Propositional Calculus in Coq

Floris van Doorn

May 9, 2014

Abstract

I formalize important theorems about classical propositional logic in the proof assistant Coq. The main theorems I prove are (1) the soundness and completeness of natural deduction calculus, (2) the equivalence between natural deduction calculus, Hilbert systems and sequent calculus and (3) cut elimination for sequent calculus.

1 Introduction

Proof assistants (or interactive theorem provers) are computer programs which help to formalize and check the correctness of proofs. Proof assistants are used for two things. First, they are used for formal verification of hardware or software, which means proving that the hardware or software conforms to the specifications. Second, they are used to formally verify mathematical theorems. I will only discuss the second use in this paper.
To prove a mathematical theorem in a proof assistant, the user has to write down the proof in a language that the proof assistant can parse, and then the proof assistant will check correctness. Usually the proof assistant provides small amounts of automation, so that the user does not have to write down every detail. Still, a user typically has to give a lot more details to a proof assistant than is usually given on a paper proof. This means that a formalization of a mathematical theorem is much more time consuming than proving the theorem on paper, hence most research mathematics is not formalized. Still, there are some deep theorems which have been fully formalized in proof assistants, such as the four color theorem[2] or Feit-Thompson's odd order theorem[3], which were both formalised by Georges Gonthier in the proof assistant Coq. It is very rare that a new result is immediately accompanied by a formalization in a proof assistant, the author knows only two instances where this has been done.[6,7]
The goal of this project is to formalize some theorems in proof theory in a proof assistant. I chose to use the proof assistant Coq[4] for this. Coq is a proof assistant based on the predicative Calculus of Inductive Constructions, which uses the expressive power of dependent type theory as language. It uses constructive reasoning, which is very suitable for the meta-logic of proof theory, since most proofs can be given constrictively.
I have formalized the following three theorems.

1. The completeness theorem of classical natural deduction calculus.
2. The equivalence between the classical versions of natural deduction calculus (Nc), Hilbert-style deduction calculus (Hc) and Gentzen's sequent calculus (Gc).

3. Cut elimination for Gentzen's sequent calculus.

To simplify, I only proved this for propositional calculus. Formalizing full predicate calculus gives rise to additional difficulties such as variable encodings, variable renaming, alpha-equivalence and induction over type families such as vectors.
The following table is a summary of the formalization. The number of lines in each file is rounded.

file	**Defines**	**Proves**	**# lines**
`a_base`	Variable	–	20
`b_soundness`	Provability in Nc, validity	Soundness of Nc.	150
`c_completeness`	Conjunctive Normal Form	Completeness of Nc.	400
`d_hilbert_calculus`	Provability in Hc	Equivalence of Nc and Hc.	100
`e_sequent_calculus`	Provability in Gc	Equivalence of Nc and Gc.	200
`f_cut_elimination`	Cut-free sequent calculus	Cut elimination.	180

In total there are just over 1000 lines, 66 theorems and 43 definitions.
In the next sections I will discuss the formalization in more detail. I will focus on the considerations which went into the formalization. I will skip all proofs and most Lemmas, but I will give informal proof sketches for the main theorems. To view all Lemmas, check the coqdoc html files. To view all proofs, check the Coq source files.

2 Basic Definitions

In `a_base` I do some preparatory work, by defining variables. The relevant lines are

```
Parameter PropVars : Set.
Hypothesis Varseq_dec : ∀ x y:PropVars, {x = y} + {x ≠ y}.
```

The first line indicates that the set of (propositional) variables is some set. The second line indicates that equality is decidable on this set. Note that since Coq is based on constructive logic, so this is not vacuous. This hypothesis is necessary for the completeness theorem. If we did not assume this, then one could take for example the constructive reals as variables. It is consistent with the constructive reals that all valuations (functions from variables to bool) are constant.[1] This means that it is possible that $\#0 \vee \#1$ (we denote variables with '#') is valid, but it is not provable. So in this case completeness would fail. These are the only axioms or parameters in the formalization. All results work even if there are only finitely many variables, even if there is just one variable, or even none at all (though that wouldn't be a very interesting case).
We can now define formulas in `b_soundness`.

```
Inductive PropF : Set :=
 | Var : PropVars → PropF
 | Bot : PropF
 | Conj : PropF → PropF → PropF
 | Disj : PropF → PropF → PropF
 | Impl : PropF → PropF → PropF
.

Notation "# P" := (Var P) (at level 1) : My_scope.
Notation "A ∨ B" := (Disj A B) (at level 15, right associativity) : My_scope.
Notation "A ∧ B" := (Conj A B) (at level 15, right associativity) : My_scope.
Notation "A ⇀ B" := (Impl A B) (at level 16, right associativity) : My_scope.
Notation "⊥" := Bot (at level 0) : My_scope.
Definition Neg A := A ⇀ ⊥.
Notation "¬ A" := (Neg A) (at level 5) : My_scope.
Definition Top := ¬⊥.
Notation "⊤" := Top (at level 0) : My_scope.
```

