Logic and Functional Programming

Florent Schaffhauser

2026.05.15

Table of contents

Disclaimer

This is a draft script for the lecture course *Logic and functional programming* held at Heidelberg University in the Summer semester of 2026. It is intended for the students of the course and not apt for wider distribution.

Course announcement

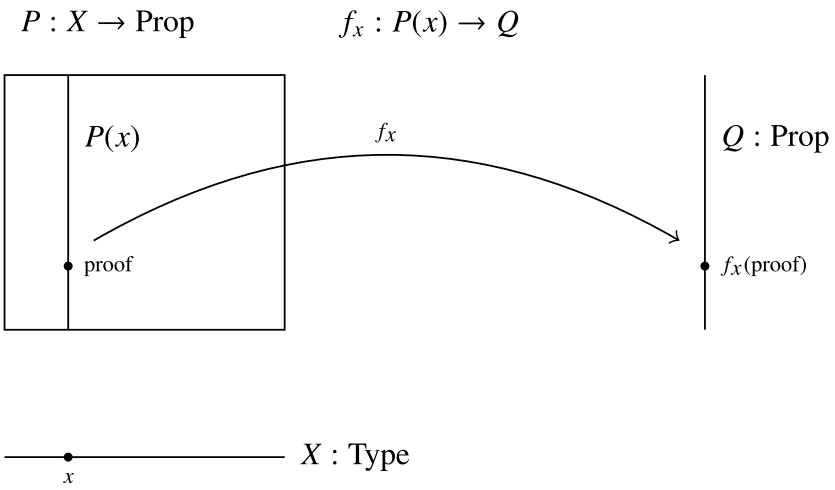

Figure 1: The elimination rule for the existential quantifier

Time and place

- **Lectures:** Tuesday-Thursday 14:00-16:00 in SR 3 (Mathematikon).
- **Exercises:** Monday 16:00-18:00 in SR 4 (Mathematikon).

Overview

Martin-Löf's type theory provides a common **foundation** for mathematics, logic and functional programming. In this course, we give an account of the basics of **type theory** with a view towards applications in **mathematics**. The course will combine a theoretical exposition of the concepts with their implementation in a proof assistant.

- Target audience: Bachelor students of mathematics or computer science.
- Instructor: Florent Schaffhauser.
- Language of instruction: English.

Syllabus

- Types and terms. Judgments and contexts.
- Inference rules. Natural deduction.
- Propositional calculus. Decidability.
- Axioms. Models. Soundness.
- Inductive types. Recursion. Pattern matching.
- Type families. Dependent pairs.
- Dependent function types. Induction.
- Propositions as types. Predicates. Subtypes.
- Relations. Ordered structures.
- Identity types. Sets.
- Number systems. Elementary arithmetic.
- Algebraic data types.
- Type equivalences.
- Propositional truncation. Principle of unique choice.

Proof assistants

As a means to experiment with the logic and explore formal proofs of mathematical statements, students will be taught to use a proof assistant. At the end of the course, they will have developed an understanding of the type theory and logic underpinning modern proof assistants and acquired basic programming skills related to the formalisation of mathematics and software verification.

```
From Stdlib Require Import Utf8.

Theorem exists_elim (X : Set) (P : X → Prop) (Q : Prop) :
    (∀ x : X, P x → Q) ∧ (∃ x : X, P x) → Q.
Proof.
  intro term.
  destruct term as [f [x proof]].
  apply (f x).
  exact proof.
Qed.

Check exists_elim.
Print exists_elim.
```

Prerequisites

There are no formal prerequisites for this course but familiarity with abstract reasoning and algebraic structures might be helpful.

Evaluation

The course is completed with a graded oral or written exam. To be admitted to the exam, students will need to collect at least 50% of the points from the graded problem sets. The final grade is determined by the grade of the exam. The requirements for the assignment of credits follows the regulations in section modalities for examinations.

1 Well-formed formulas

Lecture 1 deals with the basic *syntax of propositional logic*. We define expressions called *well-formed formulas*, in two equivalent ways:

1. Inductively, by listing all the ways to introduce such expressions.
2. As a subset of a larger set of more general expressions.

1.1 Inductively

The type of **well-formed formulas** is the collection Wff defined inductively as follows:

- Every so-called *basic proposition* $P_0, P_1, P_2, ...$ is a well-formed formula.
- If F and F' are well-formed formulas and $\diamond$ is one of the symbols $\Rightarrow$, $\wedge$, $\vee$, then $F \diamond F'$ is a well-formed formula.
- If F is a well-formed formula, then $\neg F$ is a well-formed formula.

This makes the functions $P : \mathbb{N} \to \text{Wff}$, $(\Rightarrow)$, $(\wedge)$ and $(\vee) : \text{Wff} \times \text{Wff} \to \text{Wff}$ *constructors* for well-formed formulas. For instance, $(\Rightarrow)(F, F') := F \Rightarrow F'$. Note that the codomain of such a constructor is always Wff.

Remarks.

- Here I am using Haskell syntax, to distinguish between *infix* notation such as $F \Rightarrow F'$ and *prefix* notation such as $(\Rightarrow)(F, F')$. This is just for convenience, but it also helps us realize that we need to use a different piece of notation for each syntax.
- During the lecture, we also discussed informally the tree representation of a well-formed formula, as well as associativity rules for the binary constructors: by convention, $F \Rightarrow F' \Rightarrow F''$ is parsed as $F \Rightarrow (F' \Rightarrow F'')$, and is thus a well-formed formula, different from $(F \Rightarrow F') \Rightarrow F''$.
- Similarly, an expression such as $P_1 \wedge P_2 \vee P_3$ shoulb be parsed as $(P_1 \wedge P_2) \vee P_3$, not $P_1 \wedge (P_2 \vee P_3)$ because, *by convention*, the binary operator $\wedge$ has *higher precedence* than $\vee$.

  We can choose to add one more constructor $(\Leftrightarrow) : \text{Wff} \times \text{Wff} \to \text{Wff}$. This simply says that if F and F' are well-formed formulas, then $F \Leftrightarrow F'$ is a well-formed formula. At this stage, there is no relation between the well-formed formulas $F \Leftrightarrow F'$ and $(F \Rightarrow F') \wedge (F' \Rightarrow F)$.

We could add more constructors yet, such as quotation marks ‘ . ’ : String → Wff. Then ‘It is raining’ and ‘23?x@’ would also be well-formed formulas. We will not do that. Note that our basic propositions are indexed by the natural numbers (the corresponding constructor is $P : \mathbb{N} \to \text{Wff}$) but that $\mathbb{N}$ could be replaced by an arbitrary set I, possibly a finite one.

1.2 As a subset

Given a set of basic propositions $\mathcal{B} := \{P_0, P_1, P_2, ...\}$ and a set of logical connectives $\mathcal{C} := \{\Rightarrow, \wedge, \vee, \Leftrightarrow, \neg\}$, we form the set $\mathcal{V} := \mathcal{B} \cup \mathcal{C} \cup \{(\ ,)\}$, called an **alphabet**, and the set

$$\mathcal{V}^* := \mathcal{V}^0 \cup \mathcal{V}^1 \cup \mathcal{V}^2 \cup ...$$

of **lists** of elements of $\mathcal{V}$. Here, for all natural number n, the set $\mathcal{V}^n$ is the set of lists of elements of $\mathcal{V}$ that have length n. In particular, the set $\mathcal{V}^0$ is a singleton, containing only the empty list: $\mathcal{V}_0 = \{[\,]\}$. The set $\mathcal{V}^*$ contains lists such as $(P_0 \Rightarrow \neg P_1)$, but also $P_3)) \Rightarrow \neg$, which is not an expression we want to include in our definition of a well-formed formula.

> We define the set $\mathcal{F}(\mathcal{V})$ of **propositional formulas** based on the alphabet $\mathcal{V}$ as the smallest subset $\mathcal{W} \subset \mathcal{V}^*$ satisfying the following properties:
>
> 1. $\mathcal{B} \subset \mathcal{W}$.
> 2. If $F \in \mathcal{W}$ and $F' \in \mathcal{W}$, then, for all $\diamond \in \{\Rightarrow, \wedge, \vee, \Leftrightarrow\}$, $(F \diamond F') \in \mathcal{W}$.
> 3. If $F \in \mathcal{W}$ then $(\neg F) \in \mathcal{W}$.

