The Coq Proof Assistant
A Tutorial

April 27, 2004

Version 8.0 [1]

Gérard Huet, Gilles Kahn and Christine Paulin-Mohring

LogiCal Project

[1] This research was partly supported by IST working group "Types"

V8.0, April 27, 2004

Getting started

COQ is a Proof Assistant for a Logical Framework known as the Calculus of Inductive Constructions. It allows the interactive construction of formal proofs, and also the manipulation of functional programs consistently with their specifications. It runs as a computer program on many architectures. It is available with a variety of user interfaces. The present document does not attempt to present a comprehensive view of all the possibilities of COQ, but rather to present in the most elementary manner a tutorial on the basic specification language, called Gallina, in which formal axiomatisations may be developed, and on the main proof tools. For more advanced information, the reader could refer to the COQ Reference Manual or the *Coq'Art*, a new book by Y. Bertot and P. Castéran on practical uses of the COQ system.

We assume here that the potential user has installed COQ on his workstation, that he calls COQ from a standard teletype-like shell window, and that he does not use any special interface. Instructions on installation procedures, as well as more comprehensive documentation, may be found in the standard distribution of COQ, which may be obtained from COQ web site `http://coq.inria.fr`.

In the following, all examples preceded by the prompting sequence `Coq <` represent user input, terminated by a period. The following lines usually show COQ's answer as it appears on the users screen. The sequence of such examples is a valid COQ session, unless otherwise specified. This version of the tutorial has been prepared on a PC workstation running Linux. The standard invocation of COQ delivers a message such as:

```
unix:~> coqtop
Welcome to Coq 8.0 (Mar 2004)

Coq <
```

The first line gives a banner stating the precise version of COQ used. You should always return this banner when you report an anomaly to our hot-line `coq-bugs@pauillac.inria.fr` or on our bug-tracking system :`http`//coq.inria.fr/bin/coq-bugs:

Chapter 1

Basic Predicate Calculus

1.1 An overview of the specification language Gallina

A formal development in Gallina consists in a sequence of *declarations* and *definitions*. You may also send COQ *commands* which are not really part of the formal development, but correspond to information requests, or service routine invocations. For instance, the command:

```
Coq < Quit.
```

terminates the current session.

1.1.1 Declarations

A declaration associates a *name* with a *specification*. A name corresponds roughly to an identifier in a programming language, i.e. to a string of letters, digits, and a few ASCII symbols like underscore (_) and prime ($'$), starting with a letter. We use case distinction, so that the names `A` and `a` are distinct. Certain strings are reserved as key-words of COQ, and thus are forbidden as user identifiers.

A specification is a formal expression which classifies the notion which is being declared. There are basically three kinds of specifications: *logical propositions*, *mathematical collections*, and *abstract types*. They are classified by the three basic sorts of the system, called respectively `Prop`, `Set`, and `Type`, which are themselves atomic abstract types.

Every valid expression e in Gallina is associated with a specification, itself a valid expression, called its *type* $\tau(E)$. We write $e : \tau(E)$ for the judgment that e is of type E. You may request COQ to return to you the type of a valid expression by using the command `Check`:

```
Coq < Check O.
0
     : nat
```

Thus we know that the identifier `O` (the name 'O', not to be confused with the numeral '0' which is not a proper identifier!) is known in the current context, and that its type is the specification `nat`. This specification is itself classified as a mathematical collection, as we may readily check:

```
Coq < Check nat.
nat
     : Set
```

The specification `Set` is an abstract type, one of the basic sorts of the Gallina language, whereas the notions *nat* and *O* are notions which are defined in the arithmetic prelude, automatically loaded when running the COQ system.

We start by introducing a so-called section name. The role of sections is to structure the modelisation by limiting the scope of parameters, hypotheses and definitions. It will also give a convenient way to reset part of the development.

```
Coq < Section Declaration.
```

With what we already know, we may now enter in the system a declaration, corresponding to the informal mathematics *let n be a natural number*.

```
Coq < Variable n : nat.
n is assumed
```

If we want to translate a more precise statement, such as *let n be a positive natural number*, we have to add another declaration, which will declare explicitly the hypothesis `Pos_n`, with specification the proper logical proposition:

```
Coq < Hypothesis Pos_n : (gt n 0).
Pos_n is assumed
```

Indeed we may check that the relation `gt` is known with the right type in the current context:

```
Coq < Check gt.
gt
     : nat -> nat -> Prop
```

which tells us that `gt` is a function expecting two arguments of type `nat` in order to build a logical proposition. What happens here is similar to what we are used to in a functional programming language: we may compose the (specification) type `nat` with the (abstract) type `Prop` of logical propositions through the arrow function constructor, in order to get a functional type `nat->Prop`:

```
Coq < Check (nat -> Prop).
nat -> Prop
     : Type
```

which may be composed again with `nat` in order to obtain the type `nat->nat->Prop` of binary relations over natural numbers. Actually `nat->nat->Prop` is an abbreviation for `nat->(nat->Prop)`.

Functional notions may be composed in the usual way. An expression f of type $A \rightarrow B$ may be applied to an expression e of type A in order to form the expression $(f\ e)$ of type B. Here we get that the expression `(gt n)` is well-formed of type `nat->Prop`, and thus that the expression `(gt n O)`, which abbreviates `((gt n) O)`, is a well-formed proposition.

```
Coq < Check gt n O.
n > 0
     : Prop
```

1.1.2 Definitions

The initial prelude contains a few arithmetic definitions: `nat` is defined as a mathematical collection (type `Set`), constants `O`, `S`, `plus`, are defined as objects of types respectively `nat`, `nat->nat`, and `nat->nat->nat`. You may introduce new definitions, which link a name to a well-typed value. For instance, we may introduce the constant `one` as being defined to be equal to the successor of zero:

```
Coq < Definition one := (S O).
one is defined
```

We may optionally indicate the required type:

```
Coq < Definition two : nat := S one.
two is defined
```

Actually COQ allows several possible syntaxes:

```
Coq < Definition three : nat := S two.
three is defined
```

Here is a way to define the doubling function, which expects an argument `m` of type `nat` in order to build its result as `(plus m m)`:

```
Coq < Definition double (m:nat) := plus m m.
double is defined
```

This definition introduces the constant `double` defined as the expression `fun m:nat => plus m m`. The abstraction introduced by `fun` is explained as follows. The expression `fun x:A => e` is well formed of type `A->B` in a context whenever the expression `e` is well-formed of type `B` in the given context to which we add the declaration that `x` is of type `A`. Here `x` is a bound, or dummy variable in the expression `fun x:A => e`. For instance we could as well have defined `double` as `fun n:nat => (plus n n)`.

Bound (local) variables and free (global) variables may be mixed. For instance, we may define the function which adds the constant `n` to its argument as

```
Coq < Definition add_n (m:nat) := plus m n.
add_n is defined
```

However, note that here we may not rename the formal argument m into n without capturing the free occurrence of n, and thus changing the meaning of the defined notion.

Binding operations are well known for instance in logic, where they are called quantifiers. Thus we may universally quantify a proposition such as $m > 0$ in order to get a universal proposition $\forall m \cdot m > 0$. Indeed this operator is available in COQ, with the following syntax: `forall m:nat, gt m O`. Similarly to the case of the functional abstraction binding, we are obliged to declare explicitly the type of the quantified variable. We check:

```
Coq < Check (forall m:nat, gt m 0).
forall m : nat, m > 0
     : Prop
```

We may clean-up the development by removing the contents of the current section:

```
Coq < Reset Declaration.
```

1.2 Introduction to the proof engine: Minimal Logic

In the following, we are going to consider various propositions, built from atomic propositions A, B, C. This may be done easily, by introducing these atoms as global variables declared of type `Prop`. It is easy to declare several names with the same specification:

```
Coq < Section Minimal_Logic.

Coq < Variables A B C : Prop.
A is assumed
B is assumed
C is assumed
```

We shall consider simple implications, such as $A \rightarrow B$, read as "A implies B". Remark that we overload the arrow symbol, which has been used above as the functionality type constructor, and which may be used as well as propositional connective:

```
Coq < Check (A -> B).
A -> B
     : Prop
```

Let us now embark on a simple proof. We want to prove the easy tautology $((A \rightarrow (B \rightarrow C)) \rightarrow (A \rightarrow B) \rightarrow (A \rightarrow C)$. We enter the proof engine by the command `Goal`, followed by the conjecture we want to verify:

```
Coq < Goal (A -> B -> C) -> (A -> B) -> A -> C.
1 subgoal

  A : Prop
  B : Prop
  C : Prop
  ============================
   (A -> B -> C) -> (A -> B) -> A -> C
```

The system displays the current goal below a double line, local hypotheses (there are none initially) being displayed above the line. We call the combination of local hypotheses with a goal a *judgment*. We are now in an inner loop of the system, in proof mode. New commands are available in this mode, such as *tactics*, which are proof combining primitives. A tactic operates on the current goal by attempting to construct a proof of the corresponding judgment, possibly from proofs of some hypothetical judgments, which are then added to the current list of conjectured judgments. For instance, the `intro` tactic is applicable to any judgment whose goal is an implication, by moving the proposition to the left of the application to the list of local hypotheses:

```
Coq < intro H.
1 subgoal

  A : Prop
  B : Prop
  C : Prop
  H : A -> B -> C
  ============================
   (A -> B) -> A -> C
```

Several introductions may be done in one step:

```
Coq < intros H' HA.
1 subgoal

  A : Prop
  B : Prop
  C : Prop
  H : A -> B -> C
  H' : A -> B
  HA : A
  ============================
   C
```

We notice that C, the current goal, may be obtained from hypothesis `H`, provided the truth of A and B are established. The tactic `apply` implements this piece of reasoning:

```
Coq < apply H.
2 subgoals

  A : Prop
  B : Prop
  C : Prop
  H : A -> B -> C
  H' : A -> B
  HA : A
  ============================
   A
subgoal 2 is:
 B
```

We are now in the situation where we have two judgments as conjectures that remain to be proved. Only the first is listed in full, for the others the system displays only the corresponding subgoal, without its local hypotheses list. Remark that `apply` has kept the local hypotheses of its father judgment, which are still available for the judgments it generated.