In the formalization I used the unicode character → for implication, but to distinguish it from the function type of Coq, I use ⇀ in this document instead.
I use defined negation, because that simplifies induction proofs by having one fewer connective. Other than this, I take all connectives as primitive. I could have defined conjunction and disjunction in terms of implication and falsum and retain an equivalent system, since I only formalize classical logic. However, I decided against this, to make the formalization easier to adapt to intuitionistic or minimal logic, where these connectives aren't interdefinable. We defined negation because it is definable in all these calculi.
In an earlier version of the formalization I used a special variable ⊥ which corresponds to the false formula. This had as advantages that the variables corresponded exactly to the atomic formulae and that there is one fewer induction base case when proving things by induction over formulae. In the end I decided against it, since it is more natural to have ⊥ not as a variable, but as a separate constant, and the alternative required to define valuations as assigning 'false' to the variable ⊥, while no such requirement is necessary now.
A *valuation* is an element of the space *PropVars* → **bool**. I define the truth of a formula A under valuation v in the obvious way. This allows us to define the validity of formulae. The map Is_true sends a boolean value to the corresponding (true or false) proposition. [] denotes the empty list.

```
Fixpoint TrueQ v A : bool := match A with
 | # P ⇒ v P
 | ⊥ ⇒ false
 | B ∨ C ⇒ (TrueQ v B) || (TrueQ v C)
 | B ∧ C ⇒ (TrueQ v B) && (TrueQ v C)
 | B ⇀ C ⇒ (negb (TrueQ v B)) || (TrueQ v C)
end.
Definition Satisfies v Γ := ∀ A, In A Γ → Is_true (TrueQ v A).
Definition Models Γ A := ∀ v,Satisfies v Γ→Is_true (TrueQ v A).
Notation "Γ ⊨ A" := (Models Γ A) (at level 80).
Definition Valid A := [] ⊨ A.
```

The notion of provability is naturally defined inductively. I use a context sharing version of classical natural deduction. Using the context sharing version simplifies proofs, because changes in the context are usually hard to deal with for proofs of meta-theoretic theorems. Note that I am using lists, not sets, for the context, because it is easier to reason about lists in proofs.

```
Reserved Notation "Γ ⊢ A" (at level 80).
Inductive Nc : list PropF→ PropF→Prop :=
| Nax : ∀ Γ A , In A Γ → Γ ⊢ A
| ImpI : ∀ Γ A B, A::Γ ⊢ B → Γ ⊢ A ⇀ B
| ImpE : ∀ Γ A B, Γ ⊢ A ⇀ B → Γ ⊢ A → Γ ⊢ B
| BotC : ∀ Γ A , ¬A::Γ ⊢ ⊥ → Γ ⊢ A
| AndI : ∀ Γ A B, Γ ⊢ A → Γ ⊢ B → Γ ⊢ A∧B
| AndE1 : ∀ Γ A B, Γ ⊢ A∧B → Γ ⊢ A
| AndE2 : ∀ Γ A B, Γ ⊢ A∧B → Γ ⊢ B
| OrI1 : ∀ Γ A B, Γ ⊢ A → Γ ⊢ A∨B
| OrI2 : ∀ Γ A B, Γ ⊢ B → Γ ⊢ A∨B
| OrE : ∀ Γ A B C, Γ ⊢ A∨B → A::Γ ⊢ C → B::Γ ⊢ C → Γ ⊢ C
where "Γ ⊢ A" := (Nc Γ A) : My_scope.

Definition Provable A := [] ⊢ A.
```

This allows us to define the propositions we're aiming to prove.

```
Definition Prop_Soundness := ∀ A,Provable A→Valid A.
Definition Prop_Completeness := ∀ A,Valid A→Provable A.
```

Given those definitions, Soundness can be proved directly by induction on the derivation (no additional lemmas are needed). Indeed, in the formalization this only requires 17 lines.[1]

```
Theorem Soundness : Prop_Soundness.
```

3 Completeness

The formalization continues to prove Completeness. I will first give a proof sketch, before I turn my attention to the formalization.

