Logic and Proofs

CSE 203

Ecole polytechnique

Bachelor Program

Pierre-Yves Strub and Benjamin Werner

Fall semester 2021

Contents

Chapter 1

Introduction

1.1 What are mathematics ?

For most high-school pupils or even university students, it is not easy to give a precise definition of what mathematics are. Yet, there is a clear feeling that certain pieces of reasoning are mathematical. This perception relies on the fact that mathematical reasoning is somehow *more precise*, that mathematics leave no room for ambiguity. The task of *mathematical logic* is to make this explicit by writing down the exact rules which define what is a correct mathematical reasoning, that is a mathematical *proof*.

The greek philosopher Aristotle gives a famous example:

> All men are mortal,
>
> Socrates is a man,
>
> *therefore*, Socrates is mortal.

The point of this example is not to point out that Socrates is mortal; we all know he died long ago. It has neither anything to do with the nature of a man of what it means to be mortal. The point is that the validity of this little piece of reasoning relies only on the *syntax* of the sentences involved: *if* all men are mortal and Socrates is a man, *then* Socrates has to be mortal.

The fundamental idea of mathematical logic is that a correct mathematical proof can be detailled all the way down to such syntactical rules. In other words, these syntactical rules are the *elementary bricks* out of which mathematics are constructed.

One aim of this course is to make this explicit and show what these logical rules look like.

There are several such rules and several ways to write or represent them. Let us give a very simple example; if σ_A is a proof of a mathematical proposition A and σ_B a proof of a proposition B, then these two proofs can be combined to form a new proof of the proposition $A \wedge B$.

$$\frac{\dfrac{\sigma_A}{\vdash A} \quad \dfrac{\sigma_B}{\vdash B}}{\vdash A \wedge B}$$

Rules can be stacked onto each other in order to build more complex proofs. For instance if we have another proof σ_C of another proposition C :

$$\dfrac{\dfrac{\dfrac{\sigma_A}{\vdash A}\quad\dfrac{\sigma_B}{\vdash B}}{\vdash A\wedge B}\quad\dfrac{\sigma_C}{\vdash C}}{\vdash (A\wedge B)\wedge C}$$

This particular notation for formal mathematical proofs is called *natural deduction* and we will go back to it further down. But let us already point out that it involves a data-structure which is widely used in computer science and programming: proofs are *trees*.

1.2 Where the computer comes in

Let us elaborate on this last remark.

Formal mathematical logic was born towards the end of the XIXth century. The idea that mathematical correctness could be perfectly defined without any room for ambiguity was reassuring for mathematicians. But in practice, it was impossible to write down complex proofs formally, because totally detailled proofs are too long: one would spend a life-time filling pages with formal symbols. Thus, the common mathematical practice was, and still is, the following:

- A mathematical statement is true if a formal proof exists *in principle*.
- A mathematical text is written in natural language (*ie.* english, french, chinese...) and aims at convincing the reader that this formal proof does exists. In other words, a mathematical text is an informal description of a formal proof.

This situation changed with the arrival of the computer. The computer can manipulate huge syntactical structures including formal proofs. It can help construct them, and it can certify that they are correct by checking that each single step is indeed built through a valid logical rule.

Such a software which manipulates formal mathematical proofs is called a *proof system*. The first proof system was called *Automath* and was developed around 1968 onwards by the team of the important dutch mathematician Nicolaas Govert de Bruijn. Proof systems have evolved since and modern proof systems include the softwares *Isabelle*, HOL and HOL-light, Alf, PVS. In this course we will work with the system *Coq* (french for rooster), developed mainly in France around Inria (*Institut National de Recherche en Informatique et Automatique*).

It is important to stress what we await from the computer in a proof system. The machine can help the user to prove a theorem, that is to construct a proof. But the key task is to *verify* the formal proof. For this latter task, we do *not* want the computer to be intelligent; we rather expect it to be precise and unforgiving. A human mathematician tries to *understand* a proof using his intelligence. The computer just formally verifies to proof step by step without global understanding. The advantage is that by doing that, the computer, unlike the intelligent mathematician, does not make mistakes: if a formal proof is cleared by the proof-checking program, we are certain that it is error-free; or at least we have the highest possible level of trust that the proof is correct.

In other words, doing a formal proof is often more tedious than doing the same proof informally, "on paper". The reward is perfect correctness.

1.3 The quest for certainty

It is satisfying to be sure that a proof is correct. There are however cases where this certainty has concrete material consequences. A very important application field is the one of correctness

proofs for software. There are several reasons for that:

- The proof that some piece of software has a certain behavior is a particular form of mathematical proof.
- These proofs can be, in part, quite tedious and repetitive. Which means it is easy to make a "stupid" mistake when proving a program correct.
- Even such a stupid error is enough to let a software bug slip through and thus have a program that crashes at runtime.

There is, therefore, a strong incentive for using formal proofs, or more generally *formal methods*, to prove the correctness of software. Because formally proving a programm correct is a potentially long and difficult task, one does so especially for softwares whose correctness is critical. This often means that human life, or large sums of money are at stake. Examples include:

- Embedded software, especially when it controls vehicles (automobiles, trains) or planes (fly-by-wire systems like since the A320 for Airbus or the 777 for Boeing).
- Medical software (chirurgical robots, pacemakers...)
- Communication protocols (on the internet, but also between a chipcard and its terminal for instance)

Formal proofs are also of interest for mainstream mathematicians. The proofs of some theorems being notoriously difficult to check "by hand". Milestone examples include:

- The proof of the four-color theorem which was first proved in 1976. The point being that all known proofs of this theorem involve symbolic computations which are too long to be performed by hand.
- The proof of Kepler's conjecture about the dense packing of spheres. This result was conjectured in 1612 and was proved only almost 400 years later by Thomas Hales (Univ. of Pittsburgh). The proof is mathematically more complex than the one of the four-color theorem, and also involves, predominantly numerical, computations. The proof for totally formalized in the proof-system HOL after 10 years of work.
- The classification of odd-order finite groups. This proof is more "conventional" in the sense that it does not involve machine computations. But the sheer length of this mathematical work (over 200 pages) was considered enough by a team at Inria and Microsoft, led by Georges Gonthier, to embark on a formalization effort of this proof in Coq. It was completed in 2012.

