Γ ⇒ {#*F*#} ∧ {#*F*#} ⇒ Δ ∧ *atoms F* ⊆ *atoms-mset* Γ ∧ *atoms F* ⊆ *atoms-mset* Δ))
proof(*intro iffI allI impI*, *goal-cases*)
  case (*1* Γ Δ)
  hence Γ + {#} ⇒ {#} + Δ ⟶ (∃*F*. Γ ⇒ *F*, {#} ∧ *F*, {#} ⇒ Δ ∧ *atoms F* ⊆ *atoms-mset* (Γ + {#}) ∧ *atoms F* ⊆ *atoms-mset* ({#} + Δ)) **by** *presburger*
  with *1* **show** *?case* **by** *auto*
next
  case (*2* Γ Δ Γ′ Δ′)
  from *2*(*2*) **have** Γ ⇒ Δ + *image-mset Not* Γ′ + Δ′ **by**(*induction* Γ′ *arbitrary*: Γ; *simp add*: *SCp.NotR*)
  hence Γ + *image-mset Not* Δ ⇒ *image-mset Not* Γ′ + Δ′ **by**(*induction* Δ *arbitrary*: Δ′; *simp add*: *SCp.NotL*) (*metis SCp.NotL union-mset-add-mset-right*)
  from *2*(*1*)[*THEN spec*, *THEN spec*, *THEN mp*, *OF this*]
  have ∃*F*. Γ + *image-mset* ¬ Δ ⇒ {#*F*#} ∧ {#*F*#} ⇒ *image-mset* ¬ Γ′ + Δ′ ∧ *atoms F* ⊆ *atoms-mset* (Γ + *image-mset* ¬ Δ) ∧ *atoms F* ⊆ *atoms-mset* (*image-mset* ¬ Γ′ + Δ′) **.**
  then obtain *F* **where** *n*: Γ + *image-mset* ¬ Δ ⇒ {#*F*#} **and** *p*: {#*F*#} ⇒ *image-mset* ¬ Γ′ + Δ′ **and** *at*: *atoms F* ⊆ *atoms-mset* (Γ + Δ) *atoms F* ⊆ *atoms-mset* (Γ′ + Δ′) **by** *auto*
  from *n* **have** *n*: Γ ⇒ *F*, Δ **by**(*induction* Δ *arbitrary*: Γ; *simp add*: *NotL-inv inR1*)
  from *p* **have** *p*: *F*,Γ′ ⇒ Δ′ **by**(*induction* Γ′ *arbitrary*: Δ′; *simp add*: *NotR-inv inL1*)
  show *?case* **using** *p at n* **by** *blast*
qed

The original version of the interpolation theorem is due to Craig [1]. Our proof partly follows the approach of Troelstra and Schwichtenberg [11] but, especially with the mini formulas, adds its own spin.

theorem *SC-Craig-interpolation*:
  assumes Γ + Γ′ ⇒ Δ + Δ′
  obtains *F* **where**
    Γ ⇒ *F*,Δ
    F,Γ′ ⇒ Δ′
    atoms F ⊆ *atoms-mset* (Γ + Δ)
    atoms F ⊆ *atoms-mset* (Γ′ + Δ′)
proof −
  have *split-seq*: (∃*H*′. *H* = *f F J*,*H*′) ∨ (∃*I*′. *I* = *f F J*,*I*′) **if** *f F J*, *G* = *H* + *I* **for** *f F G H I J*
  proof −
   from *that* **have** *f F J* ∈# *H* + *I* **by**(*metis* (*mono-tags*) *add-ac*(*2*) *union-single-eq-member*)
    thus *?thesis* **by** (*meson multi-member-split union-iff*)
  qed
  have *mini*: ∃ *F*. Γ ⇒ *F*, Δ ∧ *F*, Γ′ ⇒ Δ′ ∧

atoms $F \subseteq$ *atoms-mset* $(\Gamma + \Delta) \wedge$ *atoms* $F \subseteq$ *atoms-mset* $(\Gamma' + \Delta') \wedge$ *is-mini-formula* F

if $\Gamma + \Gamma' \Rightarrow \Delta + \Delta'$ *is-mini-mset* $(\Gamma+\Gamma'+\Delta+\Delta')$ **for** $\Gamma\ \Gamma'\ \Delta\ \Delta'$

using *that* **proof**(*induction* $\Gamma + \Gamma'\ \Delta + \Delta'$ *arbitrary*: $\Gamma\ \Gamma'\ \Delta\ \Delta'$ *rule*: *SCp.induct*)

case *BotL* **thus** *?case* **proof**(*cases*; *intro exI*)

assume $\bot \in\#\ \Gamma$ **with** *BotL*

show $\Gamma \Rightarrow \bot, \Delta \wedge \bot, \Gamma' \Rightarrow \Delta' \wedge$ *atoms* $\bot \subseteq$ *atoms-mset* $(\Gamma + \Delta) \wedge$ *atoms* $\bot \subseteq$ *atoms-mset* $(\Gamma' + \Delta') \wedge$ *is-mini-formula* $\bot$

by(*simp add*: *SCp.BotL*)

next

assume $\neg(\bot \in\#\ \Gamma)$ **with** *BotL*

show $\Gamma \Rightarrow \top, \Delta \wedge \top, \Gamma' \Rightarrow \Delta' \wedge$ *atoms* $\top \subseteq$ *atoms-mset* $(\Gamma + \Delta) \wedge$ *atoms* $\top \subseteq$ *atoms-mset* $(\Gamma' + \Delta') \wedge$ *is-mini-formula* $\top$