In order to solve the current goal, we just have to notice that it is exactly available as hypothesis HA:

```
Coq < exact HA.
1 subgoal

  A : Prop
  B : Prop
  C : Prop
  H : A -> B -> C
  H' : A -> B
  HA : A
  ============================
   B
```

Now H' applies:

```
Coq < apply H'.
1 subgoal

  A : Prop
```

```
  B : Prop
  C : Prop
  H : A -> B -> C
  H' : A -> B
  HA : A
  ============================
   A
```

And we may now conclude the proof as before, with `exact HA`. Actually, we may not bother with the name `HA`, and just state that the current goal is solvable from the current local assumptions:

```
Coq < assumption.
Proof completed.
```

The proof is now finished. We may either discard it, by using the command `Abort` which returns to the standard COQ toplevel loop without further ado, or else save it as a lemma in the current context, under name say `trivial_lemma`:

```
Coq < Save trivial_lemma.
intro H.
intros H' HA.
apply H.
 exact HA.
 apply H'.
   assumption.
trivial_lemma is defined
```

As a comment, the system shows the proof script listing all tactic commands used in the proof.

Let us redo the same proof with a few variations. First of all we may name the initial goal as a conjectured lemma:

```
Coq < Lemma distr_impl : (A -> B -> C) -> (A -> B) -> A -> C.
1 subgoal

  A : Prop
  B : Prop
  C : Prop
  ============================
   (A -> B -> C) -> (A -> B) -> A -> C
```

Next, we may omit the names of local assumptions created by the introduction tactics, they can be automatically created by the proof engine as new non-clashing names.

```
Coq < intros.
1 subgoal

  A : Prop
  B : Prop
  C : Prop
  H : A -> B -> C
  H0 : A -> B
  H1 : A
  ============================
   C
```

The `intros` tactic, with no arguments, effects as many individual applications of `intro` as is legal.

Then, we may compose several tactics together in sequence, or in parallel, through *tacticals*, that is tactic combinators. The main constructions are the following:

- $T_1; T_2$ (read T_1 then T_2) applies tactic T_1 to the current goal, and then tactic T_2 to all the subgoals generated by T_1.
- $T; [T_1|T_2|...|T_n]$ applies tactic T to the current goal, and then tactic T_1 to the first newly generated subgoal, ..., T_n to the nth.

We may thus complete the proof of `distr_impl` with one composite tactic:

```
Coq < apply H; [ assumption | apply H0; assumption ].
Proof completed.
```

Let us now save lemma `distr_impl`:

```
Coq < Save.
intros.
apply H; [ assumption | apply H0; assumption ].
distr_impl is defined
```

Here `Save` needs no argument, since we gave the name `distr_impl` in advance; it is however possible to override the given name by giving a different argument to command `Save`.

Actually, such an easy combination of tactics `intro`, `apply` and `assumption` may be found completely automatically by an automatic tactic, called `auto`, without user guidance:

```
Coq < Lemma distr_imp : (A -> B -> C) -> (A -> B) -> A -> C.
1 subgoal

  A : Prop
  B : Prop
  C : Prop
  ============================
   (A -> B -> C) -> (A -> B) -> A -> C

Coq < auto.
Proof completed.
```

This time, we do not save the proof, we just discard it with the `Abort` command:

```
Coq < Abort.
Current goal aborted
```

At any point during a proof, we may use `Abort` to exit the proof mode and go back to Coq's main loop. We may also use `Restart` to restart from scratch the proof of the same lemma. We may also use `Undo` to backtrack one step, and more generally `Undo n` to backtrack n steps.

We end this section by showing a useful command, `Inspect n.`, which inspects the global COQ environment, showing the last `n` declared notions:

```
Coq < Inspect 3.
*** [C : Prop]
trivial_lemma : (A -> B -> C) -> (A -> B) -> A -> C
distr_impl : (A -> B -> C) -> (A -> B) -> A -> C
```

The declarations, whether global parameters or axioms, are shown preceded by `***`; definitions and lemmas are stated with their specification, but their value (or proof-term) is omitted.

1.3 Propositional Calculus

1.3.1 Conjunction

We have seen how `intro` and `apply` tactics could be combined in order to prove implicational statements. More generally, COQ favors a style of reasoning, called *Natural Deduction*, which decomposes reasoning into so called *introduction rules*, which tell how to prove a goal whose main operator is a given propositional connective, and *elimination rules*, which tell how to use an hypothesis whose main operator is the propositional connective. Let us show how to use these ideas for the propositional connectives `/\` and `\/`.

```
Coq < Lemma and_commutative : A /\ B -> B /\ A.
1 subgoal

  A : Prop
  B : Prop
  C : Prop
  ============================
   A /\ B -> B /\ A
```

```
Coq < intro.
1 subgoal

  A : Prop
  B : Prop
  C : Prop
  H : A /\ B
  ============================
   B /\ A
```

We make use of the conjunctive hypothesis `H` with the `elim` tactic, which breaks it into its components:

```
Coq < elim H.
1 subgoal

  A : Prop
  B : Prop
  C : Prop
  H : A /\ B
  ============================
   A -> B -> B /\ A
```

We now use the conjunction introduction tactic `split`, which splits the conjunctive goal into the two subgoals:

```
Coq < split.
2 subgoals

  A : Prop
  B : Prop
  C : Prop
  H : A /\ B
  H0 : A
  H1 : B
```

```
  ============================
   B
subgoal 2 is:
 A
```

and the proof is now trivial. Indeed, the whole proof is obtainable as follows:

```
Coq < Restart.
1 subgoal

  A : Prop
  B : Prop
  C : Prop
  ============================
   A /\ B -> B /\ A

Coq < intro H; elim H; auto.
Proof completed.

Coq < Qed.
intro H; elim H; auto.
and_commutative is defined
```

The tactic `auto` succeeded here because it knows as a hint the conjunction introduction operator `conj`

```
Coq < Check conj.
conj
     : forall A B : Prop, A -> B -> A /\ B
```

Actually, the tactic `Split` is just an abbreviation for `apply conj`.

What we have just seen is that the `auto` tactic is more powerful than just a simple application of local hypotheses; it tries to apply as well lemmas which have been specified as hints. A `Hint Resolve` command registers a lemma as a hint to be used from now on by the `auto` tactic, whose power may thus be incrementally augmented.

1.3.2 Disjunction

In a similar fashion, let us consider disjunction:

```
Coq < Lemma or_commutative : A \/ B -> B \/ A.
1 subgoal

  A : Prop
  B : Prop
  C : Prop
  ============================
   A \/ B -> B \/ A

Coq < intro H; elim H.
2 subgoals

  A : Prop
  B : Prop
  C : Prop
```

```
  H : A \/ B
  ============================
   A -> B \/ A
subgoal 2 is:
 B -> B \/ A
```

Let us prove the first subgoal in detail. We use `intro` in order to be left to prove `B\/A` from `A`:

```
Coq < intro HA.
2 subgoals

  A : Prop
  B : Prop
  C : Prop
  H : A \/ B
  HA : A
  ============================
   B \/ A
subgoal 2 is:
 B -> B \/ A
```

Here the hypothesis `H` is not needed anymore. We could choose to actually erase it with the tactic `clear`; in this simple proof it does not really matter, but in bigger proof developments it is useful to clear away unnecessary hypotheses which may clutter your screen.

```
Coq < clear H.
2 subgoals

  A : Prop
  B : Prop
  C : Prop
  HA : A
  ============================
   B \/ A
subgoal 2 is:
 B -> B \/ A
```

The disjunction connective has two introduction rules, since `P\/Q` may be obtained from `P` or from `Q`; the two corresponding proof constructors are called respectively `or_introl` and `or_intror`; they are applied to the current goal by tactics `left` and `right` respectively. For instance:

```
Coq < right.
2 subgoals

  A : Prop
  B : Prop
  C : Prop
  HA : A
  ============================
   A
subgoal 2 is:
 B -> B \/ A
```

```
Coq < trivial.
1 subgoal

  A : Prop
  B : Prop
  C : Prop
  H : A \/ B
  ============================
   B -> B \/ A
```

The tactic `trivial` works like `auto` with the hints database, but it only tries those tactics that can solve the goal in one step.

