Propositional Logic in Coq: a tutorial

Rodrigo Machado Karina Girardi Roggia

April 29, 2020

Abstract

This is a tutorial on using the Coq Proof Assistant to formalize the complete meta-theory of the classical propositional logic.

Contents

Chapter 1

Introduction

Logic, as in the study of the human reasoning, is a milenar tradition, going back to Aristotle in the ancient Greece. Formal logic refers to the mathematical treatment of reasoning, and by itself has been developed progressively since the 19th century following thinkers such as Boole, deMorgan, Hilbert, Frege, Russel, Gentzen, Gödel, among others. Formal logic is one of the pillars from which computer science arose, being directed related to topics such as theory of computation and artificial inteligence. Due to its importance, almost all curriculae of computer science includes one or more courses on formal logic, covering at least the propositional and first-order variations.

Currently there are several **proof assistants** available, i.e. tools that allow the representation and validation of mathematical reasoning. Many systems such as Coq [2] and Agda [1] have been used to certify the correctness of important mathematical results. Although these systems are very powerful, they are still not popular. Part of this lies on the fact that they are not as known outside the community, and only recently introductory materials have been made available.

This text documents the process of formalizing a complete logic meta-theory from scratch using a proof assistant, alternating between explaining the object being represented and the tool to represent it. The choices for proof assistant and logic are:

1. the Coq proof assistant: an important tool with several useful libraries;

2. the classical propositional logic: usually the first logic to be developed in formal logic courses, with a relatively simple meta-theory.

We hope this text may be of value to

1. readers familiar with logic, which intend to put some effort into learning the Coq proof assistant;

2. readers familiar with Coq which intend to understand a little better the meta-theory of classical propositional logic.
3. readers unfamiliar with both worlds, which are trying to grasp both subjects symultaneously.

Chapter 2

Syntax of formulas

We assume a countable, infinite set Literal of *basic propositions* (or literals) which we will refer by the lowercase letters $\mathsf{p}, \mathsf{q}, \mathsf{r}, \mathsf{s} \ldots$. A *formula* in Propositional Logic is defined by the successive applications of logical operators for negation ($\neg$), conjunction ($\wedge$), disjunction ($\vee$) and implication ($\Rightarrow$), starting with literals. In this text we use uppercase letters $\mathsf{A}, \mathsf{B}, \ldots, \mathsf{P}, \mathsf{Q}, \mathsf{R}, \mathsf{S}, \ldots$ to represent formulas. The construction of formulas is shown in the abstract syntax below.

$$\begin{array}{lcl} \mathsf{p}, \mathsf{q}, \mathsf{r}, \ldots & \in & \mathsf{Literal} \\ \mathsf{A}, \mathsf{A}_1, \mathsf{B}, \ldots & \in & \mathsf{Formula} \\ \mathsf{A} & ::= & \mathsf{p} \mid \neg \mathsf{A}_1 \mid \mathsf{A}_1 \wedge \mathsf{A}_2 \mid \mathsf{A}_1 \vee \mathsf{A}_2 \mid \mathsf{A}_1 \Rightarrow \mathsf{A}_2 \end{array}$$

Some examples of formulas:

- $\mathsf{p} \vee \mathsf{q}$
- $\mathsf{q} \Rightarrow (\neg \mathsf{r} \Rightarrow \mathsf{p} \vee (\mathsf{q} \vee \mathsf{r}))$
- $\neg\neg \mathsf{p} \Rightarrow \mathsf{q}$

To represent the syntax for formulas in Coq, we need to choose an infinite set to represent basic propositions and define the set of formulas using induction. It is convenient to use natural numbers as basic propositions, since they are widely supported and come already equipped with a broad library of operations. Notice that natural numbers in Coq start with 0, not with 1. The following code in Coq describe the set of formulas.

```
Require Import Arith.  (* include natural numbers *)
Inductive formula : Set :=
 | Lit     : nat -> formula
 | Neg     : formula -> formula
 | And     : formula -> formula -> formula
 | Or      : formula -> formula -> formula
 | Implies : formula -> formula -> formula.
Check (Or (Lit 0) (Lit 1)).
Check (Implies (Lit 1)
               (Implies (Neg (Lit 2))
                        (Or  (Lit 0) (Or (Lit 1) (Lit 2))))).
Check (Implies (Neg (Neg (Lit 0))) (Lit 1)).
```

The Inductive command describes a set formed by induction based on its constructors. The Check command verifies the type of a given expression passed as parameter, and can be used to test values of previously defined sets. Notice that the representation of formulas may not be very easy to understand in prefix notation, in particular for big formulas. In order to to ease the visualization of inductively defined expressions, the command Notation allows one to specify a representation more similar to the mathematical notation, i.e. infix operators with precedence and associativity rules. A small number means high precedence, and conversely, a big number means low precedence. Levels range from 0 to 200.

```
Notation " X .-> Y "  := (Implies X Y)  (at level 5, right associativity).
Notation " X .\/ Y "  := (Or X Y)       (at level 4, left associativity).
Notation " X ./\ Y"   := (And X Y)      (at level 3, left associativity).
Notation " .~ X "     := (Neg X)        (at level 2, right associativity).
Notation " # X "      := (Lit X)        (at level 1, no associativity).
Check #0 .\/ #1.
Check #1 .-> .~ #2 .-> #0 .\/ (#1 .\/ #2).
Check .~.~ #0 .-> #1.
```

It is not possible to use the symbols $\sim$, \/, /\ and $->$ because these operators are native Coq operators. To differentiate native Coq operators from the operators of the object logic we are modeling, we prefix the latter with a dot.

For convenience, it is possible to provide names to particular values of interest using the Definition command. In our scenario, we will name some terms that will be used for testing purposes. The same command can be used to define non-recursive functions.

```
Definition ex1 := #0 .\/ #1.
Definition ex2 := #1 .-> .~ #2 .-> #0 .\/ (#1 .\/ #2).
Definition ex3 := .~.~ #0 .-> #1.
```

Besides defining a set by induction, we also need to define functions that receive elements of such sets and produce values based on them. For instance, consider the function size presented below.

$$
\begin{array}{lcl}
\mathsf{size} & : & \mathsf{Formula} \to \mathbb{N} \\
\mathsf{size}(p) & = & 1 \\
\mathsf{size}(\neg A) & = & 1 + \mathsf{size}(A) \\
\mathsf{size}(A \wedge B) & = & 1 + \mathsf{size}(A) + \mathsf{size}(B) \\
\mathsf{size}(A \vee B) & = & 1 + \mathsf{size}(A) + \mathsf{size}(B) \\
\mathsf{size}(A \Rightarrow B) & = & 1 + \mathsf{size}(A) + \mathsf{size}(B)
\end{array}
$$

Notice that this function is *recursive*, i.e. it uses the result of calling itself in order to calculate the value for some points, such as conjunctions. Moreover, notice that this function uses *structural recursion*, which means that recursive calls are only applied over subcomponents of the original argument. In Coq, structurally recursive functions are defined by means of the `Fixpoint` command. The following code implements the size function in Coq. The `match` operator is used to define the resulting value for all possible constructors of an inductive type. The command `Compute` is used to evaluate a given expression passed as parameter.

```
Fixpoint size (f:formula) : nat :=
match f with
 | Lit     p   => 1
 | Neg     a   => 1 + (size a)
 | And     a b => 1 + (size a) + (size b)
 | Or      a b => 1 + (size a) + (size b)
 | Implies a b => 1 + (size a) + (size b)
end.

Check size ex1.
Compute size ex1.
```

It is important to be aware that *arbitrary recursive functions*, i.e. functions which are defined using recursive calls but not necessarily of the structural kind (over a subcomponent of the argument) cannot be defined using the `Fixpoint` command. The reason is that in Coq all functions are total (the set of Coq-definable functions is a subset of the recursive functions), and the termination of the function must be proved at the moment of definition. Termination of structural recursive functions is straightforward and the proof is automatically generated by the `Fixpoint` command. It is possible to define functions using general recursion by using the `Program` command, however the user must provide a valid termination proof, which may not be trivial in many cases.

Another relevant operation regarding formulas is to collect all literals that occur on them. For such, we need a representation for a finite collection of literals. Although there are many libraries for sets in Coq, we will prefer to employ lists as collections, given that they have a very simple inductive definition. As in other functional programming languages, lists are one of the most used data structures, so much that the inductive definition for lists is loaded at the start of a Coq session.

```
(* the definition for lists in Coq's default library *)

Inductive list (A : Type) : Type :=
 | nil : list A
 | cons : A -> list A -> list A.

(* load additional list definitions, operations and facts  *)
Require Import List.

Definition list_ex1 : list nat := cons 3 (cons 2 nil).
Definition list_ex2 : list nat := 3 :: 2 :: nil.
```

Notice that the list constructor requires a *type parameter* describing the type of elements stored in the data structure. The module `List` is required to access additional functions and a more convenient notation for lists, as shown in the definition of `list_ex2`.