by(*auto simp add*: *Top-def SCp.ImpR SCp.ImpL SCp.BotL intro*!: *SCp.intros*(*3*−))

qed

next

case (*Ax k*)

let *?ss* = λF. $(\Gamma \Rightarrow F, \Delta \wedge F, \Gamma' \Rightarrow \Delta' \wedge$ *is-mini-formula* $F)$

have *ff*: *?ss* $\bot$ **if** *Atom k* $\in\#\ \Gamma$ *Atom k* $\in\#\ \Delta$

using *SCp.BotL SCp.Ax*[*of k*] *that* **by** *auto*

have *fs*: *?ss* (*Atom k*) **if** *Atom k* $\in\#\ \Gamma$ *Atom k* $\in\#\ \Delta'$

using *that* **by**(*auto intro*!: *SCp.Ax*[**where** *k*=*k*])

have *sf*: *?ss* ((*Atom k*) $\rightarrow \bot$) **if** *Atom k* $\in\#\ \Gamma'$ *Atom k* $\in\#\ \Delta$

using *that* **by**(*auto intro*!: *SCp.ImpR SCp.ImpL intro*: *SCp.Ax*[**where** *k*=*k*] *SCp.BotL*)

have *ss*: *?ss* $\top$ **if** *Atom k* $\in\#\ \Gamma'$ *Atom k* $\in\#\ \Delta'$

unfolding *Top-def* **using** *that SCp.ImpR SCp.Ax BotL-canonical* **by** *fastforce*

have *in-sumE*: $[\![A \in\# (F + G); A \in\# F \Longrightarrow P; A \in\# G \Longrightarrow P]\!] \Longrightarrow P$ **for** $A\ F\ G\ P$ **by** *fastforce*

have *trust-firstE*: $P\ F \Longrightarrow Q\ F \Longrightarrow \exists F.\ P\ F \wedge Q\ F$ **for** $P\ Q\ F$ **by** *blast*

from *Ax* **show** *?case* **by**(*elim in-sumE*) (*frule* (*1*) *ff fs sf ss*; *elim conjE trust-firstE*; *force*)+

next

case (*ImpR F G* Δ'')

note *split-seq*[*of Imp*, *OF ImpR*(*3*)]

thus *?case* **proof**(*elim disjE exE*)

fix H' **assume** Δ: $\Delta = F \rightarrow G, H'$

have $F, \Gamma + \Gamma' = (F, \Gamma) + \Gamma'$ $G, \Delta'' = (G, \Delta - \{\#F \rightarrow G\#\}) + \Delta'$ *is-mini-mset* $((F, \Gamma) + \Gamma' + (G, \Delta - \{\#F \rightarrow G\#\}) + \Delta')$

using *that ImpR*(*3*−) **by** (*simp-all add*: *union-assoc* Δ)

from *ImpR*(*2*)[*OF this*] **obtain** *Fa* **where** *Fam*:

$F, \Gamma \Rightarrow$ *Fa*, G, H' *Fa*, $\Gamma' \Rightarrow \Delta'$ *is-mini-formula Fa*

atoms Fa $\subseteq$ *atoms-mset* $((F, \Gamma) + (G, H'))$ *atoms Fa* $\subseteq$ *atoms-mset* $(\Gamma' + \Delta')$ **unfolding** Δ **by** *auto*

thus *?thesis* **unfolding** Δ **proof**(*intro exI*[**where** *x*=*Fa*] *conjI* ‹*is-mini-formula Fa*›)

show $\Gamma \Rightarrow$ *Fa*, $F \rightarrow G, H'$ **using** *Fam* **by**(*intro SCp.ImpR*[*THEN inR1*]; *fast*)