As before, all these tedious elementary steps may be performed automatically, as shown for the second symmetric case:

```
Coq < auto.
Proof completed.
```

However, `auto` alone does not succeed in proving the full lemma, because it does not try any elimination step. It is a bit disappointing that `auto` is not able to prove automatically such a simple tautology. The reason is that we want to keep `auto` efficient, so that it is always effective to use.

1.3.3 Tauto

A complete tactic for propositional tautologies is indeed available in COQ as the `tauto` tactic.

```
Coq < Restart.
1 subgoal

  A : Prop
  B : Prop
  C : Prop
  ============================
   A \/ B -> B \/ A

Coq < tauto.
Proof completed.

Coq < Qed.
tauto.
or_commutative is defined
```

It is possible to inspect the actual proof tree constructed by `tauto`, using a standard command of the system, which prints the value of any notion currently defined in the context:

```
Coq < Print or_commutative.
or_commutative =
fun H : A \/ B =>
or_ind (fun H0 : A => or_intror B H0) (fun H0 : B => or_introl A H0) H
     : A \/ B -> B \/ A
```

It is not easy to understand the notation for proof terms without a few explanations. The `fun` prefix, such as `fun H:A\/B =>`, corresponds to `intro H`, whereas a subterm such as `(or_intror B H0)` corresponds to the sequence `apply or_intror; exact H0`. The generic combinator `or_intror` needs to be instantiated by the two properties `B` and `A`. Because `A` can be deduced from the type of `H0`, only `B` is printed. The two instantiations are effected automatically by the tactic `apply` when pattern-matching a goal. The specialist will of course recognize our proof term as a λ-term, used as notation for the natural deduction proof term through the Curry-Howard isomorphism. The naive user of Coq may safely ignore these formal details.

Let us exercise the `tauto` tactic on a more complex example:

```
Coq < Lemma distr_and : A -> B /\ C -> (A -> B) /\ (A -> C).
1 subgoal

  A : Prop
  B : Prop
  C : Prop
  ============================
   A -> B /\ C -> (A -> B) /\ (A -> C)

Coq < tauto.
Proof completed.

Coq < Qed.
tauto.
distr_and is defined
```

1.3.4 Classical reasoning

`tauto` always comes back with an answer. Here is an example where it fails:

```
Coq < Lemma Peirce : ((A -> B) -> A) -> A.
1 subgoal

  A : Prop
  B : Prop
  C : Prop
  ============================
   ((A -> B) -> A) -> A

Coq < try tauto.
1 subgoal

  A : Prop
  B : Prop
  C : Prop
  ============================
   ((A -> B) -> A) -> A
```

Note the use of the `Try` tactical, which does nothing if its tactic argument fails.

This may come as a surprise to someone familiar with classical reasoning. Peirce's lemma is true in Boolean logic, i.e. it evaluates to `true` for every truth-assignment to `A` and `B`. Indeed the double negation of Peirce's law may be proved in Coq using `tauto`:

```
Coq < Abort.
Current goal aborted

Coq < Lemma NNPeirce : ~ ~ (((A -> B) -> A) -> A).
```

```
1 subgoal

  A : Prop
  B : Prop
  C : Prop
  ============================
   ~ ~ (((A -> B) -> A) -> A)

Coq < tauto.
Proof completed.

Coq < Qed.
tauto.
NNPeirce is defined
```

In classical logic, the double negation of a proposition is equivalent to this proposition, but in the constructive logic of COQ this is not so. If you want to use classical logic in COQ, you have to import explicitly the `Classical` module, which will declare the axiom `classic` of excluded middle, and classical tautologies such as de Morgan's laws. The `Require` command is used to import a module from COQ's library:

```
Coq < Require Import Classical.

Coq < Check NNPP.
NNPP
     : forall p : Prop, ~ ~ p -> p
```

and it is now easy (although admittedly not the most direct way) to prove a classical law such as Peirce's:

```
Coq < Lemma Peirce : ((A -> B) -> A) -> A.
1 subgoal

  A : Prop
  B : Prop
  C : Prop
  ============================
   ((A -> B) -> A) -> A

Coq < apply NNPP; tauto.
Proof completed.

Coq < Qed.
apply NNPP; tauto.
Peirce is defined
```

Here is one more example of propositional reasoning, in the shape of a Scottish puzzle. A private club has the following rules:

1. Every non-scottish member wears red socks
2. Every member wears a kilt or doesn't wear red socks
3. The married members don't go out on Sunday
4. A member goes out on Sunday if and only if he is Scottish
5. Every member who wears a kilt is Scottish and married

6. Every scottish member wears a kilt

Now, we show that these rules are so strict that no one can be accepted.

```
Coq < Section club.

Coq < Variables Scottish RedSocks WearKilt Married GoOutSunday : Prop.
Scottish is assumed
RedSocks is assumed
WearKilt is assumed
Married is assumed
GoOutSunday is assumed

Coq < Hypothesis rule1 : ~ Scottish -> RedSocks.
rule1 is assumed

Coq < Hypothesis rule2 : WearKilt \/ ~ RedSocks.
rule2 is assumed

Coq < Hypothesis rule3 : Married -> ~ GoOutSunday.
rule3 is assumed

Coq < Hypothesis rule4 : GoOutSunday <-> Scottish.
rule4 is assumed

Coq < Hypothesis rule5 : WearKilt -> Scottish /\ Married.
rule5 is assumed

Coq < Hypothesis rule6 : Scottish -> WearKilt.
rule6 is assumed

Coq < Lemma NoMember : False.
1 subgoal

  A : Prop
  B : Prop
  C : Prop
  Scottish : Prop
  RedSocks : Prop
  WearKilt : Prop
  Married : Prop
  GoOutSunday : Prop
  rule1 : ~ Scottish -> RedSocks
  rule2 : WearKilt \/ ~ RedSocks
  rule3 : Married -> ~ GoOutSunday
  rule4 : GoOutSunday <-> Scottish
  rule5 : WearKilt -> Scottish /\ Married
  rule6 : Scottish -> WearKilt
  ============================
   False

Coq < tauto.
Proof completed.

Coq < Qed.
tauto.
NoMember is defined
```

At that point `NoMember` is a proof of the absurdity depending on hypotheses. We may end the section, in that case, the variables and hypotheses will be discharged, and the type of `NoMember` will be generalised.

```
Coq < End club.
NoMember is discharged.

Coq < Check NoMember.
NoMember
     : forall Scottish RedSocks WearKilt Married GoOutSunday : Prop,
       (~ Scottish -> RedSocks) ->
       WearKilt \/ ~ RedSocks ->
       (Married -> ~ GoOutSunday) ->
       (GoOutSunday <-> Scottish) ->
       (WearKilt -> Scottish /\ Married) ->
       (Scottish -> WearKilt) -> False
```

1.4 Predicate Calculus

Let us now move into predicate logic, and first of all into first-order predicate calculus. The essence of predicate calculus is that to try to prove theorems in the most abstract possible way, without using the definitions of the mathematical notions, but by formal manipulations of uninterpreted function and predicate symbols.

1.4.1 Sections and signatures

Usually one works in some domain of discourse, over which range the individual variables and function symbols. In COQ we speak in a language with a rich variety of types, so me may mix several domains of discourse, in our multi-sorted language. For the moment, we just do a few exercises, over a domain of discourse `D` axiomatised as a `Set`, and we consider two predicate symbols `P` and `R` over `D`, of arities respectively 1 and 2. Such abstract entities may be entered in the context as global variables. But we must be careful about the pollution of our global environment by such declarations. For instance, we have already polluted our COQ session by declaring the variables `n`, `Pos_n`, `A`, `B`, and `C`. If we want to revert to the clean state of our initial session, we may use the COQ `Reset` command, which returns to the state just prior the given global notion as we did before to remove a section, or we may return to the initial state using :

```
Coq < Reset Initial.
```

We shall now declare a new `Section`, which will allow us to define notions local to a well-delimited scope. We start by assuming a domain of discourse `D`, and a binary relation `R` over `D`:

```
Coq < Section Predicate_calculus.

Coq < Variable D : Set.
D is assumed

Coq < Variable R : D -> D -> Prop.
R is assumed
```

As a simple example of predicate calculus reasoning, let us assume that relation `R` is symmetric and transitive, and let us show that `R` is reflexive in any point `x` which has an `R` successor. Since we do not want to make the assumptions about `R` global axioms of a theory, but rather local hypotheses to a theorem, we open a specific section to this effect.

```
Coq < Section R_sym_trans.

Coq < Hypothesis R_symmetric : forall x y:D, R x y -> R y x.
R_symmetric is assumed

Coq < Hypothesis R_transitive : forall x y z:D, R x y -> R y z -> R x z.
R_transitive is assumed
```

Remark the syntax `forall x:D`, which stands for universal quantification $\forall x : D$.