Returning to the issue of collecting all literals of a given formulas, this can be done by the following recursive function literals.

$$\begin{array}{lcl}
\mathsf{literals} & : & \mathsf{Formula} \rightarrow \mathcal{P}\,(\mathsf{Literal}) \\
\mathsf{literals}(\mathsf{p}) & = & \{\mathsf{p}\} \\
\mathsf{literals}(\neg \mathsf{A}) & = & \mathsf{literals}(\mathsf{A}) \\
\mathsf{literals}(\mathsf{A} \wedge \mathsf{B}) & = & \mathsf{literals}(\mathsf{A}) \cup \mathsf{literals}(\mathsf{B}) \\
\mathsf{literals}(\mathsf{A} \vee \mathsf{B}) & = & \mathsf{literals}(\mathsf{A}) \cup \mathsf{literals}(\mathsf{B}) \\
\mathsf{literals}(\mathsf{A} \Rightarrow \mathsf{B}) & = & \mathsf{literals}(\mathsf{A}) \cup \mathsf{literals}(\mathsf{B})
\end{array}$$

The function literals translate directly to Coq code as follows. We use lists of nats instead of sets, and the operation app corresponds to the default implementation in Coq of *list concatenation.*

```
Fixpoint literals (f:formula) : list nat :=
match f with
| Lit      x   => x::nil
| Neg      a   => literals a
| And      a b => app (literals a) (literals b)
| Or       a b => app (literals a) (literals b)
| Implies  a b => app (literals a) (literals b)
end.
```

Chapter 3

Semantics of Propositional Logic

3.1 Truth-table semantics

In Classical Propositional Logic, there are two truth-values, which we denote $\top$ (true) and $\bot$ (false). To ease the reading, we will use the notation 1 and 0 as synonyms for $\top$ and $\bot$, respectively. To be able to evaluate a formula φ, i.e. associate a truth-value to it, it is usually required to evaluate the truth-value of the literals within the formula. Since the meaning behind each literal is opaque to the logic, a particular model of world can be represented by means of a literal evaluation function

$$\mathcal{V} : \mathsf{Literal} \rightarrow \{1, 0\}$$

associating each literal to a specific truth-value. One example of such function would be even, presented below. The even function requires a fixed enumeration of literals and it alternates the associated truth-value between 1 and 0 along the enumeration.

$$\begin{array}{lcl}
\mathsf{Literal} & = & \{\mathsf{p}, \mathsf{q}, \mathsf{r}, \mathsf{s}, \ldots\} \\
\mathsf{even} & : & \mathsf{Literal} \rightarrow \{1, 0\} \\
\mathsf{even}(\mathsf{p}) & = & 1 \\
\mathsf{even}(\mathsf{q}) & = & 0 \\
\mathsf{even}(\mathsf{r}) & = & 1 \\
\mathsf{even}(\mathsf{s}) & = & 0 \\
 & \vdots &
\end{array}$$

The function above could be implemented by the following code, taking advantage of the fact that we already have an adequate enumeration of literals since we use natural numbers to represent them.

```
Fixpoint evenb (x:nat) : bool :=
match x with
  | O   => true
  | S x => negb (evenb x)
end.
```

Let us clarify a bit about the implementation of natural numbers and booleans in the standard library before proceeding. Booleans are inductively defined, and the usual logical operations on booleans are named `andb`, `orb`, `implb`, `xorb` and `negb`. Notice the trailing "b" on their name, which indicate that the return value is the *boolean datatype* and not a *logic proposition* (the distinction will be explained soon). Notation rules allow one to use the infix symbols `||` and `&&` as `orb` and `andb`, respectively.

```
Inductive bool : Set :=
| true : bool
| false : bool.

Definition negb (b:bool)      : bool :=  if b  then false else true.
Definition andb (b1 b2:bool) : bool :=  if b1 then b2    else false.
Definition orb  (b1 b2:bool) : bool :=  if b1 then true  else b2.

Definition bool_ex1 := false || (true && (negb true)).
Check      bool_ex1.
Compute    bool_ex1.
```

Natural numbers are represented and manipulated using Peano arithmetic, i.e. using two constructors, `O` (the uppercase letter "o", representing zero) and `S` (representing the successor operation). Operations such as addition are implemented using primitive recursion in one of the arguments. The usual decimal notation for natural number and associated operations are defined by means of the `Notation` system of Coq.

```
Inductive nat : Set :=
| O : nat
| S : nat -> nat.

Definition two  := S (S O).
Definition five := S (S (S (S (S O)))).

Fixpoint add n m :=
  match n with
  | 0 => m
  | S p => S (add p m)
  end

Check   add two five.
Check   2 + 5.
Compute 2 + 5.
```

Returning to logic, we must determine a formula-evaluation function based on a given literal-evaluation function. The usual semantics for Classical Propositional Logic determines that the evaluation of an logical operation is a function of its arguments, as presented by the following *truth-tables*.

A	$\neg A$
0	1
1	0

A	B	$A \wedge B$	$A \vee B$	$A \Rightarrow B$
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	1

From the truth-tables above we obtain the following recursive definition for the function $\mathcal{V}' : \mathsf{Formula} \to \{1,0\}$, parameterized by the literal evaluation function $\mathcal{V} : \mathsf{Literal} \to \{1,0\}$.

$$\begin{array}{lcl}
\mathcal{V}' & : & \mathsf{Formula} \to \{1,0\} \\
\mathcal{V}'(p) & = & \mathcal{V}(p) \\
\mathcal{V}'(\neg A) & = & 1 \text{ iff } \mathcal{V}'(A) = 0 \\
\mathcal{V}'(A \wedge B) & = & 1 \text{ iff } \mathcal{V}'(A) = 1 \text{ and } \mathcal{V}'(B) = 1 \\
\mathcal{V}'(A \vee B) & = & 1 \text{ iff } \mathcal{V}'(A) = 1 \text{ or } \mathcal{V}'(B) = 1 \\
\mathcal{V}'(A \Rightarrow B) & = & 1 \text{ iff } \mathcal{V}'(A) = 0 \text{ or } \mathcal{V}'(B) = 1
\end{array}$$

Following the usual notation of the area, we write $\mathcal{V} \vDash A$ to denote $\mathcal{V}'(A) = 1$ and $\mathcal{V} \nvDash A$ to denote $\mathcal{V}'(A) = 0$.

Since Coq is a higher-order language (functions can receive functions as arguments, and return functions as result), it is straightforward to define the parameterization of $\mathcal{V}'$ by adding the literal evaluation function $\mathcal{V}$ as its first argument. The function evaluation, depicted in the following code, implements $\mathcal{V}'$ as a function of both a literal-evaluation function and a formula. The operations of the boolean datatype are used to compute the final boolean value. The command `Compute` can be used to completely evaluate the truth value of a formula, given a literal-evaluation function.

```
Fixpoint evaluation (v: nat -> bool) (f:formula) : bool :=
match f with
| Lit     p     => v p
| Neg     a     => negb (evaluation v a)
| And     a b   => (evaluation v a) && (evaluation v b)
| Or      a b   => (evaluation v a) || (evaluation v b)
| Implies a b   => (negb (evaluation v a)) || (evaluation v b)
end.

Check   evaluation evenb ex1.
Compute evaluation evenb ex1.
```

3.2 Programs and proofs in Coq

It seems that now is a good moment to provide some insight why there is a distinction between the *boolean datatype* and *logical propositions*. Up to now, we have been using Coq only as a *programming language*: we have defined new datatypes (by induction) and also some functions that operate on them (by structural recursion). Notice that `formula`, `nat`, `bool` were defined of elements of kind `Set` (a kind for defined datatypes). We can typecheck expressions using the `Check` command, and run them using the `Compute` command.

However, Coq is more than a programming language, it is also a proof assistant which one can use to *express properties* and *prove them*. One of the simplest examples would be to state and prove that the addition on natural numbers is associative. We first state this fact by using the `Theorem` command, which receives a *name* for the stated property, and a meaning for it in Coq's logic language. After this, the system waits for a proof of this fact. Proofs start with the `Proof` command, followed by many *proof-state* manipulation commands, which end by the `Qed` command if the proof effort is successful. The language of commands that can be used between `Proof` and `Qed` is quite large, and we will postpone a detailed explanation of them. For now, just observe the overall structure of the aforementioned associativity proof in Coq.

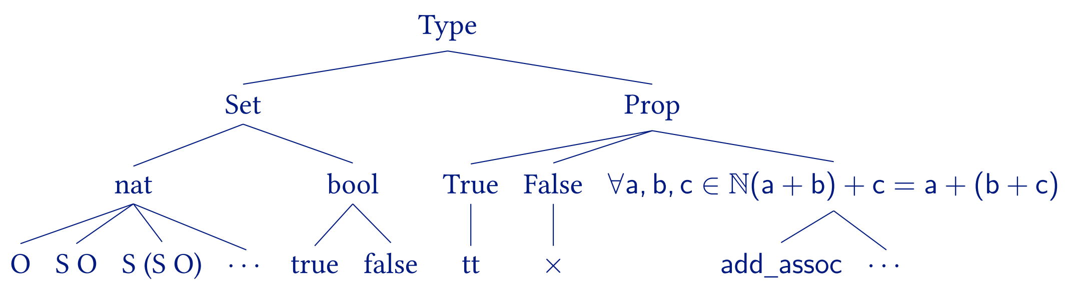
```
Theorem add_assoc : forall (a b c: nat), (a + b) + c = a + (b+c).
Proof.
induction a.
 - intros. simpl. auto.
 - intros. simpl. rewrite IHa. auto.
Qed.

Check add_assoc.
Check forall a b c : nat, a + b + c = a + (b + c).
```

When typechecking add_assoc, we observe that its type is

$$\forall a, b, c \in \mathbb{N}, (a + b) + c = a + (b + c)$$

This illustrates the fact that, from the perspective of Coq, *proofs* are terms and *properties* are types. Moreover, if we check the type of $\forall a, b, c \in \mathbb{N}, (a + b) + c = a + (b + c)$, we obtain that it is an object of type Prop (a kind for logical propositions). To provide a proof to a proposition is to construct an object of its type. This principle is known as *propositions as types*. In Coq, a proposition which is **true** has at least one proof (i.e. the type is inhabited), and a proposition which is **false** has none (i.e. the type is not inhabited).

