SoSe 26 Type Theory & Logic & Functional Programming Notes

Igor Dimitrov

2026-04-21

Table of contents

Preface

This is a Quarto book.

To learn more about Quarto books visit https://quarto.org/docs/books.

Part I

Logic

1 Lecture 1 - Propositional Logic: Syntax

1.1 Overview

Brief summary of what this lecture does (2–4 lines).

Goal: Define the expression we can write, these are called **well-formed-formulas**, or **wff**s.

There are various ways of defining what counts as a wff. One of them is the inductive approach.

1.2 Inductive Approach

First we start with **atomic formulas**:

$$P_0, P_1, P_2, ...$$
$$(P_i)_{i \in \mathbb{N}}$$

are atomic propositions.

We give some rules to inductively define new formulas from old ones.

1) if φ, ψ are **wff** then $\varphi \Rightarrow \psi$ is a **wff**
2) if φ, ψ are **wff** then $\varphi \wedge \psi$ is a **wff**
3) if φ, ψ are **wff** then $\varphi \vee \psi$ is a **wff**
4) if φ is a **wff** then $\neg\varphi$ is a **wff**

1.3 Constructor Approach

It is useful to express these rules with **constructors**, which define wffs as values of some functions (with codomain WFF)

In this approach atomic formulas are values of functions:

Definition 1.1 (Atomic formulas)**.** Atomic formulas are values of functions like:

$$P : \mathbb{N} \to \text{WFF}, \quad : i \mapsto P_i$$

or

$$''\cdot'' : \text{String} \to \text{WFF}, \quad : x \mapsto "x"$$

Example 1.1 (String Constructor)**.** with the latter we can construct arbitrary propositions like:

"today␣is␣sunny"

We also have constructors for the logical connectives:

Definition 1.2 (Constructors for Logical Connectives)**.**

$$\begin{aligned}
&(\neg) : \text{WFF} \to \text{WFF}, \quad : \varphi \mapsto \neg\varphi \\
&(\wedge) : \text{WFF} \times \text{WFF} \to \text{WFF}, \quad : (\varphi, \psi) \mapsto \varphi \wedge \psi \\
&(\vee) : \text{WFF} \times \text{WFF} \to \text{WFF}, \quad : (\varphi, \psi) \mapsto \varphi \vee \psi \\
&(\Rightarrow) : \text{WFF} \times \text{WFF} \to \text{WFF}, \quad : (\varphi, \psi) \mapsto \varphi \Rightarrow \psi
\end{aligned}$$

Remark 1.1 (Constructor vs Symbol View of Logical Connectives). Earlier we wrote $\varphi \Rightarrow \psi$ and then we denoted $\Rightarrow$ symbol as a function of type $\text{WFF} \times \text{WFF} \to \text{WFF}$, we use $(\Rightarrow)$ to distinguish between the two concepts.

A different but related question that arises si the ambiguity of $P_1 \Rightarrow P_2 \Rightarrow P_3$. According to the inductive definition there are two ways this formula could have been constructed which corresponds to the different formulas:

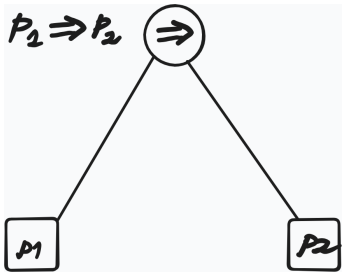
1. $(P_1 \Rightarrow P_2) \Rightarrow P_3$
2. $P_1 \Rightarrow (P_2 \Rightarrow P_3)$

To resolve this ambiguity we define the associativity rule:

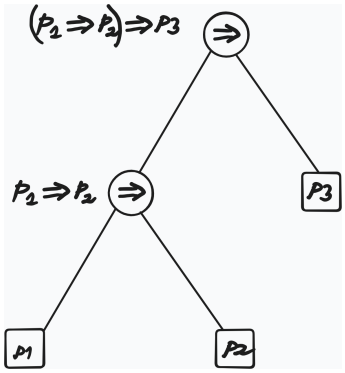
Definition 1.3 (Associativity Rule for $\Rightarrow$)**.**

$$P_1 \Rightarrow P_2 \Rightarrow P_3 := (P_1 \Rightarrow P_2) \Rightarrow P_3$$

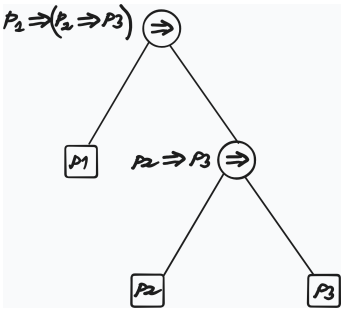
Remark 1.2 (Wffs as Trees). Wffs can be viewed as trees. Trees encode the derivation structure and resolve the ambiguity



(a) tree 1



(b) tree 2



(c) tree 3

our collection of wffs can be further enriched with another binary constructor:

$$\begin{aligned}(\Leftrightarrow) &: \mathrm{WFF} \times \mathrm{WFF} \to \mathrm{WFF} \\ &: (\varphi, \psi) \mapsto \varphi \Leftrightarrow \psi\end{aligned}$$

Remark 1.3 (Meaning of $\Leftrightarrow$). At this stage there is no relation between

$$\varphi \Leftrightarrow \psi \quad \text{and} \quad (\varphi \Rightarrow \psi) \wedge (\psi \Rightarrow \varphi)$$

Let is give another 'set-theoretic' approach

1.4 Set Theoretic Approach

First we define the alphabets:

Definition 1.4 (Set Theoeretic Approach to Defining Wffs)**.** We start with the alphabets:

$$\begin{array}{ll}\mathcal{A} := \{P_i\}_{i\in\mathbb{N}} & \text{(sub-alphabet of atomic formulas)} \\ \mathcal{C} := \{\Rightarrow, \wedge, \vee, \neg, \Leftrightarrow\} & \text{(sub-alphabet of logical connectives)} \\ \mathcal{V} := \mathcal{A} \cup \mathcal{C} \cup \{(,)\} & \text{(alphabet of propositional logic, including punctuation marks)}\end{array}$$

Then we define the so-called **Kleene Closure** of $\mathcal{V}$:

Definition 1.5 (Kleene Closure of $\mathcal{V}$)**.**

$$\mathcal{V}^* := \bigcup_{i=0}^{\infty} \mathcal{V}_i$$

$\mathcal{V}^*$ is the language containing all sorts of strings that can be constructed from the alphabet $\mathcal{V}$, also senseless ones.

To define the language propositional logic that contains only the formulas that we want and nothing else, we take the usual set-theoretic approach of taking the infinite union of all sets $\mathcal{W} \subseteq \mathcal{V}$ that satisfy the following properties:

1) $\mathcal{A} \subseteq \mathcal{W}$
2) if $\varphi, \psi \in \mathcal{W}$ then $\varphi \diamond \psi \in \mathcal{W}$, where $\diamond \in \{\Rightarrow, \wedge, \vee, \Leftrightarrow\}$
3) if $\varphi \in \mathcal{W}$ then $\neg\varphi \in \mathcal{W}$

Then the language of propositinal logic is defined as:

Definition 1.6 (Language Propositional Logic as Infinite Intersection)**.**

$$\mathcal{F}(\mathcal{A}) := \bigcap \{\mathcal{W} \subseteq \mathcal{V}^* | \mathcal{W} \text{ satisfies (1), (2), and (3)}\}$$

Theorem 1.1.

With this construction $\mathcal{F}(\mathcal{A})$ is the smallest set that satisfies the properties (1), (2) and (3), i.e.

