INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET AUTOMATIQUE

The Coq Proof Assistant
A Tutorial

Version 5.10

Gérard Huet, Gilles Kahn and Christine Paulin-Mohring

N° 0178

July 1995

PROGRAMME 2

Calcul symbolique,
programmation
et génie logiciel

Rapport technique

ISSN 0249-6399

The Coq Proof Assistant
A Tutorial

Version 5.10 *

Gérard Huet, Gilles Kahn and Christine Paulin-Mohring

Projet Coq

INRIA-Rocquencourt† — **CNRS - ENS Lyon** ‡

*This research was partly supported by ESPRIT Basic Research Action "Types" and by the GDR "Programmation" co-financed by MRE-PRC and CNRS.

†B.P. 105, 78153 Le Chesnay CEDEX, France.

‡LIP URA 1398 du CNRS, 46 Allée d'Italie, 69364 Lyon CEDEX 07, France.

1^{st} Printing February 1^{st}, 1995

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, and mainly on Unix machines. 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.

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 such as Emacs or Centaur. Instructions on installation procedures, as well as more comprehensive documentation, may be found in the standard distribution of Coq, which may be obtained by anonymous FTP from site `ftp.inria.fr`, directory `INRIA/coq/V5.10`.

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's screen. The sequence of such examples is a valid Coq session, unless otherwise specified. This version of the tutorial has been prepared on a SPARC station running UNIX. It assumes that the user has prepared in his home directory an initialisation file `.coqrc.5.10` containing the following lines:

```
Coq < AddPath ".".

Coq < AddPath "$COQTH/LISTS".

Coq < AddPath "$COQTH/SETS".

Coq < AddPath "$COQTH/BOOL".
```

so that the standard invocation of Coq delivers a message such as:

```
unix: coqtop
>       Caml Light version 0.61

Welcome to Coq V5.10 - Thu Nov 10 18:27:12 MET 1994

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 hotline `coq@pauillac.inria.fr`.

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 judgement 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.
O
     : 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 axiomatised notions which are defined in the arithmetic prelude, automatically loaded when executing the command '`Require Basis.`' in the initialisation file `.coqrc`.

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 O).
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).
(gt n O)
     : Prop
```

1.1.2 Definitions

The initial prelude `Basis` 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 := (S two) : nat.
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
```

The abstraction brackets are explained as follows. The expression `[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 `[x:A]e`. For instance we could as well have defined `double` as `[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: `(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 (m:nat)(gt m O).
(m:nat)(gt m O)
      : Prop
```

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 < 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->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 *judgement*. The new prompt `Unnamed_thm <` indicates that 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 judgement, possibly from proofs of some hypothetical judgements, which are then added to the current list of conjectured judgements. For instance, the `Intro` tactic is applicable to any judgement 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
  H : A->B->C
  ============================
   (A->B)->A->C
```

Warning to users of `Coq` previous versions: The display of a sequent in older versions of `Coq` is inverse of this convention: the goal is displayed above the double line, the hypotheses below.

Several introductions may be done in one step:

```
Coq < Intros H' HA.
1 subgoal
  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
  H : A->B->C
  H' : A->B
  HA : A
  ============================
   A
subgoal 2 is:
 B
```

We are now in the situation where we have two judgements 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 judgement, which are still available for the judgements 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
  H : A->B->C
  H' : A->B
  HA : A
  ============================
   B
```

Now H' applies:

```
Coq < Apply H'.
1 subgoal
  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.
Subtree proved!
```

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

Warning to users of Coq older versions: In order to enter the proof engine, at this point a dummy `Goal.` command had to be typed in.

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
  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].
Subtree proved!
```

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->B->C)->(A->B)->A->C

Coq < Auto.
Subtree proved!
```

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 that 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/\B->B/\A

Coq < Intro.
1 subgoal
  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
  H : A/\B
  ============================
   A->B->B/\A
```

We now use the conjuction introduction tactic Split, which splits the conjunctive goal into the two subgoals:

```
Coq < Split.
2 subgoals
  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/\B->B/\A

Coq < Intro H; Elim H; Auto.
Subtree proved!