show *Fa*, $\Gamma' \Rightarrow \Delta'$ **using** *Fam* **by** *blast*

show *atoms Fa* ⊆ *atoms-mset* (Γ + (*F* → *G*, *H*′)) *atoms Fa* ⊆ *atoms-mset* (Γ′ + Δ′) **using** *Fam* **by** *auto*
qed
next
fix *I*′ **assume** Δ′: Δ′ = *F* → *G*, *I*′
have *F*, Γ + Γ′ = Γ + (*F*,Γ′) *G*, Δ′′ = Δ + (*G*, *I*′) *is-mini-mset* (Γ + (*F*, Γ′) + Δ + (*G*, *I*′))
using *ImpR*(*3*−) **by** (*simp-all add*: *add.left-commute* Δ′)
from *ImpR*(*2*)[*OF this*] **obtain** *Fa m* **where** *Fam*:
Γ ⇒ *Fa*, Δ *Fa*, *F*, Γ′ ⇒ *G*, *I*′ *is-mini-formula Fa*
atoms Fa ⊆ *atoms-mset* (Γ + Δ) *atoms Fa* ⊆ *atoms-mset* ((*F*, Γ′) + (*G*, *I*′)) **unfolding** Δ′ **by** *auto*
show *?thesis* **unfolding** Δ′ **proof**(*intro exI*[**where** *x*=*Fa*] *conjI* ‹*is-mini-formula Fa*›)
show Γ ⇒ *Fa*, Δ **using** *Fam* **by** *fast*
show *Fa*, Γ′ ⇒ *F* → *G*, *I*′ **using** *Fam* **by** (*simp add*: *SCp.ImpR inL1*)
show *atoms Fa* ⊆ *atoms-mset* (Γ + Δ) *atoms Fa* ⊆ *atoms-mset* (Γ′ + (*F* → *G*, *I*′)) **using** *Fam* **by** *auto*
qed
qed
next
case (*ImpL* Γ′′ *F G*)
note *split-seq*[*of Imp*, *OF ImpL*(*5*)]
thus *?case* **proof**(*elim disjE exE*)
fix *H*′ **assume** Γ: Γ = *F* → *G*, *H*′
from Γ **have** ∗: Γ′′ = Γ′ + *H*′ *F*, Δ + Δ′ = Δ′ + (*F*,Δ)
using *ImpL*(*5*−) **by** (*simp-all add*: *union-assoc* Γ)
hence *is-mini-mset* (Γ′ + *H*′ + Δ′ + (*F*, Δ)) **using** *ImpL*(*6*) **by**(*auto simp add*: Γ)
from *ImpL*(*2*)[*OF* ∗ *this*] **obtain** *Fa* **where** *IH1*: Γ′ ⇒ *Fa*, Δ′ *Fa*, *H*′ ⇒ *F*, Δ
atoms Fa ⊆ *atoms-mset* (*H*′ + (*F*,Δ)) *atoms Fa* ⊆ *atoms-mset* (Γ′ + Δ′) *is-mini-formula Fa* **by** *blast*
from Γ **have** ∗: *G*, Γ′′ = (*G*, *H*′) + Γ′ Δ + Δ′ = Δ + Δ′
using *ImpL*(*5*−) **by** (*simp-all add*: *union-assoc*)
hence *is-mini-mset* ((*G*, *H*′) + Γ′ + Δ + Δ′) **using** *ImpL*(*6*) **by**(*simp add*: Γ)
from *ImpL*(*4*)[*OF* ∗ *this*] **obtain** *Ga* **where** *IH2*: *G*, *H*′ ⇒ *Ga*, Δ *Ga*, Γ′ ⇒ Δ′
atoms Ga ⊆ *atoms-mset* ((*G*, *H*′) + Δ) *atoms Ga* ⊆ *atoms-mset* (Γ′ + Δ′) *is-mini-formula Ga* **by** *blast*

A big part of the difficulty of this proof is finding a way to instantiate the IHs. Interestingly, this is not the only way that works. Originally, we used Γ′′ = *H*′ + Γ′ and *F*, Δ + Δ′ = (*F*, Δ) + Δ′ to instantiate the IH (which is in some sense more natural, less use of commutativity). This gave us a *Fa* that verifies *H*′ ⇒ *Fa*, *F*, Δ and *Fa*, Γ′ ⇒ Δ′ and resulted in the interpolant *to-mini-formula* (*Fa* ∨ *Ga*).

let *?w* = *Fa* → *Ga*

show *?thesis* **proof**(*intro exI*[**where** x=*?w*] *conjI*)
from *IH1*(*1*) *IH2*(*2*) **show** *?w*, $\Gamma' \Rightarrow \Delta'$
by (*simp add*: *SCp.ImpL*)
from *IH1*(*2*) *IH2*(*1*) **show** $\Gamma \Rightarrow$ *?w*, Δ **unfolding** Γ
by(*intro SCp.ImpL inR1*[*OF SCp.ImpR*] *SCp.ImpR*) (*simp-all add*: *weakenR weakenL*)
show *atoms ?w* $\subseteq$ *atoms-mset* ($\Gamma + \Delta$) *atoms ?w* $\subseteq$ *atoms-mset* ($\Gamma' + \Delta'$)
using *IH1*(*3*−) *IH2*(*3*−) **unfolding** Γ **by** *auto*
show *is-mini-formula ?w* **using** ‹*is-mini-formula Fa*› ‹*is-mini-formula Ga*› **by** *simp*
qed
next
fix I' **assume** Γ': $\Gamma' = F \rightarrow G, I'$ **note** *ImpL*(*5*)[*unfolded* Γ']
from Γ' **have** $*$: $\Gamma'' = \Gamma + I'\ F$, $\Delta + \Delta' = \Delta + (F,\Delta')$
using *ImpL*(*5*−) **by**(*simp-all add*: *union-assoc add-ac*(*2*,*3*))
hence *is-mini-mset* ($\Gamma + I' + \Delta + (F, \Delta')$) **using** *ImpL*(*6*) **by**(*auto simp add*: Γ')
from *ImpL*(*2*)[*OF* $*$ *this*] **obtain** *Fa*
where *IH1*: $\Gamma \Rightarrow$ *Fa*, Δ *Fa*, $I' \Rightarrow F$, Δ' *is-mini-formula Fa*
atoms Fa $\subseteq$ *atoms-mset* ($I' + (F,\Delta')$) *atoms Fa* $\subseteq$ *atoms-mset* ($\Gamma + \Delta$) **by** *blast*
from Γ' **have** $*$: $G, \Gamma'' = \Gamma + (G, I')$ $\Delta + \Delta' = \Delta + \Delta'$
using *ImpL*(*5*) **by** (*simp-all add*: *add-ac* ‹$\Gamma'' = \Gamma + I'$›)
have *is-mini-mset* ($\Gamma + (G, I') + \Delta + \Delta'$) **using** *ImpL*(*6*) **by**(*auto simp add*: Γ')
from *ImpL*(*4*)[*OF* $*$ *this*] **obtain** *Ga l*
where *IH2*: $\Gamma \Rightarrow$ *Ga*, Δ *Ga*, G, $I' \Rightarrow \Delta'$ *is-mini-formula Ga*
atoms Ga $\subseteq$ *atoms-mset* ($(G, I') + \Delta'$) *atoms Ga* $\subseteq$ *atoms-mset* ($\Gamma + \Delta$) **by** *blast*