Explicitly:

$$\mathcal{F}(\mathcal{V}) := \bigcap_{\mathcal{W} \subset \mathcal{V}^* \ |\ \mathcal{W} \text{ satisfies 1, 2, 3}} \mathcal{W}$$

Note that $\mathcal{V}^*$ satisfies Conditions 1, 2, 3, so the set $\{\mathcal{W} \subset \mathcal{V}^* \mid \mathcal{W} \text{ satisfies 1, 2, 3}\}$ is non-empty. Moreover, $\mathcal{B} \subset \mathcal{F}(\mathcal{V})$, so $\mathcal{F}(\mathcal{V}) \neq \emptyset$.

Also note that the empty list $[\,]$ (which is an element of $\mathcal{V}^*$) is not an element of $\mathcal{F}(\mathcal{V})$. If it were, then $\mathcal{F}(\mathcal{V}) \setminus \{[\,]\}$ would also satisfy Conditions 1, 2 and 3 above, contradicting the minimality of $\mathcal{F}(\mathcal{V})$.

1.3 Equivalence of the two constructions

We claim without proof that $\mathcal{F}(\mathcal{V}) = \text{Wff}$ as subsets of $\mathcal{V}^*$, or more accurately that there exists a bijection between these two sets (because Wff is not really a subset of $\mathcal{V}^*$).

2 Height and sub-formulas

2.1 Yet another construction of the set of propositional formulas

We give one more construction of $\mathcal{F}(\mathcal{V})$, again as a subset of $\mathcal{V}^*$. To avoid confusion, we will denote that subset by $\mathcal{F}$ and then prove that $\mathcal{F} = \mathcal{F}(\mathcal{V})$.

Let $(\mathcal{F}_n)_{n\in\mathbb{N}}$ be the *sequence* of subsets of $\mathcal{V}^*$ defined as follows:

- $\mathcal{F}_0 := \mathcal{B}$ (basic propositions).
- $\mathcal{F}_{n+1} :=$

  $\mathcal{F}_n \cup \{\neg F \text{ where } F \in \mathcal{F}_n\} \cup \{F \diamond F' \text{ where } F, F' \in \mathcal{F}_n \text{ and } \diamond \in \{\Rightarrow, \wedge, \vee, \Leftrightarrow\}\}$.

Then we set

$$\mathcal{F} := \bigcup_{n\in\mathbb{N}} \mathcal{F}_n.$$

The construction of $\mathcal{F}$ as $\cup_{n\in\mathbb{N}}\mathcal{F}_n$ is often referred to as a construction "from below", as opposed to the construction "from above" of $\mathcal{F}(\mathcal{V})$ seen in Lecture 1. Note that for all natural number n, we have $\mathcal{F}_n \subset \mathcal{F}_{n+1}$ and that the union defining $\mathcal{F}_{n+1}$ is a disjoint union (exercise!).

We then claim that $\mathcal{F} = \mathcal{F}(\mathcal{V})$, the latter being the set of propositional formulas constructed in Lecture 1. This is a good exercise, where you have to prove a double inclusion and use induction on n to prove one of them.

2.2 Height

The height is supposed to measure the complexity of a formula: basic propositions have height 0 and the higher the height of a compound formula, the more complex it is. Given a propositional formula F, we define the **height** of F as

$$\mathrm{ht}(F) := \min\{n \in \mathbb{N} \mid F \in \mathcal{F}_n\}.$$

This defines a function $\mathrm{ht} : \mathcal{F} \to \mathbb{N}$ satisfying $\mathrm{ht}(F) = 0$ if $F = P_i$ for some i and $h(F) = n+1$ if $F \in \mathcal{F}_{n+1}$ but $F \notin \mathcal{F}_n$.

Note that $F \in \mathcal{F}_{\mathrm{ht}(F)}$ but $F \notin \mathcal{F}_{\mathrm{ht}(F)-1}$ and that this characterises $\mathrm{ht}(F)$.

The upshot of the construction of the set of propositional formulas as a union indexed by the natural numbers is that we can use induction on the height to prove properties about propositional formulas.

However, the definition of ht is not excessively convenient for practical computations or implementation in a functional programming language. The next construction remedies that and is more direct: the induction is performed *directly on the well-formed formula* F (this is a consequence of the fact that the type Wff is defined inductively).

2.3 Computational approach to the height

Recall that $\mathcal{F} = \mathcal{F}(\mathcal{V}) = \text{Wff}$. Using Wff, we can construct the height function by *pattern matching* on constructors of well-formed formulas, which is done as follows (we will be using the same notation ht as above, to denote the height function).

Given a well-formed formula $F : \text{Wff}$, we set:

1. $\text{ht}(P_i) := 0$ if $F = P_i$ for some natural number i.
2. $\text{ht}(\neg G) := 1 + \text{ht}(G)$ if $F = \neg G$ for some well-formed formula G.
3. $\text{ht}(G \diamond G') := 1 + \max(\text{ht}(G), \text{ht}(G'))$ if $F = G \diamond G'$ for some well-formed formulas G, G' and some binary connective $\diamond \in \{\Rightarrow, \wedge, \vee, \Leftrightarrow\}$.

Note that the height function is defined *recursively*. More precisely, it is defined via *primitive recursion*. This just means that the number of constructors in the expression for the argument of the function diminishes by exactly one at each step (see how $P, \neg$ and $\diamond$ disappear when passing from the left-hand side to the right-hand side).

It is not entirely straightforward to check that the two definitions of the height function coincide when Wff and $\mathcal{F}(\mathcal{V})$ are identified. In this course, we will favour the second definition and see the height as a function $\text{ht} : \text{Wff} \to \mathbb{N}$. As seen in examples, the quantity $\text{ht}(F)$ then computes out nicely.

2.4 Sub-formulas

Similarly to the height, we can define the collection of sub-formulas of a given formula F in two different ways, depending on whether we think of F as an element of $\mathcal{F}(\mathcal{V})$ or Wff.

2.4.1 Set-theoretic approach

We define a function $\text{sf} : \mathcal{F} \to \mathcal{P}(\mathcal{F})$ from the set of propositional formulas to the power set of that set.

Using that $\mathcal{F} = \cup_{n\in\mathbb{N}} \mathcal{F}_n$, we first define a sequence of functions $(\text{sf}_n)_{n\in\mathbb{N}}$ such that $\text{sf}_n : \mathcal{F}_n \to \mathcal{P}(\mathcal{F})$ satisfying, for all natural number n, the equality $\text{sf}_{n+1}|_{\mathcal{F}_n} = \text{sf}_n$, and then set

$\mathrm{sf}(F) := \mathrm{sf}_{\mathrm{ht}(F)}(F)$. The resulting (well-defined!) function sf| satisfies, for all natural number n, the condition $\mathrm{sf}|_{\mathcal{F}_n} = \mathrm{sf}_n$.

1. For all natural number i, we set $\mathrm{sf}_0(P_i) := \{P_i\}$.
2. For all natural number n and all propositional formula $F \in F_{n+1}$, we set

- $\mathrm{sf}_{n+1}(F) := \mathrm{sf}_n(F)$ if $F \in F_n \subset F_{n+1}$.
- $\mathrm{sf}_{n+1}(F) := \{\neg G\} \cup \mathrm{sf}_n(G)$ if $F = \neg G$ with $G \in F_n$.
- $\mathrm{sf}_{n+1}(F) := \{G \diamond G'\} \cup \mathrm{sf}_n(G) \cup \mathrm{sf}_n(G')$ if $F = G \diamond G'$ with $G, G' \in F_n$ and $\diamond \in \{\Rightarrow, \wedge, \vee, \Leftrightarrow\}$.

Note that if $\mathrm{sf}(G) \cap \mathrm{sf}(G') \neq \emptyset$, then any element at the intersection will only be listed once in the subset $\mathrm{sf}(G \diamond G')$. This will not be the case in the approach below, where we use lists of well-formed formulas instead of the power set.

2.4.2 Type-theoretic approach

We now define a function strict_sf : Wff $\to$ List Wff from well-formed formulas to lists of well-formed formulas. The goal is to send a well-formed formula F to the list of formulas *strictly contained* in F. As in the definition of the height function, there are essentially three cases to consider.

1. strict_sf$(P_i) :=$ nil
2. strict_sf$(\neg G) := G ::$ strict_sf(G).
3. strict_sf$(G \diamond G') := (G ::$ strict_sf$(G)) +\!\!+ (G' ::$ strict_sf$(G'))$.

In the formulas above, the empty list is denoted by nil and the function cons : Wff $\times$ List Wff $\to$ List Wff (= the constructor for so-called *linked lists*) is denoted by _ :: _ (with infix notation). The binary operator $+\!\!+$ denotes the concatenation of lists.