1) $\mathcal{F}(\mathcal{A})$ satisfies the properties (1), (2) and (3)
2) For any set $\mathcal{W}$ that satisfies the properties (1), (2) and (3) we have $\mathcal{W} \subseteq \mathcal{F}(\mathcal{A})$

1.5 Summary

- WFF is defined inductively

- Constructors that generate WFFs
- Set theoretic construction of WFFs

2 Lecture 2 - Height and Subformulas

2.1 Overview

- We provide a fourth, iterative set construction for the language of propositional logic.
- We give two equivalent definitions for the height of a formula.
- We give two equivalent definitions of a function that gives the list of subformulas of a formula

2.2 Recurisve Set Construction

We define $\mathcal{F}_0$ iteratively as follows:

$$\begin{aligned}
&\mathcal{F}_0 := \mathcal{A} \\
&\mathcal{F}_1 := \mathcal{F}_0 \cup \{\, \neg\varphi \mid \varphi \in \mathcal{F}_0 \,\} \cup \{\, \varphi \diamond \psi \mid \varphi, \psi \in \mathcal{F}_0 \,\} && (\diamond \in \mathcal{C}) \\
&\mathcal{F}_n := \mathcal{F}_{n-1} \cup \{\, \neg\varphi \mid \varphi \in \mathcal{F}_{n-1} \setminus \mathcal{F}_{n-2} \,\} \cup \{\, \varphi \diamond \psi \mid \varphi, \psi \in \mathcal{F}_{n-1} \,\} && (n \geq 2)
\end{aligned}$$

With this construction it follows that:

$$\mathcal{F}_n \subseteq \mathcal{F}_{n+1}$$

Example 2.1. $\neg\neg P_4 \in \mathcal{F}_2 \subseteq \mathcal{F}_3 \subseteq ...$

but $\neg\neg P_4 \notin \mathcal{F}_1$, because $\neg P_4 \notin \mathcal{F}_0$

Now we make the following definition:

Definition 2.1.

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

and we claim:

Lemma 2.1.

$$\mathcal{F} = \mathcal{F}(\mathcal{A})$$

Where $\mathcal{F}(\mathcal{A})$ is defined in Definition 1.6

Proof. To prove $\mathcal{F} \subseteq \mathcal{F}(\mathcal{A})$, it suffices to show that

$$\mathcal{F}_n \subseteq \mathcal{F}(\mathcal{A}), \quad \forall n, \text{ by induction on } n$$

To conclude the proof we need to show that $\mathcal{F}_A \subseteq \mathcal{F}$ □

Definition 2.2 (Height of Formula - Set Approach)**.** For all $\varphi \in \mathcal{F}$, there is a well defined natural number $\mathrm{ht}(\varphi) \in \mathbb{N}$:

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

called the **height** of φ

We give another, a more computational definition for the height.

Definition 2.3 (Height of a Formula - Computational Approach)**.** We define $\mathrm{ht}(\cdot)$ recursively on the structure of a formula ϕ as follows:

$$\begin{aligned}
&\mathrm{ht}(P_i) = 0 && (P_i \in \mathcal{A}) \\
&\mathrm{ht}(\neg\varphi) = 1 + \mathrm{ht}(\varphi) \\
&\mathrm{ht}(\varphi \diamond \psi) = 1 + \max(\mathrm{ht}(\varphi), \mathrm{ht}(\psi)) && (\diamond \in \mathcal{C})
\end{aligned}$$

Remark 2.1 (Pattern Matching). Such inductive (recursive) definitions are also called **pattern matching**

Example 2.2 (Derivation of Height)**.**

$$\begin{aligned}
\mathrm{ht}(\neg\neg(P_1 \Rightarrow P_2)) &= 1 + \mathrm{ht}(\neg(P_1 \Rightarrow P_2)) \\
&= 1 + 1 + \mathrm{ht}(P_1 \Rightarrow P_2) \\
&= 1 + 1 + \max(\mathrm{ht}(P_1), \mathrm{ht}(P_2)) \\
&= 1 + 1 + \max(1, 1) \\
&= 1 + 1 + 1 \\
&= 3
\end{aligned}$$

Obervation:

We observe that the height of a formula is the height of its syntax tree

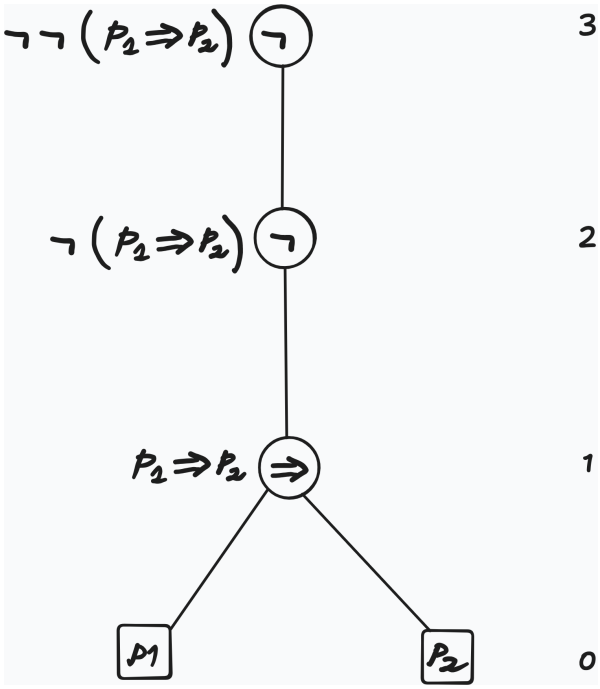


Figure 2.1: height of the syntax tree

We want to define a function

$$\mathrm{sf} : \mathcal{F}(\mathcal{A}) \to \mathcal{P}(\mathcal{F}(\mathcal{A}))$$

mapping a formula to its set of subformulas. For this we are going to define functions $\mathrm{sf}_i : \mathcal{F}_i \to \mathcal{P}(\mathcal{F})$ as follows:

Definition 2.4 (Set Construction of sf)**.**

$$\begin{aligned}
&\mathrm{sf}_0(P_i) := \{P_i\} \\
&\mathrm{sf}_n(\neg\varphi) := \{\neg\varphi\} \cup \mathrm{sf}_{n-1}(\varphi) \\
&\mathrm{sf}_n(\varphi \diamond \psi) := \{\varphi \diamond \psi\} \cup \mathrm{sf}_{n-1}(\varphi) \cup \mathrm{sf}_{n-1}(\psi)
\end{aligned}$$

Then sf can be defined as

$$\begin{aligned}
\mathrm{sf} &: \mathcal{F} \to \mathcal{P}(\mathcal{F}) \\
&: \varphi \mapsto \mathrm{sf}_{\mathrm{ht}(\varphi)}(\varphi)
\end{aligned}$$

We give another, more computational approach, by defining sf recursively on the structure of wffs as a function $\mathrm{sf} : \mathrm{WFF} \to [\mathrm{WFF}]$

Definition 2.5 (List Construction of sf)**.**

$$\begin{aligned}
&\mathrm{sf}(P_i) := [P_i] \\
&\mathrm{sf}(\neg\varphi) := [\neg\varphi] +\!\!+ \mathrm{sf}(\varphi) \\
&\mathrm{sf}(\varphi \diamond \psi) := [\varphi \diamond \psi] +\!\!+ \mathrm{sf}(\varphi) +\!\!+ \mathrm{sf}(\psi)
\end{aligned}$$

Example 2.3 (Derivation of the List of Subformulas with sf)**.**

$$\begin{aligned}
\mathrm{sf}(P_1 \vee \neg P_2) &= [P_1 \vee \neg P_2] +\!\!+ \mathrm{sf}(P_1) +\!\!+ \mathrm{sf}(\neg P_2) \\
&= [P_1 \vee \neg P_2] +\!\!+ [P_1] +\!\!+ [\neg P_2] +\!\!+ \mathrm{sf}(P_2) \\
&= [P_1 \vee \neg P_2] +\!\!+ [P_1] +\!\!+ [\neg P_2] +\!\!+ [P_2] \\
&= [P_1 \vee \neg P_2, P_1, \neg P_2, P_2]
\end{aligned}$$

2.3 Summary

- iterative set construction $\mathcal{F}_i$ - the language of propositional logic as infinite union of such
- two definitions for ht - height of a formula
- two definitions for sf - list or set of subformulas of a formula

3 Lecture 3: Basics of Rocq

Logic and Functional Programming
Florent Schaffhauser
Heidelberg University
(Summer 2026)

This note introduces Lambda Calculus, basics of Rocq function definitions, Boolean values in Rocq, basic pattern matching, curried functions, and the first small proofs about Boolean functions. The original lecture file was a Rocq source file; here the surrounding comments have been turned into prose, and Rocq commands are placed in executable Quarto code blocks.