1.4.2 Existential quantification

We now state our lemma, and enter proof mode.

```
Coq < Lemma refl_if : forall x:D, (exists y, R x y) -> R x x.
1 subgoal

  D : Set
  R : D -> D -> Prop
  R_symmetric : forall x y : D, R x y -> R y x
  R_transitive : forall x y z : D, R x y -> R y z -> R x z
  ============================
   forall x : D, (exists y : D, R x y) -> R x x
```

Remark that the hypotheses which are local to the currently opened sections are listed as local hypotheses to the current goals. The rationale is that these hypotheses are going to be discharged, as we shall see, when we shall close the corresponding sections.

Note the functional syntax for existential quantification. The existential quantifier is built from the operator `ex`, which expects a predicate as argument:

```
Coq < Check ex.
ex
     : forall A : Type, (A -> Prop) -> Prop
```

and the notation `(exists x:D, P x)` is just concrete syntax for `(ex D (fun x:D => P x))`. Existential quantification is handled in COQ in a similar fashion to the connectives `/\` and `\/` : it is introduced by the proof combinator `ex_intro`, which is invoked by the specific tactic `Exists`, and its elimination provides a witness `a:D` to `P`, together with an assumption `h:(P a)` that indeed `a` verifies `P`. Let us see how this works on this simple example.

```
Coq < intros x x_Rlinked.
1 subgoal

  D : Set
  R : D -> D -> Prop
  R_symmetric : forall x y : D, R x y -> R y x
  R_transitive : forall x y z : D, R x y -> R y z -> R x z
  x : D
  x_Rlinked : exists y : D, R x y
  ============================
   R x x
```

Remark that `intros` treats universal quantification in the same way as the premises of implications. Renaming of bound variables occurs when it is needed; for instance, had we started with `intro y`, we would have obtained the goal:

```
Coq < intro y.
1 subgoal

  D : Set
  R : D -> D -> Prop
  R_symmetric : forall x y : D, R x y -> R y x
  R_transitive : forall x y z : D, R x y -> R y z -> R x z
  y : D
  ============================
   (exists y0 : D, R y y0) -> R y y
```

Let us now use the existential hypothesis `x_Rlinked` to exhibit an R-successor y of x. This is done in two steps, first with `elim`, then with `intros`

```
Coq < elim x_Rlinked.
1 subgoal

  D : Set
  R : D -> D -> Prop
  R_symmetric : forall x y : D, R x y -> R y x
  R_transitive : forall x y z : D, R x y -> R y z -> R x z
  x : D
  x_Rlinked : exists y : D, R x y
  ============================
   forall x0 : D, R x x0 -> R x x

Coq < intros y Rxy.
1 subgoal

  D : Set
  R : D -> D -> Prop
  R_symmetric : forall x y : D, R x y -> R y x
  R_transitive : forall x y z : D, R x y -> R y z -> R x z
  x : D
  x_Rlinked : exists y : D, R x y
  y : D
  Rxy : R x y
  ============================
   R x x
```

Now we want to use `R_transitive`. The `apply` tactic will know how to match `x` with `x`, and `z` with `x`, but needs help on how to instantiate `y`, which appear in the hypotheses of `R_transitive`, but not in its conclusion. We give the proper hint to `apply` in a `with` clause, as follows:

```
Coq < apply R_transitive with y.
2 subgoals

  D : Set
  R : D -> D -> Prop
  R_symmetric : forall x y : D, R x y -> R y x
  R_transitive : forall x y z : D, R x y -> R y z -> R x z
  x : D
  x_Rlinked : exists y : D, R x y
```

```
  y : D
  Rxy : R x y
  ============================
   R x y
subgoal 2 is:
 R y x
```

The rest of the proof is routine:

```
Coq < assumption.
1 subgoal

  D : Set
  R : D -> D -> Prop
  R_symmetric : forall x y : D, R x y -> R y x
  R_transitive : forall x y z : D, R x y -> R y z -> R x z
  x : D
  x_Rlinked : exists y : D, R x y
  y : D
  Rxy : R x y
  ============================
   R y x
```

```
Coq < apply R_symmetric; assumption.
Proof completed.
```

```
Coq < Qed.
```

Let us now close the current section.

```
Coq < End R_sym_trans.
refl_if is discharged.
```

Here COQ's printout is a warning that all local hypotheses have been discharged in the statement of `refl_if`, which now becomes a general theorem in the first-order language declared in section `Predicate_calculus`. In this particular example, the use of section `R_sym_trans` has not been really significant, since we could have instead stated theorem `refl_if` in its general form, and done basically the same proof, obtaining `R_symmetric` and `R_transitive` as local hypotheses by initial `intros` rather than as global hypotheses in the context. But if we had pursued the theory by proving more theorems about relation `R`, we would have obtained all general statements at the closing of the section, with minimal dependencies on the hypotheses of symmetry and transitivity.

1.4.3 Paradoxes of classical predicate calculus

Let us illustrate this feature by pursuing our `Predicate_calculus` section with an enrichment of our language: we declare a unary predicate `P` and a constant `d`:

```
Coq < Variable P :  D -> Prop.
P is assumed
```

```
Coq < Variable d : D.
d is assumed
```

We shall now prove a well-known fact from first-order logic: a universal predicate is non-empty, or in other terms existential quantification follows from universal quantification.

```
Coq < Lemma weird : (forall x:D, P x) ->  exists a, P a.
1 subgoal

  D : Set
  R : D -> D -> Prop
  P : D -> Prop
  d : D
  ============================
   (forall x : D, P x) -> exists a : D, P a
```

```
Coq <  intro UnivP.
1 subgoal

  D : Set
  R : D -> D -> Prop
  P : D -> Prop
  d : D
  UnivP : forall x : D, P x
  ============================
   exists a : D, P a
```

First of all, notice the pair of parentheses around `forall x:D, P x` in the statement of lemma `weird`. If we had omitted them, COQ's parser would have interpreted the statement as a truly trivial fact, since we would postulate an `x` verifying `(P x)`. Here the situation is indeed more problematic. If we have some element in `Set D`, we may apply `UnivP` to it and conclude, otherwise we are stuck. Indeed such an element `d` exists, but this is just by virtue of our new signature. This points out a subtle difference between standard predicate calculus and COQ. In standard first-order logic, the equivalent of lemma `weird` always holds, because such a rule is wired in the inference rules for quantifiers, the semantic justification being that the interpretation domain is assumed to be non-empty. Whereas in COQ, where types are not assumed to be systematically inhabited, lemma `weird` only holds in signatures which allow the explicit construction of an element in the domain of the predicate.

Let us conclude the proof, in order to show the use of the `Exists` tactic:

```
Coq < exists d; trivial.
Proof completed.
```

```
Coq < Qed.
intro UnivP.
exists d; trivial.
weird is defined
```

Another fact which illustrates the sometimes disconcerting rules of classical predicate calculus is Smullyan's drinkers' paradox: "In any non-empty bar, there is a person such that if she drinks, then everyone drinks". We modelize the bar by Set `D`, drinking by predicate `P`. We shall need classical reasoning. Instead of loading the `Classical` module as we did above, we just state the law of excluded middle as a local hypothesis schema at this point:

```
Coq < Hypothesis EM : forall A:Prop, A \/ ~ A.
EM is assumed
```

```
Coq < Lemma drinker :  exists x:D, P x -> forall x:D, P x.
1 subgoal
```

```
  D : Set
  R : D -> D -> Prop
  P : D -> Prop
  d : D
  EM : forall A : Prop, A \/ ~ A
  ============================
   exists x : D, P x -> forall x0 : D, P x0
```

The proof goes by cases on whether or not there is someone who does not drink. Such reasoning by cases proceeds by invoking the excluded middle principle, via `elim` of the proper instance of `EM`:

```
Coq < elim (EM (exists x, ~ P x)).
2 subgoals

  D : Set
  R : D -> D -> Prop
  P : D -> Prop
  d : D
  EM : forall A : Prop, A \/ ~ A
  ============================
   (exists x : D, ~ P x) -> exists x : D, P x -> forall x0 : D, P x0
subgoal 2 is:
 ~ (exists x : D, ~ P x) -> exists x : D, P x -> forall x0 : D, P x0
```

We first look at the first case. Let Tom be the non-drinker:

```
Coq < intro Non_drinker; elim Non_drinker; intros Tom Tom_does_not_drink.
2 subgoals

  D : Set
  R : D -> D -> Prop
  P : D -> Prop
  d : D
  EM : forall A : Prop, A \/ ~ A
  Non_drinker : exists x : D, ~ P x
  Tom : D
  Tom_does_not_drink : ~ P Tom
  ============================
   exists x : D, P x -> forall x0 : D, P x0
subgoal 2 is:
 ~ (exists x : D, ~ P x) -> exists x : D, P x -> forall x0 : D, P x0
```

We conclude in that case by considering Tom, since his drinking leads to a contradiction:

```
Coq < exists Tom; intro Tom_drinks.
2 subgoals

  D : Set
  R : D -> D -> Prop
  P : D -> Prop
  d : D
  EM : forall A : Prop, A \/ ~ A
  Non_drinker : exists x : D, ~ P x
  Tom : D
```

```
  Tom_does_not_drink : ~ P Tom
  Tom_drinks : P Tom
  ============================
   forall x : D, P x
subgoal 2 is:
 ~ (exists x : D, ~ P x) -> exists x : D, P x -> forall x0 : D, P x0
```