With these conventions, the function strict_sf is defined by pattern matching on F and it is a (primitive) recursive function. Then we define a function sf : Wff $\to$ ListWff by prepending F to strict_sf(F).

$\mathrm{sf}(F) := [F] +\!\!+$ strict_sf(F).

In a forthcoming lecture, we will implement the function sf in Rocq.

2.5 Height and sub-formulas in the tree representation of a formula

As an exercise, take $F := (P_1 \vee P_2) \wedge \neg P_3$ and compute $\mathrm{ht}(F)$ and $\mathrm{sf}(F)$. Represent F as a tree whose leafs (at the bottom) are the atomic sub-formulas P_1, P_2, P_3 of F, whose nodes are the connectives $\vee, \wedge$ and $\neg$ appearing in F, and whose sub-trees are labelled by (or in bijection with) sub-formulas of F (the whole tree corresponding to $[F]$). Note that the height of the formula F corresponds to how "tall" the tree is.

3 Boolean values and pattern matching

3.1 Booleans in Rocq's core library

Boolean values are available without any imports in Rocq. We have two canonical Boolean values, called `true` and `false`.

```
Check true.                    (* true : bool *)
Check false.                   (* false : bool *)
```

The type of Booleans is called `bool` and is recognised as something called `Set` in Rocq. We will conveniently ignore this for now.

```
Check bool.                    (* bool : Set *)
About bool.
Check Set.
Check Type.
```

Certain pre-defined functions are also available out of the box and can be used to compute expressions.

```
Check negb.                    (* negb : : bool -> bool *)
About negb.

Check negb false.              (* negb false : bool *)
Compute negb false.            (* = true *)
```

A function of two variables like `orb` are written in **curried notation**. This means that the value or `orb` on a pair `such as (true , false)` is written `orb true false`, not `orb (true , false)`. In turn, the expression `orb true`, for instance, denotes a *function* from `bool` to `bool`.

```
Check orb true false.
Check orb.                     (* orb : bool -> bool -> bool *)
Check bool -> (bool -> bool).  (* bool -> bool -> bool : Set *)
Check orb true.                (* orb true : bool -> bool *)
```

We can compute using this syntax.

```
Compute negb (orb true false).  (* = false *)
```

In the rest of the file, we construct basic functions from `bool` to `bool`. Most of them, if not all, are contained in Rocq's core library, but we do this to learn how to use **pattern matching** in order to construct functions.

3.2 Basic pattern matching

To avoid conflicts with Rocq's core library, we wrap everything in a module that we choose to call `BooleanValues`. This is optional here, since we do not intend to use the current file anywhere else later on.

```
Module BooleanValues.
```

3.2.1 Boolean NOT

Let us start with the definition of the `negb` function, which sends `true` to `false` and vice versa. We define it by pattern matching on Boolean values. Note that we can use the same name and that this does not cause any conflict with the existing Rocq `negb` function.

```
Definition negb (b : bool) : bool :=
  match b with
  | true  => false
  | false => true
  end.

Check negb.                          (* negb : bool -> bool *)
```

From now on, when we write `negb`, it will refer to the `negb` function defined above. You can check this by modifying the definition (sending everything to `false`, for instance, and computing a few values.

```
About negb.
Compute negb false.                  (* = true *)
```

We can chain also give expressions a name to manipulate them in computations.

```
Compute negb (negb false).           (* = false *)

Definition example_bool := negb false.

Compute negb example_bool.           (* = false *)
```

3.2.2 Boolean AND

Now we want to define an `andb` function, that takes two Booleans `b, b'` and returns `andb b b'`. The value of `andb b b'` a priori depends on the values of `b` and `b'`, so we have four cases to consider when pattern matching.

```
Definition andb (b b' : bool) : bool :=
  match b, b' with
  | true,  true  => true
  | true,  false => false
  | false, true  => false
  | false, false => false
  end.
```

In fact, some simplifications are possible.

```
match b, b' with
| true,  b' => b'
| false, _  => false
end.
```

Another way of doing the pattern matching, that you may want to avoid, is pattern matching first on `b`, then on `b'`, which results in a nested program. It typechecks, though. Note that the only the final `end` has a period following it.

```
match b with
| true =>
  match b' with
  | true  => true
  | false => false
  end
| false =>
  match b' with
  | true  => false
  | false => false
  end
end.
```

As with `negb`, we can use `andb` to compute using Boolean values.

```
Compute andb true false.            (* = false *)
Compute andb true (negb example_bool).
  (* = false *)
```

Recall that `andb` is a function of two variables, but it is written in **curried notation**, and that you can check that as follows: if you pass `Check andb.`, the elaborator returns `andb : bool -> bool -> bool`, not `andb : bool × bool -> bool`. This means that an expression like `andb true`, for instance, is a *function* from `bool` to `bool`.

```
Check andb.                          (* andb : bool -> bool -> bool *)
Check andb true.                     (* andb true : bool -> bool *)
Compute andb true.
```

We see in particular, that arrow types associate to the right:

```
(bool -> bool -> bool) = (bool -> (bool -> bool))
```

which *forces* the following expression to be parsed as follows:

```
andb b b' = (andb b) b'
```

In contrast, the arrow type `(bool -> bool) -> bool` takes a function `f : bool -> bool` and returns a `bool`. We can define an example of a function with type signature `(bool -> bool) -> bool` by evaluating `f : bool -> bool` at a given Boolean value.‘

```
Definition eval_at_true : (bool -> bool) -> bool :=
  fun f => f true.

Compute eval_at_true negb.           (* = false *)
Compute eval_at_true (andb true).    (* =true *)

Definition eval_at (b : bool) : (bool -> bool) -> bool :=
  fun f => f b.

Compute eval_at false negb.          (* = true *)
Compute eval_at false (andb true).   (* = false *)
```

3.2.3 Boolean OR

Let us define an `orb` function on Booleans.

```
Definition orb (b b' : bool) : bool :=
  match b, b' with
  | true,  _  => true
  | false, b' => b'
  end.
```

In fact, we are not using `b'` as a pattern, so we can also write the following.

```
Definition orb (b b' : bool) : bool :=
  match b with
  | true  => true
  | false => b'
  end.

Compute orb false false.             (* = false *)
Compute orb false (negb false).      (* = true *)
```

3.2.4 Boolean implication

Let us define a function called implication (between two Boolean values). For this one, we will use Rocq's if-then-else function, which uses pattern matching behind the scenes. Note the syntax using mixfix notation.

```
Definition implb (b b':bool) : bool := if b then b' else true.
```

> **Exercise.** Implement the function `implb` using pattern matching. The result of the following computation should be the Boolean `true`.
>
> ```
> Compute implb false true.          (* = true *)
> ```

As one more exercise, implement the Boolean if-then-else function as a function with type signature `bool -> bool -> bool`.

3.3 Introduction to theorem proving

Next, we start using Rocq as a means to prove certain properties of the function we have defined. The syntax looks a bit different at first, but in fact we will see in the course that stating a theorem is not that different from declaring a function.

3.3.1 Negation is involutive

The first property that we will prove is that, for all Boolean value `b`, we have an equality 'negb (negb b) = b'. Let us start with some special cases, in which we prove our equality by a direct computation. Note the interactive **tactic state** on the right-hand-side of the editor, which guides us in our proof.

```
Theorem negb_negb_true_eq_true : negb (negb true) = true.
Proof.
  simpl.
    (* The `simpl` tactic simplifies the terms for you,
    essentially by computation. *)
  reflexivity.
    (* You can always use `reflexivity` to *test* whether the
    equality follows from a direct computation. *)
Qed.
```

If you are just getting started with theorem provers, you should focus here on the block that starts with `Proof` and ends with `Qed`. This is a Rocq program, like the one defining the `negb` or `andb` functions, but *written in a different language*, namely the tactic language `ltac`, which gets superposed onto the basic Rocq language, called *Gallina*.