3.1 Basics of Lambda Calculus

In lambda calculus, the basic syntactic forms are:

- variables: $x, y, f, \ldots$. These are **atomic terms**
- lambda abstractions: $\lambda x.t$. These **create functions** by binding a variable
- Applicaitons: $t\,u$: **Uses** a function by applying it to an argument.

So the two central operations are:

- abstraction: $\lambda x.t$. (t is called the function **body**)
- application: $t\,u$.

These can be seen as **complementary** operations. For example:

$$\lambda x.\, x \cdot 2 + 1$$

is the function $x \to x \cdot 2 + 1$, and applying it to 3 gives:

$$\begin{aligned}(\lambda x.\, x \cdot 2 + 1)\, 3 &\to 3 \cdot 2 + 1 \quad (\to \text{ denotes reduction}) \\ &\to 7\end{aligned}$$

This reduction rule is called β-reduction:

Definition 3.1 (β-reduction)**.**

$$\lambda x.\, t\, a \to_\beta t[x := a]$$

It says: applying a lambda abstraction to an argument means substituting the argument for the bound variable in the body.

> **i** Why is it called *abstraction*?
>
> - abstraction: *generalize* an expression into a function
> - application: *specialize* a function to a concrete argument

Bound and Free Variables

In a term like:

$$\lambda x.\, x \cdot 2 + 1$$

The variable x is bound the λ. Its **scope** is the **body** $x \cdot 2 + 1$.

In constrast consider:

$$\lambda x.\, x \cdot y + 1$$

Here x is bound and y is not. The variable y is **free**. Nevertheless this term is still syntactically meaningful. It represents a function of x, but one that depends on an external parameter y. For example, consider the reduction:

$$(\lambda x.\, x \cdot y + 1)\, 3 \to_\beta 3 \cdot y + 1$$

The result still contains the free variable y, and this term is not automatically interpreted as a function of two variables. For a function with x and y, we must bind both variables explicitly:

$$\lambda x.\, \lambda y.\, x \cdot y + 1$$

This is a **curried** two argument function: It first takes x, then returns a function that takes y. Consider the following reduction:

$$\begin{aligned}((\lambda x.\, \lambda y.\, x \cdot y + 1)\, 3)\, 4 &\to_\beta (\lambda y.\, 3 \cdot y + 1)\, 4 \\ &\to_\beta 3 \cdot 4 + 1 \\ &\to_\beta 13\end{aligned}$$

Typed Lambda Calculus

In typed lambda calculus, variables and functions have types.

$$\lambda x : \mathbb{N}.\, x \cdot 2 + 1 : \mathbb{N} \to \mathbb{N}$$

A two-argument function can be written as:

$$\lambda x : \mathbb{N}.\, \lambda y : \mathbb{N}.\, x \cdot y + 1 : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$$

By convention, arrow types associate to the right, so

$$\mathbb{N} \to \mathbb{N} \to \mathbb{N}$$

means:

$$\mathbb{N} \to (\mathbb{N} \to \mathbb{N})$$

So a two-argument function is really a function that takes one argument and returns another function. (**Curried** function notation)

Open Terms and Contexts

The term

$$\lambda x : \mathbb{N}.\, x \cdot y + 1$$

contains the free variable y. In typed lambda calculus, this term is not typeable in the empty context, because the type of y is unknown. But it is typeable in a context where $y : \mathbb{N}$ is assumed.

We can write this as

$$y : \mathbb{N} \vdash \lambda x : \mathbb{N}.\, x \cdot y + 1 : \mathbb{N} \to \mathbb{N}$$

This means:

$$\text{Assuming } y \text{ is a natural number} \lambda x : \mathbb{N}.\, x \cdot y + 1 \text{ is a function from } \mathbb{N} \text{ to } \mathbb{N}$$

By contrast,

$$\vdash \lambda x : \mathbb{N}.\, x \cdot 2 + 1 : \mathbb{N} \to \mathbb{N}$$

has no assumptions. It is a **closed term**.

Lambda Abstraction and Application as Introduction and Elimination

In type theory, abstraction and application can be understood as the introduction and elimination forms for function types, analogous to introduction and elimination of implication symbol in natural deduction calculus.

These should be understood as type judgment rules.

Fur a function type $A \to B$ consider:

Introduction

Consider the following:

$$\Gamma, x : A \vdash t : B$$

This judgment means:

> Assuming the type judgments in Γ (the context) and that the additional variable x has type A, then the term t has the type B in the same context Γ.

The lambda-introduction rule says:

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.\, t : A \to B}$$

Meaning:

> if t has type B under temporary assumptions $x : A$ and context Γ, then $\lambda x.\, t$ has type $A \to B$ under the original context Γ.

Elimination

The elimination rule for the function type $A \to B$ (also called function application and implication elimination):

With context, the rule is written as

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash a : A}{\Gamma \vdash f\, a : B}$$

The first premise

$$\Gamma \vdash f : A \to B$$

says:

> Under the assumptions Γ, f is a function that takes input A and produces output of type B.

The second premise

$$\Gamma \vdash a : A$$

says:

> Under the same assumptions in Γ, a is a term of type A.

The conclusion

$$\Gamma \vdash f\, a : B$$

says:

> Applying f to a gives a term of type B.

Lambda Abstraction in Rocq

Rocq uses they keyword `fun` for λ. So the lambda term:

$$\lambda x : \mathbb{N}. x \cdot 2 + 1$$

is written in Rocq as:

```
fun x : nat => x * 2 + 1
```

So in general

$$\lambda x : A. t$$

corresponds to

```
fun x : A => t
```

Lambda Abstraction in Rocq Function Definitions

Rocq lets us define fucntions in two equivalent styles. For example, Boolean negation can be written in the familiar argument style:

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

But it also can be written by explicitly giving a function value:

```
Definition negb : bool -> bool :=
  fun b : bool =>
    match b with
    | true => false
    | false => true
    end.
```

These two definitions are equivalent. So in general

```
Definition f (x : A) : B :=
  body.
```

```
Definition f : A -> B :=
  fun x : A => body.
```

are equivalent.

Multiple Arguments

Same idea applies. For example:

```
Definition g (x y : nat) : nat :=
  x * y + 1.
```

is shorthand for

```
Definition g : nat -> nat -> nat :=
  fun x y : nat => x * y + 1
```

where `fun x y : nat => x * y + 1` is itself a shorthand for:

```
fun x : nat => fun y : nat => x * y + 1.
```

Or in lambda calculus notation:

$$\lambda x\, y.\, x * y + 1$$

is a shorthand for:

$$\lambda x.\, \lambda y.\, x * y + 1$$

3.2 Booleans in Rocq's Core Library

Boolean values are available without any imports in Rocq. There are two canonical Boolean values:

- `true`
- `false`

Both are values of type `bool`.

```
Check true.
Check false.
```

```
true
     : bool

false
     : bool
```

In Rocq, every expression has a type. The expressions `true` and `false` have type `bool`. The type `bool` itself also has a type: Rocq classifies it as `Set`.

Roughly speaking, `Set` is the universe of computational data types, such as Booleans, natural numbers, lists, and other datatypes. More generally, `Set` itself lives in a larger universe called `Type`.

```
Check bool.
About bool.
Check Set.
Check Type.
```

```
bool
     : Set

bool : Set

bool is not universe polymorphic
Expands to: Inductive Corelib.Init.Datatypes.bool
Declared in library Corelib.Init.Datatypes, line 39, characters 10-14

Set
     : Type
```

```
Type
     : Type
```

A useful first picture is:

```
true  : bool
false : bool
bool  : Set
Set   : Type
```

So the hierarchy starts with ordinary values, then their types, and then universes classifying those types.

> **Note**
>
> In type theory, types are themselves objects that can be checked and classified. This is why commands like `Check bool`, `Check Set`, and `Check Type` are meaningful.