1.4 Why this course

Glancing through the exemples above, one can see that they often lie at the intersection of mathematical reasoning and mechanical computation. We believe that understanding how to reason about computations on one hand, and how to decompose mathematical reasoning into computational simple steps on the other, paves the way for a better understanding of both computation and deduction.

As said above, formal methods are already important in the advanced software industry. But even when one writes a program which is not meant to be formally verified, being knowledgeable in formal certification can give insights which help to design cleaner, simpler and more robust programs.

In the same way, having gone through the process of formalizing mathematical proofs underlines the importance of subtle choices which are to be made when designing and then writing down a proof.

1.5 Our approach

Classically, a first course of mathematical logic presents the rules of mathematical logic and then goes on proving properties about them. In this course we put the proof system forward. In other words, we focus less on the detail of the mathematical rules and more on how they are used in practice. This is much closer to the way one learns computer programming.

Chapter 2

Propositional calculus

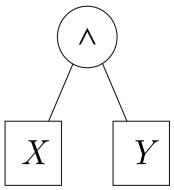
2.1 A fragment of logic

Propositional logic is a small formal language which is an important part of the language of mathematics. Although is it not sufficient to express interesting mathematical statements, it is at the core of about all mathematical formalisms. It is also a good way to apprehend what are logical rules and the way they are used in Coq.

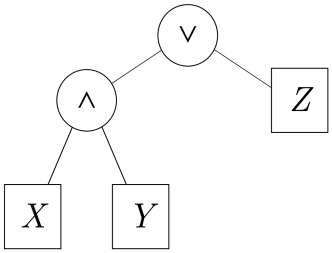
The set of propositional formulas is the smallest set verifying the following clauses:

- a propositional variable $X, Y, Z \ldots$ is a propositional formula.
- If A and B are propositional formulas, then $A \wedge B$, $A \vee B$ and $A \Rightarrow B$ are propositional formulas.
- $\bot$ (false) is a propositional formula.

Such a definition is called an *inductive definition*. This way of defining sets is often used in logic and we will become more familiar in the process of this course. For the moment let us just note that it means that propositions can be viewed as tree-structures. The proposition $X \wedge Y$ can be drawn as :



or $(X \wedge Y) \vee Z$ can be drawn as :



The operators allowing to build formulas are called *connectors*:

- The connector $\wedge$ stand for "and" and is called *conjunction*.
- The connector $\vee$ stand for "or" and is called *disjunction*.
- The connector $\Rightarrow$ stand for the *logical implication*.

2.2 Some syntax

In Coq, the connectors $\wedge, \vee$ and $\Rightarrow$ are respectively written `/\`, `\/` and `->`.

The negation of a formula A, or "not A", is written $\neg A$ on paper and `~A` in Coq. We can consider it as an abbreviation for $A \Rightarrow \bot$.

In Coq

In Coq, propositions are objects of a type Prop, which is a special constant.

We can introduce propositional variables by:

```
Coq <   Parameter X Y Z T : Prop.
X is declared
Y is declared
Z is declared
T is declared
```

We can now construct propositions. The Coq command `Check` verifies that an object is well-typed:

```
Coq < Check X.
X
     : Prop
Coq < Check X /\ Y.
X /\ Y
     : Prop
Coq < Check X -> (Y \/ Z).
X -> Y \/ Z
     : Prop
```

2.3 Rules

2.3.1 Logical statements

As in the example given in the previous chapter, proofs deal with hypotheses (*all men are mortal*) and conclusions (*Socrates is mortal*). The hypotheses and the conclusion have thus to be separated in the logical statement. We present logical rules using the following notation for logical statements: $\Gamma \vdash A$, where Γ is a set of propositions that are assumed true in the judgement. On the other hand, A is the conclusion of the statement. In other words, $\Gamma \vdash A$ stands for: "If the propositions Γ are true, then A is true.

This can be seen in the axiom rule which states that under hypothesis A, A is true.

$$\frac{A \in \Gamma}{\Gamma \vdash A}$$

2.3.2 Rules for conjunction

For every connectors, there are so-called introduction rules and elimination rules.

Let us look at the case of conjunction; the introduction rule gives a way to prove a conjunction: out of two proofs of A and B one can build a proof of $A \wedge B$:

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}$$

The elimination rules, on the other hand, show how to use a connector. In the case of conjunction there are two rules which say that from a proof of $A \wedge B$ one can get respectively a proof of A and a proof of B:

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}$$

2.3.3 Rules for implication

The rules for implication are interesting because they involve manipulations of the set of hypotheses.

To prove $A \Rightarrow B$ means proving B under the hypothesis A. This is precisely what is stated by the introduction rule of implication:

$$\frac{\Gamma; A \vdash B}{\Gamma \vdash A \Rightarrow B}$$

The other rule involving implication is famous under the latin denomination of *modus ponens*. If we know $A \Rightarrow B$ and A then we can deduce B:

$$\frac{\Gamma \vdash A \Rightarrow B \qquad \Gamma \vdash A}{\Gamma \vdash B}$$

2.3.4 The false proposition

The proposition "false", written $\bot$ is special. It is special enough to have a rule of its own: if we can prove $\bot$ under some hypotheses, then we can also prove any other proposition out of it:

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash A}$$

Note that there is no constraint on the conclusion A here. We say that the context Γ is inconsistent (in french *incohérent*).