If you try to prove something incorrect, the prover will let you know.

```
Theorem negb_negb_false_eq_false : negb (negb false) = false.
Proof.
  reflexivity.
Qed.
```

Here is the same program, written in Gallina. Note that it requires changing `Theorem` to `Definition`.

```
Definition negb_negb_false_eq_false' :
  negb (negb false) = false :=
eq_refl.
```

You can check that this is indeed the same program using the `Print` command.

```
Print negb_negb_false_eq_false.
Print negb_negb_false_eq_false'.
```

If you try to prove something incorrect, the prover will let you know.

```
Theorem fail_to_unify : negb (negb false) = true.
Proof.
  Fail reflexivity.
Admitted.
```

Now let us see how to use pattern matching in proofs. The `destruct` tactic plays the same role as the `match` keyword in Gallina. After that, there are two so-called **subgoals** to close and we can isolate each one by using a focusing dash (or plus sign). The `Qed` keyword is optional and lets you visually check that your proof is complete.

```
Theorem negb_inv (b : bool) : negb (negb b) = b.
Proof.
  destruct b.
  - simpl. reflexivity.
  - reflexivity.
    (* Note that the `simpl` tactic is not actually needed to
    test the equality. *)
Qed.
```

> Note that the `destruct` tactic changes the goal. Before it, the goal is `negb (negb b) = b`, while after it, there are tow sub-goals, one where `b` has been replaced by `true`, and one where `b` has been replaced by `false`.

You can also call previous proofs while doing a proof, using the `exact` tactic. In the proof above, for instance, you can use the following in place of the second `reflexivity` tactic.

```
destruct b.
  - simpl. reflexivity.
  - exact negb_negb_false_eq_false.
```

One may observe that, in this formalism, the expression `negb_inv true` is a proof of the equality `negb (negb true) = true`, and `negb_inv` is a proof of the proposition `∀ b : bool, negb (negb b) =b`.

```
Check negb_inv true.  (* negb (negb true) = true *)
Check negb_inv.       (* negb_inv : : forall b : bool, negb (negb b) = b *)
```

3.3.2 Equivalent definitions of implication

Recall the formulas

`(P ⇒ Q) ⇔ ¬P ∨ Q`

and

`(P ⇒ Q) ⇔ ¬(P ∧ ¬Q)`

from classical logic. The first is in fact often taken as a definition of implication in classical logic. We will further comment on this later in the course.

In what follows, we interpret ⇒ as `implb`, ¬ as `negb`, ∨ as `orb`, ∧ as `andb` and ⇔ as Rocq's equality `=` to prove that we can define `implb` in two more ways, equivalent to the choice we made in the sense that the resulting function of two variables takes the same value as `implb` on all `b b' : bool`.

This exercise also shows how we can structure a proof with pattern matching on several variables in it. First, we present a nested version: we first pattern match on `b` and then we pattern match on `b'` in each branch. This results in two cases (or branches, corresponding to the possible values of `b`) with two sub-cases each. (corresponding to the possible values of `b'`).

Here is the formalisation and proof of `(P ⇒ Q) ⇔ ¬P ∨ Q` using Booleans. In the second branch, we close all goals in one move.

```
Theorem implb_eq_orb_negb (b b' : bool) :
  implb b b' = orb (negb b) b'.
Proof.
  destruct b.
  - destruct b'.
    + reflexivity.
    + reflexivity.
  - destruct b'.
    all: reflexivity.
Qed.
```

And here is the formalisation and proof of `(P ⇒ Q) ⇔ ¬(P ∧ ¬Q)`. Here, we are *not* nesting the proof: we are pattern matching on `b` and `b'` simultaneously, right from the start, much like we did for the definition of `andb`, for instance. This immediately creates four cases, without any sub-cases. It so happens here, that we can close them all by reflexivity.

```
Theorem implb_eq_negb_andb_negb {b b' : bool} :
  implb b b' = negb (andb b (negb b')).
Proof.
  destruct b, b'.
  all: reflexivity.
Qed.
```

To conclude, we only need to close the module `BooleanValues` we opened earlier.

```
End BooleanValues.
```

4 A toy implementation of propositional formulas

To be able to use Rocq files that contain Unicode characters like

```
ℕ, ∧, ¬ or →
```

in our code, we add support for UTF8. The notation must be defined manually.

```
From Stdlib Require Import Utf8.

Definition ℕ := nat.
```

To input Unicode characters in VS Code/Codium, you can use extensions such as `latex-input` or `Unicode shortcuts`. In the online scratchpad, ASCII encoding like `->` will render as → on the screen, but if you copy-paste your code in a text file, you will see `->` again.

In the Rocq scratchpad, use the following block instead.

```
From Coq Require Import Utf8.

Definition ℕ := nat.
```

4.1 Well-formed formulas

Recall that the collection of well-formed formulas is the datatype `Wff` defined inductively as follows:

- Every so-called basic proposition `P₀, P₁, P₂, ...` is a well-formed formula.
- If `F` and `F'` are well-formed formulas and ◇ is one of the symbols `⇒, ∧, ∨`, then `F ◇ F'` is a well-formed formula.
- If `F` is a well-formed formula, then `¬F` is a well-formed formula.

We now declare the type of well-formed formulas in Rocq. Take a good look at the syntax of this declaration and compare it to the informal definition above.

```
Inductive Wff :=
  | P    : ℕ   → Wff
  | Neg  : Wff → Wff
  | Impl : Wff → Wff → Wff
  | Conj : Wff → Wff → Wff
  | Disj : Wff → Wff → Wff.
```

This in particular defines functions

```
P    : ℕ → Wff
Neg  : Wff → Wff
Impl : Wff → Wff → Wff
Conj : Wff → Wff → Wff
Disj : Wff → Wff → Wff
```

called *constructors*, all of whose codomain is `Wff`.

```
Check P.                          (* P : ℕ → Wff *)
Check Neg.                        (* P : Wff → Wff *)
Check Impl.                       (* P : Wff → Wff → Wff *)
```

Per our declaration, basic propositions are indexed by natural numbers and type-check as well-formed formulas.

```
Check P 0.                        (* P 0  : Wff *)
Check P 42.                       (* P 42 : Wff *)
```

We can compose constructors to construct new well-formed formulas out of old ones. The resulting expressions not only typecheck but can also be simplified automatically by the typechecker, using the keyword `Compute`.

```
Check Neg (P 0).                  (* Neg (P 0) : Wff *)

Check Impl (P 1) (P 2).           (* Impl (P 1) (P 2) : Wff *)
Check Conj (P 1) (P 2).           (* Conj (P 1) (P 2) : Wff *)

Definition F₁ := Neg (P 0).
Definition F₂ := Conj (P 1) (P 2).

Check F₁.                         (* F₁ : Wff *)
Check F₂.                         (* F₂ : Wff *)

Compute F₁.                       (* = Neg (P 0) *)
Compute F₂.                       (* = Conj (P 1) (P 2) *)

Check Impl F₁ F₂.                 (* Impl F₁ F₂ : Wff *)
Compute Impl F₁ F₂.               (* = Impl (Neg (P 0)) (Conj (P 1) (P 2)) *)
```

Note that `Impl`, `Conj` and `Disj` are functions of two variables but they are written in "curried notation". This means that `Imp`, for instance, is not defined on the Cartesian product`Wff × Wff`, but instead sends a given well formed formula `F` to a function `Imp F : Wff → Wff`.

```
Section Scratch.

  Variable (dummy_formula : Wff).
  Check Impl dummy_formula.          (* Impl dummy_formula : Wff → Wff *)

End Scratch.
```

As a side remark, note that Roq recognises `Wff` as something called `Set`. In the declaration, we could have written `Inductive Wff : Set :=` but this is not necessary; it is *inferred* by the type checker.

```
Check Wff                            (* Wff : Set *)
```

4.2 Notation

For greater convenience, let us implement infix notation for the binary logical connectives. The associativity rules we set are consistent with the ones in Rocq's Corelib.Init.Notations.

```
Notation "a ∧ b" := (Conj a b) (at level 80, right associativity).
Notation "a ∨ b" := (Disj a b) (at level 85, right associativity).
Notation "a ⇒ b" := (Impl a b) (at level 99, right associativity).
```

With these notations, we get the associativity rules that we talked about in the lecture, where expressions such as `P 1 ⇒ P 2 ⇒ P 3` are parsed as `P 1 ⇒ (P 2 ⇒ P 3)`. Note that we do not need brackets around `P 1` or `P 2` (functions bind tighter that anything else, so to speak).

```
Check P 1 ⇒ P 2 ⇒ P 3.               (* P 1 ⇒ P 2 ⇒ P 3   : Wff *)
Check P 1 ⇒ (P 2 ⇒ P 3).             (* P 1 ⇒ P 2 ⇒ P 3   : Wff *)
Check (P 1 ⇒ P 2) ⇒ P 3.             (* (P 1 ⇒ P 2) ⇒ P 3 : Wff *)

Check P 1 ∧ P 2 ∧ P 3.               (* P 1 ∧ P 2 ∧ P 3   : Wff *)
Check P 1 ∧ (P 2 ∧ P 3).             (* P 1 ∧ P 2 ∧ P 3   : Wff *)
Check (P 1 ∧ P 2) ∧ P 3.             (* (P 1 ∧ P 2) ∧ P 3 : Wff *)
```

Since `∧` has been set at so-called "level" 80 and `∨` has been set at level 85, the binary operator `∧` binds *tighter* than `∨`. This is an arbitrary convention, but it is common (just like `*` usually binds tighter than `+` when the two pieces of notation are used together).