3.1 Proof Sketch

The formalization uses the following proof. Recall that a literal is an atomic formula or the negation of an atomic formula, a clause is a disjunction of literals, and that a formula is in conjunctive normal form (CNF) if it's a conjunction of clauses. Also, a formula is in *negation normal form* (NNF), if it only consists of conjunctions, disjunctions and literals. In this paper I will use CNF and NNF as nouns indicating formulae in CNF resp. NNF.

Proof sketch of Completeness. Let A be a formula, A_{NNF} its negation normal form and A_{CNF} its conjunctive normal form. We say that a clause is *syntactically invalid* if either it contains both p and $\neg p$ for some atomic formula p, or if it contains $\neg\bot$. We say that A_{CNF} is *syntactically valid* if it contains no syntactically invalid clauses. Then Completeness follows from the following statements

(1) If A is valid, then A_{NNF} is valid.

(2) If A_{NNF} is valid, then A_{CNF} is valid.

(3) If A_{CNF} is valid then A_{CNF} is syntactically valid.

(4) If A_{CNF} is syntactically valid then A_{CNF} is provable.

(5) $A_{\text{CNF}} \to A_{\text{NNF}}$ is provable.

(6) $A_{\text{NNF}} \to A$ is provable.

All statements except (3) are proven by induction to the structure of the formula in question. Many of these proofs need additional lemmas, which are also proven by induction. Statement (3) is most naturally proved using contraposition, by showing that if A_{CNF} is syntactically invalid, then there is a countervaluation v for it. First one does this for a clause. The countervaluation v of a syntactically invalid clause C can be defined as $v(x) = \text{true}$ iff $\neg x$ occurs in C. One can then

[1] Counting the lines for theorems Soundness_general and Soundness and for tactics *case_bool* and *prove_satisfaction*. No other lemmas or tactics are used.

show that v is indeed a valuation, and that v makes C false. Then if A_{CNF} is syntactically invalid, it contains some syntactically invalid clause C, and the countervaluation for that clause is also a countervaluation for A_{CNF}. □

3.2 Considerations for the formalization

In the formalization I defined clauses as lists of literals and CNFs (formulae in CNF) as lists of clauses. The alternative, to say that only some class of propositional formulae are in CNF, will be very hard to work with. One of the problems which one has to deal with is associativity. Which of the following do you call clauses?

$$(((x \vee \neg y) \vee z) \vee w \qquad (x \vee (\neg y \vee (z \vee w))) \qquad (x \vee \neg y) \vee (z \vee w).$$

You can call all of them clauses, but then clauses are not in a canonical form, and proving theorems about them will probably be harder. Also, induction proofs will be harder, because you're constantly proving or using the fact that some formula is in the shape of a CNF. Using lists, you can make sure that all clauses have the same shape of parentheses, and a term of the correct type is automatically in CNF. The disadvantage, that you have to define a mapping from clauses to actual formulas, is very minor.
A viable alternative to using lists would be using non-empty lists, i.e. lists which have at least one element. When transforming a formula to conjunctive normal form, it turns out we never end up with an empty clause (i.e. an empty list of literals) or an empty formula in CNF (i.e. an empty list of clauses). Using non-empty lists might be more natural in that case, but I decided against this, because I thought this would make meta-theorems harder to prove. With non-empty lists, the base case is a list of one element, and that case is probably harder than the case of an empty list. However, allowing for empty lists gives an unexpected artefact in the translation from lists to formulae. When transforming a clause (a list of literals) to a formulae, it is natural to say $f(\mathsf{nil}) = \bot$ (nil is the empty list) and $f(a::l) = a \vee f(l)$. However, this means that, for example, $f([x;y;z]) = x \vee (y \vee (z \vee \bot))$ instead of the more natural $f([x;y;z]) = x \vee (y \vee z)$. This poses no problems except that it is unnatural.