Same thing as in the other case, just with $G, \Gamma'' = (G, I') + \Gamma$, $\Delta + \Delta' = \Delta' + \Delta$, $\Gamma'' = I' + \Gamma$, and $F, \Delta + \Delta' = (F, \Delta') + \Delta$ resulting in *to-mini-formula* ($\neg$ (*Fa* $\vee$ *Ga*))

let *?w* = (*Ga* $\rightarrow$ (*Fa* $\rightarrow \bot$)) $\rightarrow \bot$
have *?w* = *to-mini-formula* (*Ga* $\wedge$ *Fa*) **by** (*simp add*: *IH1*(*3*) *IH2*(*3*) *mini-to-mini*)
show *?thesis* **proof**(*intro exI*[*of - ?w*] *conjI*)
from *IH1*(*1*) *IH2*(*1*) **show** $\Gamma \Rightarrow$ *?w*, Δ
by(*intro SCp.ImpR SCp.ImpL*) (*simp-all add*: *inR1 weakenR BotL-canonical*)
from *IH1*(*2*) *IH2*(*2*) **show** *?w*, $\Gamma' \Rightarrow \Delta'$ **unfolding** Γ'
by(*blast intro*!: *SCp.ImpL SCp.ImpR dest*: *weakenL weakenR*)+
show *atoms ?w* $\subseteq$ *atoms-mset* ($\Gamma + \Delta$)
atoms ?w $\subseteq$ *atoms-mset* ($\Gamma' + \Delta'$) **using** *IH1*(*3*−) *IH2*(*3*−) **unfolding** Γ' **by** *auto*
show *is-mini-formula ?w* **using** ‹*is-mini-formula Fa*› ‹*is-mini-formula Ga*› **by** *simp*
qed
qed
next

The rest is just those cases that can't happen because of the mini formula property.

qed (*metis add.commute is-mini-formula.simps union-iff union-single-eq-member*)+
define *tms* :: *'a formula multiset* ⇒ *'a formula multiset*
where *tms* = *image-mset to-mini-formula*
have [*simp*]: *tms* (*A* + *B*) = *tms A* + *tms B tms* {#*F*#} = {#*to-mini-formula F*#} **for** *A B F* **unfolding** *tms-def* **by** *simp-all*
have [*simp*]: *atoms-mset* (*tms* Γ) = *atoms-mset* Γ **for** Γ **unfolding** *tms-def* **using** *mini-formula-atoms* **by** *fastforce*
have *imm*: *is-mini-mset* (*tms* Γ + *tms* Γ′ + *tms* Δ + *tms* Δ′) **unfolding** *tms-def* **by** *auto*
from *assms* **have** *tms* Γ + *tms* Γ′ ⇒ *tms* Δ + *tms* Δ′ **unfolding** *tms-def* **using** *SC-full-to-mini* **by** *force*
from *mini*[*OF this imm*] **obtain** *F* **where** *hp*:
tms Γ ⇒ *F*, *tms* Δ *F*, *tms* Γ′ ⇒ *tms* Δ′
and *su*: *atoms F* ⊆ *atoms-mset* (*tms* Γ + *tms* Δ) *atoms F* ⊆ *atoms-mset* (*tms* Γ′ + *tms* Δ′)
and *mf*: *is-mini-formula F* **by** *blast*
from *hp mf* **have** *tms* Γ ⇒ *tms* (*F*, Δ) *tms* (*F*, Γ′) ⇒ *tms* Δ′ **using** *mini-to-mini*[**where** *'a='a*] **unfolding** *tms-def* **by** *simp-all*
hence Γ ⇒ *F*, Δ *F*, Γ′ ⇒ Δ′ **using** *SC-mini-to-full* **unfolding** *tms-def* **by** *blast+*
with *su* **show** *?thesis* **using** ‹⋀Γ. *atoms-mset* (*tms* Γ) = *atoms-mset* Γ› *image-mset-union that* **by** *auto*
qed

Note that there is an extension to Craig interpolation: One can show that atoms that only appear positively/negatively in the original formulas will only appear positively/negatively in the interpolant.

abbreviation *patoms-mset S* ≡ ⋃*F*∈*set-mset S*. *fst* (*pn-atoms F*)
abbreviation *natoms-mset S* ≡ ⋃*F*∈*set-mset S*. *snd* (*pn-atoms F*)

theorem *SC-Craig-interpolation-pn*:
assumes Γ + Γ′ ⇒ Δ + Δ′
obtains *F* **where**
Γ ⇒ *F*,Δ
F,Γ′ ⇒ Δ′
fst (*pn-atoms F*) ⊆ (*patoms-mset* Γ ∪ *natoms-mset* Δ) ∩ (*natoms-mset* Γ′ ∪ *patoms-mset* Δ′)
snd (*pn-atoms F*) ⊆ (*natoms-mset* Γ ∪ *patoms-mset* Δ) ∩ (*patoms-mset* Γ′ ∪ *natoms-mset* Δ′)
proof −
have *split-seq*: (∃ *H′*. *H* = *f F J*,*H′*) ∨ (∃ *I′*. *I* = *f F J*,*I′*) **if** *f F J*, *G* = *H* + *I* **for** *f F G H I J*
proof −
from *that* **have** *f F J* ∈# *H* + *I* **by**(*metis* (*mono-tags*) *add-ac*(*2*) *union-single-eq-member*)
thus *?thesis* **by** (*meson multi-member-split union-iff*)
qed
have *mini*: ∃ *F* :: *'a formula*. Γ ⇒ *F*, Δ ∧ *F*, Γ′ ⇒ Δ′ ∧