In practice, this means that `P 1 ∧ P 2 ∨ P 3` is parsed as `(P 1 ∧ P 2) ∨ P 3`, not `P 1 ∧ (P 2 ∨ P 3)`, and similarly for `P 1 ∨ P 2 ∧ P 3`.

```
Check (P 1 ∧ P 2) ∨ P 3.          (* P 1 ∧ P 2 ∨ P 3   : Wff *)
Check P 1 ∧ (P 2 ∨ P 3).          (* P 1 ∧ (P 2 ∨ P 3) : Wff *)
Check P 1 ∧ P 2 ∨ P 3.            (* P 1 ∧ P 2 ∨ P 3   : Wff *)
```

It may be helpful to think of Rocq's *levels* as a measure of *distance* between the operator and its arguments: the shorter the distance, the higher the precedence. For instance, in the expression `P 1 ∧ P 2 ∧ P 3`, the distance between `∧` and its arguments is set to 80, while that of `∨` is set to 85. So `P 2` is closer to `∧` than it is to `∨` and the expression is parsed as `(P 1 ∧ P 2) ∨ P 3`.

Similar considerations apply with `⇒`, which binds looser than `∨`, hence also than `∧`.

```
Check F₁ ⇒ ((F₂ ⇒ P 42) ∨ P 3).   (* F₁ ⇒ (F₂ ⇒ P 42) ∨ P 3 : Wff *)
Check F₁ ⇒ ((F₂ ⇒ P 42) ∧ P 3).   (* F₁ ⇒ (F₂ ⇒ P 42) ∧ P 3 : Wff *)
```

Note that Rocq does not require blank spaces around the operators, but that we usually insert them.

```
Check F₁∧F₂⇒F₁.                   (* F₁ ∧ F₂ ⇒ F₁ : Wff *)
```

Similarly for `Neg`, we can introduce a convenient notation. No associativity rule here (`Neg` is not a binary operator), but a formatting rule to guarantee that the pretty printer shows `¬P 0` instead of `¬ P 0` (try it!).

```
Notation "¬ a" := (Neg a) (at level 75, format "¬ a").

Check ¬P 0.                       (* ¬P 0 : Wff *)
Compute F₁.                       (* = ¬P 0 : Wff *)
Compute F₁ ⇒ F₂.                  (* = ¬P 0 ⇒ P 1 ∧ P 2 *)
```

Note that, in the scratchpad file available for download, the character `!` is used, instead of `¬`, to denote the constructor `Neg`.

```
Notation "! a" := (Neg a) (at level 75, format "! a").
Check !P 0.
```

4.3 The height function

When we constructed functions *out of* the type of Booleans, we did so by pattern matching. We can do that because Booleans are inductively declared in Rocq. More precisely, here is the declaration, as found in Rocq's Corelib.Init.Datatypes.

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

The intuition is that, in order to define a function `f : bool → X` (where `X` is an arbitrary set/type), it suffices to specify `f true` and `f false` as elements/terms of `X`. Similarly, we should be able to define a function `f : Wff → X` by defining it on each constructor of `Wff`.

Let us for instance declare the height function on well-formed formulas that we introduced in Lecture 2. Two observations are in order:

- A cosmetic one: we can use the notation for `Neg`, `And`, etc as valid syntax in the pattern matching, which makes the code more readable.
- A fundamental one: our height function is defined recursively (it calls upon itself in three out of four subcases), which forces us, in Rocq, to replace the keyword `Definition` with `Fixpoint`.

```
Fixpoint ht (F : Wff) :=
  match F with
  | P i    => 0
  | ¬F     => 1 + ht F
  | F ∧ F' => 1 + max (ht F) (ht F')
  | F ∨ F' => 1 + max (ht F) (ht F')
  | F ⇒ F' => 1 + max (ht F) (ht F')
  end.
```

Note that the return type of the height function is inferred by the type checker (because the first case returns `0` which is parsed as a natural number by the type checker).

```
Check ht.                          (* ht : Wff → ℕ *)
```

Then not only does this typecheck but it also computes 🎉.

```
Check   ht ((P 1 ∧ P 2) ∨ ¬¬P 3).   (* ht ((P 1 ∧ P 2) ∨ ¬¬P 3) *)
Compute ht (P 1 ∧ P 2 ∨ ¬¬P 3).     (* = 3 *)
```

With `Fixpoint`, the syntax

```
Fixpoint ht : Wff → ℕ :=
  fun F => match F with
```

is *not* accepted by the typechecker.

If you want to desugar the `Fixpoint` syntax, you can write the same program as follows. Also note that the last three cases in the pattern matching are treated in one go, which is a syntax you can of course use elsewhere, too!

```
Definition ht₀ : Wff → ℕ :=
  fix ht₀ (F : Wff) : nat :=
    match F with
    | P _                        => 0
    | ¬F                         => 1 + ht F
    | F ⇒ F' | F ∧ F' | F ∨ F' => 1 + max (ht F) (ht F')
    end.
```

We can check that the two functions `ht` and `ht₀` are indeed equal. This is a proof by reflexivity, which *implies* that the two functions take the same values on every well-formed formula.

```
Goal ht = ht₀.
Proof.
  reflexivity.
Qed.
```

It is possible to use Rocq's tactic language to define the above height function. Namely, one can use the `induction` tactic. It is worth taking a look at, but in practice I recommend using the `Fixpoint` construction. Note that the `induction` tactic can introduce the five sub-cases automatically, as well as name the relevant terms and induction hypotheses in each case, but you can also choose your own names, by replacing

```
induction F.
```

by

```
induction F as [n | F htF | F htF F' htF' | F htF F' htF' | F htF F' htF'].
```

which nonetheless seems to break the proof by reflexivity that `ht = ht₁`.

```
Definition ht₁ (F : Wff) : ℕ.
Proof.
  induction F
    (* as [n | F htF | F htF F' htF' | F htF F' htF' | F htF F' htF'] *).
  - exact 0.
  - exact (1 + IHF).                (* exact (1 + htF). *)
  - exact (1 + (max IHF1 IHF2)).    (* exact (1 + (max htF htF')). *)
  - exact (1 + (max IHF1 IHF2)).    (* exact (1 + (max htF htF')). *)
  - exact (1 + (max IHF1 IHF2)).    (* exact (1 + (max htF htF')). *)
Defined.
(* Qed *)

Compute ht₁ (P 1 ∧ P 2 ∨ ¬¬P 3).    (* = 3 *)

Goal ht = ht₁.
Proof.
  reflexivity.
Qed.
```

Careful! If we use `Qed` instead of `Defined` in the above construction of `ht₁`, then the definition becomes opaque and the computation does not return `3` (try it!). It just prints out the expression again…

Note that if you do not use focusing points, you can also write the proof as follows. I personally find this less informative.

```
Definition ht₂ (F : Wff) : ℕ.
Proof.
  induction F.
  exact 0.
  exact (1 + IHF).
  all: exact (1 + (max IHF1 IHF2)).
Defined.

Compute ht₂ (P 1 ∧ P 2 ∨ ¬¬P 3).    (* = 3 *)

Goal ht = ht₂.
Proof.
  reflexivity.
Qed.
```

4.4 Subformulas

We will require Rocq's standard library on linked lists. More precisely, linked lists are defined in Rocq's Corelib.Init.Datatype but the notation we will be using is defined in the Stdlib.Lists.List. Let us first analyse the definition of lists.

```
Inductive list (A : Type) : Type :=
  | nil  : list A
  | cons : A → list A → list A.

Arguments nil {A}.
Arguments cons {A} a l.
```

It is quite general. We see that it depends on a parameter `A` which is a type. This enables us to consider lists of natural numbers, lists of booleans, etc. For some reason, Rocq recognises these as sets rather than types.

```
Check list ℕ.                    (* list ℕ : Set *)
Check list bool.                 (* list bool : Set *)
```