3.3 Definitions

The definitions needed for Completeness are given below, given in `c_completeness`.

```
Inductive NNF : Set :=
 | NPos : PropVars → NNF
 | NNeg : PropVars → NNF
 | NBot : NNF
 | NTop : NNF
 | NConj : NNF → NNF → NNF
 | NDisj : NNF → NNF → NNF
Fixpoint MakeNNF (A:PropF) : NNF := match A with
 | # P ⇒ NPos P
 | ⊥ ⇒ NBot
 | B ∨ C ⇒ NDisj (MakeNNF B) (MakeNNF C)
 | B ∧ C ⇒ NConj (MakeNNF B) (MakeNNF C)
 | B → C ⇒ NDisj (MakeNNFN B) (MakeNNF C)
 end
with MakeNNFN (A:PropF) : NNF := match A with
 | # P ⇒ NNeg P
 | ⊥ ⇒ NTop
 | B ∨ C ⇒ NConj (MakeNNFN B) (MakeNNFN C)
 | B ∧ C ⇒ NDisj (MakeNNFN B) (MakeNNFN C)
 | B ⇀ C ⇒ NConj (MakeNNF B) (MakeNNFN C)
 end.
Inductive Literal :=
| LPos : PropVars → Literal
| LNeg : PropVars → Literal
| LBot : Literal
| LTop : Literal
.
```

The inclusions NNFtoPropF and LiteraltoPropF from respectively NNF and literals to propositional formulae are defined in the obvious way.
In a_base I define the constructor map_fold_right, such that for $f : B \to A$, $g : A \to A \to A$, $a : A$ and $x_i : B$ we have

$$\mathsf{map\_fold\_right}\, f\, g\, a\, [x_0; \cdots ; x_n] = g(f(x_0), g(f(x_1), \cdots g(f(x_n), a) \cdots).$$

The definition is below, and allows us to define the inclusion from Clause and CNF to PropF.

```
Fixpoint map_fold_right (A B:Type) (f : B → A) (g : A → A → A) a l := match l with
 | nil ⇒ a
 | b::l2 ⇒ g (f b) (map_fold_right f g a l2)
end.
Definition Clause := list Literal.
Definition ClausetoPropF := map_fold_right LiteraltoPropF Disj ⊥.
Definition CNF := list Clause.
Definition CNFtoPropF := map_fold_right ClausetoPropF Conj ⊤.
```

We still need to define the map which transforms a NNF to a CNF. The conjunction of two CNFs is just concatenation of the corresponding lists. However, the disjunction is harder to define. Doing first the simpler case of taking the disjunction of a clause with a CNF, we see that this corresponds to just adding this clause in front of every other clause:

$$C \vee (C_1 \wedge C_2 \wedge \cdots) \rightsquigarrow (C \vee C_1) \wedge (C \vee C_2) \wedge \cdots ,$$

see AddClause below ($\mathsf{map}\, f\, l$ means applying f to every element of l). We can then define the disjunction of two CNFs as follows:

$$(C_1 \wedge C_2 \wedge \cdots) \vee A \rightsquigarrow (C_1 \vee A) \wedge (C_2 \vee A) \wedge \cdots$$

where A is an CNF, and where $C_1 \vee A$ is defined using AddClause. This gives rise to the definition of Disjunct below. The term flat_map is defined in the Coq library such that if $f : A \to \mathbf{list} B$ and $x_i : A$ then

$$\mathsf{flat\_map}\, f\, [x_0; \cdots ; x_n] = f(x_0) \mathbin{++} f(x_1) \mathbin{++} \cdots \mathbin{++} f(x_n).$$

This allows us to define the transformation from NNF to CNF.

```
Definition AddClause (l:Clause) (ll:CNF) : CNF := map (fun l2 ⇒ l++l2) ll.
Definition Disjunct (ll ll2:CNF) : CNF := flat_map (fun l ⇒ AddClause l ll2) ll.

Fixpoint MakeCNF (A:NNF) : CNF := match A with
 | NPos P ⇒ [[LPos P]]
 | NNeg P ⇒ [[LNeg P]]
 | NBot ⇒ [[LBot]]
 | NTop ⇒ [[LTop]]
 | NConj B C ⇒ MakeCNF B ++ MakeCNF C
 | NDisj B C ⇒ Disjunct (MakeCNF B) (MakeCNF C)
 end.
```