The following tree depicts in a high level the two kinds of objects in Coq: computational and logic. The relation depicted by the tree is *typing*, i.e. the children are elements whose type is the parent. True (with an uppercase "t") is a proposition with a canonical proof called tt. False (with an uppercase "f") is a proposition without a proof (the absence of a proof is represented by the $\times$ mark). The associativity of addition is a property which has add_assoc as a proof (however there are other distinct proofs of the same fact). Notice the relative position between the boolean datatype bool and the type of all propositions Prop.



The boolean operators negb, andb, orb and implb are replicated in the level of propositions as the logical operators not (~), and (/\), or (\/) and implication (->). The usage of both is quite distinct: for simplicity, let us just say that booleans are used in computations and propositions are used in proofs. Therefore, if our focus is to prove properties regarding objects we have defined, it is more convenient use Prop in our definitions rather than booleans. This will become more evident when considering our first proof of meta-properties of the Propositional Logic. For this purpose, we now present a version of the encoded semantics of Classical Propositional Logic using Prop instead of booleans.

```
Fixpoint evenP (x:nat) : Prop :=
match x with
  | O   => True
  | S x => not (evenP x)
end.

Fixpoint formulaSAT (v: nat -> Prop) (f:formula) : Prop :=
match f with
| Lit     p     => v p
| Neg     a     => ~ (formulaSAT v a)
| And     a b   => (formulaSAT v a) /\ (formulaSAT v b)
| Or      a b   => (formulaSAT v a) \/ (formulaSAT v b)
| Implies a b   => ~ (formulaSAT v a) \/ (formulaSAT v b)
end.

Check   formulaSAT evenP (#0 .\/ #1).
Compute formulaSAT evenP (#0 .\/ #1).
```

Notice that the structure of the code is essentially the same: we only changed the return type and associated operations. When typechecking the example we obtain the type Prop, and when computing it we obtain the proposition True \/ (True -> False), rather than the simplified data value true. Notice that this happens because evenP #0 evaluates to True, and evenP #1 evaluates to (not True). In the logic of Coq, a negation (not A) is actually an alias for the implication (A -> False). Notice that, as elements of type Prop, True and True \/ (True -> False) are distinct, even though they are logically equivalent.

Chapter 4

Logic entailment

4.1 The entailment relation

A *theory* is a set of formulas. For example, the set

$$\Gamma = \{\, p, p \Rightarrow q, \neg p \vee (\neg p \wedge r) \,\}$$

constitutes a theory. We say that an evaluation function $\mathcal{V}' : \mathsf{Formula} \to \{1, 0\}$ *satisfies* a theory, denoted

$$\mathcal{V} \vDash \Gamma$$

when for all $A \in \Gamma$, we have $\mathcal{V} \vDash A$.

The relation *logical consequence* or *logic entailment* connects a theory with a single formula. We denote

$$\Gamma \vDash A$$

to say that Γ entails A, or, in other words, that A is a logical consequence of Γ. This relation holds when, for all possible literal evaluation $\mathcal{V}$, if $\mathcal{V} \vDash \Gamma$ then $\mathcal{V} \vDash A$. In other words: *all evaluations that satisfy the theory also satisfy the conclusion.* The following code implements theories, theory satisfaction and logic entailment.

```
Definition theory := list formula.

Fixpoint theorySAT (v:nat -> Prop) (Gamma:theory) : Prop :=
match Gamma with
  | nil      => True
  | cons h t => (formulaSAT v h) /\ (theorySAT v t)
end.
```

```

Definition entails (Gamma:theory) (f:formula) : Prop :=
forall (v: nat -> Prop), theorySAT v Gamma -> formulaSAT v f.


Notation " A |= B " := (entails A B) (at level 110, no associativity).
```

We chose lists to represent theories instead of sets in Coq. There are two reasons for that: list are easy to manipulate in Coq, and they are a good choice if we want to model other kinds of logics like substructural logic, for instance. Note that we are using finite lists. That is not a problem since our logic is compact[1]

In our encoding, we restricted theories to be finite since we are using the List datatype instead of actual sets. One advantage of this is that we can manipulate the underlying set as a simple list, for instance when defining the function theorySAT that determines the satisfaction of a theory for a formula. Notice also that, because evaluation functions for formulas are uniquely defined by means of literal evaluation, we use the latter as parameter. The proposition entails uses universal quantification over all literal evaluation functions. A disavantage of the proposed formalization, however, is that we cannot express and prove properties about infinite theories.

4.2 Proving theorems in Coq

Now we will start explaining the language used in Coq to describe proofs. After introducing the basic ideas, we will state and prove properties of the entailment relation in Coq.

To write a property in Coq, we start with the keyword Theorem, followed by an identifier, a colon and finally the property we intend to prove in the logic language of Coq. It is important to note that the Theorem command only provides a name for a property: after it, a proof of the property is expected. A proof starts with the keyword Proof, and after it a sequence of *proof commands* follow. If the proof is successful, this sequence end with the command Qed. For example, consider the *modus ponen* rule

```
Theorem modus_ponens : forall (A:Prop) (B:Prop), (A -> B) -> A -> B.
```

[1]An infinite theory Γ is satisfiable, if and only if, every finite subset of Γ is also satisfiable.

A **proof state**, **goal** or **sequent** in Coq is a structure consisting of many *named premisses* and one single *conclusion*, as the following example shows:

$$\boxed{\begin{array}{c} \mathsf{p1 : P \to Q} \\ \mathsf{p2 : P \land R} \\ \hline \mathsf{Q \lor S} \end{array}}$$

After the `Theorem` keyword is received by Coq, an initial goal is created. It consists of an empty collection of premisses with the desired property as the conclusion. For example, the initial goal of `modus_ponens` is shown below.

$$\boxed{\begin{array}{c} \\ \hline \mathsf{\forall(A : Prop), \forall(B : Prop), (A \to B) \to A \to B} \end{array}}$$

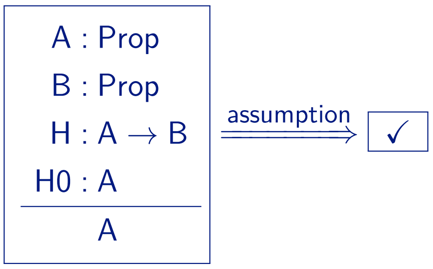
Each *proof manipulation command* can either *close* the current goal or create one or more new goals. For instance, the `intros` command puts all antecedents of a sequence of implications as premisses, leaving the last consequent as the conclusion. Below we present the effect of intros over the initial goal of `modus_ponens`:

$$\boxed{\begin{array}{c} \\ \hline \mathsf{\forall(A : Prop), \forall(B : Prop), (A \to B) \to A \to B} \end{array}} \xrightarrow{\text{intros}} \boxed{\begin{array}{c} \mathsf{A : Prop} \\ \mathsf{B : Prop} \\ \mathsf{H : A \to B} \\ \mathsf{H0 : A} \\ \hline \mathsf{B} \end{array}}$$

The `apply` command proves a conclusion based on an implication as premisse. Given the previous proof state, the command `apply H` changes the conclusion, altering it to be the antecedent of the conditional `H`. The idea behing this command is that if we can show that A is obtained from the current premisses, then by means of the implication `H` we can prove the original goal B.

$$\boxed{\begin{array}{c} \mathsf{A : Prop} \\ \mathsf{B : Prop} \\ \mathsf{H : A \to B} \\ \mathsf{H0 : A} \\ \hline \mathsf{B} \end{array}} \xrightarrow{\text{apply H}} \boxed{\begin{array}{c} \mathsf{A : Prop} \\ \mathsf{B : Prop} \\ \mathsf{H : A \to B} \\ \mathsf{H0 : A} \\ \hline \mathsf{A} \end{array}}$$

The `assumption` command closes the proof state when the conclusion appears as one of the premisses (in this case, H0).



When all created goals are closed, the proof is finished and the command `Qed` can be used to delimit its conclusion.

```
Theorem modus_ponens : forall (A:Prop) (B:Prop), (A -> B) -> A -> B.
Proof. intros. apply H. assumption. Qed.
```

The construction of a proof in Coq is usually an interactive process: starting with the initial goal, the user attempts several commands until all goals are closed. If the property originally stated is not valid (i.e. not a theorem), it will not be possible to close all of its goals, and the proof effort will not be successful.

Now that the basic notions of proof construction were introduced, let us start proving properties of the entailment relation. We begin with basic structural properties and conclude with an important result: the deduction theorem.

4.3 Structural properties of the entailment relation

Some properties of the entailment relation are known as *structural properties*. Two of them, namely *reflexivity* and *transitivity*, are very basic, and are expected to be satisfied by the entailment relation of any logic:

Reflexivity: $\Gamma, A \vDash A$
Proof: Let Γ be a theory and A a formula, given $\mathcal{V}$ a valuation such that $\mathcal{V} \vDash \Gamma \cup \{A\}$. In particular, $\mathcal{V} \vDash A$, which completes the proof.

```
Theorem  reflexive_deduction:
  forall (Gamma:theory) (A:formula) ,
    (A::Gamma |= A).
```

```
4  Proof.
5    intros.
6    unfold entails.
7    intros.
8    simpl in H.
9    destruct H as [Hleft Hright].
10   apply Hleft.
11 Qed.
```

In the proof above we used the tactic `unfold`, which unfolds a name by its definition. On line 6 the `|=` operator in the goal is rewrited as

```
forall v : nat -> Prop, theorySAT v (A :: Gamma) -> formulaSAT v A
```

.