Once a type `A` is fixed, the type `list A` is defined inductively, using two constructors, `nil {A} : list A` and `cons {A} : A → list A → list A`. Because of the `Arguments` command placed after the definition, the parameter `A` is implicit when using `nil` and `cons`.

```
Check nil.                       (* nil : list ?A *)
Check cons.                      (* cons : ?A → list ?A → list ?A *)

Check @nil.                      (* nil : ∀ A : Type, list A *)
Check @cons.                     (* cons : ∀ A : Type, A → list A → list A *)
```

```
Check @nil bool.                    (* nil : list bool *)
Check @cons ℕ.                      (* cons : ℕ → list ℕ → list ℕ *)
```

In general, this implicit parameter can be inferred by the typechecker, as we shall see below when we construct our first examples of lists.

```
Check (nil : list ℕ).               (* nil : list ℕ *)
Check cons true (cons false nil).   (* cons true (cons false nil) : list bool)
```

The syntax with `cons` and `nil` is hard to parse. This is why `cons` is replaced by the following infix notation.

```
Infix "::" := cons (at level 60, right associativity) : list_scope.
```

To access it, we can either put the notation in scope explicitly, or import a module that opens it (see below).

```
Open Scope list_scope.
Check 1 :: 2 :: 3 :: nil.           (* 1 :: 2 :: 3 :: nil : list nat *)
```

This is better, but not yet optimal in practice. So we import the following module.

```
From Stdlib Require Import List.
Import ListNotations.
```

In the Rocq scratchpad, use the following block instead.

```
From Coq Require Import List.
Import ListNotations.
```

This makes the pretty printer view of lists much nicer. And more importantly, we can use that same syntax to denote lists in our code, in a variety of ways.

```
Check 1 :: 2 :: 3 :: nil.           (* [1; 2; 3] : list nat *)
Check [1; 2; 3].                    (* [1; 2; 3] : list nat *)
Check 1 :: [2; 3].                  (* [1; 2; 3] : list nat *)
Check [42].                         (* [42] : list nat *)
```

Let us now use lists to implement the function `strict_sf` seen in Lecture 2. Here, we do this using the constructors actual names, not the notation we introduced for them, but you can change `Neg F` to `¬F`, `Conj F F'` to `F ∧ F'` etc. Note that the order in which you enter the constructor does not matter for the pattern matching, and that you can use the notation `F` in the pattern `Neg F` etc, even though you are pattern matching on a parameter called `F` in the body of the function.

```
Fixpoint strict_sf (F : Wff) : list Wff :=
  match F with
  | P i       => nil
  | Neg F     => F :: strict_sf F
  | Conj F F' => (F :: strict_sf F) ++ (F' :: strict_sf F')
  | Disj F F' => (F :: strict_sf F) ++ (F' :: strict_sf F')
  | Impl F F' => (cons F (strict_sf F)) ++ (cons F' (strict_sf F'))
  end.
```

The resulting function returns a list of the strict subformulas of a given formula, as expected. If you remove `list Wff` from the declaration of `strict_sf`, the return type will be inferred by the type checker, because of the syntax `F :: _` in the second case, for `F` of type `Wff` (try it!).

```
Check   strict_sf.                  (* strict_sf : Wff → Wff *)
Compute strict_sf ((P 1 ∧ P 2) ∨ ¬¬P 3).
  (* = [P 1 ∧ P 2; P 1; P 2; ¬¬P 3; ¬P 3; P 3] *)
```

We can then define the subformula function as in Lecture 2.

```
Definition sf (F : Wff) : list Wff :=
F :: (strict_sf F).

Compute sf ((P 1 ∧ P 2) ∨ ¬¬P 3).
  (* = [P 1 ∧ P 2 ∨ ¬¬P 3; P 1 ∧ P 2; P 1; P 2; ¬¬P 3; ¬P 3; P 3])
```

4.5 Substitutions

Given a well-formed formula `F`, a subset `I ⊂ ℕ` and a family of well-formed formulas `G : ℕ → Wff`, we have seen in the lecture how to replace the atomic formulas `P n` by `G n` in `F`, for all `n ∈ I`. Let us now implement this function in practice.

```
Fixpoint subst_Wff (F : Wff) (I : ℕ → bool) (G : ℕ → Wff) : Wff :=
  match F with
  | P j     => if I j then G j else P j
  | ¬F      => ¬(subst_Wff F I G)
  | F₁ ⇒ F₂ => (subst_Wff F₁ I G) ⇒ ((subst_Wff F₂ I G))
  | F₁ ∧ F₂ => (subst_Wff F₁ I G) ∧ ((subst_Wff F₂ I G))
  | F₁ ∨ F₂ => (subst_Wff F₁ I G) ∨ ((subst_Wff F₂ I G))
  end.
```

For the sake of simplicity, we will use the same characteristic function `I : ℕ → bool` in all our examples. Note how we define `I` using pattern matching when we want it to represent the finite subset `{0; 26; 41; 42}` of the set ℕ.

```
Definition I (n : ℕ) : bool :=
  match n with
  | 0  => true
  | 26 => true
  | 41 => true
  | 42 => true
  | _  => false
  end.
```

4.5.1 Example 1

Let us define a family of formulas to be used when performing the substitution of certain atomic formulas. In this example `G₁ n` is different from `P n` for all `n`, but in the substitution, only those `G n` for which `I n = true` will matter.

```
Definition G₁ : ℕ → Wff :=
  fun n => P (n + 2) ∧ P (n + 1).
```

In the example below, every atomic subformula `P j` for which `I j = true` (so, everything except 25) is replaced by something new, because `G₁ j ≠ P j` for all `ij. Note thatI` 41 = true`but that this plays no role becauseP` 41`does not appear in our example forF`‘.

```
Compute subst_Wff (P 0 ∨ P 26 ⇒ P 42 ∧ P 25) I G₁.
  (* = P 2 ∧ P 1 ∨ P 28 ∧ P 27 ⇒ (P 44 ∧ P 43) ∧ P 25 *)
```

4.5.2 Example 2

We can also define a family of formulas in a way similar to what we did for `I`, meaning that we only care about defining `G₂ n` for a finite number of `n` and for all other `n` we simply set `G n = P n`.

```
Definition G₂ : ℕ → Wff :=
  fun n =>
    match n with
    | 25  => P 44
    | 26  => P 1 ⇒ P 2
    | _   => P n
    end.
```

In the example below, only `P 26` is replaced by `G 26` (even though `G 2025 ≠ P 2025` and `P 2025` appears in our example for `F`), because `I 2025 ≠ true`.

```
Compute subst_Wff (P 0 ∨ P 26 ⇒ P 42 ∧ P 2025) I G₂.
  (* =  P 0 ∨ (P 1 ⇒ P 2) ⇒ P 42 ∧ P 25 *)
```

4.5.3 Example 3

For the third example, we define a family of formulas using a conditional statement. The point is to show an example where `G n ≠ P n` for an infinite quantity of `n` but also `G n = P n` for an infinite quantity of `n`. We will do so by defining the characteristic function of even numbers.

```
From Corelib Require Import Init.Nat.
```

In the Rocq scratchpad, use the following block instead.

```
From Coq Require PeanoNat.
Import Nat.
```

Back to the main definition

```
Definition is_even (n : nat) : bool :=
  eqb (n mod 2) 0.

Definition G₃ (n : ℕ) : Wff :=
  if is_even n then P (n / 2) else P n.
```

In the example below, an atomic formula `P j` is replaced by `G (j / 2)` if and only if `I j = true` and `is_even j = true`.

```
Compute subst_Wff (P 0 ∨ P 26 ⇒ P 41 ∧ P 2025) I G₃.
  (* = P 0 ∨ P 13 ⇒ P 41 ∧ P 25 *)
```

4.5.4 Substitutions in indexed families

The definition of the function `subst_Wff` suggests a general method to substitute terms in a family `P : Y → X` (family of terms of type `X` indexed by the type `Y`). All you need is a subtype of `Y` (represented by a characteristic function `I : Y → X`) and a new family `G : Y → X`, indexed by the same type `Y`.