Finally, we can define the syntactical validity.

```
Definition Valid_Clause (l:Clause) := In LTop l∨∃ A,(In (LPos A) l∧In (LNeg A) l).
Definition Valid_CNF ll := ∀ l, In l ll→Valid_Clause l.
```

3.4 Formalized Proof

We now turn to prove the 6 statements described in Section 3.1.

```
Lemma NNF_equiv_valid : ∀ v A, TrueQ v (NNFtoPropF (MakeNNF A))=TrueQ v A ∧
                               TrueQ v (NNFtoPropF (MakeNNFN A))=TrueQ v ¬A.
Theorem CNF_equiv_valid : ∀ v A, TrueQ v (CNFtoPropF (MakeCNF A)) = TrueQ v (NNFtoPropF A).
Theorem CNF_valid : ∀ ll, Valid (CNFtoPropF ll) → Valid_CNF ll.
Theorem CNF_provable : ∀ ll, Valid_CNF ll → Provable (CNFtoPropF ll).
Theorem CNF_impl_prov : ∀ A, Provable (CNFtoPropF (MakeCNF A) ⇀ NNFtoPropF A).
Lemma NNF_impl_prov : ∀ A, Provable (NNFtoPropF (MakeNNF A) ⇀ A) ∧
                          Provable (NNFtoPropF (MakeNNFN A) ⇀ ¬A).
```

In a previous version I proved CNF_valid with contraposition as described above. In that version I needed to first prove that syntactical validity was decidable. In the latest version I prove this directly, by showing that a clause is valid if it is valid under the countervaluation for it (which assigns true to a variable p iff $\neg p$ occurs in the clause).
This allows us to prove completeness:

```
Theorem Completeness : Prop_Completeness.
```

4 Equivalence between the calculi

We now turn to formalizing the equivalence between the natural deduction calculus Nc, Hilbert-style calculus Hc and Gentzen's sequent calculus Gc.

4.1 Nc and Hc

In `d_hilbert_calculus` I define Hilbert-style calculus. Natural deduction calculus and Hilbert-style calculus are quite similar. The difference is that in Hilbert-style calculus we have axioms, which allows us to only have implication elimination (modus ponens) as inference rule. The Hilbert-style calculus is defined below.

```
Inductive AxiomH : PropF → Prop :=
| HOrI1 : ∀ A B , AxiomH (A ⇀ A∨B)
| HOrI2 : ∀ A B , AxiomH (B ⇀ A∨B)
| HAndI : ∀ A B , AxiomH (A ⇀ B ⇀ A∧B)
| HOrE : ∀ A B C, AxiomH (A∨B ⇀ (A ⇀ C) ⇀ (B ⇀ C) ⇀ C)
| HAndE1 : ∀ A B , AxiomH (A∧B ⇀ A)
| HAndE2 : ∀ A B , AxiomH (A∧B ⇀ B)
| HS : ∀ A B C, AxiomH ((A ⇀ B ⇀ C) ⇀ (A ⇀ B) ⇀ A ⇀ C)
| HK : ∀ A B , AxiomH (A ⇀ B ⇀ A)
| HClas : ∀ A , AxiomH (¬(¬A) ⇀ A)
.
Inductive Hc : list PropF→ PropF→Prop :=
| Hass : ∀ A Γ, In A Γ → Γ ⊢H A
| Hax : ∀ A Γ, AxiomH A → Γ ⊢H A
| HImpE : ∀ Γ A B, Γ ⊢H A ⇀ B → Γ ⊢H A → Γ ⊢H B
where "Γ ⊢H A" := (Hc Γ A) : My_scope.
```

I then prove the equivalence of these systems.

```
Theorem Nc_equiv_Hc : ∀ Γ A, Γ ⊢ A ↔ Γ ⊢H A.
```