This means $\bot$ is a connector with an elimination rule (written above) and no introduction rule (since there is no canonical way to prove false).

2.3.5 The rules for disjunction

The introduction rules for disjunction, that is the connector "or" are not surprising. On can prove $A \vee B$ out of a proof of A or out of a proof of B. So there are two rules:

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}$$

The elimination rule which defines how to use disjunction is less intuitive. It says if we can deduce a proposition C from A and from B, then one can deduce C from $A \vee B$.

$$\frac{\Gamma; A \vdash C \qquad \Gamma; B \vdash C}{\Gamma; A \vee B \vdash C}$$

This rule will become clearer when we will start to build proofs in Coq.

2.4 Doing proofs in Coq

One interacts with Coq through text commands. These commands are the same in the different user interfaces; mainly the proof-general mode under emacs, or the GUI CoqIDE. We therefore do not go into details regarding these interfaces here.

2.4.1 Starting

In this course, we will use a special version of Coq's proof language, called Ssreflect. We switch to ssreflect mode by the command:

```
Coq < Require Import ssreflect.
```

In order not to have to type this line at each time, you can put it in the file `.coqrc` in your home directory.

We have already mentioned the command to introduce some propositional variables:

```
Coq <   Parameter X Y Z T : Prop.
X is declared
Y is declared
Z is declared
T is declared
```

And the command `Check` which verifies that an expression is well-built:

```
Coq <   Check X.
X
     : Prop
Coq <   Check X -> Y.
X -> Y
     : Prop
Coq <   Check False.
False
     : Prop
```

2.4.2 First proof

Let us prove the possibly simplest theorem, that is the fact that the proposition `X` implies itself. We start by stating this theorem, and we give it a name:

```
Coq <   Theorem ex1 : X -> X.
1 subgoal

  ============================
  X -> X
```

We have not proven the theorem yet; we are in an *interactive* proof mode, where the *goal* to be proven is under the double bar.

This goal will evolve as we construct the proof through special commands called *proof tactics*. Here, the first tactic is the command `move =>` which allows us to push the left part of the goal over the bar:

```
Coq <   move => x.
1 subgoal

  x : X
  ============================
  X
```

The goal to be proved is now `X`, and over the bar we have an *assumption* `X`. Note that this assumption, or hypothesis, is named; here we gave it the name `x`.

Formally one can see that this tactic actually corresponded to using the introduction rule for $\Rightarrow$. But this is not what we focus on now.

We can now use the assumption `x` to finish the proof:

```
Coq <   exact x.
No more subgoals.
```

The system now states that there is no goal left, which means that the proof is finished. We can now save the results by the command `Qed` which is the abbreviation for *quod erat demonstratum*[1].

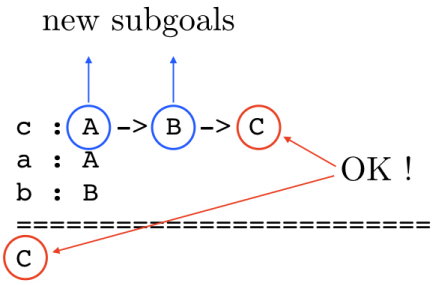
```
Coq <   Qed.
ex1 is defined
```

After this, the "theorem" `ex1` is part of the database of proved statements of the system and can be used in proofs to come.

2.4.3 Chains of implications

There is a syntactical convention that $A \Rightarrow B \Rightarrow C \Rightarrow D$ (or in Coq notation `A->B->C->D` corresponds to $A \Rightarrow (B \Rightarrow (C \Rightarrow D))$ which means that under assumptions A, B and C, the proposition D holds. In other words, the statement is equivalent to $A \wedge B \wedge C \Rightarrow D$.

[1] Latin for "that which was to be proved"; in french one often uses the corresponding sentence *"ce qu'il fallait démontrer"* or c.q.f.d.



Effect of the command `apply c` on a subgoal.

Figure 2.1: The `apply` tactic

Exercise 2.4.1 *Prove* `X -> Y -> X`.

Note that one combine several move steps in a single command by, for instance:

```
move => x y.
```

2.4.4 Using implications

A dual of the `move =>` tactic is the `apply` tactic.

Here is a simple example.

```
Coq <   Lemma ex3 : (X -> Y -> Z) -> Y -> X -> Z.

Coq < move => h y x.
1 subgoal

  h : X -> Y -> Z
  y : Y
  x : X
  ============================
  Z
```

We now want to use the hypothesis `h`. The tactic `apply h` will:

- check that `h` indeed allows to prove the goal `Z`,
- generate two subgoals `X` and `Y` since this is what is need to prove `Z` out of `h`.

```
Coq <   apply h.
2 subgoals
```

```
  h : X -> Y -> Z
  y : Y
  x : X
  ============================
  X
subgoal 2 is:
 Y
```

```
Coq <   exact x.
1 subgoal

  h : X -> Y -> Z
  y : Y
  x : X
  ============================
  Y
```

```
Coq <   exact y.
No more subgoals.
```

```
Coq < Qed.
ex3 is defined
```

The figure 2.1 schematize the action of the `apply` tactic.