Then for all element `y` of `Y`, if `y` belongs to the subset `I` (which means that the characteristic function `I` evaluates to `true` on `y`), one replaces `P y` with `G y`, and if `y` does not belong to `I`, then one leaves `P y` unchanged.

```
Definition substFamily {Y} {X} : (Y → X) → (Y → bool) → (Y → X) → Y → X :=
  fun P I G y => if I y then G y else P y.
```

One can then redefine `subst_Wff` as follows and check that it behaves as expected on our examples. This time we place `(F : Wff)` last, so `subst_Wff' I G : Wff → Wff`.

```
Fixpoint subst_Wff' (I : ℕ → bool) (G : ℕ → Wff) (F : Wff) : Wff :=
  match F with
  | P j     => substFamily P I G j (* if I j then G P I j else P j *)
  | ¬F      => ¬(subst_Wff' I G F)
  | F₁ ⇒ F₂ => (subst_Wff' I G F₁) ⇒ ((subst_Wff' I G F₂))
  | F₁ ∧ F₂ => (subst_Wff' I G F₁) ∧ ((subst_Wff' I G  F₂))
  | F₁ ∨ F₂ => (subst_Wff' I G F₁) ∨ ((subst_Wff' I G F₂))
  end.

Compute subst_Wff  (P 0 ∨ P 26 ⇒ P 42 ∧ P 2025) I G₁.
  (* = P 2 ∧ P 1 ∨ P 28 ∧ P 27 ⇒ (P 44 ∧ P 43) ∧ P 25 *)
Compute subst_Wff' I G₁ (P 0 ∨ P 26 ⇒ P 42 ∧ P 2025).
  (* = P 2 ∧ P 1 ∨ P 28 ∧ P 27 ⇒ (P 44 ∧ P 43) ∧ P 25 *)

Compute subst_Wff  (P 0 ∨ P 26 ⇒ P 42 ∧ P 2025) I G₂.
  (* = P 0 ∨ (P 1 ⇒ P 2) ⇒ P 42 ∧ P 25 *)
Compute subst_Wff' I G₂ (P 0 ∨ P 26 ⇒ P 42 ∧ P 2025).
  (* = P 0 ∨ (P 1 ⇒ P 2) ⇒ P 42 ∧ P 25 *)

Compute subst_Wff  (P 0 ∨ P 26 ⇒ P 42 ∧ P 2025) I G₃.
  (* = P 0 ∨ P 13 ⇒ P 21 ∧ P 25 *)
Compute subst_Wff' I G₃ (P 0 ∨ P 26 ⇒ P 42 ∧ P 25).
  (* = P 0 ∨ P 13 ⇒ P 21 ∧ P 25 *)
```

4.6 Exercise

As an exercise to practice the techniques seen in this file, add one more constructor

```
Iff : Wff → Wff → Wff
```

to the type of well-formed formulas, then introduce the (non-associative) infix notation

```
`⇔` (or `<=>`)
```

for it, and complete the definitions of `ht`, `strict_sf` and `subst_Wff` accordingly.

Be careful with the precedence level for ⇔. In Rocq's notation for propositional connectives, the formulas `P 1 ⇒ P 2 ⇔ P 3` and `P 2 ⇔ P 3 ⇒ P 1` parse respectively as `P 1 ⇒ (P 2 ⇔ P 3)` and `(P 2 ⇔ P 3) ⇒ P 1`.

5 Semantics of propositional logic

To give a *meaning* to well-formed formulas, we will construct a set of *truth values* $\mathcal{B}$ and a function

$$\text{eval} : \text{Wff} \to \mathcal{B}.$$

In fact, we will restrict ourselves to the case when $\mathcal{B} := \text{bool} = \{\text{false}, \text{true}\}$. These two possible truth values could also be denoted by $\{0, 1\}$.

5.1 Evaluation of well-formed formulas

Recall that well-formed formulas are defined inductively, with constructors

- $P : \mathbb{N} \to \text{Wff}$
- $\neg : \text{Wff} \to \text{Wff}$
- $\wedge : \text{Wff} \to \text{Wff} \to \text{Wff}$
- $\vee : \text{Wff} \to \text{Wff} \to \text{Wff}$
- $\Rightarrow : \text{Wff} \to \text{Wff} \to \text{Wff}$
- $\Leftrightarrow : \text{Wff} \to \text{Wff} \to \text{Wff}$

so we can define a function *out of* Wff by *pattern matching* on the constructors (primitive recursion).

5.1.1 Valuations

The first case is that of so-called *atomic formulas*, meaning well-formed formulas F of the form $F = P_i$ for some natural number $i : \mathbb{N}$. In practice, what we want here is to define $\text{eval}(P_i)$ for all i. The constructor's name P is not relevant here: what we want is a function $\nu : \mathbb{N} \to \text{bool}$, with the *same domain as* P. Then we can use this function to set $\text{eval}(P_i) := \nu(i)$ for all i.

For convenience, in these notes we will think of the set of atomic formulas as a subset of the set of well-formed formulas

$$\mathcal{A} := \{P_0, P_1, \ \ldots\ \} \subset \text{Wff}$$

so we are able to identify $\nu : \mathbb{N} \to \text{bool}$ and $\text{eval}|_{\mathcal{A}} : \mathcal{A} \to \text{bool}$.

Definition. A function $\nu : \mathcal{A} \to \text{bool}$ will be called a **valuation** for $\mathcal{A}$.

A valuation assigns a truth value to every atomic formula. In our case, the possible truth values are true and false, but this is arbitrary. In particular, we could have more than two truth values (take for instance $\mathcal{B} := \{\text{false}, \text{undefined}, \text{true}\}$).

Suppose we are given a valuation $\nu : \mathcal{A} \to \text{bool}$. Then we want to construct an *extension* $\hat{\nu} : \text{Wff} \to \text{bool}$ of ν, meaning a function $\hat{\nu}$ from Wff to bool such that $\hat{\nu}|_{\mathcal{A}} = \nu$. To achieve this, it suffices to carry on with the pattern matching and define:

- Given a well-formed formula F, a boolean $\hat{\nu}(\neg F)$, and
- Given well-formed formulas F and F', a boolean $\hat{\nu}(F \diamond F')$ for every binary constructor $\diamond \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$.

In this process, we are allowed to consider that $\hat{\nu}(F)$ and $\hat{\nu}(F')$ are already defined (recursion principle).

5.1.2 Negation

Let F be a well-formed formula. We want to define $\hat{\nu}(\neg F) : \text{bool}$, assuming that $\hat{\nu}(F) : \text{bool}$ has already been defined. The natural thing to do is to set

$$\hat{\nu}(\neg F) :=\, !\hat{\nu}(F)$$

where $!b$ is the *negation* of the boolean b (meaning that $!b = \text{false}$ if $b = \text{true}$ and $!b = \text{true}$ if $b = \text{false}$). This is precisely the function `negb : bool -> bool` that we defined in the introduction to pattern matching.

So, to interpret the function$\neg : \text{Wff} \to \text{Wff}$, we have used a function $! : \text{bool} \to \text{bool}$, replacing Wff by bool everywhere it appears in the type signature of $\neg$ (which here is both domain and codomain). We will proceed similarly for other constructors.

5.1.3 Binary constructors

To define $\hat{\nu}(F \diamond F')$ for each binary constructor $\diamond \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$, we proceed as follows. As a function, the binary constructors have type signature

$$\diamond : \text{Wff} \to \text{Wff} \to \text{Wff}\ ,$$

so to interpret them we will use a function $[\diamond] : \text{bool} \to \text{bool} \to \text{bool}$, and set

$$\hat{\nu}(F \diamond F') := \hat{\nu}(F)\ [\diamond]\ \hat{\nu}(F').$$

Since we are proceeding recursively, this is sufficient to compute $\hat{\nu}(F \diamond F')$ as soon as $\hat{\nu}(F)$ and $\hat{\nu}(F')$ are known.

The binary constructors $\wedge, \vee, \Rightarrow, \Leftrightarrow$ will be interpreted, respectively, by the functions `andb`, `orb`, `implb` and `eqb`, which we have introduced in the lecture on pattern matching. We will use the following notation for boolean operations.

- $\hat{\nu}(F \wedge F') := \hat{\nu}(F)$ && $\hat{\nu}(F')$
- $\hat{\nu}(F \vee F') := \hat{\nu}(F)$ || $\hat{\nu}(F')$
- $\hat{\nu}(F \Rightarrow F') := \hat{\nu}(F) \preccurlyeq \hat{\nu}(F')$
- $\hat{\nu}(F \Leftrightarrow F') := \hat{\nu}(F) == \hat{\nu}(F')$

Note that if we think of bool as $\{0, 1\}$ (equipped with the usual ordering $0 \leqslant 1$, an addition law $+$ such that $1 + 1 = 0$, and a multiplication law $\times$), then we can also use the following notation:

- $[\neg]b := 1 + b$
- $b\ [\wedge]\ b' := \min(b, b')$
- $b\ [\vee]\ b' := \max(b, b')$
- $b\ [\Rightarrow]\ b' := 1 + x \times y + x$
- $b\ [\Leftrightarrow]\ b' := b == b'$

With this we have defined a function $\hat{\nu} : \text{Wff} \to \text{bool}$ by primitive recursion. Note that part of it is entirely mechanical: the interpretation $[\diamond]$ of the constructors $\neg, \wedge, \vee, \Rightarrow$ and $\Leftrightarrow$ does not depend on the valuation $\nu : \mathcal{A} \to \text{bool}$. So what we have seen above is a procedure to construct an evaluation/interpretation function $\hat{\nu} : \text{Wff} \to \text{bool}$ from a valuation $\nu : \mathcal{A} \to \text{bool}$.

$$\begin{array}{ccc} (\mathcal{A} \to \text{bool}) & \longrightarrow & (\text{Wff} \to \text{bool}) \\ \nu & \longmapsto & \hat{\nu} \end{array}$$

We will come back to this later when we discuss recursion principles in further generality.