Certain pre-defined functions are also available out of the box and can be used to compute expressions. For example, `negb` is Boolean negation.

```
Check negb.
About negb.

Check negb false.
Compute negb false.
```

```
negb
     : bool -> bool

negb : bool -> bool

negb is not universe polymorphic
Arguments negb b%bool_scope
negb is transparent
Expands to: Constant Corelib.Init.Datatypes.negb
Declared in library Corelib.Init.Datatypes, line 84, characters 11-15

negb false
     : bool
```

```
     = true
     : bool
```

Curried Notation

A function of two variables like `orb` is written in curried notation. This means that the value of `orb` on two Boolean arguments is written as

```
orb true false
```

not as

```
orb (true, false)
```

The second expression would suggest that `orb` takes a pair as input. But Rocq's `orb` takes one Boolean argument first, and then returns a new function waiting for the second Boolean argument.

```
Check orb true false.
Check orb.
Check bool -> (bool -> bool).
Check orb true.
```

```
(true || false)%bool
     : bool

orb
     : bool -> bool -> bool

bool -> bool -> bool
     : Set

orb true
     : bool -> bool
```

The type

```
bool -> bool -> bool
```

should be read as

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

because arrow types associate to the right. Therefore, `orb true` is a function of type `bool -> bool`.

> **Note**
>
> A function type `A -> B -> C` means `A -> (B -> C)`. A function of two arguments is therefore a function that takes one argument and returns another function.

We can compute using this syntax.

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

```
     = false
     : bool
```

In the rest of the file, we construct basic functions from `bool` to `bool` and from `bool` to `bool -> bool`. Most of them are already contained in Rocq's core library, but redefining them is a good way to learn pattern matching.

3.3 Basic Pattern Matching

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

```
Module BooleanValues.
```

Inside the module, our own definitions can have the same names as functions from Rocq's core library. For example, when we define our own `negb`, it shadows the library function of the same name inside this module. The original library function still exists, but the unqualified name `negb` will refer to our local definition.

> **Note**
>
> A module gives us a local namespace. This lets us experiment with names like `negb`, `andb`, and `orb` without permanently overwriting Rocq's library definitions.

Boolean Negation

Let us start with the definition of the `negb` function, which sends `true` to `false` and `false` to `true`. We define it by pattern matching on Boolean values.

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

Check negb.
```

```
negb
     : bool -> bool
```

The definition says: to compute `negb b`, inspect the possible forms of `b`.

- If `b` is `true`, return `false`.
- If `b` is `false`, return `true`.

Since `bool` has exactly two constructors, `true` and `false`, these two cases are exhaustive.

```
About negb.
Compute negb false.
```

```
negb : bool -> bool

negb is not universe polymorphic
Arguments negb b%bool_scope
negb is transparent
Expands to: Constant Top.BooleanValues.negb
Declared in toplevel input, characters 11-15

     = true
     : bool
```

We can also give expressions a name and then use that name in later computations.

```
Compute negb (negb false).

Definition example_bool := negb false.

Compute negb example_bool.
```

```
     = false
     : bool

     = false
     : bool
```

> **Note**
>
> Pattern matching is Rocq's way of defining functions by cases. For `bool`, there are exactly two cases: `true` and `false`.

Boolean Conjunction

Now we define an `andb` function. It takes two Booleans `b` and `b'`, and returns their Boolean conjunction.

The truth table for Boolean conjunction is:

b	b'	andb b b'
true	true	true
true	false	false
false	true	false
false	false	false

This truth table can be translated directly into a pattern match with four cases.

```
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. If the first Boolean is `true`, then the result is just the second Boolean. If the first Boolean is `false`, then the result is always `false`, regardless of the second Boolean.

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

The symbol `_` is a wildcard pattern. It means: there is some value here, but we do not need to name it because the result does not depend on it.

Another possible, but less readable, way is to pattern match first on `b` and then on `b'`. This produces a nested program. It typechecks, but it is more verbose. Note that only the final `end` is followed by a period.

```
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 Boolean values.

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

```
     = false
     : bool

     = false
     : bool
```

Recall that `andb` is a function of two variables, but it is written in curried notation. If we ask Rocq to check its type, we get `bool -> bool -> bool`, not a function type from pairs.

```
Check andb.
Check andb true.
Compute andb true.
```

```
andb
     : bool -> bool -> bool

andb true
     : bool -> bool

     = fun b' : bool => if b' then true else false
     : bool -> bool
```

The expression `andb true` is a partially applied function. It is the function that takes a Boolean `b'` and returns `andb true b'`, which is just `b'`.

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

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

This means that an expression like

```
andb b b'
```

is parsed as

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

In contrast, the arrow type `(bool -> bool) -> bool` describes a function which takes a Boolean function as input and returns a Boolean value. We can define an example by evaluating a function `f : bool -> bool` at a given Boolean value.

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

Compute eval_at_true negb.
```

```
Compute eval_at_true (andb true).

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

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

```
     = false
     : bool

     = true
     : bool

     = true
     : bool

     = false
     : bool
```

Boolean Disjunction

Let us define an `orb` function on Booleans.

The truth table for Boolean disjunction is:

b	b'	orb b b'
true	true	true
true	false	true
false	true	true
false	false	false

If the first Boolean is `true`, then the result is always `true`. If the first Boolean is `false`, then the result is the second Boolean.

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

Since the first branch does not actually use `b'`, and the second branch only returns `b'`, we can also write the same idea as a match on `b` alone.

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

```
Compute orb false false.
Compute orb false (negb false).
```

```
     = false
     : bool

     = true
     : bool
```

Boolean Implication

Let us define Boolean implication. The intended truth table is:

b	b'	implb b b'
true	true	true
true	false	false
false	true	true
false	false	true

This corresponds to the classical idea that `P -> Q` is false only when `P` is true and `Q` is false.

For this one, we use Rocq's `if ... then ... else ...` syntax.

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

Here

```
if b then b' else true
```

is essentially shorthand for the following pattern match:

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

So `if ... then ... else ...` is not a separate mysterious mechanism. For Booleans, it is a convenient notation for pattern matching on a Boolean condition.

Exercise: implement the function `implb` using pattern matching. The result of the following computation should be the Boolean `true`.

```
Compute implb false true.
```

```
     = true
     : bool
```

Exercise: implement the Boolean if-then-else function as a function with type signature

```
bool -> bool -> bool -> bool
```

It should take three Boolean arguments:

```
condition -> then_branch -> else_branch -> result
```

One possible implementation is:

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

3.4 Theorem Proving

Next, we start using Rocq as a tool for proving properties of the functions we have defined. The syntax looks different at first, but in type theory stating a theorem is closely related to declaring a type, and proving a theorem is closely related to constructing a term of that type.

For example, the statement

```
negb (negb true) = true
```

is an equality proposition. To prove it means to construct a proof object inhabiting that proposition.

> **Note**
>
> In Rocq, a theorem statement is a type. Proving the theorem means constructing a term whose type is the theorem statement.

Negation Is Involutive

The first property that we prove is that, for every Boolean value `b`, we have

```
negb (negb b) = b
```

In ordinary language: applying Boolean negation twice gives us back the original Boolean value.

Let us start with some special cases, in which we prove our equality by direct computation. When using an interactive Rocq editor, the tactic state shows the current goal and helps guide the proof.

```
Theorem negb_negb_true_eq_true : negb (negb true) = true.
Proof.
  simpl.
```

```
Proving: negb_negb_true_eq_true

1 subgoal

-------------- (1/1)
true = true
```

The `simpl` tactic simplifies the goal by computation. In this case, `negb (negb true)` computes to `true`.

```
  reflexivity.
```

```
Proving: negb_negb_true_eq_true
```

```
No more subgoals
```

The `reflexivity` tactic proves goals where the two sides of an equality reduce to the same expression. It is stronger than a purely textual check: it can unfold definitions and compute enough to see that both sides are definitionally equal.

```
Qed.
```

If you are just getting started with theorem provers, focus here on the block that starts with `Proof` and ends with `Qed`. This is a Rocq proof script. It is written in the tactic language, while function definitions like `negb` and `andb` are written in the core term language, called Gallina.

In this simple example, `simpl` is not actually necessary, because `reflexivity` can perform the needed computation itself.

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

Here is the same proof written directly as a Gallina term. Note that we use `Definition` rather than `Theorem`.

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

The term `eq_refl` is the canonical proof that an expression is equal to itself. Since both sides reduce to the same Boolean, `eq_refl` is accepted.

You can check that the theorem and the definition produce essentially the same proof object using the `Print` command.

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

```
negb_negb_false_eq_false = eq_refl
     : negb (negb false) = false

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

```
The command has indeed failed with message:
Unable to unify "true" with "negb (negb false)".
```

The command `Fail reflexivity.` asks Rocq to check that `reflexivity` fails. This is useful for experimentation. The theorem is then closed with `Admitted`, which means that the statement is accepted without proof.