2.4.5 Tactics for conjunction

The canonical way to prove $A \wedge B$ is to first prove A and then prove B. This is precisely what is proposed by the `split` tactic which splits a goal `A /\ B` into two subgoals `A` and `B`:

```
1 subgoal

  ============================
  X /\ Y
```

```
Coq <   split.
2 subgoals

  ============================
  X
subgoal 2 is:
 Y
```

The tactics to use hypotheses which are conjunctions are a little more subtle. The idea, is that a proof of $A \wedge B$ can be thought of as being built from a proof a A and a proof of B, and that these two proofs can be recovered.

Let us look at a proof-state where we have a conjunction hypothesis:

```
1 subgoal

  c : X /\ Y
  ============================
  Y
```

The `case` tactic allows us to destruct `c` into two hypotheses; for instance:

```
Coq <   case c => [x y].
1 subgoal

  c : X /\ Y
  x : X
  y : Y
  ============================
  Y
```

The square brackets are here to illustrate the decomposition into two subproofs. Here `x` a proof witness for proposition `X` and `y` a proof witness for `Y`.

A variant is to use `case:` instead of `case`, in which case the original hypothesis is erased from the context and gives:

```
1 subgoal

  x : X
  y : Y
  ============================
  Y
```

When doing `move` one can also directly deconstruct a proof:

```
1 subgoal

  ============================
  X /\ Y -> X

Coq <   move => [x y].
1 subgoal

  x : X
  y : Y
  ============================
  X
```

2.4.6 Tactics for disjunction

There are two canonical ways to prove $A \vee B$: by proving A and by proving B. The corresponding tactics are `left` and `right`.

The first one:

```
1 subgoal

  ============================
  X \/ Y

Coq <   left.
1 subgoal

  ============================
  X
```

And the second one:

```
1 subgoal

  ============================
  X \/ Y
```

```
Coq <   right.
1 subgoal

  ============================
  Y
```

Again, the interesting part are the tactics to deal with disjunctions when they come as hypotheses. The situation is similar to conjunctions, except that a proof of $A \vee B$ is *either* a proof of A *or* a proof of B.

Consider the following situation:

```
1 subgoal

  d : X \/ Y
  ============================
  Y \/ X
```

We can destruct the hypothesis `d` which yields two cases: the one where we have a proof of `X` and the one where we have a proof of `Y`. These two cases mean we have two subgoals.

```
Coq <   move: d => [x | y].
2 subgoals

  x : X
  ============================
  Y \/ X
subgoal 2 is:
 Y \/ X
```

Here we see that in the first subgoal we have a new hypothesis `x:X`. We can check that we also have a hypothesis `y:Y` in the second subgoal:

```
Coq <   Show 2.
subgoal 2 is:

  y : Y
  ============================
  Y \/ X
```

Note that `Show` is a command which does not construct any new proof.

2.5 The false proposition

There is no tactic for proving the proposition `False` (precisely because it is false). When one has `False` as an assumption, one can prove any goal with the tactic `case`.

```
1 subgoal

  f : False
  ============================
  X

Coq <   case: f.
No more subgoals.
```

The use of this tactic here can be understood through the idea that if there are, for instance, two cases for a proof of $A \vee B$ (namely A and B) there is no case for a proof of `False`.

Negation

The negation $\neg A$ is written `~A` in Coq. It is an abbreviation of `A->False` and can be used as such.

For instance:

```
1 subgoal

  f : ~ X
  x : X
  ============================
  False

Coq <   apply f.
1 subgoal

  f : ~ X
  x : X
  ============================
  X
```

Chapter 3

Predicate calculus

3.1 Comes in the object

We now discover how the language of logic, and in particular how the language of Coq can yield with more meaningful propositions. For that, we want to be able to talk about (mathematical) objects. For instance, the number 2 is an object, and we want to be able to state propositions like *"2 is even"*.

In such a proposition there are two components:

- the number 2 is the object,
- and *being even* is the property, or the *predicate*.

In Coq, everything is typed, liked in most programming languages[1]. Therefore, an object must have a type. In the initial state of Coq, some types are already defined. In particular the type `nat` of natural numbers:

```
Coq <   Check 2.
2
     : nat
```

In this chapter, we will:

- start to explore how (some) objects are constructed in Coq,
- how these objects can be used in propositions,
- and in particular how the quantifiers $\forall$ and $\exists$ are handeld.

A first family of propositions dealing with objects is obtained by the equality property. When two objects `a` and `b` are of the same type, then `a=b` is a proposition. For instance:

```
Coq <   Check 2=2.
2 = 2
     : Prop
```

Note that `2=2` is a true proposition, but the moment we focus on the fact that it is a *well-formed* or *meaningful* proposition. In that respect, `2=3` is also well-formed, although it is false:

[1] But not Python!

```
Coq <   Check 2=3.
2 = 3
     : Prop
```

We will see later how to prove that such equalities are true or false.

3.2 Constants

When doing mathematics in practice, it is essential to be able to name objects. For instance, one can say: "*Let the real number π be the limit of the serie:*

$$1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \dots$$

In Coq, naming an object is done through the `Definition` command. For instance, the following command gives the name `two` to the object `2`:

```
Coq < Definition two := 2.
two is defined
```

We can then manipulate `two` as a Coq object:

```
Coq < Check two.
two
     : nat
```

and also check the content of its definition:

```
Coq < Print two.
two = 2
     : nat
```

3.3 Functions and applications

In Coq, we can construct functions. Let us define the identity function over natural numbers. In conventional mathematical notation, we would write:

$$\begin{array}{rcl} \mathrm{Id} & : & \mathbb{N} \to \mathbb{N} \\ & & x \mapsto x \end{array}$$

In Coq, we write this in the following way:

```
Coq < Definition Id := fun x : nat => x.
Id is defined
```

There are actually two novel points in this short command:

– The first is the notation `fun x : nat => x` which designates the function which maps the natural number `x` to `x`.