There are several ways in which we may eliminate a contradictory case; a simple one is to use the `absurd` tactic as follows:

```
Coq < absurd (P Tom); trivial.
1 subgoal

  D : Set
  R : D -> D -> Prop
  P : D -> Prop
  d : D
  EM : forall A : Prop, A \/ ~ A
  ============================
   ~ (exists x : D, ~ P x) -> exists x : D, P x -> forall x0 : D, P x0
```

We now proceed with the second case, in which actually any person will do; such a John Doe is given by the non-emptiness witness `d`:

```
Coq < intro No_nondrinker; exists d; intro d_drinks.
1 subgoal

  D : Set
  R : D -> D -> Prop
  P : D -> Prop
  d : D
  EM : forall A : Prop, A \/ ~ A
  No_nondrinker : ~ (exists x : D, ~ P x)
  d_drinks : P d
  ============================
   forall x : D, P x
```

Now we consider any Dick in the bar, and reason by cases according to its drinking or not:

```
Coq < intro Dick; elim (EM (P Dick)); trivial.
1 subgoal

  D : Set
  R : D -> D -> Prop
  P : D -> Prop
  d : D
  EM : forall A : Prop, A \/ ~ A
  No_nondrinker : ~ (exists x : D, ~ P x)
  d_drinks : P d
  Dick : D
  ============================
   ~ P Dick -> P Dick
```

The only non-trivial case is again treated by contradiction:

```
Coq < intro Dick_does_not_drink; absurd (exists x, ~ P x); trivial.
1 subgoal

  D : Set
  R : D -> D -> Prop
  P : D -> Prop
  d : D
  EM : forall A : Prop, A \/ ~ A
  No_nondrinker : ~ (exists x : D, ~ P x)
  d_drinks : P d
  Dick : D
  Dick_does_not_drink : ~ P Dick
  ============================
   exists x : D, ~ P x

Coq < exists Dick; trivial.
Proof completed.

Coq < Qed.
elim (EM (exists x : _, ~ P x)).
 intro Non_drinker; elim Non_drinker; intros Tom Tom_does_not_drink.
   exists Tom; intro Tom_drinks.
   absurd (P Tom); trivial.
 intro No_nondrinker; exists d; intro d_drinks.
   intro Dick; elim (EM (P Dick)); trivial.
   intro Dick_does_not_drink; absurd (exists x : _, ~ P x); trivial.
   exists Dick; trivial.
drinker is defined
```

Now, let us close the main section and look at the complete statements we proved:

```
Coq < End Predicate_calculus.
refl_if is discharged.
weird is discharged.
drinker is discharged.

Coq < Check refl_if.
refl_if
     : forall (D : Set) (R : D -> D -> Prop),
       (forall x y : D, R x y -> R y x) ->
       (forall x y z : D, R x y -> R y z -> R x z) ->
       forall x : D, (exists y : D, R x y) -> R x x

Coq < Check weird.
weird
     : forall (D : Set) (P : D -> Prop),
       D -> (forall x : D, P x) -> exists a : D, P a

Coq < Check drinker.
drinker
     : forall (D : Set) (P : D -> Prop),
       D ->
       (forall A : Prop, A \/ ~ A) ->
       exists x : D, P x -> forall x0 : D, P x0
```

Remark how the three theorems are completely generic in the most general fashion; the domain `D` is discharged in all of them, `R` is discharged in `refl_if` only, `P` is discharged only in `weird` and `drinker`, along with the hypothesis that `D` is inhabited. Finally, the excluded middle hypothesis is discharged only in `drinker`.

Note that the name `d` has vanished as well from the statements of `weird` and `drinker`, since COQ's pretty-printer replaces systematically a quantification such as `forall d:D, E`, where `d` does not occur in `E`, by the functional notation `D->E`. Similarly the name `EM` does not appear in `drinker`.

Actually, universal quantification, implication, as well as function formation, are all special cases of one general construct of type theory called *dependent product*. This is the mathematical construction corresponding to an indexed family of functions. A function $f \in \Pi x : D \cdot Cx$ maps an element x of its domain D to its (indexed) codomain Cx. Thus a proof of $\forall x : D \cdot Px$ is a function mapping an element x of D to a proof of proposition Px.

1.4.4 Flexible use of local assumptions

Very often during the course of a proof we want to retrieve a local assumption and reintroduce it explicitly in the goal, for instance in order to get a more general induction hypothesis. The tactic `generalize` is what is needed here:

```
Coq < Section Predicate_Calculus.

Coq < Variables P Q : nat -> Prop.
P is assumed
Q is assumed

Coq < Variable R :  nat -> nat -> Prop.
R is assumed

Coq < Lemma PQR :
Coq <  forall x y:nat, (R x x -> P x -> Q x) -> P x -> R x y -> Q x.
1 subgoal

  P : nat -> Prop
  Q : nat -> Prop
  R : nat -> nat -> Prop
  ============================
   forall x y : nat, (R x x -> P x -> Q x) -> P x -> R x y -> Q x

Coq < intros.
1 subgoal

  P : nat -> Prop
  Q : nat -> Prop
  R : nat -> nat -> Prop
  x : nat
  y : nat
  H : R x x -> P x -> Q x
  H0 : P x
  H1 : R x y
  ============================
   Q x

Coq < generalize H0.
```

```
1 subgoal

  P : nat -> Prop
  Q : nat -> Prop
  R : nat -> nat -> Prop
  x : nat
  y : nat
  H : R x x -> P x -> Q x
  H0 : P x
  H1 : R x y
  ============================
   P x -> Q x
```

Sometimes it may be convenient to use a lemma, although we do not have a direct way to appeal to such an already proven fact. The tactic `cut` permits to use the lemma at this point, keeping the corresponding proof obligation as a new subgoal:

```
Coq < cut (R x x); trivial.
1 subgoal

  P : nat -> Prop
  Q : nat -> Prop
  R : nat -> nat -> Prop
  x : nat
  y : nat
  H : R x x -> P x -> Q x
  H0 : P x
  H1 : R x y
  ============================
   R x x
```

We clean the goal by doing an `Abort` command.

```
Coq < Abort.
```

1.4.5 Equality

The basic equality provided in Coq is Leibniz equality, noted infix like `x=y`, when `x` and `y` are two expressions of type the same Set. The replacement of `x` by `y` in any term is effected by a variety of tactics, such as `rewrite` and `replace`.

Let us give a few examples of equality replacement. Let us assume that some arithmetic function `f` is null in zero:

```
Coq < Variable f : nat -> nat.
f is assumed

Coq < Hypothesis foo : f 0 = 0.
foo is assumed
```

We want to prove the following conditional equality:

```
Coq < Lemma L1 : forall k:nat, k = 0 -> f k = k.
```

As usual, we first get rid of local assumptions with `intro`:

```
Coq < intros k E.
1 subgoal

  P : nat -> Prop
  Q : nat -> Prop
  R : nat -> nat -> Prop
  f : nat -> nat
  foo : f 0 = 0
  k : nat
  E : k = 0
  ============================
   f k = k
```

Let us now use equation `E` as a left-to-right rewriting:

```
Coq < rewrite E.
1 subgoal

  P : nat -> Prop
  Q : nat -> Prop
  R : nat -> nat -> Prop
  f : nat -> nat
  foo : f 0 = 0
  k : nat
  E : k = 0
  ============================
   f 0 = 0
```

This replaced both occurrences of `k` by `O`.

Now `apply foo` will finish the proof:

```
Coq < apply foo.
Proof completed.

Coq < Qed.
intros k E.
rewrite E.
apply foo.
L1 is defined
```

When one wants to rewrite an equality in a right to left fashion, we should use `rewrite <- E` rather than `rewrite E` or the equivalent `rewrite -> E`. Let us now illustrate the tactic `replace`.

```
Coq < Hypothesis f10 : f 1 = f 0.
f10 is assumed

Coq < Lemma L2 : f (f 1) = 0.
1 subgoal

  P : nat -> Prop
  Q : nat -> Prop
  R : nat -> nat -> Prop
  f : nat -> nat
  foo : f 0 = 0
```

```
  f10 : f 1 = f 0
  ============================
   f (f 1) = 0
```

```
Coq < replace (f 1) with 0.
2 subgoals

  P : nat -> Prop
  Q : nat -> Prop
  R : nat -> nat -> Prop
  f : nat -> nat
  foo : f 0 = 0
  f10 : f 1 = f 0
  ============================
   f 0 = 0
subgoal 2 is:
 0 = f 1
```

What happened here is that the replacement left the first subgoal to be proved, but another proof obligation was generated by the `replace` tactic, as the second subgoal. The first subgoal is solved immediately by applying lemma `foo`; the second one transitivity and then symmetry of equality, for instance with tactics `transitivity` and `symmetry`:

```
Coq < apply foo.
1 subgoal

  P : nat -> Prop
  Q : nat -> Prop
  R : nat -> nat -> Prop
  f : nat -> nat
  foo : f 0 = 0
  f10 : f 1 = f 0
  ============================
   0 = f 1
```

```
Coq < transitivity (f 0); symmetry; trivial.
Proof completed.
```

In case the equality $t = u$ generated by `replace` u `with` t is an assumption (possibly modulo symmetry), it will be automatically proved and the corresponding goal will not appear. For instance:

```
Coq < Restart.
1 subgoal

  P : nat -> Prop
  Q : nat -> Prop
  R : nat -> nat -> Prop
  f : nat -> nat
  foo : f 0 = 0
  f10 : f 1 = f 0
  ============================
   f (f 1) = 0
```

```
Coq < replace (f 0) with 0.
1 subgoal
```

```
  P : nat -> Prop
  Q : nat -> Prop
  R : nat -> nat -> Prop
  f : nat -> nat
  foo : f 0 = 0
  f10 : f 1 = f 0
  ============================
   f (f 1) = 0
```

```
Coq < rewrite f10; rewrite foo; trivial.
Proof completed.
```

```
Coq < Qed.
replace (f 0) with 0.
rewrite f10; rewrite foo; trivial.
L2 is defined
```

1.5 Using definitions

The development of mathematics does not simply proceed by logical argumentation from first principles: definitions are used in an essential way. A formal development proceeds by a dual process of abstraction, where one proves abstract statements in predicate calculus, and use of definitions, which in the contrary one instantiates general statements with particular notions in order to use the structure of mathematical values for the proof of more specialised properties.