> **Warning**
>
> `Admitted` is not a real proof. It is useful while experimenting or developing a file, but admitted theorems should not be treated as established results in finished work.

Now let us prove the general theorem. We need to prove the statement for an arbitrary Boolean `b`. Since `b` can only be `true` or `false`, it is natural to split the proof into two cases.

The `destruct` tactic plays the same role in proofs that the `match` keyword plays in function definitions.

- In a function definition, `match b with ... end` defines a result by considering all possible forms of `b`.
- In a proof, `destruct b` proves the current goal by considering all possible forms of `b`.

Since `bool` has exactly two constructors, `destruct b` creates two subgoals: one for `b = true` and one for `b = false`.

```
Theorem negb_inv (b : bool) : negb (negb b) = b.
Proof.
  destruct b.
```

```
Proving: negb_inv

2 subgoals

-------------- (1/2)
negb (negb true) = true
-------------- (2/2)
negb (negb false) = false
```

After `destruct b`, the goal has been modified. In the first case, `b` has been replaced by `true`; in the second case, `b` has been replaced by `false`.

We use focusing dashes to separate the two subgoals.

```
  - simpl. reflexivity.
```

```
Proving: negb_inv

This subproof is complete, but there are some unfocused goals:

-------------- (1/1)
negb (negb false) = false
```

The second case can be solved by `reflexivity` alone.

```
  - reflexivity.
```

Instead of using `reflexivity`, we can also reuse a previous proof with the `exact` tactic. The theorem `negb_negb_false_eq_false` has exactly the type needed for the second goal.

```
  - exact negb_negb_false_eq_false.
Qed.
```

One may observe that `negb_inv true` is a proof of the equality

```
negb (negb true) = true
```

and that `negb_inv` itself is a proof of the universally quantified proposition

```
forall b : bool, negb (negb b) = b
```

```
Check negb_inv true.
Check negb_inv.
```

```
negb_inv true
     : negb (negb true) = true

negb_inv
     : forall b : bool, negb (negb b) = b
```

> **Note**
>
> A theorem with a variable, such as `Theorem negb_inv (b : bool) : ...`, behaves like a function that takes an argument `b : bool` and returns a proof of the corresponding statement.

Equivalent Definitions of Implication

Recall the formulas from classical logic:

```
(P   Q)   ¬P   Q
(P   Q)   ¬(P   ¬Q)
```

The first formula is often taken as a definition of implication in classical logic.

In this section, we interpret the logical connectives as Boolean functions:

Logical notation	Boolean function
`P   Q`	`implb b b'`
`¬P`	`negb b`
`P   Q`	`orb b b'`
`P   Q`	`andb b b'`

Since we are working with Boolean-valued functions, saying that two Boolean formulas are equivalent means saying that they compute to the same Boolean value for all inputs. Therefore, we express equivalence here using equality of Booleans.

Later, when working directly with propositions in `Prop`, logical equivalence will be expressed differently, for example using `<->`.

We first prove the Boolean version of

```
(P ∨ Q) ∧ ¬P ∧ Q
```

This exercise also shows how to structure a proof with pattern matching on several variables. First, we present a nested version: we destruct `b`, and then destruct `b'` in each branch. This gives two cases corresponding to the possible values of `b`, with two subcases corresponding to the possible values of `b'`.

```
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.
```

The tactic `all: reflexivity` applies `reflexivity` to all remaining subgoals. Here, after destructing `b'`, both remaining cases can be solved in the same way.

Next, we prove the Boolean version of

```
(P ∨ Q) ∧ ¬(P ∧ ¬Q)
```

Here we destruct `b` and `b'` simultaneously. This immediately creates four cases, much like the simultaneous pattern matching used in the definition of `andb`.

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

Notice that we used explicit parameters `(b b' : bool)` rather than implicit parameters `{b b' : bool}`. Braces introduce implicit arguments, which Rocq tries to infer from context. This is useful later, but explicit parameters are clearer at this introductory stage.

```
End BooleanValues.
```

Curry-Howard Correspondence

In Rocq, there is a deep analogy between ordinary functions and theorem with parameters. This is an instance of the Curry-Howard correspondence:

> Propositions behave like types Proofs behave like terms/values inhabiting those types

For example `bool` is a type, and its inhabitants are the Boolean values:

```
true : bool
false : bool
```

Similarly, a proposition such as

```
neg (neg b) = b
```

can be understood as a type. Its inhabitants are proofs of that proposition. So if we have

```
p : negb (negb b) = b
```

Then `p` is a proof object whose type is the proposition

```
negb (negb b) = b
```

Analogy between Ordinary Functions and Theorems

Consider the ordinary boolean function:

```
Definition f (b : bool) : bool :=
  negb b.
```

This says:

> `f` takes an input `b : bool` and returns an output of type `bool`

it's type is

```
f : bool -> bool
```

And compare with the theorem:

```
Theorem negb_inv (b : bool) : negb (negb b) = b.
```

This can be understood as saying:

> `negb_inv` takes `b : bool` and returns a proof of `negb (negb b) = b`.

After the theorem is proved, Rocq gives it the type:

```
negb_inv : forall b : bool, negb (negb b) = b
```

This is a dependent function type. It is called dependent becaus the result type depends on the input `b`. For example, if we apply `negb_inv` to `true`, we get:

```
negb_inv true : negb (negb true) = true
```

If we apply it to `false`, we get:

`negb_inv false : negb (negb false) = false.`

The important point is that `negb_inv true` is **not** a Boolean value. It is a proof object. More precisely, it s a proof of the proposition.

```
negb (negb true) = true
```

We can compare theorems and functions as follows:

Expression	Input	Output
`f`	`b : bool`	`negb b : bool`
`negb_inv`	`b : bool`	proof of `negb (negb b) = b`

The theorem

```
Theorem negb_inv (b : bool) : negb (negb b) = b.
Proof.
  destruct b.
  - reflexivity.
  - reflexivity.
Qed.
```

is conceptually similar to:

```
Definition negb_inv : forall b : bool, negb (negb b) = b :=
  fun b =>
    match b with
    | true => eq_refl
    | false => eq_refl
    end.
```

Set vs Type

The type `bool` lives in the universe `Set`:

```
Check bool.
```

```
bool
     : Set
```

whereas propositions such as equalities live in Prop:

```
Check negb (negb true) = true.
```

```
negb (negb true) = true
     : Prop
```

Both are type-like objects, but they have different roles:

Set: computational objects
Prop: logical propositions / proof types

Meaning of `Destruct`

The tactic `destruct` performs case analysis on an object according to the possible ways that object could have been constructed.

For example, if

```
b : bool
```

then `b` has only two possible constructors:

```
true : bool
false : bool
```

So when we write

```
destruct b.
```

Rocq splits the proof into two cases:

Case 1: `b = true`
Case 2: `b = false`

The word `destruct` is therefore connected to the idea of deconstructing a value: we inspect its outermost constructor and split into the corresponding cases.

The same idea applies to other inductive types. For natural numbers, a value of type `nat` is constructed either as `0` or as the successor of antoher natural nubmer. Therefore `destruct n.` splits the proof into the cases

Case 1: n = 0
Case 2: n = S n'

This is closely related to pattern matching. In a function definition, we write:

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

In a proof, we write:

```
destruct b.
```

The difference is:

> match: case-splits inside a term/program
> destruct: case-splits inside an interactive proof

3.5 Summary

In this lecture, we saw the following ideas:

- `true` and `false` are the two canonical values of type `bool`.
- `bool` is classified by the universe `Set`.
- Boolean functions can be defined by pattern matching.
- Functions of several arguments are usually curried in Rocq.
- The wildcard pattern `_` means that the value is irrelevant in that branch.
- `if ... then ... else ...` is a convenient notation for pattern matching on Booleans.
- Theorems are types, and proofs are terms inhabiting those types.
- `simpl` computes in the goal, while `reflexivity` proves equalities whose two sides reduce to the same expression.
- `destruct` performs case distinction in proofs, analogous to `match` in programs.
- Boolean equivalence of Boolean-valued expressions can be expressed as equality of Booleans.