fst (*pn-atoms* F) $\subseteq$ (*patoms-mset* $\Gamma \cup$ *natoms-mset* Δ) $\cap$ (*natoms-mset* $\Gamma' \cup$ *patoms-mset* Δ') $\wedge$
snd (*pn-atoms* F) $\subseteq$ (*natoms-mset* $\Gamma \cup$ *patoms-mset* Δ) $\cap$ (*patoms-mset* $\Gamma' \cup$ *natoms-mset* Δ') $\wedge$ *is-mini-formula* F
if $\Gamma + \Gamma' \Rightarrow \Delta + \Delta'$ *is-mini-mset* $(\Gamma+\Gamma'+\Delta+\Delta')$ **for** $\Gamma\ \Gamma'\ \Delta\ \Delta'$
using *that* **proof**(*induction* $\Gamma + \Gamma'\ \Delta + \Delta'$ *arbitrary*: $\Gamma\ \Gamma'\ \Delta\ \Delta'$ *rule*: *SCp.induct*)
case *BotL*
let *?om* = λF. *fst* (*pn-atoms* F) $\subseteq$ (*patoms-mset* $\Gamma \cup$ *natoms-mset* Δ) $\cap$ (*natoms-mset* $\Gamma' \cup$ *patoms-mset* Δ') $\wedge$
snd (*pn-atoms* F) $\subseteq$ (*natoms-mset* $\Gamma \cup$ *patoms-mset* Δ) $\cap$ (*patoms-mset* Γ' $\cup$ *natoms-mset* Δ') $\wedge$ *is-mini-formula* (F :: *'a formula*)
show *?case* **proof**(*cases*; *intro exI*)
assume $\bot \in\#\ \Gamma$ **with** *BotL*
show $\Gamma \Rightarrow \bot, \Delta \wedge \bot, \Gamma' \Rightarrow \Delta' \wedge$ *?om* $\bot$ **by**(*simp add*: *SCp.BotL*)
next
assume $\neg(\bot \in\#\ \Gamma)$ **with** *BotL*
show $\Gamma \Rightarrow \top, \Delta \wedge \top, \Gamma' \Rightarrow \Delta' \wedge$ *?om* $\top$
by(*auto simp add*: *Top-def SCp.ImpR SCp.ImpL SCp.BotL prod-unions-def intro*!: *SCp.intros*(*3*−))
qed
next
case (*Ax k*)
let *?ss* = λF. ($\Gamma \Rightarrow F, \Delta \wedge F, \Gamma' \Rightarrow \Delta' \wedge$ *fst* (*pn-atoms* F) $\subseteq$ (*patoms-mset* $\Gamma \cup$ *natoms-mset* Δ) $\cap$ (*natoms-mset* $\Gamma' \cup$ *patoms-mset* Δ') $\wedge$
snd (*pn-atoms* F) $\subseteq$ (*natoms-mset* $\Gamma \cup$ *patoms-mset* Δ) $\cap$ (*patoms-mset* Γ' $\cup$ *natoms-mset* Δ') $\wedge$ *is-mini-formula* F)
have *ff*: *?ss* $\bot$ **if** *Atom k* $\in\#\ \Gamma$ *Atom k* $\in\#\ \Delta$
using *SCp.BotL SCp.Ax*[*of k*] *that* **by** *auto*
have *fs*: *?ss* (*Atom k*) **if** *Atom k* $\in\#\ \Gamma$ *Atom k* $\in\#\ \Delta'$
using *that* **by**(*force intro*!: *SCp.Ax*[**where** *k*=*k*])
have *sf*: *?ss* ((*Atom k*) $\rightarrow \bot$) **if** *Atom k* $\in\#\ \Gamma'$ *Atom k* $\in\#\ \Delta$
using *that* **by**(*auto intro*!: *SCp.ImpR SCp.ImpL intro*: *SCp.Ax*[**where** *k*=*k*] *SCp.BotL exI*[**where** *x*=*Atom k*] *simp add*: *prod-unions-def*; *force*)
have *ss*: *?ss* $\top$ **if** *Atom k* $\in\#\ \Gamma'$ *Atom k* $\in\#\ \Delta'$
unfolding *Top-def* **using** *that SCp.ImpR* **by** (*auto simp add*: *prod-unions-def SCp.Ax*)
have *in-sumE*: $[\![A \in\# (F + G); A \in\# F \Longrightarrow P; A \in\# G \Longrightarrow P]\!] \Longrightarrow P$ **for** $A\ F\ G\ P$ **by** *fastforce*
have *trust-firstE*: $P\ F \Longrightarrow Q\ F \Longrightarrow \exists F.\ P\ F \wedge Q\ F$ **for** $P\ Q\ F$ **by** *blast*
from *Ax* **show** *?case* **by**(*elim in-sumE*) (*frule* (*1*) *ff fs sf ss*; *elim conjE trust-firstE*; *force*)+
next
next
case (*ImpR F G* Δ'')
note *split-seq*[*of Imp*, *OF ImpR*(*3*)]
thus *?case* **proof**(*elim disjE exE*)
fix H' **assume** Δ: $\Delta = F \rightarrow G, H'$
have $F, \Gamma + \Gamma' = (F, \Gamma) + \Gamma'\ G, \Delta'' = (G, \Delta - \{\#F \rightarrow G\#\}) + \Delta'$ *is-mini-mset* $((F, \Gamma) + \Gamma' + (G, \Delta - \{\#F \rightarrow G\#\}) + \Delta')$