– The second, is the *function type*. The object we have defined is of the type `nat->nat`, which is the type of functions from the type `nat` to itself.

We can visualize the second point by checking the type of the defined object:

```
Coq < Check Id.
Id
     : nat -> nat
```

or equivalently:

```
Coq < Check fun x:nat => x.
fun x : nat => x
     : nat -> nat
```

A function is meant to be applied to an argument. In usual mathematics, we would write $Id(4)$ for designate the application of the argument 4 to the function Id. There is a slight difference in Coq, where we put the brackets differently; we write `(Id 4)`. For instance:

```
Coq < Check (Id 5).
Id 5
     : nat
```

Note that here we do not ask the system to *compute* `(Id 5)` but only to type-check it.

3.4 Assuming the existence of objects

In a mathematical, one often encounters sentences starting like "let f be some function from $\mathbb{N}$ to $\mathbb{N}\ldots$". This means that we *assume* the existence of the function f but we have not *defined* it. In Coq, one can introduce such new variables with the `Parameter` command, and when a new variable is introduced, one has to give its type:

```
Coq <   Parameter f : nat -> nat.
f is declared
Coq <   Parameter x y : nat.
x is declared
y is declared
Coq <   Check (f y).
f y
     : nat
```

3.5 Functions with multiple arguments

We see that the function type uses the same arrow notation `->` as the implication. It also uses the same bracketing convention: `A->B->C` is a shorthand for `A->(B->C)` that is the type of a function which first takes an argument of type `A` and then an argument of type `B` before returning a result of type `C`.

```
Coq <   Parameter g : nat -> nat -> nat.
```

```
g is declared

Coq <   Check (g x y).
g x y
     : nat
```

Exercise 3.5.1 *Start by assuming the existence of three types:*

```
Parameter A B C : Type.
```

You now know enough to define a function of type:

```
(A -> B -> C) -> (B -> A -> C).
```

3.6 Predefined arithmetic functions

In Coq, the basic functions of addition and multiplication are predefined. For instance:

```
Coq <   Check 2+2.
2 + 2
     : nat

Coq <   Check 3*5.
3 * 5
     : nat
```

We see that they use the usual infix notation. Fundamentally however, these notation correspond to two predefined functions named `Nat.add` and `Nat.mul`. So `2+2` is just a pretty-printing for `(Nat.mul 2 2)`.

```
Coq <   Check (Nat.mul 2 2).
2 * 2
     : nat
```

The name of these underlying functions is not really important here. We mention this point to stress how ubiquitous functions and functions with multiple arguments are.

3.7 Some first computations

In this chapter we mainly focus on the construction of objects, and their manipulation inside proofs. But Coq also allows computations like is conventional programming languages. A command to perform such computations in `Eval compute`. For instance, using the identity function defined previously:

```
Coq <   Eval compute in (Id 3).
     = 3
     : nat
```

Note that these computations also work when the object is not totally defined and contains variables:

```
Coq <   Eval compute in (Id x).
     = x
     : nat
```

They also work for the arithmetic operations mentioned above:

```
Coq <   Eval compute in (Id (2+2)).
     = 4
     : nat
```

3.8 Propositions are objects

In Coq a (well-formed) proposition is a well-formed object of the special type Prop. Therefore, a property over a type `A` will be an object of type `A -> Prop`. For instance, `even` will be of type `nat -> Prop`.

The same goes for relations which become functions mapping several arguments to `Prop`. The following command assumes the existence of a binary relation over natural numbers:

```
Coq <   Parameter R : nat -> nat -> Prop.
R is declared
```

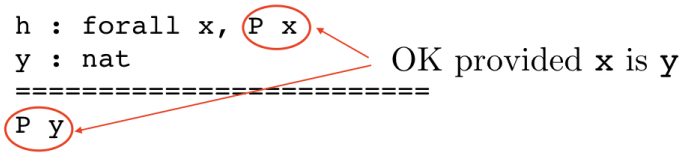
3.9 Quantifiers and bindings

A crucial way to build interesting propositions are the quantifiers $\forall$ and $\exists$. Here are some simple examples to illustrate their syntax in Coq:

```
Coq <   Check forall x : nat, P x.
forall x : nat, P x
     : Prop

Coq <   Check forall x, x = 0.
forall x : nat, x = 0
     : Prop

Coq <   Check forall x : nat, exists y : nat, x = y.
forall x : nat, exists y : nat, x = y
     : Prop

Coq <   Check (forall x : nat, exists y : nat, (R x y)) ->
                            exists f :  nat->nat, forall x, (R x (f x)).
(forall x : nat, exists y : nat, R x y) ->
exists f : nat -> nat, forall x : nat, R x (f x)
     : Prop
```

Note that, in general, when you quantify over a variable, you have to give its type. But the type can be omitted when the system can infer it from the context (like in the second example above).

When one writes `forall x :  nat, x=x`, the variable `x` is *bound* in the right hand part (here in `x=x`). This means that the proposition above is actually totally identical to the following `forall y :  nat, y=y`.



Effect of the command `apply` with forall assumptions.

Figure 3.1: The `apply` tactic for $\forall$

In order to restrict the binding of the quantifier, one has to use brackets. For instance the following proposition will be true:

```
(forall X, P x) -> forall y, P y
```

while this one will not be provable:

```
forall X, P x -> forall y, P y.
```

3.10 Tactics for the $\forall$ quantifier

The generic way to prove $\forall x, A$ is to prove A without any assumption over x. The corresponding tactic in Coq is the same as the one for implication:

```
1 subgoal

  ============================
  forall z : nat, P z

Coq <   move => z.
1 subgoal

  z : nat
  ============================
  P z
```

Note that we could have changed the name of the variable here, for instance by `move => y`.