The proof is not hard. In the direction from left to right, one basically needs to prove that all axioms for Hc are provable in Nc. In the converse direction, one needs to prove the Deduction Theorem for Hc, which states that

$$\Gamma, A \vdash_{\mathrm{Hc}} B \iff \Gamma \vdash_{\mathrm{Hc}} A \to B.$$

4.2 Nc and Gc

In `e_sequent_calculus` I define Gentzen's sequent calculus. Gentzen's sequent calculus is quite different than natural deduction. In sequent calculus the propositions are *sequents* of the form $\Gamma \vdash \Delta$, where Γ and Δ are either sequences, multisets or sets of formulae. A sequent intuitively means that the conjunction of Γ implies the disjunction of Δ. Instead of having introduction and elimination rules as in natural deduction, there are left rules and right rules for each connective, which introduce that connective on that side of the sequent.
I define the sequent calculus in the formalization below. Since lists are easiest to work with, I use them in sequents. Usually when lists are used in sequent calculus, there are some structural rules allowing to move formulas around in the list. However, having these rules make induction proofs very hard, and requires more nonstructural induction proofs, so instead I defined the sequent calculus in such a way that it is possible to apply a rule anywhere in the lists (ordinarily the relevant formulas appear only on one end of the lists). I use the symbol $\supset$ as separator in a sequent.

```
Inductive G : list PropF→list PropF→Prop :=
| Gax : ∀ A Γ Δ , In A Γ → In A Δ → Γ ⊃ Δ
| GBot : ∀ Γ Δ , In ⊥ Γ → Γ ⊃ Δ
| AndL : ∀ A B Γ1 Γ2 Δ, Γ1++A::B::Γ2 ⊃ Δ → Γ1++A∧B::Γ2 ⊃ Δ
| AndR : ∀ A B Γ Δ1 Δ2, Γ ⊃ Δ1++A::Δ2 → Γ ⊃ Δ1++B::Δ2 → Γ ⊃ Δ1++A∧B::Δ2
| OrL : ∀ A B Γ1 Γ2 Δ, Γ1++A::Γ2 ⊃ Δ → Γ1++B::Γ2 ⊃ Δ → Γ1++A∨B::Γ2 ⊃ Δ
| OrR : ∀ A B Γ Δ1 Δ2, Γ ⊃ Δ1++A::B::Δ2 → Γ ⊃ Δ1++A∨B::Δ2
| ImpL : ∀ A B Γ1 Γ2 Δ, Γ1++B::Γ2 ⊃ Δ → Γ1++Γ2 ⊃ A::Δ → Γ1++A⇀B::Γ2 ⊃ Δ
| ImpR : ∀ A B Γ Δ1 Δ2, A::Γ ⊃ Δ1++B::Δ2 → Γ ⊃ Δ1++A⇀B::Δ2
| Cut : ∀ A Γ Δ , Γ ⊃ A::Δ → A::Γ ⊃ Δ → Γ ⊃ Δ
where "Γ ⊃ Δ" := (G Γ Δ) : My_scope.
```

We now turn to proving the equivalence between Nc and Gc. Neither direction is easy.

In the direction from Gc to Nc we prove that

$$\Gamma \supset \Delta \implies \Gamma \vdash \bigvee \Delta,$$

where $\bigvee \Delta$ is the disjunction of the formulae in Δ (similar to a clause, except that the entries can now be any formulae instead of only literals). However, proving this directly is very hard, since in the definition of sequents we make changes in the middle of the disjunction. This means that in every step we have to use multiple or-eliminations and then or-introductions, which gets really ugly. Instead, we prove the following intermediate lemma:

$$\Gamma \supset \Delta \implies \Gamma, \neg\Delta \vdash \bot,$$

where $\neg\Delta$ is the element-wise negation of formulae in Δ. In the formalization this looks as follows.

```
Definition BigOr := fold_right Disj ⊥.
Notation "⋁ Δ" := (BigOr Δ) (at level 19).
Notation "¬l Γ" := (map Neg Γ) (at level 40).
Lemma G_to_Nc_Neg : ∀ Γ Δ, Γ ⊃ Δ → Γ++¬l Δ ⊢ ⊥.
Theorem G_to_Nc : ∀ Γ Δ, Γ ⊃ Δ → Γ ⊢ ⋁Δ.
```

In the direction from Gc to Nc the major difficulty is proving weakening for Gc. On paper proving weakening is not very hard, but in the formalization the fact that we can apply rules anywhere in the list, and the fact that the weakening also occurs anywhere in the list, makes it quite hard to prove this (for example, we need to distinguish cases in every induction step which of the two formulae occurs earlier in the list).

```
Lemma WeakL : ∀ Γ1 Γ2 Δ A, Γ1++Γ2 ⊃ Δ → Γ1++A::Γ2 ⊃ Δ.
Lemma WeakR : ∀ Γ Δ1 Δ2 A, Γ ⊃ Δ1++Δ2 → Γ ⊃ Δ1++A::Δ2.
Theorem Nc_to_G : ∀ Γ A, Γ ⊢ A → Γ ⊃ [A].
```

5 Cut Elimination

In `f_cut_elimination` I prove the cut elimination theorem. I first introduce the cut-free sequent calculus, which has no cut rule. To strengthen the cut elimination a bit, I only allow the axiom rule for atomic formulae in the cut-free calculus. I denote the cut-free sequents as $\Gamma \supset\mathsf{c}\, \Delta$ (I don't give the definition here; it is very similar to the definition **G**).