using *that ImpR*(*3*−) **by** (*simp-all add*: *union-assoc* Δ)
from *ImpR*(*2*)[*OF this*] **obtain** *Fa* **where** *Fam*:
F, Γ ⇒ *Fa*, G, H' *Fa*, Γ′ ⇒ Δ′ *is-mini-formula Fa*
fst (*pn-atoms Fa*) ⊆ (*patoms-mset* (F, Γ) ∪ *natoms-mset* (G, H')) ∩ (*natoms-mset* Γ′ ∪ *patoms-mset* Δ′)
snd (*pn-atoms Fa*) ⊆ (*natoms-mset* (F, Γ) ∪ *patoms-mset* (G, H')) ∩ (*patoms-mset* Γ′ ∪ *natoms-mset* Δ′) **unfolding** Δ **by** *auto*
thus *?thesis* **unfolding** Δ **proof**(*intro exI*[**where** *x*=*Fa*] *conjI* ‹*is-mini-formula Fa*›)
show Γ ⇒ *Fa*, F → G, H' **using** *Fam* **by**(*intro SCp.ImpR*[*THEN inR1*]; *fast*)
show *Fa*, Γ′ ⇒ Δ′ **using** *Fam* **by** *blast*
show *fst* (*pn-atoms Fa*) ⊆ (*patoms-mset* Γ ∪ *natoms-mset* (F → G, H')) ∩ (*natoms-mset* Γ′ ∪ *patoms-mset* Δ′)
snd (*pn-atoms Fa*) ⊆ (*natoms-mset* Γ ∪ *patoms-mset* (F → G, H')) ∩ (*patoms-mset* Γ′ ∪ *natoms-mset* Δ′)
using *Fam*(*4*−) **by** (*auto simp*: *prod-unions-def split*: *prod.splits*)
qed
next
fix I' **assume** Δ′: Δ′ = F → G, I'
have F, Γ + Γ′ = Γ + (F,Γ′) G, Δ″ = Δ + (G, I') *is-mini-mset* (Γ + (F, Γ′) + Δ + (G, I'))
using *ImpR*(*3*−) **by** (*simp-all add*: *add.left-commute* Δ′)
from *ImpR*(*2*)[*OF this*] **obtain** *Fa m* **where** *Fam*:
Γ ⇒ *Fa*, Δ *Fa*, F, Γ′ ⇒ G, I' *is-mini-formula Fa*
fst (*pn-atoms Fa*) ⊆ (*patoms-mset* Γ ∪ *natoms-mset* Δ) ∩ (*natoms-mset* (F, Γ′) ∪ *patoms-mset* (G, I'))
snd (*pn-atoms Fa*) ⊆ (*natoms-mset* Γ ∪ *patoms-mset* Δ) ∩ (*patoms-mset* (F, Γ′) ∪ *natoms-mset* (G, I')) **unfolding** Δ′ **by** *auto*
show *?thesis* **unfolding** Δ′ **proof**(*intro exI*[**where** *x*=*Fa*] *conjI* ‹*is-mini-formula Fa*›)
show Γ ⇒ *Fa*, Δ **using** *Fam* **by** *fast*
show *Fa*, Γ′ ⇒ F → G, I' **using** *Fam* **by** (*simp add*: *SCp.ImpR inL1*)
show *fst* (*pn-atoms Fa*) ⊆ (*patoms-mset* Γ ∪ *natoms-mset* Δ) ∩ (*natoms-mset* Γ′ ∪ *patoms-mset* (F → G, I'))
snd (*pn-atoms Fa*) ⊆ (*natoms-mset* Γ ∪ *patoms-mset* Δ) ∩ (*patoms-mset* Γ′ ∪ *natoms-mset* (F → G, I'))
using *Fam* **by** (*auto simp*: *prod-unions-def split*: *prod.splits*)
qed
qed
next
next
case (*ImpL* Γ″ F G)
note *split-seq*[*of Imp*, *OF ImpL*(*5*)]
thus *?case* **proof**(*elim disjE exE*)
fix H' **assume** Γ: Γ = F → G, H'
from Γ **have** ∗: Γ″ = Γ′ + H' F, Δ + Δ′ = Δ′ + (F,Δ)
using *ImpL*(*5*−) **by** (*simp-all add*: *union-assoc* Γ)
hence *is-mini-mset* (Γ′ + H' + Δ′ + (F, Δ)) **using** *ImpL*(*6*) **by**(*auto simp*