The tactic `destruct` in line 9 split the hypothesis (a logical "and") `H` in two (named `Hright` and `Hleft` by the `as [...]` structure).

Transitivity: $(\Gamma, A \vDash B) \wedge (\Delta, B \vDash C) \Rightarrow (\Gamma, \Delta, A \vDash C)$

Proof: Suppose $\Gamma, A \vDash B$ and $\Delta, B \vDash C$, that is,

1. given any valuation $\mathcal{V}$ such that $\mathcal{V} \vDash \Gamma \cup \{A\}$, we have $\mathcal{V} \vDash B$, and
2. given any valuation $\mathcal{U}$ such that $\mathcal{U} \vDash \Delta \cup \{B\}$, we have $\mathcal{U} \vDash C$.

Take a valuation $\mathcal{W}$ such that $\mathcal{W} \vDash \Gamma \cup \Delta \cup \{A\}$. In particular, we have that $\mathcal{W} \vDash \Gamma \cup \{A\}$, so by 1 we have $\mathcal{W} \vDash B$. We also have that $\mathcal{W} \vDash \Delta$, so $\mathcal{W} \vDash \Delta \cup \{B\}$. By 2 we have $\mathcal{W} \vDash C$, which completes the proof.

```
Lemma theorySAT_union: forall (v: nat -> Prop) (Gamma Delta:theory),
(theorySAT v (Gamma++Delta)) -> ((theorySAT v Gamma) /\ (theorySAT v Delta)).
Proof.
  intros.
  induction Gamma.
    + simpl in *. split. auto. apply H.
    + simpl in *. apply and_assoc. destruct H as [left right]. split.
      - apply left.
      - apply IHGamma. apply right.
Qed.

```

```
Theorem  transitive_deduction_topdown:
   forall (Gamma Delta:theory) (A B C:formula) ,
      (A::Gamma |= B) /\ (B::Delta |= C) -> (A::Gamma++Delta |= C).
Proof.
  intros.
  unfold entails in *.
  destruct H as [H1 H2].
  intros.
  simpl in H.
    destruct H as [left right].
    pose right as x. apply theorySAT_union in x.
    destruct x as [x1 x2].
    assert ( theorySAT p (A::Gamma)). simpl. split. assumption. assumption.
    pose H as y. apply H1 in y.
    assert ( theorySAT p (B::Delta)). simpl. split. assumption. assumption.
    pose H0 as z. apply H2 in z.
    assumption.
Qed.
```

The code above is a translation of the last proof in Coq. However, this is not the "natural" way to prove things in Coq. In a "pen and paper" proof style is frequent to see a top-down reasoning: from the hypothesis we manipulate definitions and properties to find out the conclusion.

Parágrafo fazendo a "bijeção" entre a prova escrita e a do código acima!

On the other hand, Coq uses a bottom-up reasoning: from our goal, we look for properties and assumptions that lead to it. The following code prove the transitivity of the deduction relation using the bottom-up strategy.

```
Theorem  transitive_deduction:
   forall (Gamma Delta:theory) (A B C:formula) ,
      (A::Gamma |= B) /\ (B::Delta |= C) -> (A::Gamma++Delta |= C).
Proof.
  intros.
  unfold entails in *.
  destruct H as [H1 H2].
  intros. apply H2.
  simpl in *. destruct H as [left right].
  apply theorySAT_union in right. destruct right as [SatG SatD]. split.
  - apply H1. split. apply left. apply SatG.
  - apply SatD.
Qed.
```

When theories are presented as lists (and not as sets), the following two properties (exchange and contraction) ensure that these lists of formulas behave essentially as sets, at least with respect to the deduction relation.

Exchange: $A, B, \Gamma \vDash C$ implies $B, A, \Gamma \vDash C$.

Prova “informal” da propriedade

```
(* exchange *)

Theorem exchange:
  forall (p: nat -> Prop) (t:theory) (a:formula) (b:formula) (c:formula),
    (a::b::t |= c) -> (b::a::t |= c).
Proof.
intros. unfold entails in *.
intros. simpl in H0. destruct H0.
destruct H1. apply H. simpl. auto.
Qed.
```

Contraction: $A, A, \Gamma \vDash B$ implies $A, \Gamma \vDash B$.

Prova “informal” da propriedade

```
(* contraction *)

Theorem contraction:
  forall (p: nat -> Prop) (t:theory) (a:formula) (b:formula),
    (a::a::t |= b) -> (a::t |= b).
Proof.
intros. unfold entails in *.
intros. apply H. simpl in H0. destruct H0. simpl. auto.
Qed.
```

The last structural property we considerer is monotonicity, also called weakening. Roughly speaking, it states that adding formulas to the theory does not invalidate any deduction made before the addition.

Monotonicity: $\Gamma \vDash \mathsf{A}$ implies $\Gamma, \Delta \vDash \mathsf{A}$

Prova "informal" da propriedade

```
(* monotonicity *)
Theorem  monotonicity:
   forall (t u:theory) (a:formula) ,
      (t |= a) -> (t++u |= a).
Proof.
induction t.
 * intros. simpl in *. unfold entails. intros.  apply H. constructor 1.
 * intros. simpl in *. unfold entails. intros. apply H. simpl in H0. destruct
 ↪  H0.
   simpl. split. auto. apply theorySAT_union1 in H1. destruct H1. auto.
Qed.
```

4.4 The deduction theorem for propositional logic

Informally, the deduction theorem allows formulas to "cross" the $\vDash$ symbol. Formally, it is stated as

$$\Gamma, \mathsf{A} \vDash \mathsf{B} \text{ if, and only if, } \Gamma \vDash \mathsf{A} \Rightarrow \mathsf{B}$$

in which we use Γ, A as a synonym for $\Gamma \cup \{\mathsf{A}\}$.

Proof: Since the deduction theorem is a "if and only if" statement, we will prove it spliting in two subproofs.

First, we prove that if $\Gamma, \mathsf{A} \vDash \mathsf{B}$ then $\Gamma \vDash \mathsf{A} \Rightarrow \mathsf{B}$.

Suppose $\Gamma, \mathsf{A} \vDash \mathsf{B}$, which means that every valuation $\mathcal{V}_0 : \mathsf{Literal} \to \{1, 0\}$ that satisfies $\Gamma \cup \{\mathsf{A}\}$ also satisfies B. Let $\mathcal{V}'_0$ be some valuation such that $\mathcal{V}'_0 \vDash \Gamma$. There are two cases to analyze:

1. $\mathcal{V}'_0 \vDash \mathsf{A}$: in this case, $\mathcal{V}'_0 \vDash \Gamma \cup \{\mathsf{A}\}$. By hypothesis, $\mathcal{V}'_0 \vDash \mathsf{B}$. Therefore, $\mathcal{V}'_0 \vDash \mathsf{A} \Rightarrow \mathsf{B}$ by the truth-table of implication.

2. $\mathcal{V}' \nvDash \mathsf{A}$: so $\mathcal{V}'_0 \vDash \mathsf{A} \Rightarrow \mathsf{B}$ by the truth-table of implication.

Thus $\Gamma \vDash \mathsf{A} \Rightarrow \mathsf{B}$.

Now the other way, if $\Gamma \vDash \mathsf{A} \Rightarrow \mathsf{B}$ then $\Gamma, \mathsf{A} \vDash \mathsf{B}$.

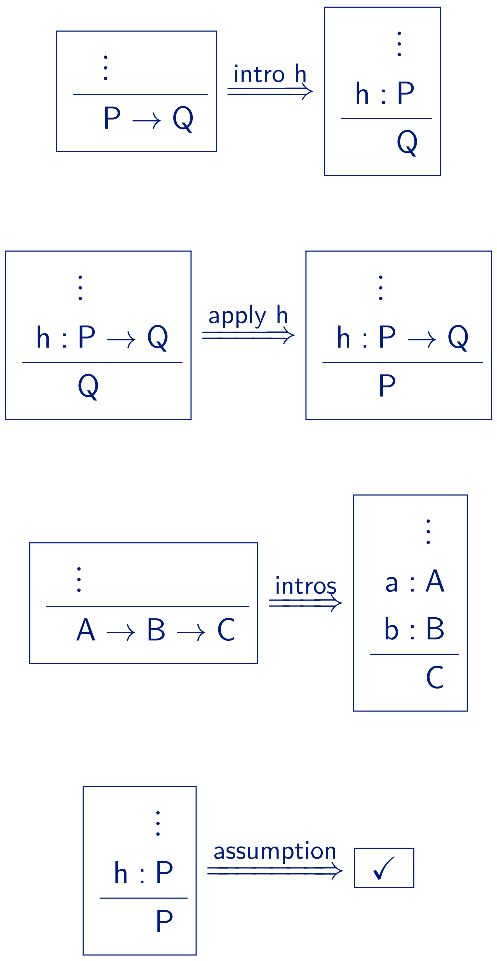
Suppose $\Gamma \vDash A \Rightarrow B$, so every valuation $\mathcal{V}_0$ that satisfies Γ also satisfies the formula $A \Rightarrow B$. Let …