Let us now look at how to use universally quantified assumptions. Consider the following situation:

```
1 subgoal
```

```
  h : forall x : nat, P x
  y : nat
  ============================
  P y
```

We want to use the assumption `h` and specify the quantified variable `x` should be *instantiated* by `y`. We can do by `apply` :

```
Coq <   apply h.
No more subgoals.
```

The figure 3.1 schematizes the principle of the `apply` tactic in this case.

3.11 Tactics for the ∃ quantifier

The generic way to prove an existential statement $\exists x, A(x)$ is to provide an object t, such that $A(t)$ is provable; this object is sometimes called the witness. The corresponding tactic is `exists` which, of course, needs to be given the witness.

```
1 subgoal

  ============================
  exists z : nat, z = 5

Coq <   exists 5.
1 subgoal

  ============================
  5 = 5
```

We see that the goal then becomes the right-hand part of the original existential proposition, where the *bound variable* (here `x`) is replaced by the witness.

A little more subtle is the way to use an existential assumption. Consider the following situation:

```
1 subgoal

  ============================
  (exists y : nat, P y) ->
  (forall x : nat, P x -> Q (x + 2)) -> exists z : nat, Q z

1 subgoal

  he : exists y : nat, P y
  ha : forall x : nat, P x -> Q (x + 2)
  ============================
  exists z : nat, Q z
```

We want to name the witness corresponding to the hypothesis `he` in order to be able to construct the witness for the goal.

The idea is that a proof of an existential (here `he`) can be destructed into the witness `y` and the proof of the property it verifies (which we may want to call `hy`). This is done by the following tactic:

```
Coq < move: he => [y hy].
1 subgoal

  ha : forall x : nat, P x -> Q (x + 2)
  y : nat
  hy : P y
  ============================
  exists z : nat, Q z
```

Equivalently one could use `destruct: he => [y hy]`, or `destruct he => [y hy]`. The latter version does not erase the hypothesis `he` from the context.

3.12 The equality predicate

Equality is an ubiquitous relation in mathematics. We have seen that in Coq it is written in the usual way, provided the two terms of the equality are of the same type.

The canonical way tactic for proving equalities is the `reflexivity` tactics; as the name stresses, it works provided the two terms are identical.

On the other hand, when one has an assumption `e:a=b`, one can use it to replace all occurrences of `a` by `b` in the goal by `rewrite e`.

Variants are:

- `rewrite -e` which will rewrite the other way around, replacing `b` by `a`.
- `rewrite {4 5}e` which will replace the fourth and fifth occurrences of `a` by `b`.
- combinations of the two constructs above.

Chapter 4

Basic Functional Programming

This chapter goes into more detail in the description of the objects of Coq.

We have seen that:

- These objects are (always) typed,
- these objects include functions, whose types are function types `A->B`,
- these functions give rise to computations, for instance $2+2$ computes to (or "reduces to") 4.

Actually, the language of Coq's objects can be understood as a quite powerful programming language. This language belongs to the family called functional programming, which is a programming principle which is interesting in its own right.

The two main components of this programming style are:

- The fact that functions are themselves objects of the language; for instance a function can take another function as an argument. On sometimes also syas that functions are *first-class* objects of the language.
- The presence of concrete datatypes, defined by a certain number of constructors. In Coq, these are called *inductive datatypes*.

4.1 Functions are objects

In mathematics, one often uses the notation $x \mapsto x+2$ to define a function. This notation can be translated to Coq, provided one gives the *type* of the argument x.

Indeed, there are two big families of functional programming languages: typed and untyped languages and Coq belongs to the first category: functions are typed, and this means that their arguments and their results both bear types.

For instance, the identity function $x \mapsto x$ over the type `nat` of natural numbers is written `fun x : nat => x`. We can check its type and see that it is the function type `nat -> nat`:

```
Coq <   Check fun x : nat => x.
fun x : nat => x
     : nat -> nat
```

We can define constants which are functions:

```
Coq <   Definition idnat := fun x : nat => x.
idnat is defined
```

Note there is an equivalent syntax for arguments when defining a function:

```
Coq <   Definition idnat (x : nat) := x.
```

Also, the type of the argument must be unambiguous but can be given in various ways. This is also equivalent to the definition above:

```
Coq <   Definition idnat : nat -> nat := fun x => x.
```

Once one has a function, one can apply it to an argument. The notation in Coq is slightly different from informal mathematics practice: instead of writing $f(x)$ one puts the brackets around the function too $(f\ x)$ or in the present case, `(idnat 3)`. Furthermore, one can ask the system to perform some *computations*:

```
Coq <   Eval compute in (idnat 3).
     = 3
     : nat
```

The following example really shows we are in a functional language: we can define a function which takes another function as argument. Take:

```
Coq <   Definition app4 (f : nat -> nat) := (f 4).
app4 is defined
```

The type of this function (or functional) `app4` is `(nat -> nat) -> nat`. One can verify that `(app4 idnat)` indeed reduces to `4`.