add: Γ)
from *ImpL*(*2*)[*OF* ∗ *this*] **obtain** *Fa* **where** *IH1*: $\Gamma' \Rightarrow Fa, \Delta'$ $Fa, H' \Rightarrow F, \Delta$
fst (*pn-atoms Fa*) ⊆ (*patoms-mset* Γ′ ∪ *natoms-mset* Δ′) ∩ (*natoms-mset* H′ ∪ *patoms-mset* (F, Δ))
snd (*pn-atoms Fa*) ⊆ (*natoms-mset* Γ′ ∪ *patoms-mset* Δ′) ∩ (*patoms-mset* H′ ∪ *natoms-mset* (F, Δ)) *is-mini-formula Fa* **by** *blast*
from Γ **have** ∗: $G, \Gamma'' = (G, H') + \Gamma'$ $\Delta + \Delta' = \Delta + \Delta'$
using *ImpL*(*5*−) **by** (*simp-all add*: *union-assoc*)
hence *is-mini-mset* $((G, H') + \Gamma' + \Delta + \Delta')$ **using** *ImpL*(*6*) **by**(*simp add*: Γ)
from *ImpL*(*4*)[*OF* ∗ *this*] **obtain** *Ga* **where** *IH2*: $G, H' \Rightarrow Ga, \Delta$ $Ga, \Gamma' \Rightarrow \Delta'$
fst (*pn-atoms Ga*) ⊆ (*patoms-mset* (G, H′) ∪ *natoms-mset* Δ) ∩ (*natoms-mset* Γ′ ∪ *patoms-mset* Δ′)
snd (*pn-atoms Ga*) ⊆ (*natoms-mset* (G, H′) ∪ *patoms-mset* Δ) ∩ (*patoms-mset* Γ′ ∪ *natoms-mset* Δ′) *is-mini-formula Ga* **by** *blast*
let *?w* = *Fa* → *Ga*
show *?thesis* **proof**(*intro exI*[**where** *x*=*?w*] *conjI*)
from *IH1*(*1*) *IH2*(*2*) **show** *?w*, $\Gamma' \Rightarrow \Delta'$
by (*simp add*: *SCp.ImpL*)
from *IH1*(*2*) *IH2*(*1*) **show** Γ ⇒ *?w*, Δ **unfolding** Γ
by(*intro SCp.ImpL inR1*[*OF SCp.ImpR*] *SCp.ImpR*) (*simp-all add*: *weakenR weakenL*)
show *fst* (*pn-atoms ?w*) ⊆ (*patoms-mset* Γ ∪ *natoms-mset* Δ) ∩ (*natoms-mset* Γ′ ∪ *patoms-mset* Δ′)
snd (*pn-atoms ?w*) ⊆ (*natoms-mset* Γ ∪ *patoms-mset* Δ) ∩ (*patoms-mset* Γ′ ∪ *natoms-mset* Δ′)
using *IH1*(*3*−) *IH2*(*3*−) **unfolding** Γ **by** (*auto simp*: *prod-unions-def split*: *prod.splits*)
show *is-mini-formula ?w* **using** ‹*is-mini-formula Fa*› ‹*is-mini-formula Ga*› **by** *simp*
qed
next
fix *I′* **assume** Γ′: $\Gamma' = F \rightarrow G, I'$ **note** *ImpL*(*5*)[*unfolded* Γ′]
from Γ′ **have** ∗: $\Gamma'' = \Gamma + I'$ $F, \Delta + \Delta' = \Delta + (F,\Delta')$
using *ImpL*(*5*−) **by**(*simp-all add*: *union-assoc add-ac*(*2*,*3*))
hence *is-mini-mset* $(\Gamma + I' + \Delta + (F, \Delta'))$ **using** *ImpL*(*6*) **by**(*auto simp add*: Γ′)
from *ImpL*(*2*)[*OF* ∗ *this*] **obtain** *Fa*
where *IH1*: $\Gamma \Rightarrow Fa, \Delta$ $Fa, I' \Rightarrow F, \Delta'$ *is-mini-formula Fa*
fst (*pn-atoms Fa*) ⊆ (*patoms-mset* Γ ∪ *natoms-mset* Δ) ∩ (*natoms-mset* I′ ∪ *patoms-mset* (F, Δ′))
snd (*pn-atoms Fa*) ⊆ (*natoms-mset* Γ ∪ *patoms-mset* Δ) ∩ (*patoms-mset* I′ ∪ *natoms-mset* (F, Δ′)) **by** *blast*
from Γ′ **have** ∗: $G, \Gamma'' = (G, I') + \Gamma$ $\Delta + \Delta' = \Delta' + \Delta$
using *ImpL*(*5*) **by** (*simp-all add*: *add-ac* ‹$\Gamma'' = \Gamma + I'$›)
have *is-mini-mset* $((G, I') + \Gamma + \Delta' + \Delta)$ **using** *ImpL*(*6*) **by**(*auto simp add*: Γ′)