Fazer a prova em linguagem "natural"

```
(*verificar se já apareceu: unfold, assumption*)

Theorem deduction1 :

  forall (p: nat -> Prop) (t:theory) (a:formula) (b:formula),
         (a::t |= b) -> (t |= a .-> b).
Proof.
  intros p t a b H.
  unfold entails in H.
  unfold entails.
  intros p0 H0.
  assert ((formulaSAT p0 a) \/ ~(formulaSAT p0 a)). { apply classic. (*new!*)
  ↪  }
  destruct H1. (* new! destruct or *)
  - simpl. right. apply H. simpl. split. assumption. assumption. (*new!
  ↪  right/left + split*)
  - simpl. left. assumption.
Qed.
```

```
Theorem deduction2 :

  forall (p: nat -> Prop) (t:theory) (a:formula) (b:formula),
      (t |= a .-> b) -> (a::t |= b).

Proof.
  intros p t a b H.
  unfold entails in *.
  intros p0 H0.
  simpl in H0.
  destruct H0 as [H' H''].
  pose (H p0 H'') as H_used. (*new!*)
  simpl in H_used.
  destruct H_used.
    - contradiction. (*new!*)
    - assumption.
Qed.
```

```
Theorem deduction :
  forall (p: nat -> Prop) (t:theory) (a:formula) (b:formula),
         (a::t |= b) <-> (t |= a .-> b).
Proof.
intros.
split.
  - intros. apply deduction1. assumption. assumption.
  - intros. apply deduction2. assumption. assumption.
Qed.
```



$$\boxed{\begin{array}{c} \vdots \\ a : P \\ b : \text{not } P \\ \hline Q \end{array}} \xRightarrow{\text{contradiction}} \boxed{\checkmark}$$

$$\boxed{\begin{array}{c} \vdots \\ \hline P = P \end{array}} \xRightarrow{\text{reflexivity}} \boxed{\checkmark}$$

$$\boxed{\begin{array}{c} a : P \\ b : (P \to Q) \lor Q \\ \hline Q \end{array}} \xRightarrow{\text{destruct b}} \boxed{\begin{array}{c} a : P \\ b : (P \to Q) \\ \hline Q \end{array}} \boxed{\begin{array}{c} a : P \\ b : Q \\ \hline Q \end{array}}$$

$$\boxed{\begin{array}{c} a : P \to Q \\ b : R \land P \\ \hline Q \end{array}} \xRightarrow{\text{destruct b as [b1 b2]}} \boxed{\begin{array}{c} a : P \to Q \\ b1 : R \\ b2 : P \\ \hline Q \end{array}}$$

$$\boxed{\begin{array}{c} \vdots \\ \hline P \lor Q \end{array}} \xRightarrow{\text{left}} \boxed{\begin{array}{c} \vdots \\ \hline P \end{array}}$$

$$\boxed{\begin{array}{c} \vdots \\ \hline P \lor Q \end{array}} \xRightarrow{\text{right}} \boxed{\begin{array}{c} \vdots \\ \hline Q \end{array}}$$

$$\boxed{\begin{array}{c} \vdots \\ \hline P \land Q \end{array}} \xRightarrow{\text{split}} \boxed{\begin{array}{c} \vdots \\ \hline P \end{array}} \boxed{\begin{array}{c} \vdots \\ \hline Q \end{array}}$$

$$\boxed{\begin{array}{c} \vdots \\ \hline P \leftrightarrow Q \end{array}} \xRightarrow{\text{split}} \boxed{\begin{array}{c} \vdots \\ \hline P \to Q \end{array}} \boxed{\begin{array}{c} \vdots \\ \hline Q \to P \end{array}}$$

Chapter 5

Logical equivalence

5.1 Equivalence of formulas

The entailment relation presented in the previous chapter induces a very important relation between formulas: *logical equivalence*. We say two formulas A and B are (logically) equivalent, written $A \equiv B$, when one entails the other and vice-versa, i.e.

$$A \vDash B \qquad \text{and} \qquad B \vDash A$$

This definition translates directly to the following Coq code, in which we use `=|=` to denote $\equiv$.

```
Definition equivalence (a b:formula):Prop := (f::nil |= g) /\ (g::nil |= f).

Notation " A =|= B " := (equivalence A B) (at level 110, no associativity).
```

Equivalence can be observed by means of truth-tables: two formulas are equivalent whenever they have *exactly* the same *truth-table*. For example, we can show that $\neg p \Rightarrow q \vDash p \vee q$ and also $p \vee q \vDash \neg p \Rightarrow q$ by constructing their truth-tables an observing that their respective logical values coincide for all possible valuations of p and q.

p	q	$\neg p$	$p \vee q$	$\neg p \Rightarrow q$
0	0	1	0	0
0	1	1	1	1
1	0	0	1	1
1	1	0	1	1

The *de Morgan identities* are famous equivalences that describe how the negation operation distributes over conjunctions and disjunctions. Let us consider one of them as a detailed example: for all formulas A and B, we have the following equivalence

$$\neg(\mathsf{A} \wedge \mathsf{B}) \equiv (\neg\mathsf{A}) \vee (\neg\mathsf{B})$$

Proof: we need to show two entailments:

(1) $\neg(\mathsf{A} \wedge \mathsf{B}) \vDash (\neg\mathsf{A}) \vee (\neg\mathsf{B})$

Take a valuation[1] $\mathcal{V}$ such that $\mathcal{V}'(\neg(\mathsf{A} \wedge \mathsf{B})) = 1$. Notice that this is equivalent to $\mathcal{V}'(\mathsf{A} \wedge \mathsf{B}) = 0$ (by the definition of $\mathcal{V}'$). There are two possibilities for $\mathcal{V}'(\mathsf{A})$:

- case $\mathcal{V}'(\mathsf{A}) = 1$: in this situation, it is only possible that $\mathcal{V}'(\mathsf{B}) = 0$, because from $\mathcal{V}'(\mathsf{B}) = 1$ we would obtain a contradiction: if $\mathcal{V}'(\mathsf{A}) = 1$ and $\mathcal{V}'(\mathsf{B}) = 1$ are both true, we would have $\mathcal{V}'(\mathsf{A} \wedge \mathsf{B}) = 1$, which contradicts the hypothesis $\mathcal{V}'(\mathsf{A} \wedge \mathsf{B}) = 0$. Since $\mathcal{V}'(\mathsf{B}) = 0$, we have $\mathcal{V}'(\neg\mathsf{B}) = 1$ and, subsequently, $\mathcal{V}'((\neg\mathsf{A}) \vee (\neg\mathsf{B})) = 1$ (by the definition of $\mathcal{V}'$).
- case $\mathcal{V}'(\mathsf{A}) = 0$: in this situation, $\mathcal{V}'(\neg\mathsf{A}) = 1$ and $\mathcal{V}'((\neg\mathsf{A}) \vee (\neg\mathsf{B})) = 1$ (by the definition of $\mathcal{V}'$).

(2) $(\neg\mathsf{A}) \vee (\neg\mathsf{B}) \vDash \neg(\mathsf{A} \wedge \mathsf{B})$

Take a valuation $\mathcal{V}$ such that $\mathcal{V}'((\neg\mathsf{A}) \vee (\neg\mathsf{B})) = 1$. By the definition of $\mathcal{V}'$, either $\mathcal{V}'((\neg\mathsf{A})) = 1$ or $\mathcal{V}'((\neg\mathsf{B})) = 1$:

- case $\mathcal{V}'(\neg\mathsf{A}) = 1$: we then have $\mathcal{V}'(\mathsf{A}) = 0$, which leads to $\mathcal{V}'(\mathsf{A} \wedge \mathsf{B}) = 0$ and, subsequently, $\mathcal{V}'(\neg(\mathsf{A} \wedge \mathsf{B})) = 1$.
- case $\mathcal{V}'(\neg\mathsf{B}) = 1$: we then have $\mathcal{V}'(\mathsf{B}) = 0$, which leads to $\mathcal{V}'(\mathsf{A} \wedge \mathsf{B}) = 0$ and, subsequently, $\mathcal{V}'(\neg(\mathsf{A} \wedge \mathsf{B})) = 1$.

The reasoning presented above can be directly translated to a Coq Theorem, as follows.

```
Theorem deMorgan_and :
   forall (a b: formula), .~ (a ./\ b) =|= (.~a) .\/ (.~b).
Proof.
  intros.
  unfold equivalence.
```

[1]Remember that $\mathcal{V} : \mathsf{Literal} \to \{1, 0\}$ while $\mathcal{V}' : \mathsf{Formula} \to \{1, 0\}$

```
  split.
    - intro. unfold entails. intros. simpl in H. destruct H.
      simpl. pose (classic (formulaSAT v a)) as lem.
      destruct lem.
         * assert (~ (formulaSAT v b)). intro.
           assert (formulaSAT v a /\ formulaSAT v b).
             split. auto. auto. contradiction.
           right. auto.
         * left. auto.
    - intro. intro. simpl in *.
      destruct H. destruct H as [ left | right ].
      intro. destruct H. contradiction.
      intro. destruct H. contradiction.
Qed.
```

There are many distinct equivalences in classical propositional logic. Notice that in many demonstrations based on valuations, the law of excluded middle (LEM), which states

$$A \lor \neg A$$

for any proposition A is used to reason by cases. To replicate this, we used the `classic` tactic in coq to introduce such assertion as a premisse in the proof state.

5.2 Structural properties of formula equivalence

Although we have been calling the $\equiv$ relation (`=|=` in Coq) an equivalence, we did not ensure that it satisfies the requirements to be called as such. Formally, a relation $R \subseteq U \times U$ is an *equivalence relation* whenever it is

- reflexive: $\forall u \in U, uRu$;
- symmetric: $\forall u, v \in U,$ if uRv then vRu;
- transitive: $\forall u, v, w \in U,$ if uRv and vRw then uRw.