1.5.1 Unfolding definitions

Assume that we want to develop the theory of sets represented as characteristic predicates over some universe `U`. For instance:

```
Coq < Variable U : Type.
U is assumed
```

```
Coq < Definition set := U -> Prop.
set is defined
```

```
Coq < Definition element (x:U) (S:set) := S x.
element is defined
```

```
Coq < Definition subset (A B:set) := forall x:U, element x A -> element x B.
subset is defined
```

Now, assume that we have loaded a module of general properties about relations over some abstract type `T`, such as transitivity:

```
Coq < Definition transitive (T:Type) (R:T -> T -> Prop) :=
Coq <   forall x y z:T, R x y -> R y z -> R x z.
transitive is defined
```

Now, assume that we want to prove that `subset` is a `transitive` relation.

```
Coq < Lemma subset_transitive : transitive set subset.
1 subgoal

  P : nat -> Prop
  Q : nat -> Prop
  R : nat -> nat -> Prop
  f : nat -> nat
  foo : f 0 = 0
  f10 : f 1 = f 0
  U : Type
  ============================
   transitive set subset
```

In order to make any progress, one needs to use the definition of `transitive`. The `unfold` tactic, which replaces all occurrences of a defined notion by its definition in the current goal, may be used here.

```
Coq < unfold transitive.
1 subgoal

  P : nat -> Prop
  Q : nat -> Prop
  R : nat -> nat -> Prop
  f : nat -> nat
  foo : f 0 = 0
  f10 : f 1 = f 0
  U : Type
  ============================
   forall x y z : set, subset x y -> subset y z -> subset x z
```

Now, we must unfold `subset`:

```
Coq < unfold subset.
1 subgoal

  P : nat -> Prop
  Q : nat -> Prop
  R : nat -> nat -> Prop
  f : nat -> nat
  foo : f 0 = 0
  f10 : f 1 = f 0
  U : Type
  ============================
   forall x y z : set,
   (forall x0 : U, element x0 x -> element x0 y) ->
   (forall x : U, element x y -> element x z) ->
   forall x0 : U, element x0 x -> element x0 z
```

Now, unfolding `element` would be a mistake, because indeed a simple proof can be found by `auto`, keeping `element` an abstract predicate:

```
Coq < auto.
Proof completed.
```

Many variations on `unfold` are provided in Coq. For instance, we may selectively unfold one designated occurrence:

```
Coq < Undo 2.
1 subgoal

  P : nat -> Prop
  Q : nat -> Prop
  R : nat -> nat -> Prop
  f : nat -> nat
  foo : f 0 = 0
  f10 : f 1 = f 0
  U : Type
  ============================
   forall x y z : set, subset x y -> subset y z -> subset x z

Coq < unfold subset at 2.
1 subgoal

  P : nat -> Prop
  Q : nat -> Prop
  R : nat -> nat -> Prop
  f : nat -> nat
  foo : f 0 = 0
  f10 : f 1 = f 0
  U : Type
  ============================
   forall x y z : set,
   subset x y -> (forall x : U, element x y -> element x z) -> subset x z
```

One may also unfold a definition in a given local hypothesis, using the `in` notation:

```
Coq < intros.
1 subgoal

  P : nat -> Prop
  Q : nat -> Prop
  R : nat -> nat -> Prop
  f : nat -> nat
  foo : f 0 = 0
  f10 : f 1 = f 0
  U : Type
  x : set
  y : set
  z : set
  H : subset x y
  H0 : forall x : U, element x y -> element x z
  ============================
   subset x z

Coq < unfold subset in H.
1 subgoal

  P : nat -> Prop
  Q : nat -> Prop
  R : nat -> nat -> Prop
  f : nat -> nat
```

```
  foo : f 0 = 0
  f10 : f 1 = f 0
  U : Type
  x : set
  y : set
  z : set
  H : forall x0 : U, element x0 x -> element x0 y
  H0 : forall x : U, element x y -> element x z
  ============================
   subset x z
```

Finally, the tactic `red` does only unfolding of the head occurrence of the current goal:

```
Coq < red.
1 subgoal

  P : nat -> Prop
  Q : nat -> Prop
  R : nat -> nat -> Prop
  f : nat -> nat
  foo : f 0 = 0
  f10 : f 1 = f 0
  U : Type
  x : set
  y : set
  z : set
  H : forall x0 : U, element x0 x -> element x0 y
  H0 : forall x : U, element x y -> element x z
  ============================
   forall x0 : U, element x0 x -> element x0 z

Coq < auto.
Proof completed.

Coq < Qed.
unfold transitive in |- *.
unfold subset at 2 in |- *.
intros.
unfold subset in H.
red in |- *.
auto.
subset_transitive is defined
```

1.5.2 Principle of proof irrelevance

Even though in principle the proof term associated with a verified lemma corresponds to a defined value of the corresponding specification, such definitions cannot be unfolded in COQ: a lemma is considered an *opaque* definition. This conforms to the mathematical tradition of *proof irrelevance*: the proof of a logical proposition does not matter, and the mathematical justification of a logical development relies only on *provability* of the lemmas used in the formal proof.

Conversely, ordinary mathematical definitions can be unfolded at will, they are *transparent*.

Chapter 2

Induction

2.1 Data Types as Inductively Defined Mathematical Collections

All the notions which were studied until now pertain to traditional mathematical logic. Specifications of objects were abstract properties used in reasoning more or less constructively; we are now entering the realm of inductive types, which specify the existence of concrete mathematical constructions.

2.1.1 Booleans

Let us start with the collection of booleans, as they are specified in the COQ's `Prelude` module:

```
Coq < Inductive bool :  Set := true | false.
bool is defined
bool_rect is defined
bool_ind is defined
bool_rec is defined
```

Such a declaration defines several objects at once. First, a new `Set` is declared, with name `bool`. Then the *constructors* of this `Set` are declared, called `true` and `false`. Those are analogous to introduction rules of the new Set `bool`. Finally, a specific elimination rule for `bool` is now available, which permits to reason by cases on `bool` values. Three instances are indeed defined as new combinators in the global context: `bool_ind`, a proof combinator corresponding to reasoning by cases, `bool_rec`, an if-then-else programming construct, and `bool_rect`, a similar combinator at the level of types. Indeed:

```
Coq < Check bool_ind.
bool_ind
     : forall P : bool -> Prop, P true -> P false -> forall b : bool, P b

Coq < Check bool_rec.
bool_rec
     : forall P : bool -> Set, P true -> P false -> forall b : bool, P b

Coq < Check bool_rect.
bool_rect
     : forall P : bool -> Type, P true -> P false -> forall b : bool, P b
```

Let us for instance prove that every Boolean is true or false.

```
Coq < Lemma duality : forall b:bool, b = true \/ b = false.
```

```
1 subgoal

  P : nat -> Prop
  Q : nat -> Prop
  R : nat -> nat -> Prop
  f : nat -> nat
  foo : f 0 = 0
  f10 : f 1 = f 0
  U : Type
  ============================
   forall b : bool, b = true \/ b = false

Coq < intro b.
1 subgoal

  P : nat -> Prop
  Q : nat -> Prop
  R : nat -> nat -> Prop
  f : nat -> nat
  foo : f 0 = 0
  f10 : f 1 = f 0
  U : Type
  b : bool
  ============================
   b = true \/ b = false
```

We use the knowledge that `b` is a `bool` by calling tactic `elim`, which is this case will appeal to combinator `bool_ind` in order to split the proof according to the two cases:

```
Coq < elim b.
2 subgoals

  P : nat -> Prop
  Q : nat -> Prop
  R : nat -> nat -> Prop
  f : nat -> nat
  foo : f 0 = 0
  f10 : f 1 = f 0
  U : Type
  b : bool
  ============================
   true = true \/ true = false
subgoal 2 is:
 false = true \/ false = false
```