Coq < Save.
(Intros 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
     : (A:Prop)(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` 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\/B->B\/A
```

```
Coq < Intro H; Elim H.
2 subgoals
  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
  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
  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
  HA : A
  ============================
   A
subgoal 2 is:
 B->B\/A

Coq < Trivial.
1 subgoal
  H : A\/B
  ============================
   B->B\/A
```

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

```
Coq < Auto.
Subtree proved!
```

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\/B->B\/A

Coq < Tauto.
Subtree proved!

Coq < Save.
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 =
[H:A\/B]
 (or_ind A B B\/A [H0:A](or_intror B A H0) [H0:B](or_introl B A H0) H)
     : A\/B->B\/A
```

It is not easy to understand the notation for proof terms without a few explanations. The square brackets, such as `[HH1:A\/B]`, correspond to `Intro HH1`, whereas a subterm such as `(or_intror B A HH6)` corresponds to the sequence `Apply or_intror; Exact HH6`. The extra arguments `B` and `A` correspond to instanciations to the generic combinator `or_intror`, which 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->B/\C->A->B/\A->C

Coq < Tauto.
Subtree proved!

Coq < Save.
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->B)->A)->A

Coq < Try Tauto.
1 subgoal
  ============================
   ((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->B)->A)->A)

Coq < Tauto.
```

```
Subtree proved!

Coq < Save.
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 Classical.
[Reinterning Classical ...done]

Coq < Check NNPP.
NNPP
     : (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->B)->A)->A

Coq < Apply NNPP; Tauto.
Subtree proved!

Coq < Save.
(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 < Variable 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
  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.
Subtree proved!

Coq < Save.
Tauto.
NoMember is defined

Coq < End club.
Constant NoMember:
```

```
(Scottish:Prop)
 (RedSocks:Prop)
  (WearKilt:Prop)
   (Married:Prop)
    (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.

```
Coq < Reset n.
```

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 : (x,y:D) (R x y) -> (R y x).
R_symmetric is assumed

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

Remark the syntax (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 : (x:D)(Ex [y:D](R x y)) -> (R x x).
1 subgoal
  D : Set
  R : D->D->Prop
  R_symmetric : (x:D)(y:D)(R x y)->(R y x)
  R_transitive : (x:D)(y:D)(z:D)(R x y)->(R y z)->(R x z)
  ============================
   (x:D)(Ex [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
     : (A:Set)(A->Prop)->Prop
```

and the notation (Ex [x:D](P x)) is just concrete syntax for (ex D [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 : (x:D)(y:D)(R x y)->(R y x)
  R_transitive : (x:D)(y:D)(z:D)(R x y)->(R y z)->(R x z)
  x : D
```

```
  x_Rlinked : (Ex [y:D](R x y))
  ============================
   (R x x)
```

Remark that Intro treats universal quantification in the same way as the premisses 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 : (x:D)(y:D)(R x y)->(R y x)
  R_transitive : (x:D)(y:D)(z:D)(R x y)->(R y z)->(R x z)
  y : D
  ============================
   (Ex [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 : (x:D)(y:D)(R x y)->(R y x)
  R_transitive : (x:D)(y:D)(z:D)(R x y)->(R y z)->(R x z)
  x : D
  x_Rlinked : (Ex [y:D](R x y))
  ============================
   (x0:D)(R x x0)->(R x x)

Coq < Intros y Rxy.
1 subgoal
  D : Set
  R : D->D->Prop
  R_symmetric : (x:D)(y:D)(R x y)->(R y x)
  R_transitive : (x:D)(y:D)(z:D)(R x y)->(R y z)->(R x z)
  x : D
  x_Rlinked : (Ex [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 instanciate 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 : (x:D)(y:D)(R x y)->(R y x)
  R_transitive : (x:D)(y:D)(z:D)(R x y)->(R y z)->(R x z)
  x : D
  x_Rlinked : (Ex [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 : (x:D)(y:D)(R x y)->(R y x)
  R_transitive : (x:D)(y:D)(z:D)(R x y)->(R y z)->(R x z)
  x : D
  x_Rlinked : (Ex [y:D](R x y))
  y : D
  Rxy : (R x y)
  ============================
   (R y x)

Coq < Apply R_symmetric; Assumption.
Subtree proved!


Coq < Save.
```

Let us now close the current section.

```
Coq < End R_sym_trans.
Constant refl_if:
  ((x:D)(y:D)(R x y)->(R y x))
   ->((x:D)(y:D)(z:D)(R x y)->(R y z)->(R x z))
      ->(x:D)(Ex [y:D](R x y))->(R x x)
```

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 : ((x:D)(P x)) -> (Ex [a:D](P a)).
1 subgoal
  D : Set
  R : D->D->Prop
  P : D->Prop
  d : D
  ============================
   ((x:D)(P x))->(Ex [a:D](P a))

Coq < Intro UnivP.
1 subgoal
  D : Set
  R : D->D->Prop
  P : D->Prop
  d : D
  UnivP : (x:D)(P x)
  ============================
   (Ex [a:D](P a))
```

First of all, notice the pair of parentheses around (x:D)(P x) in the statement of lemma weird. If we had ommitted 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.
Subtree proved!