4 Lecture 3 - Rocq Programming II

4.1 Inductive Formulas, Lists, and Substitution

Below is a consolidated summary of the whole Rocq lecture file `PropFormulas.v`, together with the extra conceptual points we discussed while revising it.

Defining well-formed formulas as an inductive datatype

The central datatype is:

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

This says that `Wff` is the type of well-formed propositional formulas.

Each constructor corresponds to a formation rule:

```
P n          is an atomic proposition.
Neg F        is the negation of F.
Impl F G     is the implication F   G.
Conj F G     is the conjunction F   G.
Disj F G     is the disjunction F   G.
```

So the inductive datatype corresponds directly to the grammar of propositional formulas:

```
F ::= P  | ¬F | F   F | F   F | F   F
```

The important point is:

```
An inductive type tells Rocq how objects of that type can be constructed.
```

So `Wff` is not just a set declared abstractly. It is generated by the constructors `P`, `Neg`, `Impl`, `Conj`, and `Disj`.

Constructor types

The declaration defines constructors with types:

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

Examples:

```
Check P.
Check Neg.
Check Impl.
```

For atomic propositions:

```
Check P 0.
Check P 42.
```

Both have type:

```
Wff
```

Composite formulas can be built by applying constructors:

```
Check Neg (P 0).
Check Impl (P 1) (P 2).
Check Conj (P 1) (P 2).
```

For example:

```
Definition F  := Neg (P 0).
Definition F  := Conj (P 1) (P 2).
```

Then:

```
Impl F  F
```

is again a formula of type `Wff`.

Currying and partial application

The constructor:

```
Impl : Wff → Wff → Wff
```

is curried.

This does not mean that `Impl` takes a pair:

```
Wff × Wff → Wff
```

Instead it means:

```
Impl : Wff → (Wff → Wff)
```

So `Impl` takes one formula and returns a function waiting for the second formula.

The lecture demonstrates this using:

```
Section Scratch.

  Variable (dummy_formula : Wff).
  Check Impl dummy_formula.

End Scratch.
```

Inside the section:

```
Variable (dummy_formula : Wff).
```

means:

```
Let dummy_formula be an arbitrary formula of type Wff.
```

Then:

```
Check Impl dummy_formula.
```

shows that after applying `Impl` to one formula, the result has type:

```
Wff → Wff
```

So:

```
Impl dummy_formula : Wff → Wff
```

Then, after applying a second argument:

```
Impl dummy_formula (P 0) : Wff
```

This illustrates partial application.

Sections and local variables

The block:

```
Section Scratch.
  Variable (dummy_formula : Wff).
  Check Impl dummy_formula.
End Scratch.
```

creates a temporary local context.

Inside the section, Rocq knows:

```
dummy_formula : Wff
```

After:

```
End Scratch.
```

the variable disappears.

So `Section ... End` is useful for temporary assumptions, local experiments, and definitions that depend on parameters.

We discussed that:

```
Variable dummy_formula : Wff.
```

and:

```
Variable (dummy_formula : Wff).
```

are essentially equivalent here.

Both mean:

```
Assume dummy_formula is an arbitrary element of Wff.
```

This is not an imperative-programming variable. It is not a mutable storage cell. It is more like the mathematical phrase:

```
Let F be an arbitrary well-formed formula.
```

We clarified the difference between `Variable` and `Definition`.

A declaration like:

```
Variable F : Wff.
```

means:

```
Assume an arbitrary formula F.
```

It does not construct a concrete formula.

A definition like:

```
Definition F : Wff := P 0.
```

means:

```
Define F to be the concrete formula P 0.
```

So `Variable` is an assumption or parameter, while `Definition` introduces a named term.

You cannot generally do this:

```
Variable x : A.
Definition x := val.
```

because `x` is already a name in the current scope. Also, `Variable x : A` is not an uninitialized placeholder waiting to be assigned later. It is already an arbitrary but fixed object of type `A`.

The usual Rocq workflow is not:

```
declare x, then assign x later
```

but rather:

```
work generally with x as a parameter,
then later instantiate definitions or theorems with a concrete value.
```

Notation declarations

The lecture introduces readable notation:

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

These do not create new constructors. They only tell the parser and printer how to read and display expressions.

So:

```
P 0  P 1
```

is notation for:

```
Conj (P 0) (P 1)
```

Similarly:

```
¬ P 0
```

is notation for:

```
Neg (P 0)
```

We also discussed that one cannot type-check an infix notation by writing:

```
Check /\.
```

or similarly for Unicode operators.

Notation is not a standalone term. It is a parsing pattern.

Instead, one can check:

```
Check Conj.
Check (P 0   P 1).
Check (fun a b => a   b).
```

The last one treats the notation as a binary operation by eta-expanding it into a function.

Defining formula height with Fixpoint

The first serious recursive function is the height function:

```
Fixpoint ht (F : Wff) :   :=
  match F with
  | P i       => 0
  | Neg F     => 1 + ht F
  | Impl F F' => 1 + max (ht F) (ht F')
  | Conj F F' => 1 + max (ht F) (ht F')
  | Disj F F' => 1 + max (ht F) (ht F')
  end.
```

Conceptually:

```
The height of an atom is 0.
The height of ¬F is 1 + height of F.
The height of a binary formula F   G is 1 + max(height F, height G).
```

The important Rocq idea:

```
Fixpoint is used for recursive definitions.
```

For ordinary non-recursive functions, one writes:

```
Definition f (x : A) := ...
```

But for recursive functions, the function body refers to the function itself, so one uses:

```
Fixpoint f (x : A) := ...
```

Rocq only accepts recursive definitions when it can verify termination. In this case, recursive calls are made on structurally smaller subformulas:

```
ht F
ht F'
```

where `F` and `F'` are subformulas of the original formula.

Why Fixpoint is needed

A recursive function such as height satisfies equations like:

```
ht (Neg F) = 1 + ht F
ht (Conj F G) = 1 + max (ht F) (ht G)
```

So the function refers to itself.

A plain `Definition` cannot introduce such self-reference directly. That is why Rocq has a special recursive definition construct:

```
Fixpoint
```

We discussed that `Fixpoint` is related to the mathematical idea of fixed points.

A recursive definition can be seen abstractly as an equation:

```
f = T(f)
```

where `T` is an operator that takes a candidate function and returns an improved/new function.

For factorial, one can define an operator:

```
T(g)(0)     = 1
T(g)(n + 1) = (n + 1) · g(n)
```

Then factorial is a fixed point of `T`:

```
fact = T(fact)
```

This operator viewpoint is not necessary for ordinary Rocq programming, but it explains the word "Fixpoint" and connects to recursion theory, lambda calculus, domain theory, and denotational semantics.

In Rocq, however, the practical rule is:

```
Fixpoint = recursive function accepted only if Rocq can see termination.
```

Lower-level `fix`

The lecture then shows that the height function can also be defined using the lower-level `fix` term:

```
Definition ht  : Wff →   :=
  fix ht (F : Wff) :   :=
    match F with
    | P i       => 0
    | Neg F     => 1 + ht F
    | Impl F F' => 1 + max (ht F) (ht F')
    | Conj F F' => 1 + max (ht F) (ht F')
    | Disj F F' => 1 + max (ht F) (ht F')
    end.
```

Here:

```
fix ht (F : Wff) :   := ...
```

is an explicit recursive function term.

The name `ht` inside the `fix` expression is the self-reference used for recursive calls.

So:

```
Fixpoint ht (F : Wff) :   := ...
```

is a convenient command-level syntax, while:

```
Definition ht  : Wff →   :=
  fix ht (F : Wff) :   := ...
```

uses the lower-level recursive term former.

The lecture then proves:

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

This shows that the two definitions coincide.