Now we can state and prove these properties. The proofs are quite easy since they rely mainly on the already stated properties of $\vDash$, for which we proved reflexivity and transitivity in the previous chapter.

```
1  Theorem equiv_refl:
2          forall (f:formula), f =|= f.
3  Proof.
4  intro. unfold equivalence. split.
5  apply reflexive_deduction.
6  apply reflexive_deduction.
7  Qed.
```

```
1  Theorem equiv_sym:
2          forall (f g:formula), (f =|= g) -> (g =|= f).
3  Proof.
4  intro. unfold equivalence. split.
5    destruct H. auto.
6    destruct H. auto.
7  Qed.
```

```
1  Theorem equiv_trans :
2          forall (f g h:formula), (f =|= g) -> (g =|= h) -> (f =|= h).
3  Proof.
4  intros. unfold equivalence in *. split.
5     destruct H. destruct H0.
6       assert ((nil:list formula) = (nil:list formula) ++ (nil:list formula)).
7       simpl. auto.
8       rewrite H3.
9       apply transitive_deduction_td with (B:=g). split. auto. auto.
10    destruct H. destruct H0.
11      assert ((nil:list formula) = (nil:list formula) ++ (nil:list formula)).
12      simpl. auto.
13      rewrite H3.
14      apply transitive_deduction_td with (B:=g). split. auto. auto.
15 Qed.
```

Comentário: na linha 6, talvez fosse mais interessante escrever nil:theory ao invés de nil:list formula. A sintaxe pode conflitar os : com :: para um leitor rápido e inexperiente, e colocar o tipo como theory parece-me mais interessante para notar que é uma declaração de tipo e não um operador de listas.

Besides being an equivalence relation, the $\equiv$ relation has even more interesting properties: it is also a congruence relation. In general terms, an equivalence relation $\mathsf{R} \subseteq \mathsf{U} \times \mathsf{U}$ is *congruent to a constructor* $\mathsf{C} : \mathsf{U}^n \to \mathsf{U}$ whenever

$$u_1 \mathrel{R} v_1, \quad \ldots, \quad u_n \mathrel{R} v_n$$

implies

$$C(u_1, \ldots, u_n) \mathrel{R} C(v_1, \ldots, v_n)$$

If R is a congruence with respect to all constructors of U, we simply call it a *congruence*. For the case of formulas, we need to consider all formula constructors: negation, conjuntion, disjunction and implication. The following Coq code shows that $\equiv$ is congruent to all of them.

```
Theorem equiv_cong_not :
        forall (f g:formula), (f =|= g) -> (.~ f =|= .~ g).
Proof.
intros. unfold equivalence in *. split.
  - destruct H. unfold entails in *. simpl in *.
    intros. intro. assert (formulaSAT v g /\ True). split. auto. auto.
    pose (H0 v H3). destruct H1. contradiction.
  - destruct H. unfold entails in *. simpl in *.
    intros. intro. assert (formulaSAT v f /\ True). split. auto. auto.
    pose (H v H3). destruct H1. contradiction.
Qed.
```

```
Theorem equiv_cong_implies :
        forall (f f' g g':formula), (f =|= f') -> (g =|= g') ->
                                    (f .-> g  =|= f' .-> g').
Proof.
intros. unfold equivalence in *. split.
  - unfold entails in *. intros. simpl in *. destruct H1.
    destruct H1.
     * assert (~ formulaSAT v f'). intro.
         assert (formulaSAT v f' /\ True). split. auto. auto.
         apply H in H4. contradiction.
       left. auto.
     * assert (formulaSAT v g /\ True). split. auto. auto.
         apply H0 in H3. right. auto.
  - unfold entails in *. intros. simpl in *. destruct H1.
     destruct H1.
     * assert (~ formulaSAT v f). intro.
         assert (formulaSAT v f /\ True). split. auto. auto.
         apply H in H4. contradiction.
       left. auto.
     *  assert (formulaSAT v g' /\ True). split. auto. auto.
        apply H0 in H3. right. auto.
Qed.
```

```
Theorem equiv_cong_and :
  forall (f f' g g':formula), (f =|= f') -> (g =|= g')
                              ->   (f ./\ g =|= f' ./\ g').
Proof.
intros.  unfold equivalence in *. split.
  * unfold entails in *. intros. simpl in *. destruct H1. destruct H1. split.
     - assert (formulaSAT v f /\ True). split. auto. auto.
       apply H in H4. auto.
     - assert (formulaSAT v g /\ True). split. auto. auto.
       apply H0 in H4. auto.
  * unfold entails in *. intros. simpl in *. destruct H1. destruct H1. split.
     - assert (formulaSAT v f' /\ True). split. auto. auto.
       apply H in H4. auto.
     - assert (formulaSAT v g' /\ True). split. auto. auto.
       apply H0 in H4. auto.
Qed.
```

```
Theorem equiv_cong_or :
   forall (f f' g g':formula), (f =|= f') -> (g =|= g')
                               -> (f .\/ g =|= f' .\/ g').
Proof.
intros.  unfold equivalence in *. split.
  * unfold entails in *. intros. simpl in *. destruct H1. destruct H1.
     - assert (formulaSAT v f /\ True). split. auto. auto.
       apply H in H3. auto.
     - assert (formulaSAT v g /\ True). split. auto. auto.
       apply H0 in H3. auto.
  * unfold entails in *. intros. simpl in *. destruct H1. destruct H1.
```

```
    - assert (formulaSAT v f' /\ True). split. auto. auto.
      apply H in H3. auto.
    - assert (formulaSAT v g' /\ True). split. auto. auto.
      apply H0 in H3. auto.
Qed.
```

5.3 Operator elimination

The presented version of logical propositional logic contains four logical connectives: $\{\neg, \wedge, \vee, \Rightarrow\}$. One of the characteristics of the classical propositional logic is that there is some redundancy in terms of the available operators: if one removes conjunctions ($\wedge$) it is still possible to construct formulas representing all possible truth-tables.

We say that a collection X of logical operators is *adequate* whenever it is possible to represent all possible truth-tables by means of formulas that employ only operators in X. For instance, $\{\wedge, \neg, \vee\}$ is adequate, as it is $\{\wedge, \neg\}$, $\{\vee, \neg\}$, $\{\Rightarrow, \neg\}$. However, $\{\wedge, \vee\}$ and $\{\wedge\}$ are not adequate.

The possibility of eliminating some operators and still be able to represent the same collection of truth-tables is important. Besides reducing the complexity of circuit construction (by reducing the kinds of basic blocks used) it also allows the simplification of the study of meta-properties of the logic (by reducing the number of cases to be considered in theorems).

Operator elimination can be achieved by means of logical equivalence. For instance, consider the following classical equivalences:

- $\neg\neg A \equiv A$
- $A \vee B \equiv (\neg A) \Rightarrow B$
- $A \wedge B \equiv \neg(A \Rightarrow \neg B)$

These three important equivalences can be proved by semantic means, as confirmed by the following Coq code.

```
Theorem elim_double_negation : forall (a: formula),   .~ .~ a  =|=  a.
Proof.
intros.
split.
  - intros. unfold entails in *. intros. simpl in H. destruct H.
    assert (formulaSAT v a \/ ~formulaSAT v a). apply classic.
    destruct H1.
       * auto.
       * contradiction.

  - intro. intro. simpl in H. destruct H.
    simpl. intro. contradiction.
Qed.
```

```
Theorem or_to_implies: forall (a b: formula),  a .\/ b   =|=   (.~ a) .-> b.
Proof.
intros.
split.
  - intros. unfold entails in *. intros.
    pose (classic (formulaSAT v a)) as H1.
    destruct H1.
      * simpl. left. auto.
      * simpl in H. destruct H. destruct H. contradiction. simpl. right. auto.
  - intros. unfold entails in *. intros. simpl.
    pose (classic (formulaSAT v a)) as H1.
    destruct H1. left. auto.
    simpl in H. destruct H. destruct H. contradiction. right. auto.
Qed.
```

```
Theorem and_to_implies : forall (a b: formula),   a ./\ b  =|=  .~ (a .-> .~
 ↪  b).
Proof.
intros.
split.
  - intro. intro. simpl in H. destruct H. destruct H.

    simpl. intro. destruct H2. contradiction. contradiction.
  - intro. intro.
    pose (classic (formulaSAT v a)) as H1.
    destruct H1.
       * pose (classic (formulaSAT v b)) as H2.
         destruct H2. simpl. split. auto. auto.
         simpl in H. destruct H. assert (~formulaSAT v a \/ ~formulaSAT v b).
          ↪  right. auto. contradiction.
```

```
      * simpl in *. destruct H. assert (~formulaSAT v a \/ ~formulaSAT v b).
       ↪  left. auto. contradiction.
Qed.
```

Notice that the right-hand side of all these equivalences only involve literals, negation and implication, i.e. the set of connectives $\{\Rightarrow, \neg\}$, which is known as the *implicative fragment* of the classical propositional logic.