I give a semantic proof for cut elimination. This proof does not generalize easily to predicate calculus, the semantic proof for predicate calculus is way more involved. I also tried to give a syntactical proof, but failed due to time contraints.

The semantic proof goes as follows. First we prove soundness of the sequent calculus with cut, which states that the conclusion of any derivation (with cuts) is valid. The main part is to prove completeness of the cut-free calculus. Suppose that we are given a valid sequent. If no logical connectives occur in the sequent, then we prove that we can either apply the axiom rule or the falsum rule, using that the sequent is valid. On the other hand, if any connective occurs in the sequent, we apply the corresponding rule (in reverse) to eliminate that connective. We can then show the hypotheses to the rule are also valid, and they contain fewer connectives than the conclusion. So we can use a nonstructural induction to conclude that the hypotheses are provable, which finishes the proof.

We need the following definitions. The first line defines the notation for a valid sequent. Then we define the size of a formula and sequent, which in this case is the number of connectives occurring in it.

```
Definition Validates v Δ := ∃ A, In A Δ ∧ Is_true (TrueQ v A).
Notation "Γ =⊃ Δ" := (∀ v,Satisfies v Γ→Validates v Δ) (at level 80).
Fixpoint size A : nat := match A with
  | # P ⇒ 0
  | ⊥ ⇒ 0
  | B ∨ C ⇒ S (size B + size C)
  | B ∧ C ⇒ S (size B + size C)
  | B ⇀ C ⇒ S (size B + size C)
end.
Definition sizel := map_fold_right size plus 0.
Definition sizes Γ Δ:= sizel Γ + sizel Δ.
```

This allows us to prove the following theorem by induction to n.

```
Theorem Gcf_complete_induction : ∀ n Γ Δ, sizes Γ Δ ≤ n → Γ =⊃ Δ → Γ ⊃c Δ.
```

Together with Soundness this proves cut elimination.

```
Theorem Cut_elimination : ∀ Γ Δ, Γ ⊃ Δ → Γ ⊃c Δ.
```

6 Conclusion and Future work

In this project I have shown that the proofs of basic but important theorems about propositional calculus can be implemented in a proof assistant relatively painlessly. Since many concepts in proof theory are naturally definable using induction, Coq's excellent support for inductive datatypes and proofs by induction makes the formalization of this subject relatively easy. The lack of strong automation in Coq does require that the user has to focus a lot on details in the proof.
This project can be seen as a proof of concept for formalizing theorems in proof theory. One can extend this project in numerous ways. One can consider nonclassical logics, and prove soundness and completeness results about them, like the completeness intuitionistic logic w.r.t. Kripke models. Another way to extend this is to consider predicate calculus instead of propositional calculus. This has been done in the literature already, for example to prove Gödel's first incompleteness theorem.[5] Another possible extension is to prove other theorems, like the Beth definability theorem or Craig's interpolation theorem.

References

[1] Matt Bishop. Foundations of constructive analysis. 1967.

[2] Georges Gonthier. A computer-checked proof of the four colour theorem. 2005.

[3] Georges Gonthier. Engineering Mathematics: The Odd Order Theorem Proof. *SIGPLAN Not.*, 48(1):1–2, January 2013.

[4] The Coq development team. *The Coq proof assistant reference manual.* LogiCal Project, 2013. Version 8.4.

[5] Russell O'Connor. Essential Incompleteness of Arithmetic Verified by Coq. 2005.

[6] Vincent Siles and Hugo Herbelin. Pure Type System conversion is always typable. *Journal of Functional Programming*, 22(2):153 – 180, May 2012.

[7] Floris van Doorn, Herman Geuvers, and Freek Wiedijk. Explicit Convertibility Proofs in Pure Type Systems. LFMTP '13, pages 25–36. ACM, 2013.