Goal versus Theorem

We discussed:

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

`Goal` starts an anonymous theorem.

It is like:

```
Theorem ht_eq_ht0 : ht = ht .
Proof.
  reflexivity.
Qed.
```

except the result is not given a useful global name.

So `Goal` is often used for:

```
small experiments
lecture demonstrations
checking that something is provable
```

Why reflexivity proves `ht = ht`

The tactic:

```
reflexivity.
```

proves equalities whose two sides are equal after computation/unfolding.

It does not only check textual equality. It checks definitional equality.

Here, `ht` and `ht` reduce to the same underlying recursive function. Therefore:

```
reflexivity
```

can prove:

```
ht = ht
```

This is an example of equality by computation.

This differs from equalities requiring real reasoning, such as:

```
forall n : nat, n + 0 = n
```

which usually needs induction.

Defining a function using tactics

The lecture then defines height again, this time using proof/tactic mode:

```
Definition ht  (F : Wff) :  .
Proof.
  induction F.
  - exact 0.
  - exact (1 + IHF).
  - exact (1 + (max IHF1 IHF2)).
  - exact (1 + (max IHF1 IHF2)).
  - exact (1 + (max IHF1 IHF2)).
Defined.
```

This looks like a proof, but it defines a function.

The line:

```
Definition ht  (F : Wff) :  .
```

means:

```
Define ht .
It takes F : Wff.
It must return a natural number.
I will construct the body interactively.
```

After this, Rocq enters proof mode. The goal is:

```

```

under the assumption:

```
F : Wff
```

In type theory, proving and constructing are the same kind of activity. A goal is a type, and solving the goal means constructing a term of that type.

If the goal is a proposition, the term is a proof.

If the goal is , the term is a natural number.

So this tactic script constructs a program.

The role of `induction F`

The tactic:

```
induction F.
```

splits into cases according to the constructors of `Wff`.

Since `Wff` has five constructors, it produces five cases:

```
F = P i
F = Neg F
F = Impl F1 F2
F = Conj F1 F2
F = Disj F1 F2
```

In each case, the goal is to construct a natural number, the height.

For the atom case:

```
exact 0.
```

For the negation case, Rocq provides an induction result:

```
IHF :
```

which plays the role of the recursively computed height of the subformula. So:

```
exact (1 + IHF).
```

For binary constructors, Rocq provides:

```
IHF1 :
IHF2 :
```

corresponding to the recursively computed heights of the two subformulas. So:

```
exact (1 + (max IHF1 IHF2)).
```

The correspondence is:

```
Fixpoint recursive call      tactic version

ht F                         IHF
ht F1                        IHF1
ht F2                        IHF2
```

So the induction tactic builds the same recursive function.

Why `Defined` instead of `Qed`

The tactic-defined function ends with:

```
Defined.
```

rather than:

```
Qed.
```

The difference:

```
Qed      closes the proof opaquely.
Defined  closes it transparently.
```

For ordinary proofs, `Qed` is common.

For functions/programs built with tactics, `Defined` is usually needed if we want Rocq to compute with the definition.

So if `ht` is ended with `Defined`, then:

```
Compute ht  some_formula.
```

can unfold and compute.

If ended with `Qed`, the definition may become opaque and computation will not unfold it as expected.

The version using `all:`

The lecture then defines another version:

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

This is the same function again.

The only new syntax is:

```
all: tactic.
```

It means:

```
Apply this tactic to all remaining goals.
```

After:

```
induction F.
```

there are five goals.

The first goal is solved by:

```
exact 0.
```

The second goal is solved by:

```
exact (1 + IHF).
```

The remaining three goals are the binary cases: implication, conjunction, disjunction.

They all have the same shape, with two induction results:

```
IHF1 :
IHF2 :
```

So the same expression works for all of them:

```
all: exact (1 + (max IHF1 IHF2)).
```

This is more compact, but less explicit than the bullet version.

Introduction of lists

The next part introduces lists because they are needed to compute subformulas.

The lecture presents the inductive definition:

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

This says:

```
For every type A, list A is the type of lists whose elements have type A.
```

Examples:

```
list nat
list bool
list Wff
```

The type constructor:

```
list
```

has type:

```
Type → Type
```

So `list` itself is not a type of elements; rather, it takes a type and returns a type.

This is analogous to C++ templates:

```
std::vector<int>
std::vector<bool>
std::vector<Wff>
```

correspond to:

```
list nat
list bool
list Wff
```

The analogy is useful at the type-constructor level:

```
C++:   vector<T>
Rocq:  list A
```

But Rocq lists are pure inductive linked-list-like values, not mutable containers with capacity and memory operations.

Why `Inductive list (A : Type) : Type`

We discussed the difference between:

```
Inductive Wff := ...
```

and:

```
Inductive list (A : Type) : Type := ...
```

For `Wff`, Rocq can infer:

```
Wff : Type
```

One could explicitly write:

```
Inductive Wff : Type := ...
```

For `list`, the declaration means:

```
For each type A, list A is a Type.
```

So:

```
list : Type → Type
```

and:

```
list A : Type
```

The final `: Type` says that once `A : Type` is given, `list A` is itself a type.

We noted that Rocq can often infer the final `: Type`, so one could often write:

```
Inductive list (A : Type) := ...
```

But the lecture writes it explicitly for clarity.

List constructors

The constructors are:

```
nil  : list A
cons : A → list A → list A
```

More explicitly, because `list` is polymorphic:

```
nil  : forall A : Type, list A
cons : forall A : Type, A → list A → list A
```

So fully explicit uses look like:

```
nil bool
cons bool true (nil bool)
cons nat 3 (nil nat)
cons Wff (P 0) (nil Wff)
```

But this is verbose.

Arguments and implicit parameters

The lecture comments out or discusses:

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

These commands control implicit arguments.

The curly braces mean:

```
This argument should be implicit; Rocq should infer it when possible.
```

So:

```
Arguments nil {A}.
```

means:

```
The type parameter A of nil is implicit.
```

And:

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

means:

```
The type parameter A of cons is implicit.
The element a and tail l remain explicit.
```

Without implicit arguments, one writes:

```
cons bool true (nil bool)
```

With implicit arguments, one can write:

```
cons true nil
```

and Rocq infers:

```
A = bool
```

from `true : bool`.

We also discussed the `@` notation:

```
@nil
@cons
```

The `@` forces Rocq to show or use all arguments explicitly, including implicit ones.

So:

```
cons true nil
```

is roughly shorthand for:

```
@cons bool true (@nil bool)
```

Friendlier list notation

The lecture then introduces standard list notation:

```
Open Scope list_scope.
```

and imports list notation:

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

This allows syntax such as:

```
[1; 2; 3]
1 :: [2; 3]
[]
```

The notation:

```
x :: xs
```

means:

```
cons x xs
```

So:

```
1 :: 2 :: 3 :: nil
```

means:

```
cons 1 (cons 2 (cons 3 nil))
```

And:

```
[1; 2; 3]
```

is nicer notation for the same list.

Strict subformulas

The lecture then defines a function computing strict subformulas:

```
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 meaning:

```
An atom has no strict subformulas.

The strict subformulas of ¬F are:
  F itself, plus the strict subformulas of F.