It is easy to conclude in each case:

```
Coq < left; trivial.
1 subgoal

  P : nat -> Prop
  Q : nat -> Prop
  R : nat -> nat -> Prop
  f : nat -> nat
```

```
  foo : f 0 = 0
  f10 : f 1 = f 0
  U : Type
  b : bool
  ============================
   false = true \/ false = false
```

```
Coq < right; trivial.
Proof completed.
```

Indeed, the whole proof can be done with the combination of the `simple induction` tactic, which combines `intro` and `elim`, with good old `auto`:

```
Coq < Restart.
1 subgoal

  P : nat -> Prop
  Q : nat -> Prop
  R : nat -> nat -> Prop
  f : nat -> nat
  foo : f 0 = 0
  f10 : f 1 = f 0
  U : Type
  ============================
   forall b : bool, b = true \/ b = false
```

```
Coq < simple induction b; auto.
Proof completed.
```

```
Coq < Qed.
simple induction b; auto.
duality is defined
```

2.1.2 Natural numbers

Similarly to Booleans, natural numbers are defined in the `Prelude` module with constructors `S` and `O`:

```
Coq < Inductive nat : Set :=
Coq <   | O : nat
Coq <   | S : nat -> nat.
nat is defined
nat_rect is defined
nat_ind is defined
nat_rec is defined
```

The elimination principles which are automatically generated are Peano's induction principle, and a recursion operator:

```
Coq < Check nat_ind.
nat_ind
     : forall P : nat -> Prop,
       P O -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n
```

```
Coq < Check nat_rec.
nat_rec
     : forall P : nat -> Set,
       P O -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n
```

Let us start by showing how to program the standard primitive recursion operator `prim_rec` from the more general `nat_rec`:

```
Coq < Definition prim_rec := nat_rec (fun i:nat => nat).
prim_rec is defined
```

That is, instead of computing for natural `i` an element of the indexed `Set (P i)`, `prim_rec` computes uniformly an element of `nat`. Let us check the type of `prim_rec`:

```
Coq < Check prim_rec.
prim_rec
     : (fun _ : nat => nat) O ->
       (forall n : nat, (fun _ : nat => nat) n -> (fun _ : nat => nat) (S n)) ->
       forall n : nat, (fun _ : nat => nat) n
```

Oops! Instead of the expected type `nat->(nat->nat->nat)->nat->nat` we get an apparently more complicated expression. Indeed the type of `prim_rec` is equivalent by rule β to its expected type; this may be checked in COQ by command `Eval Cbv Beta`, which β-reduces an expression to its *normal form*:

```
Coq < Eval cbv beta in
Coq <   ((fun _:nat => nat) O ->
Coq <    (forall y:nat, (fun _:nat => nat) y -> (fun _:nat => nat) (S y)) ->
Coq <    forall n:nat, (fun _:nat => nat) n).
     = nat -> (nat -> nat -> nat) -> nat -> nat
     : Set
```

Let us now show how to program addition with primitive recursion:

```
Coq < Definition addition (n m:nat) := prim_rec m (fun p rec:nat => S rec) n.
addition is defined
```

That is, we specify that `(addition n m)` computes by cases on `n` according to its main constructor; when `n = O`, we get `m`; when `n = S p`, we get `(S rec)`, where `rec` is the result of the recursive computation `(addition p m)`. Let us verify it by asking COQ to compute for us say $2+3$:

```
Coq < Eval compute in (addition (S (S O)) (S (S (S O)))).
     = S (S (S (S (S O))))
     : (fun _ : nat => nat) (S (S O))
```

Actually, we do not have to do all explicitly. COQ provides a special syntax `Fixpoint/match` for generic primitive recursion, and we could thus have defined directly addition as:

```
Coq < Fixpoint plus (n m:nat) {struct n} : nat :=
Coq <   match n with
Coq <   | O => m
Coq <   | S p => S (plus p m)
Coq <   end.
plus is recursively defined
```

For the rest of the session, we shall clean up what we did so far with types `bool` and `nat`, in order to use the initial definitions given in COQ's `Prelude` module, and not to get confusing error messages due to our redefinitions. We thus revert to the state before our definition of `bool` with the `Reset` command:

```
Coq < Reset bool.
```

2.1.3 Simple proofs by induction

Let us now show how to do proofs by structural induction. We start with easy properties of the `plus` function we just defined. Let us first show that $n = n+0$.

```
Coq < Lemma plus_n_O : forall n:nat, n = n + 0.
1 subgoal

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

Coq < intro n; elim n.
2 subgoals

  n : nat
  ============================
   0 = 0 + 0
subgoal 2 is:
 forall n0 : nat, n0 = n0 + 0 -> S n0 = S n0 + 0
```

What happened was that `elim n`, in order to construct a `Prop` (the initial goal) from a `nat` (i.e. `n`), appealed to the corresponding induction principle `nat_ind` which we saw was indeed exactly Peano's induction scheme. Pattern-matching instantiated the corresponding predicate `P` to `fun n:nat => n = n  0+`, and we get as subgoals the corresponding instantiations of the base case `(P O)` , and of the inductive step `forall y:nat, P y -> P (S y)`. In each case we get an instance of function `plus` in which its second argument starts with a constructor, and is thus amenable to simplification by primitive recursion. The COQ tactic `simpl` can be used for this purpose:

```
Coq < simpl.
2 subgoals

  n : nat
  ============================
   0 = 0
subgoal 2 is:
 forall n0 : nat, n0 = n0 + 0 -> S n0 = S n0 + 0

Coq < auto.
1 subgoal

  n : nat
  ============================
   forall n0 : nat, n0 = n0 + 0 -> S n0 = S n0 + 0
```

We proceed in the same way for the base step:

```
Coq < simpl; auto.
Proof completed.

Coq < Qed.
intro n; elim n.
 simpl in |- *.
   auto.
 simpl in |- *; auto.
plus_n_O is defined
```

Here `auto` succeeded, because it used as a hint lemma `eq_S`, which say that successor preserves equality:

```
Coq < Check eq_S.
eq_S
     : forall x y : nat, x = y -> S x = S y
```

Actually, let us see how to declare our lemma `plus_n_O` as a hint to be used by `auto`:

```
Coq < Hint Resolve plus_n_O .
```

We now proceed to the similar property concerning the other constructor `S`:

```
Coq < Lemma plus_n_S : forall n m:nat, S (n + m) = n + S m.
1 subgoal

  ============================
   forall n m : nat, S (n + m) = n + S m
```

We now go faster, remembering that tactic `simple induction` does the necessary `intros` before applying `elim`. Factoring simplification and automation in both cases thanks to tactic composition, we prove this lemma in one line:

```
Coq < simple induction n; simpl; auto.
Proof completed.
```

```
Coq < Qed.
simple induction n; simpl in |- *; auto.
plus_n_S is defined
```

```
Coq < Hint Resolve plus_n_S .
```

Let us end this exercise with the commutativity of `plus`:

```
Coq < Lemma plus_com : forall n m:nat, n + m = m + n.
1 subgoal

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

Here we have a choice on doing an induction on `n` or on `m`, the situation being symmetric. For instance:

```
Coq < simple induction m; simpl; auto.
1 subgoal

  n : nat
  m : nat
  ============================
   forall n0 : nat, n + n0 = n0 + n -> n + S n0 = S (n0 + n)
```

Here `auto` succeeded on the base case, thanks to our hint `plus_n_O`, but the induction step requires rewriting, which `auto` does not handle:

```
Coq < intros m' E; rewrite <- E; auto.
Proof completed.
```

```
Coq < Qed.
simple induction m; simpl in |- *; auto.
intros m' E; rewrite <- E; auto.
plus_com is defined
```

2.1.4 Discriminate

It is also possible to define new propositions by primitive recursion. Let us for instance define the predicate which discriminates between the constructors `O` and `S`: it computes to `False` when its argument is `O`, and to `True` when its argument is of the form `(S n)`:

```
Coq < Definition Is_S (n:nat) := match n with
Coq <                            | O => False
Coq <                            | S p => True
Coq <                            end.
Is_S is defined
```

Now we may use the computational power of `Is_S` in order to prove trivially that `(Is_S (S n))`:

```
Coq < Lemma S_Is_S : forall n:nat, Is_S (S n).
1 subgoal

  ============================
   forall n : nat, Is_S (S n)

Coq < simpl; trivial.
Proof completed.

Coq < Qed.
simpl in |- *; trivial.
S_Is_S is defined
```

But we may also use it to transform a `False` goal into `(Is_S O)`. Let us show a particularly important use of this feature; we want to prove that `O` and `S` construct different values, one of Peano's axioms:

```
Coq < Lemma no_confusion : forall n:nat, 0 <> S n.
1 subgoal

  ============================
   forall n : nat, 0 <> S n
```

First of all, we replace negation by its definition, by reducing the goal with tactic `red`; then we get contradiction by successive `intros`:

```
Coq < red; intros n H.
1 subgoal

  n : nat
  H : 0 = S n
  ============================
   False
```

Now we use our trick:

```
Coq < change (Is_S 0).
1 subgoal

  n : nat
  H : 0 = S n
  ============================
   Is_S 0
```

Now we use equality in order to get a subgoal which computes out to `True`, which finishes the proof:

```
Coq < rewrite H; trivial.
1 subgoal

  n : nat
  H : 0 = S n
  ============================
   Is_S (S n)

Coq < simpl; trivial.
Proof completed.
```

Actually, a specific tactic `discriminate` is provided to produce mechanically such proofs, without the need for the user to define explicitly the relevant discrimination predicates:

```
Coq < Restart.
1 subgoal

  ============================
   forall n : nat, 0 <> S n

Coq < intro n; discriminate.
Proof completed.