Let us denote $\mathsf{Formula}_X$ the set of formulas obtained from literals and using only operators in X. For example, the set $\mathsf{Formula}$ defined in Chapter 2 would be denoted $\mathsf{Formula}_{\{\neg,\wedge,\vee,\Rightarrow\}}$ in this notation. Using the previously proved equivalences, it is possible to define a function

$$\mathtt{toImplic} : \mathsf{Formula}_{\{\neg,\wedge,\vee,\Rightarrow\}} \to \mathsf{Formula}_{\{\neg,\Rightarrow\}}$$

which always returns a a resulting formula equivalent to the received argument, but using only negation and implication, i.e. eliminating conjunctions and disjunctions.

```
Fixpoint toImplic (f: formula) : formula :=
match f with
  | # x     => # x
  | .~ a    => .~ (toImplic a)
  | a ./\ b => .~ (toImplic a .-> .~ (toImplic b) )
  | a .\/ b => (.~ (toImplic a) .-> (toImplic b) )
  | a .-> b => (toImplic a) .-> (toImplic b)
end.
```

As important as defining the function `toImplic` is to prove that the output formula is indeed equivalent to the input formula. The following Coq code ensures that this is the case: it is a proof by induction on the formula received as argument, using as lemmas the previously proved equivalences together with the fact that $\equiv$ is a congruence.

```
Theorem toImplic_equiv : forall (f:formula), f =|= (toImplic f).
Proof.
induction f.
 simpl. apply equiv_refl.
 simpl. apply equiv_cong_not. auto.
 simpl.
```

```
  assert (f1 ./\ f2 =|= toImplic f1 ./\ toImplic f2).
    apply equiv_cong_and. auto. auto.
  assert (toImplic f1 ./\ toImplic f2 =|=
            .~ (toImplic f1 .-> .~ toImplic f2)).
    apply and_to_implies.
  apply equiv_trans with (g:=toImplic f1 ./\ toImplic f2). auto. auto.
 simpl.
  assert (f1 .\/ f2 =|= toImplic f1 .\/ toImplic f2).
    apply equiv_cong_or. auto. auto.
  assert (toImplic f1 .\/ toImplic f2  =|=
            (.~ toImplic f1 .-> toImplic f2)).
    apply or_to_implies.
  apply equiv_trans with (g:= toImplic f1 .\/ toImplic f2 ). auto. auto.
 simpl. apply equiv_cong_implies. auto. auto.
Qed.
```

Chapter 6

Deductive Systems

In this chapter we introduce deductive systems. We start introducing an axiomatic deductive system for the implicative fragment of the classical propositional logic. After defining it and presenting some example of deductions, we prove a syntactic variant of the deduction theorem presented in Chapter 4.

6.1 Deductive systems

A *deductive system* is a collection of rules that allow one to assert certain facts (judgements). The deductive system we will be interested here are to ones that allow one to asssert logic entailment. For the classical propositional logic there are many distinct deductive system avaliable: axiomatic system, natural deduction system, sequent calculus, analytic tableaux, resolution, etc. In particular we will detail the *axiomatic method* and its properties, in particular because it allows one of the simplest proofs for its meta-theory properties.

A deductive system allows one to judge, by means of a finite set of rules, when a given formula A is a deductive (or syntactic) consequence of a theory Γ, which we denote

$$\Gamma \vdash \mathsf{A}$$

The main distinction between *logical consequence* $\Gamma \vDash \mathsf{A}$ and *deductive consequence* $\Gamma \vdash \mathsf{A}$ is that the former requires one to consider all possible valuations that satisfy Γ to conclude that A is entailed by Γ, while the latter arrives at such conclusion by *constructing a proof* that certifies this fact. Since the rules of the deductive system generaly follows the syntax of formulas in Γ and A, $\Gamma \vdash \mathsf{A}$ is also referred as *syntactic entailment* and, conversely, $\Gamma \vDash \mathsf{A}$ is referred as *semantic entailment*.

6.2 Axiomatic deductive system

Axiomatic deductive systems, also known as *Hilbert systems*, rely on a collection of *axiom schemas* and one rule of deduction: *modus ponendu ponens*, or simply *modus ponens*. A deduction $\Gamma \vdash A$ is a sequence of formulas obtained from rules in Γ, axiom instantiations and application of *modus ponens* over previous rules, which ends in formula A. We now begin to discuss each of these components in detail.

We start with axiom schemas: they are essentically an infinite collection of formulas containing the same common syntactic pattern. For instance,

$$ax_1 = A \Rightarrow (B \Rightarrow A)$$

is a pattern that is common to all of the following formulas:

- $p \Rightarrow (p \Rightarrow p)$, for $A = p$ and $B = p$
- $p \Rightarrow (r \Rightarrow p)$, for $A = p$ and $B = r$
- $(q \wedge r) \Rightarrow ((p \Rightarrow p) \Rightarrow (q \wedge r))$, for $A = (q \wedge r)$ and $B = (p \Rightarrow p)$

By substituting A and B in ax_1 by two arbitrary formulas, we obtain distinct *instances* of this pattern. For the implicative fragment of classical propositional logic, we will work with the following collection of axiom schemas:

- $ax_1 = A \Rightarrow (B \Rightarrow A)$
- $ax_2 = (A \Rightarrow (B \Rightarrow C)) \Rightarrow (A \Rightarrow B) \Rightarrow (A \Rightarrow C)$
- $ax_3 = (\neg A \Rightarrow B) \Rightarrow (\neg A \Rightarrow \neg B) \Rightarrow A$

In the axiomatic deduction system, a *deduction* of A from a theory Γ is a finite sequence $D = A_1, \ldots, A_n$ of formulas such that

- $A_n = A$
- any formula $A_i \in D$ is either
  1. an arbitrary instantiation of an axiom (ax_1, ax_2 or ax_3).
  2. a premisse in Γ
  3. the consequence of the MP rule, presented below, over two other formulas $A_x = B \Rightarrow C$ and $A_y = B$ ocurring previously in the deduction.

$$\frac{A_x = B \Rightarrow C \qquad A_y = B \qquad x, y < i}{A_i = C} \quad \text{(MP)}$$

We say $\Gamma \vdash A$, i.e. A derives from Γ, whenever exists at least one deduction D of A from Γ. For example, considering $\Gamma = \{p, p \Rightarrow q, q \Rightarrow r\}$, we can derive $p \Rightarrow r$, written

$$p,\ p \Rightarrow q,\ q \Rightarrow r \vdash p \Rightarrow r$$

because it is possible to construct the following deduction:

$$\begin{aligned}
&A_1 = p && (\text{premisse}_1)\\
&A_2 = p \Rightarrow q && (\text{premisse}_2)\\
&A_3 = q && (\text{MP } A_2, A_1)\\
&A_4 = q \Rightarrow r && (\text{premisse}_3)\\
&A_5 = r && (\text{MP } A_4, A_3)\\
&A_6 = r \Rightarrow (p \Rightarrow r) && (\text{ax}_1)\\
&A_7 = p \Rightarrow r && (\text{MP } A_6, A_5)
\end{aligned}$$

A formula A that can be deduced from the empty theory $\Gamma = \{\}$ is called a *theorem*. For example, $p \Rightarrow p$ is a theorem, written

$$\vdash p \Rightarrow p$$

because there exists the following deduction of $(p \Rightarrow p)$ from $\{\}$:

$$\begin{aligned}
&A_1 = (p \Rightarrow ((p \Rightarrow p) \Rightarrow p)) \Rightarrow ((p \Rightarrow (p \Rightarrow p)) \Rightarrow (p \Rightarrow p)) && (\text{ax}_2)\\
&A_2 = p \Rightarrow ((p \Rightarrow p) \Rightarrow p) && (\text{ax}_1)\\
&A_3 = (p \Rightarrow (p \Rightarrow p)) \Rightarrow (p \Rightarrow p) && (\text{MP } A_1, A_2)\\
&A_4 = p \Rightarrow (p \Rightarrow p) && (\text{ax}_1)\\
&A_5 = p \Rightarrow p && (\text{MP } A_3, A_4)
\end{aligned}$$

Let us now represent these concepts in Coq. Since we have proved in the previous chapter that all truth-tables can be represented using only the implicative fragment of classical propositional logic, we need to reproduce the notions of formula and entailment for such fragment.

```
Inductive Impformula : Set :=
 | ImpLit     : nat -> Impformula
 | ImpNeg     : Impformula -> Impformula
 | ImpImplies : Impformula -> Impformula -> Impformula
.

Fixpoint toFormula (i:Impformula) : formula :=
  match i with
  |   ImpLit x       => Lit x
  |   ImpNeg a       => Neg (toFormula a)
  |   ImpImplies a b => Implies (toFormula a) (toFormula b)
  end.

Notation " X ;-> Y "  := (ImpImplies X Y) (at level 13, right associativity).
Notation " ;~ X "     := (ImpNeg X)       (at level 10, right associativity).
Notation " ;# X "     := (ImpLit X)       (at level 1, no associativity).


Fixpoint ImpformulaSAT (p: nat -> Prop) (f:Impformula) : Prop :=
match f with
| ImpLit     x         => p x
| ImpNeg     x1        => ~ (ImpformulaSAT p x1)
| ImpImplies x1 x2     => (~ (ImpformulaSAT p x1)) \/ (ImpformulaSAT p x2)
end.


Definition Imptheory := list Impformula.

Fixpoint ImptheorySAT (v:nat -> Prop) (Gamma:Imptheory) : Prop :=
match Gamma with
  | nil      => True
  | cons h t => (ImpformulaSAT v h) /\ (ImptheorySAT v t)
end.

Definition Impentails (Gamma:Imptheory) (A:Impformula) : Prop :=
  forall (v: nat -> Prop), ImptheorySAT v Gamma -> ImpformulaSAT v A.

Notation " G ;|= A " := (Impentails G A) (at level 110, no associativity).
```