5.1.4 Example

Let P and Q be atomic formulas and let $F := (P \vee Q) \wedge \neg P$. What are the possible truth values for F (in bool)? Meaning, what is $\hat{\nu}(F)$ as a function of ν?

For instance, if $\nu(P) = \text{false}$ and $\nu(Q) = \text{true}$, what does $\hat{\nu}(F)$ *compute* to?

$$\begin{array}{rcl} \hat{\nu}(F) & = & \hat{\nu}((P \vee Q) \wedge \neg P) \\ & = & \hat{\nu}(P \vee Q) \text{ \&\& } \hat{\nu}(\neg P) \\ & = & (\nu(P) \text{ || } \nu(Q)) \text{ \&\& } !\nu(P) \\ & = & (\text{false} \text{ || } \text{true}) \text{ \&\& } !\text{false} \\ & = & \text{true} \text{ \&\& } !\text{true} \\ & = & \text{true} \end{array}$$

We can compute similarly for other values of $\nu(P)$ and $\nu(Q)$ and store the result in a **truth table** for F.

$\nu(P)$	$\nu(Q)$	$\hat{\nu}(P \vee Q)$	$\hat{\nu}(\neg P)$	$\hat{\nu}((P \vee Q) \wedge \neg P)$
false	false	false	true	false
false	true	true	true	true
true	false	true	false	false
true	true	true	false	false

Note that:

- There are as many columns in the truth table of F as there are subformulas in F.
- If F has n atomic subformulas, the truth table of F has 2^n rows (since each atomic formula can assume two different boolean values).
- We are labelling the columns using $\nu(P)$, $\nu(Q)$, $\hat{\nu}(P \vee Q)$, ... , *not just* P, Q, $P \vee Q$, ... , to emphasise that the truth table depends on an *evaluation* procedure of well-formed formulas. Here we have chosen to interpret well-formed formulas as booleans, in a standard way.

Depending on the particular well-formed formula F, we may be able to simplify the computation. For instance, in the example $F := (P \vee Q) \wedge \neg P$ above, since _ && true = false regardless of what appears in the placeholder "_" , we have $\hat{\nu}(F) = \text{false}$ as soon as $\nu(P) = \text{true}$.

5.2 Satisfiablity and validity

Definition. A well-formed formula F is called **satisfiable** if *there exists* a valuation $\nu : \mathcal{A} \to \text{bool}$ such that $\hat{\nu}(F) = \text{true}$.

For example, the formula $F := (P \vee Q) \wedge \neg P$ (where P and Q are atomic formulas) is satisfiable.

Observation. If $\mathcal{B} := \{\text{false}, \text{undefined}, \text{true}\}$, we can *decide* if the only accepted truth value for satisfiability is true or if undefined is also acceptable. This will usually depend on the context, in particular the applications we may have in mind.

Definition. A well-formed formula F is called **valid** if *for all* valuation $\nu : \mathcal{A} \to \text{bool}$, one has $\hat{\nu}(F) = \text{true}$. A valid formula is also called a **tautology**.

So a tautology is a compound formula whose truth value is always equal to true, regardless of the truth value of its subformulas.

For example, the formula $F := (P \vee Q) \wedge \neg P$ seen above (where P and Q are atomic formulas) is *not* a tautology. In contrast, the formula $F \Rightarrow \neg P$ is a tautology, as shown by the following truth table.

$\nu(P)$	$\nu(Q)$	$\hat{\nu}(P \vee Q)$	$\hat{\nu}(\neg P)$	$\hat{\nu}((P \vee Q) \wedge \neg P)$	$\hat{\nu}((P \vee Q) \wedge \neg P \Rightarrow \neg P)$
false	false	false	true	false	true
false	true	true	true	true	true
true	false	true	false	false	true
true	true	true	false	false	true

Theorem. A well-formed formula F is vlid if and only if $\neg F$ is not satisfiable.

Proof. Exercise.

5.3 Logical equivalence

Let us discuss the notion of logical equivalence *in the framework of Boolean semantics* (meaning when a well-formed formula can only take one of two truth values, called true and false, and the logical connectives $\neg$, $\wedge$, $\vee$, $\Rightarrow$ and $\Leftrightarrow$ are interpreted via the Boolean operations !, &&, ||, $\preccurlyeq$ and ==).

Definition. The well-formed formulas F and F' are called **logically equivalent** if, for all valuation $\nu : \mathcal{A} \to \text{bool}$, one has $\hat{\nu}(F) = \hat{\nu}(F')$.

Theorem. For all well-formed formulas F and F', the formula $F \Leftrightarrow F'$ is logically equivalent to $(F \Rightarrow F') \wedge (F' \Rightarrow F)$.

Proof. We will compute the truth tables of $F \Leftrightarrow F'$ and $(F \Rightarrow F') \wedge (F' \Rightarrow F)$ and see that they coincide.

- First we compute $\hat{\nu}(F \Leftrightarrow F')$ as much as possible:

$$\hat{\nu}(F \Leftrightarrow F') = \hat{\nu}(F) == \hat{\nu}(F')$$

  The computation is now stuck: we have to pattern match on $\hat{\nu}(F)$ and $\hat{\nu}(F')$ to trigger a case analysis.

- Next we compute $\hat{\nu}\big((F \Rightarrow F') \wedge (F' \Rightarrow F)\big)$

$$\begin{array}{rcl} \hat{\nu}\big((F \Rightarrow F') \wedge (F' \Rightarrow F)\big) & = & \hat{\nu}(F \Rightarrow F') \;\&\&\; \hat{\nu}(F' \Rightarrow F) \\ & = & \big(\hat{\nu}(F) \preccurlyeq \hat{\nu}(F')\big) \;\&\&\; \big(\hat{\nu}(F') \preccurlyeq \hat{\nu}(F)\big) \end{array}$$

  The computation is stuck again, we need to distinguish cases according to values of $\hat{\nu}(F)$ and $\hat{\nu}(F')$.

- The truth tables of $F \Leftrightarrow F'$ and $(F \Rightarrow F') \wedge (F' \Rightarrow F)$ are therefore given as follows, and the last column of the first table is indeed the same as the last column of the second table.

$\hat{\nu}(F)$	$\hat{\nu}(F')$	$\hat{\nu}(F \Leftrightarrow F')$
false	false	true
false	true	false
true	false	false
true	true	true

$\hat{\nu}(F)$	$\hat{\nu}(F')$	$\hat{\nu}(F \Rightarrow F')$	$\hat{\nu}(F' \Rightarrow F)$	$\hat{\nu}\big((F \Rightarrow F') \wedge (F' \Rightarrow F)\big)$
false	false	true	true	true
false	true	true	false	false
true	false	false	true	false
true	true	true	true	true

The set of valuations $\nu : \mathcal{A} \to \text{bool}$ that make a formula F (evaluate to) true will be called the **semantics** of F.

$$\text{sem}(F) := \{\nu : \mathcal{A} \to \text{bool} \mid \hat{\nu}(F) = \text{true}\} = \{\nu_0, \nu_1, \ \ldots\ \}$$

In particular, the formula F is satisfiable if and only if $\text{sem}(F) \neq \emptyset$, and the formulas F and F' are logically equivalent if and only if they have the same semantics: $\text{sem}(F) = \text{sem}(F')$. As an example, the semantics of the formula the formula $F := (P \vee Q) \wedge \neg P$ are given by $\nu_0 := \{P \mapsto \text{false}; Q \mapsto \text{false}\}$. To conclude, we give the following characterisation of logically equivalent formulas, which can also be taken as a definition.

Theorem. Two well-formed formulas F and F' are logically equivalent if and only if $F \Leftrightarrow F'$ is a tautology.

Proof. Exercise.

We will see in forthcoming lectures that there also exist purely syntactic methods to determine when is a given formula a tautology.