Coq < Qed.
intro n; discriminate.
no_confusion is defined
```

2.2 Logic programming

In the same way as we defined standard data-types above, we may define inductive families, and for instance inductive predicates. Here is the definition of predicate $\leq$ over type `nat`, as given in COQ's `Prelude` module:

```
Coq < Inductive le (n:nat) : nat -> Prop :=
Coq <   | le_n : le n n
Coq <   | le_S : forall m:nat, le n m -> le n (S m).
```

This definition introduces a new predicate `le:nat->nat->Prop`, and the two constructors `le_n` and `le_S`, which are the defining clauses of `le`. That is, we get not only the "axioms" `le_n` and `le_S`, but also the converse property, that `(le n m)` if and only if this statement can be obtained as a consequence of these defining clauses; that is, `le` is the minimal predicate verifying clauses `le_n` and `le_S`. This is insured, as in the case of inductive data types, by an elimination principle, which here amounts to an induction principle `le_ind`, stating this minimality property:

```
Coq < Check le.
le
     : nat -> nat -> Prop

Coq < Check le_ind.
le_ind
     : forall (n : nat) (P : nat -> Prop),
       P n ->
       (forall m : nat, le n m -> P m -> P (S m)) ->
       forall n0 : nat, le n n0 -> P n0
```

Let us show how proofs may be conducted with this principle. First we show that $n \leq m \Rightarrow n+1 \leq m+1$:

```
Coq < Lemma le_n_S : forall n m:nat, le n m -> le (S n) (S m).
1 subgoal

  ============================
   forall n m : nat, le n m -> le (S n) (S m)

Coq < intros n m n_le_m.
1 subgoal

  n : nat
  m : nat
  n_le_m : le n m
  ============================
   le (S n) (S m)

Coq < elim n_le_m.
2 subgoals

  n : nat
  m : nat
  n_le_m : le n m
  ============================
   le (S n) (S n)
subgoal 2 is:
 forall m0 : nat, le n m0 -> le (S n) (S m0) -> le (S n) (S (S m0))
```

What happens here is similar to the behaviour of `elim` on natural numbers: it appeals to the relevant induction principle, here `le_ind`, which generates the two subgoals, which may then be solved easily with the help of the defining clauses of `le`.

```
Coq < apply le_n; trivial.
1 subgoal

  n : nat
  m : nat
  n_le_m : le n m
  ============================
   forall m0 : nat, le n m0 -> le (S n) (S m0) -> le (S n) (S (S m0))

Coq < intros; apply le_S; trivial.
Proof completed.
```

Now we know that it is a good idea to give the defining clauses as hints, so that the proof may proceed with a simple combination of `induction` and `auto`.

```
Coq < Restart.
1 subgoal

  ============================
   forall n m : nat, le n m -> le (S n) (S m)

Coq < Hint Resolve le_n le_S .
```

We have a slight problem however. We want to say "Do an induction on hypothesis `(le n m)`", but we have no explicit name for it. What we do in this case is to say "Do an induction on the first unnamed hypothesis", as follows.

```
Coq < simple induction 1; auto.
Proof completed.

Coq < Qed.
simple induction 1; auto.
le_n_S is defined
```

Here is a more tricky problem. Assume we want to show that $n \leq 0 \Rightarrow n = 0$. This reasoning ought to follow simply from the fact that only the first defining clause of `le` applies.

```
Coq < Lemma tricky : forall n:nat, le n 0 -> n = 0.
1 subgoal

  ============================
   forall n : nat, le n 0 -> n = 0
```

However, here trying something like `induction 1` would lead nowhere (try it and see what happens). An induction on `n` would not be convenient either. What we must do here is analyse the definition of `le` in order to match hypothesis `(le n O)` with the defining clauses, to find that only `le_n` applies, whence the result. This analysis may be performed by the "inversion" tactic `inversion_clear` as follows:

```
Coq < intros n H; inversion_clear H.
1 subgoal

  n : nat
  ============================
   0 = 0

Coq < trivial.
Proof completed.

Coq < Qed.
intros n H; inversion_clear H.
trivial.
tricky is defined
```

Chapter 3

Modules

3.1 Opening library modules

When you start COQ without further requirements in the command line, you get a bare system with few libraries loaded. As we saw, a standard prelude module provides the standard logic connectives, and a few arithmetic notions. If you want to load and open other modules from the library, you have to use the `Require` command, as we saw for classical logic above. For instance, if you want more arithmetic constructions, you should request:

```
Coq < Require Import Arith.
```

Such a command looks for a (compiled) module file `Arith.vo` in the libraries registered by COQ. Libraries inherit the structure of the file system of the operating system and are registered with the command `Add LoadPath`. Physical directories are mapped to logical directories. Especially the standard library of COQ is pre-registered as a library of name `Coq`. Modules have absolute unique names denoting their place in COQ libraries. An absolute name is a sequence of single identifiers separated by dots. E.g. the module `Arith` has full name `Coq.Arith.Arith` and because it resides in eponym subdirectory `Arith` of the standard library, it can be as well required by the command

```
Coq < Require Import Coq.Arith.Arith.
```

This may be useful to avoid ambiguities if somewhere, in another branch of the libraries known by Coq, another module is also called `Arith`. Notice that by default, when a library is registered, all its contents, and all the contents of its subdirectories recursively are visible and accessible by a short (relative) name as `Arith`. Notice also that modules or definitions not explicitly registered in a library are put in a default library called `Top`.

The loading of a compiled file is quick, because the corresponding development is not type-checked again.

3.2 Creating your own modules

You may create your own modules, by writing COQ commands in a file, say `my_module.v`. Such a module may be simply loaded in the current context, with command `Load my_module`. It may also be compiled, in "batch" mode, using the UNIX command `coqc`. Compiling the module `my_module.v` creates a file `my_module.vo` that can be reloaded with command `Require Import my_module`.

If a required module depends on other modules then the latters are automatically required beforehand. However their contents is not automatically visible. If you want a module `M` required in a module `N` to be automatically visible when `N` is required, you should use `Require Export M` in your module `N`.

3.3 Managing the context

It is often difficult to remember the names of all lemmas and definitions available in the current context, especially if large libraries have been loaded. A convenient `SearchAbout` command is available to lookup all known facts concerning a given predicate. For instance, if you want to know all the known lemmas about the less or equal relation, just ask:

```
Coq < SearchAbout le.
tricky: forall n : nat, le n 0 -> n = 0
Top.le_n_S: forall n m : nat, le n m -> le (S n) (S m)
le_ind:
  forall (n : nat) (P : nat -> Prop),
  P n ->
  (forall m : nat, le n m -> P m -> P (S m)) ->
  forall n0 : nat, le n n0 -> P n0
le_n: forall n : nat, le n n
le_S: forall n m : nat, le n m -> le n (S m)
```

Another command `Search` displays only lemmas where the searched predicate appears at the head position in the conclusion.

```
Coq < Search le.
Top.le_n_S: forall n m : nat, le n m -> le (S n) (S m)
le_n: forall n : nat, le n n
le_S: forall n m : nat, le n m -> le n (S m)
```

A new and more convenient search tool is SearchPattern developed by Yves Bertot. It allows to find the theorems with a conclusion matching a given pattern, where _ can be used in place of an arbitrary term. We remark in this example, that COQ provides usual infix notations for arithmetic operators.

```
Coq < SearchPattern (_ + _ = _).
le_plus_minus_r: forall n m : nat, n <= m -> n + (m - n) = m
mult_acc_aux: forall n m p : nat, m + n * p = mult_acc m p n
plus_O_n: forall n : nat, 0 + n = n
plus_Sn_m: forall n m : nat, S n + m = S (n + m)
mult_n_Sm: forall n m : nat, n * m + n = n * S m
plus_0_l: forall n : nat, 0 + n = n
plus_0_r: forall n : nat, n + 0 = n
plus_comm: forall n m : nat, n + m = m + n
plus_Snm_nSm: forall n m : nat, S n + m = n + S m
plus_assoc: forall n m p : nat, n + (m + p) = n + m + p
plus_permute: forall n m p : nat, n + (m + p) = m + (n + p)
plus_assoc_reverse: forall n m p : nat, n + m + p = n + (m + p)
plus_permute_2_in_4: forall n m p q : nat, n + m + (p + q) = n + p + (m + q)
plus_tail_plus: forall n m : nat, n + m = tail_plus n m
plus_com: forall n m : nat, n + m = m + n
```

3.4 Now you are on your own

This tutorial is necessarily incomplete. If you wish to pursue serious proving in COQ, you should now get your hands on COQ's Reference Manual, which contains a complete description of all the tactics we saw, plus many more. You also should look in the library of developed theories which is distributed with COQ, in order to acquaint yourself with various proof techniques.