Remark 4.1.1 *Although Python is generally not considered to belong to the functional programming paradigm, it allows functional programming, since one can constructs functional objects. The notation corresponding to Coq's* `fun x => ...` *is called the lambda notation. The identity function in python is written:* `lambda x :x`*. Note that:*

- *The notation* `lambda` *actually comes from logic. In the 1940s, Alonzo Church introduced the λ-calculus, which used the greek letter λ for functional abstraction. In Church's calculus, the identity is written $\lambda x.x$.*
- *One major difference between Coq and python, is that in the latter, there is no typing; that is in the Python code above* `x` *can be any object of any type.*

4.2 Multiple arguments

A reason for the notation of application is that it gives a convenient way to deal with functions taking multiple arguments:

- A type `A -> (B -> C)` can be written `A -> B -> C`.

- A function of such a type awaits first an argument of type `A`, then an argument of type `B` before returning a result of type `C`.
- Given a function `f : A -> B -> C` and objects `a : A` and `b : B`, we may want to build:
  - Either the application of both arguments to the function, `((f a) b)` which gives a result of type `C`,
  - or a partial application of just the argument `a` which gives a function `(f a) : B -> C`.
- Therefore, it is possible to write the successive applications of several arguments to a function with a single pair of brackets. In this case `(f a b)`.

4.3 Inductive data types

The basic way to construct data types in Coq is inductive definitions. This is a feature that allows to define a type as "the smallest type closed for by some constructors". On of the simplest examples is the one of booleans, that is the smallest type containing the constructors `true` and `false`. The definition in goes goes as:

```
Coq <   Inductive bool : Type :=
        | true : bool
        | false : bool.
```

This definition adds the following to the environment:

- The data type `bool`,
- the two constant constructors `true` and `false` which are both of type `bool`.

We will see in the next chapter, that it also adds principles to reason about objects of type `bool`. In this chapter we focus on the programming side.

Note that this definition of `bool` *is already present in the environment when Coq is launched. To experiment, one can use another isomorphic type:*

```
Coq <   Inductive color : Type :=
        | red : color
        | blue : color.
Coq <   Check red.
red
     : color
```

4.4 Basic pattern matching

In a functional setting, programs are functions; and functions can check the value of inductive values through pattern matching. The following is the equivalent for `color` of the boolean negation:

```
Coq < Definition inv c :=
        match c with
        | red => blue
        | blue => red
        end.
inv is defined
```

One can test the behavior of the function:

```
Coq <   Eval compute in (inv red).
     = blue
     : color
Coq <   Eval compute in (inv (inv red)).
     = red
     : color
```

4.5 Functionals

Since functions are objects, it is easy to define functions taking other functions as arguments. For instance, here is a function which takes another function as an argument, and applies it twice to another argument:

```
Coq <   Definition twice (f : color -> color) (c : color) :=
        (f (f c)).
twice is defined
Coq < Eval compute in (twice inv red).
     = red
     : color
```

4.6 Constructors with argument(s)

An example of a more complex inductive type is the *option type*. We want to define functions over natural numbers with a special exceptional case (for instance to treat division by zero).

```
Coq <   Inductive option_nat :=
        | None
        | Some : nat -> option_nat.
option_nat is defined
option_nat_rect is defined
option_nat_ind is defined
option_nat_rec is defined
```

This means that a canonical element of `option_nat` is either:

- `None` which is similar to the previous examples,
- `(Some 0)`, `(Some 1)`, `(Some 2)`... that is an element of `nat` "wrapped" by the constructor `Some`.

The constructor `Some` thus injects `nat` into `option_nat`. We program the opposite translation by:

```
Coq <   Definition convert (x : option_nat) :=
        match x with
        | None => 0
        | Some n => n
        end.
convert is defined
```

Note that:

- We need to chose some value for `(convert None)` (here 0). This is because all functions are total.
- In the pattern `| Some n => ...` the variable `n` is bound in the right-hand part of the definition. Indeed, the second part of the definition should be read "if `x` is of the form `Some n`, then the result is `n`.

Here is another example of using pattern-matching. We want to define an addition function over `option_nat` such that the addition is performed only if both arguments are of the `Some` form. If at least one of the two is `None`, then the result should also be `None`:

```
Coq <   Definition opt_add n m := match n, m with
        | Some a, Some b => Some (a+b)
        | _, _ => None
        end.
opt_add is defined
```

We here use the _ pattern, which catches all the patterns which have not been captured above. This means that this definition is a shortcut for:

```
Coq <   Definition opt_add n m := match n, m with
        | Some a, Some b => Some (a+b)
        | Some a,  None => None
        | None, Some b => None
        | None, None => None
        end.
```

4.7 Recursive types

The final, very powerful feature, we present in this chapter is the possibility for inductive type definitions to be recursive. That is the argument of a constructor can be of the type we are defining.

4.7.1 A recursive type definition

The fundamental example is the definition of the type of natural numbers. It goes as follows:

```
Coq <   Inductive nat : Type :=
        | O : nat
        | S : nat -> nat.
```

It can be understood as:

- The first constructor states that `O` is a natural number.
- The second constructor states that, if we have constructed a natural number `n`, then we can build a new natural number `(S n)`.
- There is no other way to construct natural numbers, that is `nat` is the smallest type verifying the two conditions above.

Putting this together, we see that natural number values can be `O`, `(S O)`, `(S (S O))`... and so on.

The mechanisms described above can be used to deal with a recursive inductive type. For instance, this is the definition of the predecessor function, which takes off one `S` constructor of every natural number (and returns O as a default valute when given O):

```
Coq <   Definition pred n :=
        match n with
        | O => O
        | S p => p
        end.
pred is defined
```

4.7.2 Recursive function definitions

The fact that the type's definition is recursive goes hand in hand with the need for recursive functions over this type.