from *ImpL*(*4*)[*OF* ∗ *this*] **obtain** *Ga*
where *IH2*: G, I' ⇒ *Ga*, Δ' *Ga*, $\Gamma \Rightarrow \Delta$ *is-mini-formula Ga*
fst (*pn-atoms Ga*) ⊆ (*patoms-mset* (G, I') ∪ *natoms-mset* Δ') ∩ (*natoms-mset* Γ ∪ *patoms-mset* Δ)
snd (*pn-atoms Ga*) ⊆ (*natoms-mset* (G, I') ∪ *patoms-mset* Δ') ∩ (*patoms-mset* Γ ∪ *natoms-mset* Δ) **by** *blast*
let *?w* = (*Fa* → *Ga*) → ⊥
have *?w* = *to-mini-formula* (¬(*Fa* → *Ga*)) **unfolding** *to-mini-formula.simps* *mini-to-mini*[*OF IH1*(*3*)] *mini-to-mini*[*OF IH2*(*3*)] **by** (*simp add*: *IH1*(*3*) *IH2*(*3*))
show *?thesis* **proof**(*intro exI*[*of* - *?w*] *conjI*)
from *IH1*(*1*) *IH2*(*2*) **show** $\Gamma \Rightarrow$ *?w*, Δ
by(*intro SCp.ImpR SCp.ImpL*) (*simp-all add*: *inR1 weakenR BotL-canonical*)
from *IH1*(*2*) *IH2*(*1*) **show** *?w*, $\Gamma' \Rightarrow \Delta'$ **unfolding** Γ'
by(*blast intro*!: *SCp.ImpL SCp.ImpR dest*: *weakenL weakenR*)+
show *fst* (*pn-atoms ?w*) ⊆ (*patoms-mset* Γ ∪ *natoms-mset* Δ) ∩ (*natoms-mset* Γ' ∪ *patoms-mset* Δ')
snd (*pn-atoms ?w*) ⊆ (*natoms-mset* Γ ∪ *patoms-mset* Δ) ∩ (*patoms-mset* Γ' ∪ *natoms-mset* Δ')
using *IH1*(*4*−) *IH2*(*4*−) **unfolding** Γ' **by** (*auto simp*: *prod-unions-def* *split*: *prod.splits*)
show *is-mini-formula ?w* **using** ‹*is-mini-formula Fa*› ‹*is-mini-formula Ga*› **by** *simp*
qed
qed
next
qed (*metis add.commute is-mini-formula.simps union-iff union-single-eq-member*)+
define *tms* :: *'a formula multiset* ⇒ *'a formula multiset*
where *tms* = *image-mset to-mini-formula*
have [*simp*]: *tms* (A + B) = *tms* A + *tms* B *tms* {#*F*#} = {#*to-mini-formula* *F*#} **for** A B F **unfolding** *tms-def* **by** *simp-all*
have *imm*: *is-mini-mset* (*tms* Γ + *tms* Γ' + *tms* Δ + *tms* Δ') **unfolding** *tms-def* **by** *auto*
from *assms* **have** *tms* Γ + *tms* Γ' ⇒ *tms* Δ + *tms* Δ' **unfolding** *tms-def* **using** *SC-full-to-mini* **by** *force*
from *mini*[*OF this imm*] **obtain** F **where** *hp*:
tms Γ ⇒ F, *tms* Δ F, *tms* Γ' ⇒ *tms* Δ'
and *su*: *fst* (*pn-atoms* F) ⊆ (*patoms-mset* (*tms* Γ) ∪ *natoms-mset* (*tms* Δ)) ∩ (*natoms-mset* (*tms* Γ') ∪ *patoms-mset* (*tms* Δ'))
snd (*pn-atoms* F) ⊆ (*natoms-mset* (*tms* Γ) ∪ *patoms-mset* (*tms* Δ)) ∩ (*patoms-mset* (*tms* Γ') ∪ *natoms-mset* (*tms* Δ'))
and *mf*: *is-mini-formula* F **by** *blast*
from *hp mf* **have** *tms* Γ ⇒ *tms* (F, Δ) *tms* (F, Γ') ⇒ *tms* Δ' **using** *mini-to-mini*[**where** *'a*=*'a*] **unfolding** *tms-def* **by** *simp-all*
hence ∗: $\Gamma \Rightarrow F$, Δ F, $\Gamma' \Rightarrow \Delta'$ **using** *SC-mini-to-full* **unfolding** *tms-def* **by** *blast*+
have *pn-atoms* (*to-mini-formula* F) = *pn-atoms* F **for** F :: *'a formula* **by**(*induction* *F*; *simp add*: *prod-unions-def split*: *prod.splits*)
hence *pn-tms*: *patoms-mset* (*tms* Γ) = *patoms-mset* Γ *natoms-mset* (*tms* Γ) =

natoms-mset Γ **for** Γ **unfolding** *tms-def* **by** *simp-all*
  from *su*[*unfolded pn-tms*] **show** *?thesis* **using** *that*[*of F*, *OF* ∗ - -] **by** *auto*
qed

end

References

[1] W. Craig. Linear reasoning. A new form of the Herbrand-Gentzen theorem. *The Journal of Symbolic Logic*, 22(03):250–268, 1957.

[2] M. Fitting. *First-Order Logic and Automated Theorem Proving*. Graduate Texts in Computer Science. Springer, 1990.

[3] J. H. Gallier. *Logic for computer science: foundations of automatic theorem proving*. Courier Dover Publications, 2015.

[4] G. Gentzen. Untersuchungen über das logische Schließen. I. *Mathematische zeitschrift*, 39(1):176–210, 1935.

[5] J. Harrison. *Handbook of practical logic and automated reasoning*. Cambridge University Press, 2009.

[6] D. Hilbert. Die Grundlagen der Mathematik. In *Die Grundlagen der Mathematik*, pages 1–21. Springer, 1928.

[7] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge university press, 2004.

[8] I. Johansson. Der Minimalkalkül, ein reduzierter intuitionistischer Formalismus. *Compositio mathematica*, 4:119–136, 1937.

[9] U. Schöning. *Logik für Informatiker*. BI Wissenschaftsverlag Mannheim, 1987.

[10] R. M. Smullyan. A unifying principal in quantification theory. *Proceedings of the National Academy of Sciences*, 49(6):828–832, 1963.

[11] A. S. Troelstra and H. Schwichtenberg. *Basic proof theory*. Number 43 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2000.