Now, let us represent the set of possible instantiated axioms and their respective parameters. Let us define a function that performs the substitution on the axiom schema and obtain the resulting formula.

```
Inductive axiom : Set :=
| ax1 : Impformula -> Impformula -> axiom
  (* A -> (B -> A) *)
```

```
| ax2 : Impformula -> Impformula -> Impformula -> axiom
  (* (A -> B -> C) -> (A->B) -> (A -> C) *)

| ax3 : Impformula -> Impformula -> axiom
  (* (~A -> B) -> (~A -> ~B) -> A *)
.

Fixpoint instantiate (a:axiom) : Impformula :=
match a with
| ax1  p1 p2    => p1 ;-> (p2 ;-> p1)
| ax2  p1 p2 p3 => (p1 ;-> p2 ;-> p3) ;-> (p1 ;-> p2) ;-> (p1 ;-> p3)
| ax3  p1 p2    => (;~ p1 ;-> p2) ;-> (;~p1 ;-> ;~ p2) ;-> p1
end.

Definition exi1 := ;#0 ;-> ;#1 .
Definition exi2 := ;#1 ;-> ;~ ;#2 ;-> ;#0 ;-> (;#1 ;-> ;#2).
Definition exi3 := ;~;~ ;#0 ;-> ;#1.
```

Finally, let us define a datatype for deductions.

```
Inductive deduction : Imptheory -> Impformula -> Set :=

| Prem : forall (t:Imptheory) (f:Impformula) (i:nat),
                (nth_error t i = Some f) -> deduction t f

| Ax   : forall (t:Imptheory) (f:Impformula) (a:axiom),
                (instantiate a = f) -> deduction t f

| Mp   : forall (t:Imptheory) (f g:Impformula)
                (d1:deduction t (f ;-> g)) (d2:deduction t f), deduction t g
.

Notation " G ;|- A " := (exists d, d:deduction G A) (at level 110, no
 ↪  associativity).

Definition th1 := (;#0 :: ;#0 ;-> ;#1 :: nil).
Definition ded1 := (Prem th1         ;#0 0 eq_refl).
Definition ded2 := (Prem th1 (;#0 ;-> ;#1) 1 eq_refl).
Definition ded3 := (Mp   th1 ;#0 ;#1 ded2 ded1).
Check ded3.
Definition ded4 := (Ax th1 (;#1 ;-> ;#2 ;-> ;#1) (ax1 ;#1 ;#2) eq_refl).
Definition ded5 := (Mp th1  ;#1 (;#2 ;-> ;#1) ded4 ded3).
Check ded5.
```

Let us consider a few choices of this particular definition of deduction:

- note that theory is a dependent parameter that is kept constant along the constructor for the deduction;
- the presented inductive datatype for deductions in Coq describe actually a tree, not a list. The tree branches in each application of the *modus ponens* rule.
- the chosen representation in Coq makes the definition of deduction and associated proofs by structural induction easier. In fact, we can actually see the list definition as an optimization on the tree definition, allowing sharing of intermediate results between distinct branches.

Melhorar a notação de deduções

Usar let ... in ... em Coq para relacionar árvores e deduções lineares

As an example, we show the proof of $\Gamma \vdash A \Rightarrow A$ for all possible Γ and A as the following Coq lemma.

```
Lemma derive_self :
   forall (a:Impformula) (G:Imptheory), (deduction G (a ;-> a)).
Proof.
intros.
apply Mp with (f:=a;->a;->a).
  apply Mp with (f:=a;->(a;->a);->a).
  apply Ax with (a:=(ax2 a (a;->a) a)). simpl. auto.
  apply Ax  with (a:=(ax1 a (a;->a))). simpl. auto.
  apply Ax with (a:=(ax1 a a)). simpl. auto.
Qed.
```

6.3 Structural properties of syntactic entailment

Reflexivity: we can reproduce the result $\Gamma, A \vdash A$ for all A and Γ.

```
Lemma derive_refl :
 forall (G:Imptheory) (a:Impformula), deduction (a::G) a.
Proof.
intros.
apply Prem with (i:=0).
simpl.
auto.
Qed.
```

Weakening

```
Fixpoint addTheory (x:Impformula) (a:Impformula) (G:Imptheory)
                   (d1: deduction G a) : (deduction (x::G) a) :=
match d1 with
| Prem _ f i d    => Prem (x::G) f (S i) d
| Ax _ f a d      => Ax (x::G) f a d
| Mp  _ f g d1 d2  => Mp (x::G) f g (addTheory x (f ;-> g) G d1) (addTheory x
↪  f G d2)
end.

Lemma derive_weakening_left :
  forall (H:Imptheory) (a:Impformula) (G:Imptheory) (d1: deduction G a),
     (deduction (H ++ G) a).
Proof.
intros H.
induction H.
 - simpl. intros. auto.
 - simpl. intros.
   pose d1. apply IHlist in d. apply addTheory. auto.
Qed.
```

Transitivity:

TODO

6.4 The syntactic deduction theorem

The semantic deduction theorem presented in Chapter 4 states

$$\Gamma, A \vDash B \quad \text{iff} \quad \Gamma \vDash A \Rightarrow B$$

exposing the relation between increasing a theory and introducing the $\Rightarrow$ operator. There is a *syntactic deduction theorem*, which states a similar fact in the context of the axiomatic deduction system:

$$\Gamma, A \vdash B \quad \text{iff} \quad \Gamma \vdash A \Rightarrow B$$

The proof of these two theorems, however, are distinct, since the syntatic version refers to *proof construction* and the semantic version refers to *formula valuations*. The syntactic deduction theorem can be rephrased as *there is a proof of* B *from* Γ, A *whenever there is a proof of* $A \Rightarrow B$ *from* Γ*, and vice-versa.*

```
(* Deduction theorem for Hilbert systems *)

Theorem deduction_theorem_hilbert_1 :
   forall (G:Imptheory) (A B:Impformula) (d1: deduction (A::G) B), (deduction
   ↪  G (A ;-> B)).
Proof.

  * intros. induction d1.
    - subst.
      destruct i.
       + simpl in *. inversion e. apply derive_self.
       + simpl in *. assert (deduction G f). apply Prem with (i:=i). auto.
                     assert (deduction G (f ;-> (A ;-> f))). apply Ax with
                      ↪  (a:=ax1 f A). simpl. auto.
                     apply Mp with (f:=f). auto. auto.
    - subst.
        assert (deduction G  (instantiate a)). apply Ax with (a:=a). auto.
                     assert (deduction G (instantiate a ;-> (A ;-> instantiate
                      ↪  a))). apply Ax with (a:=ax1 (instantiate a) A).
                      ↪  simpl. auto.
                     apply Mp with (f:=instantiate a). auto. auto.
    - subst.
       apply Mp with (f:=A;-> f).
```

```
        apply Mp with (f:=A;-> f;-> g).
        apply Ax with (a:=(ax2 A f g)). simpl. auto. auto. auto.
Qed.



Theorem deduction_theorem_hilbert_2 :
   forall (G:Imptheory) (A B:Impformula) (d1: deduction G (A ;-> B)),
    ↪  (deduction (A::G) B).
Proof.
 intros.
 assert (deduction (A::G) (A;->B)). apply addTheory. auto.

 assert (deduction (A::G) A). apply Prem with (f:=A) (i:=0). simpl. auto.

 apply Mp with (f:=A). auto. auto.
Qed.
```

Chapter 7

Soundness and completeness

Given a semantic notion of entailment $\Gamma \vDash A$ and a syntactic notion of entailment $\Gamma \vdash A$, it would be natural to wish that both judgements agree. We can capture this agreement by means of two important properties of a deductive system with respective to a semantics: soundness and completeness.

Soundness: we say a deductive system $\vdash$ is *sound* with respect to $\vDash$ whenever $\Gamma \vdash A$ implies $\Gamma \vDash A$. In other words, all possible deductions of a formula A from a theory Γ consist are valid semantic entailments.

Completeness: we say a deductive system $\vdash$ is *complete* with respect to $\vDash$ whenever $\Gamma \vDash A$ implies $\Gamma \vdash A$. In other words, all entailments of a formula A from a theory Γ have at least one deduction in $\vdash$.

If we have a sound and complete deductive system, we can use it as a substitute for the semantic entailment relation. Due to the rule-based nature of deductive systems, they are in general easier to manipulate when compared to valuation-based semantic definitions.

It is important to notice that these properties are not automatic. When constructing a new deductive system, it is possible to achieve only soundness, only completeness or even neither of them. For instance, a deductive system that allows to deduct $\Gamma \vdash A$ for all Γ and A is *complete*, but not sound. An empty deductive system, in which $\Gamma \nvdash A$ for any Γ and A is *sound*, but not complete. A deductive system that only states $\Gamma \vdash p \land \neg p$ for every theory Γ and literal p is not sound neither complete. Although it is possible to build deductive systems without one of these properties, they tend to not be very useful from a practical perspective.

7.1 Soundness

Now let us prove that the axiomatic deductive system of Chapter 6 is **sound**, i.e.

$$\text{if} \quad \Gamma \vdash A \quad \text{then} \quad \Gamma \models A$$

Proof: by induction on the construction of proofs. We need to

- ensure that, if A is an axiom instantiations, then it is valid (i.e. $\models A$). This fact, together with weakening, allows to ensure that $\Gamma \models A$ holds for any Γ.
- ensure that $A \in \Gamma$ is sufficient to ensure $\Gamma \models A$
- ensure that, if $\Gamma \models A \Rightarrow B$ and $\Gamma \models A$ are valid entailments, then $\Gamma \models B$ holds.

Chapter 8

Conclusions

We conclude that Coq is cool!

Bibliography

[1] The Agda Development Team. Agda's documentation. v2.6.0.1.

[2] The Coq Development Team. *The Coq Reference Manual. Release 8.9.0.* 2019.