Let us consider the following function which multiplies natural numbers by two:

```
Coq <   Fixpoint double n :=
        match n with
        | O => O
        | S a => S (S (double a))
        end.
double is defined
double is recursively defined (decreasing on 1st argument)
```

The basic arithmetic operations, including addition, multiplication, are defined in a similar way. One difference being that they take more than one argument. The standard definition of addition is:

```
Coq <   Fixpoint add n m :=
        match n with
        | O => m
        | S p => S (add p m)
        end.
add is defined
add is recursively defined (decreasing on 1st argument)
```

which defines a function

```
add : nat -> nat -> nat
```

Exercise 4.7.1 *Define similarly multiplication, or subtraction. For the latter, one can chose that if* `m` *is larger than* `n`*, then* `(minus n m)` *will return 0.*

4.8 Predefined natural numbers and pretty-printing in Coq

If you test the examples involving natural numbers you will notice that:

– Most definitions are already present in the environment. The definition of `nat` is, addition is called `Nat.add`, subtraction `Nat.sub`...

– When one uses these definitions, one can see that the system prints out `O` as `0`, `(S O)` as `1`, `(S (S O)` as `2` etc/dots Also, one can call the addition function with the regular infix notation `+` and the same for the other arithmetic operations.

This pretty-printing makes the definitions easier to use, but does note change these definitions, nor the mechanisms involved.

4.9 Restrictions for termination

For the logic to remain consistent, it is essential that all functions terminate. Without going into details at this stage, we can sum up the argument by saying:

– A natural number has to be either `O` or a successor `(S n)`,

– a function application of type `nat` which does not return a result cannot be considered to be of one of these forms, and should thus not be allowed.

For instance, the following definition, which is obviously not terminating, is rejected by the system:

```
Coq <   Fixpoint foo (n : nat) : nat := (foo n).
Toplevel input, characters 2-42:
>   Fixpoint foo (n : nat) : nat := (foo n).
>   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Error:
Recursive definition of foo is ill-formed.
In environment
foo : nat -> nat
n : nat
Recursive call to foo has principal argument equal to
"n" instead of a subterm of "n".
Recursive definition is: "fun n : nat => foo n".
```

Coq thus checks that every recursive function terminates by verifying that the recursive calls get *structurally smaller*. In the case of natural numbers, this means that, when defining the value `(f (S n))` of some function `f`, one can make recursive calls to compute `(f n)`, since `n` is structurally smaller than `(S n)`.

This structural relation is actually transitive. So one can do recursive calls over some subterm of the predecessor. For instance, the following function is accepted:

```
Coq <   Fixpoint d2 (n : nat) := match n with
        | S (S p) => S (d2 p)
        | _ => 0
        end.
d2 is defined
d2 is recursively defined (decreasing on 1st argument)
```

Exercise 4.9.1 *Can you see what this function computes ?*

4.10 Reasoning about inductive objects

The fact the objects are functional programs involving inductive types and constructors yields special ways to reason about these objects in the proofs. This point will expand and be the main focus of the course. But at this stage we can already single out three main points as to how this is done.

4.10.1 Using computations in proofs

A functional program is an object which may contain some computations which are still not performed. For instance, `(inv red)` contains an unperformed computation. By performing the computation, this object transforms itself into `blue`. We also say that `(inv red)` *reduces to* `blue`.

From a logical standpoint, Coq identifies objects modulo computation. This means that there is no difference between `(inv red)` and `blue`.

The tactic `simpl` *performs the upheld computations in the goal.*

This gives us a way to *prove* some propositions, typically equalities:

```
Coq <   Lemma invol_red : inv (inv red) = red.
1 subgoal

  ============================
  inv (inv red) = red

Coq <   Proof.

Coq <   simpl.
1 subgoal

  ============================
  red = red

Coq <   reflexivity.
No more subgoals.

Coq <   Qed.
invol_red is defined
```

Note that the `simpl` tactic here is optional. Just invoking `reflexivity` will force the computations in order to check that the goal actually reduces to something of the form $x = x$.

```
Coq <   Lemma invol_blue : inv (inv blue) = blue.
1 subgoal

  ============================
  inv (inv blue) = blue

Coq <   Proof.

Coq <   reflexivity.
No more subgoals.

Coq <   Qed.
invol_blue is defined
```

Another example which involves longer computations can be:

```
Coq <   Lemma add200 : 200 + 200 = 400.
1 subgoal

  ============================
  200 + 200 = 400

Coq <   Proof.

Coq <   reflexivity.
No more subgoals.

Coq <   Qed.
add200 is defined
```

4.10.2 Reasoning by case

4.10.3 General principle

Suppose we now want to prove that the `inv` function is involutive:

```
Coq <   Lemma inv_involutive : forall c, inv (inv c) = c.
1 subgoal

  ============================
  forall c : color, inv (inv c) = c

Coq <   Proof.

Coq <   move => c.
1 subgoal

  c : color
  ============================
  inv (inv c) = c

Coq <   simpl.
1 subgoal

  c : color
  ============================
  inv (inv c) = c
```

We see that there can be no reduction performed, since `c` is a variable. We thus must reason by case over `c`, using the fact that a `color` is either `red` or `blue`.

The tactic `case` over an inductive object creates as many subgoals as there are constructors. Here, `case: c` will transform the goal into two goals `inv (inv red)` and `inv (inv blue)`. These two goals can then be solved through computations by `reflexivity`.

```
Coq <   case: c.
2 subgoals

  ============================
  inv (inv red) = red
```

```
subgoal 2 is:
 inv (inv blue) = blue
```

```
Coq <     reflexivity.
1 subgoal

  ============================
  inv (inv blue) = blue
```

```
Coq <   reflexivity.
No more subgoals.
```

4.10.4 Tactics with patterns for cases

4.10.5 Reasoning by induction