The strict subformulas of F   G are:
  F and all strict subformulas of F,
  together with G and all strict subformulas of G.
```

The operator:

```
++
```

is list concatenation.

So:

```
(F :: strict_sf F) ++ (F' :: strict_sf F')
```

means:

```
combine the left-subformula list and right-subformula list
```

Then the full subformula list is defined as:

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

So:

```
sf F = F itself plus all strict subformulas of F.
```

Substitution in formulas

The lecture then defines formula substitution:

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

Here:

```
I :   → bool
```

is a predicate/characteristic function deciding which atomic propositions should be replaced.

And:

```
G :   → Wff
```

is a replacement family. It tells us what formula should replace `P j`.

In the atomic case:

```
P j
```

the function does:

```
if I j then G j else P j
```

So:

```
If j is selected by I, replace P j with G j.
Otherwise leave P j unchanged.
```

For compound formulas, substitution recursively enters the subformulas and rebuilds the same connective.

For example:

```
subst(¬F) = ¬ subst(F)
subst(F   G) = subst(F)   subst(G)
```

So this function is a recursive traversal over the syntax tree.

Example characteristic function

The lecture defines a concrete selection function:

```
Definition I (n :  ) : bool :=
  match n with
  | 0    => true
  | 2026 => true
  | 41   => true
  | 42   => true
  | _    => false
  end.
```

This selects some natural-number indices.

We noticed that if the comment says this represents `{0; 2026; 42}`, then the code also includes `41`, so the actual function selects:

```
0, 41, 42, 2026
```

Example replacement families

The file then defines replacement functions such as:

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

This means selected atoms `P n` are replaced by:

```
P (n + 2)  P (n + 1)
```

Another example:

```
Definition G :  → Wff :=
  fun n => match n with
  | 2025 => P 44
  | 2026 => P 1  P 2
  | _    => P n
  end.
```

This replacement function changes only certain indices. But substitution still happens only when `I n = true`.

So even if `G 2025 = P 44`, the atom `P 2025` is not replaced unless `I 2025 = true`.

A third example uses evenness:

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

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

This replacement family changes even indices by halving them, and leaves odd indices unchanged.

Again, the final substitution is controlled by both:

```
I decides whether replacement happens.
G decides what replacement is used.
```

General substitution for indexed families

The lecture abstracts the atomic substitution pattern into a general function:

```
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.
```

This is polymorphic in two types:

```
Y : Type
X : Type
```

Because they are written in curly braces:

```
{Y} {X}
```

they are implicit arguments.

The function takes:

```
P : Y → X
I : Y → bool
G : Y → X
y : Y
```

and returns:

```
if I y then G y else P y
```

So it says:

```
Given an original family P,
a selection predicate I,
and a replacement family G,
produce the substituted family.
```

In the formula case:

```
Y =
X = Wff
P :   → Wff
```

where `P n` is the atomic formula indexed by `n`.

Second substitution function

The file then defines another formula substitution function:

```
Fixpoint subst_Wff' (I :   → bool) (G :   → Wff) (F : Wff) : Wff :=
  match F with
  | P j     => substFamily P I G j
  | ¬F      => ¬(subst_Wff' I G F)
  | F   F  => (subst_Wff' I G F )   (subst_Wff' I G F )
  | F   F  => (subst_Wff' I G F )   (subst_Wff' I G F )
  | F   F  => (subst_Wff' I G F )   (subst_Wff' I G F )
  end.
```

This is essentially the same as `subst_Wff`.

The difference is that the atomic case uses:

```
substFamily P I G j
```

instead of writing directly:

```
if I j then G j else P j
```

Also, the argument order is different.

Original:

```
subst_Wff : Wff → (  → bool) → (  → Wff) → Wff
```

New version:

```
subst_Wff' : (  → bool) → (  → Wff) → Wff → Wff
```

This order is often useful because, after fixing `I` and `G`:

```
subst_Wff' I G
```

becomes a function:

```
Wff → Wff
```

So it is a formula transformer.

Final exercise: adding biconditional

The file ends with an exercise: extend the language with biconditional.

Add a constructor:

```
Iff : Wff → Wff → Wff
```

Then define notation, for example:

```
Notation "a  b" := (Iff a b) ...
```

or in ASCII:

```
Notation "a <=> b" := (Iff a b) ...
```

Then update all recursive functions:

```
ht
strict_sf
subst_Wff
```

because whenever we extend an inductive datatype with a new constructor, every function that pattern-matches on that datatype must handle the new case.

This is an important structural lesson:

```
Inductive datatype = syntax/grammar.
Recursive functions over it = one case per constructor.
If you add a constructor, you must update the recursive consumers.
```

The exercise also asks you to think about precedence and associativity, because notation levels determine how expressions are parsed.

30. The central conceptual pattern of the whole lecture

The whole lecture revolves around one fundamental pattern:

```
Define syntax as an inductive datatype.
Then define recursive functions by pattern matching on that syntax.
```

For this file:

```
Wff defines the syntax of formulas.

ht consumes a formula and returns a number.
strict_sf consumes a formula and returns a list of formulas.
subst_Wff consumes a formula and returns a transformed formula.
```

All of these functions follow the same shape:

```
Fixpoint f (F : Wff) :=
  match F with
  | P i       => ...
  | Neg F     => ... f F ...
  | Impl F G  => ... f F ... f G ...
  | Conj F G  => ... f F ... f G ...
  | Disj F G  => ... f F ... f G ...
  end.
```

This is the core programming pattern for inductive syntax trees.

The deeper lessons from our supplementary discussion

Beyond the lecture itself, we clarified several important Rocq/type-theoretic ideas.

`Variable` is not an imperative variable

```
Variable x : A.
```

does not mean "create a mutable slot."

It means:

```
Assume x is an arbitrary element of A.
```

`Definition` introduces a fixed named term

```
Definition x : A := val.
```

means:

```
Define x to be val.
```

It is closer to a constant binding than to a mutable variable.

`Fixpoint` is controlled recursion

Rocq only accepts recursive definitions if it can verify termination, usually by structural recursion.

This is essential because Rocq is a proof assistant. Unrestricted recursion would endanger logical consistency.

`fix` is the lower-level recursive term

```
fix f (x : A) : B := ...
```

is the term-level recursive function construct.

`Fixpoint` is the usual top-level command syntax.

`Goal` is an anonymous theorem

```
Goal P.
Proof.
...
Qed.
```

starts a proof of `P` without naming it.

`reflexivity` proves definitional equality

It works when both sides are the same after computation/unfolding.

Tactics can build programs

A tactic script can construct a proof, but it can also construct ordinary data or functions.

This works because in type theory:

```
proofs and programs are both terms.
propositions and data types are both types.
```

`Defined` keeps the term transparent

For functions built interactively, use:

```
Defined.
```

if you want computation to unfold them.

Polymorphic inductive types are type constructors

```
list : Type → Type
```

is analogous, loosely, to a C++ template like:

```
template <typename T>
class vector;
```

Applying it to a type produces a concrete type:

```
list Wff : Type
```

Implicit arguments reduce verbosity

Constructors of polymorphic types have hidden type parameters.

With implicit arguments, one writes:

```
cons true nil
```

instead of:

```
@cons bool true (@nil bool)
```

Best mental model for the lecture

A good final mental model is:

```
Wff is a syntax tree type.

P, Neg, Impl, Conj, Disj are the constructors for building syntax trees.

Notation makes those trees look like logical formulas.

Fixpoint defines recursive tree traversals.

ht computes a number from a formula tree.

strict_sf collects formula subtrees.

subst_Wff transforms a formula tree by replacing selected atoms.

list is introduced because subformulas naturally form a list.

substFamily abstracts the indexed replacement operation.
```

So the lecture is not merely about propositional logic. It is really an introduction to the Rocq pattern:

```
inductive syntax + recursive functions over syntax
```

This pattern will return constantly in type theory, proof assistants, programming-language semantics, lambda calculus, formal grammars, operational semantics, and verified interpreters.

References

5 Resources

Books & other resources

5.1 SML

- Elements of Functional Programming. Reade
- Introduction to Programming Using SML. Hansel, Rischel
- Programming in SML. Harper
- Programmierung: Eine Einfuehrung in Standard ML. Gert Smolka

5.2 Haskell

- Functional Programming with Gofer. Fokker
- Introduction to Computation, Haskell, Logic, and Automata. Wadler
- Introduction to Functional Programming using Haskell. Bird
- Introduction to Functional Programming Systems Using Haskell
- Thinking Functionally with Haskell. Bird
- Programming in Haskell. Graham Hutton
- Haskell - The Craft of Functional Programming. Thompson

5.3 OCAML

- OCaml from the very beginning. Whitington
- More OCaml. Whitingnton
- OCaml - Functional Programming for the Masses. Madhavapeddy

5.4 Rocq / Coq

- Mathematical Components. Assia Mahboubi, Enrico Tassi.
- Software Foundations. Benjamin C. Pierce
- Programs and Proofs - Mechanizing Mathematics with Dependent Types. Ilya Sergey
- Certified Programming with Dependent Types. Adam Chipala
- Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions. Bertot, Casteran