Coq < Save.
Intros UnivP.
(Exists d ;Try 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 : (A:Prop) A \/ ~A.
EM is assumed

Coq < Lemma drinker : (Ex [x:D](P x) -> (x:D)(P x)).
1 subgoal
  D : Set
  R : D->D->Prop
  P : D->Prop
  d : D
  EM : (A:Prop)A\/~A
  ============================
   (Ex [x:D](P x)->(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  (Ex [x:D] ~(P x))).
2 subgoals
  D : Set
  R : D->D->Prop
  P : D->Prop
  d : D
  EM : (A:Prop)A\/~A
  ============================
   (Ex [x:D]~(P x))->(Ex [x:D](P x)->(x0:D)(P x0))
subgoal 2 is:
 ~(Ex [x:D]~(P x))->(Ex [x:D](P x)->(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 : (A:Prop)A\/~A
  Non_drinker : (Ex [x:D]~(P x))
  Tom : D
  Tom_does_not_drink : ~(P Tom)
  ============================
   (Ex [x:D](P x)->(x0:D)(P x0))
subgoal 2 is:
 ~(Ex [x:D]~(P x))->(Ex [x:D](P x)->(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 : (A:Prop)A\/~A
  Non_drinker : (Ex [x:D]~(P x))
  Tom : D
  Tom_does_not_drink : ~(P Tom)
  Tom_drinks : (P Tom)
  ============================
   (x:D)(P x)
subgoal 2 is:
 ~(Ex [x:D]~(P x))->(Ex [x:D](P x)->(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 : (A:Prop)A\/~A
  ============================
   ~(Ex [x:D]~(P x))->(Ex [x:D](P x)->(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-emptyness 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 : (A:Prop)A\/~A
  No_nondrinker : ~(Ex [x:D]~(P x))
  d_drinks : (P d)
  ============================
   (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 : (A:Prop)A\/~A
  No_nondrinker : ~(Ex [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 (Ex [x:D]~(P x)); Trivial.
1 subgoal
  D : Set
  R : D->D->Prop
  P : D->Prop
  d : D
  EM : (A:Prop)A\/~A
  No_nondrinker : ~(Ex [x:D]~(P x))
  d_drinks : (P d)
  Dick : D
  Dick_does_not_drink : ~(P Dick)
  ============================
   (Ex [x:D]~(P x))

Coq < Exists Dick; Trivial.
Subtree proved!

Coq < Save.
Elim (EM (Ex [x:D](not (P x)))).
(Intros Non_drinker;Elim Non_drinker;Intros Tom Tom_does_not_drink).
```

```
(Exists Tom ;Intros Tom_drinks).
(Absurd (P Tom);Try Trivial).
(Intros No_nondrinker;Exists d ;Intros d_drinks).
(Intros Dick;Elim (EM (P Dick));Try Trivial).
(Intros Dick_does_not_drink;Absurd (Ex [x:D](not (P x)));Try Trivial).
(Exists Dick ;Try Trivial).
drinker is defined
```

Now, let us close the main section:

```
Coq < End Predicate_calculus.
Constant refl_if:
  (D:Set)
   (R:D->D->Prop)
    ((x:D)(y:D)(R x y)->(R y x))
     ->((x:D)(y:D)(z:D)(R x y)->(R y z)->(R x z))
        ->(x:D)(Ex [y:D](R x y))->(R x x)
Constant weird:
  (D:Set)(P:D->Prop)D->((x:D)(P x))->(Ex [a:D](P a))
Constant drinker:
  (D:Set)(P:D->Prop)D->((A:Prop)A\/~A)->(Ex [x:D](P x)->(x0:D)(P x0))
```

Remark how the three theorems are completely generic is 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 prettyprinter replaces systematically a quantification such as (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 < Variable P,Q:nat->Prop. Variable R: nat->nat->Prop.
P is assumed
Q is assumed
R is assumed

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

```
1 subgoal
  ============================
   (x:nat)(y:nat)((R x x)->(P x)->(Q x))->(P x)->(R x y)->(Q x)

Coq < Intros.
1 subgoal
  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
  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
  x : nat
  y : nat
  H : (R x x)->(P x)->(Q x)
  H0 : (P x)
  H1 : (R x y)
  ============================
   (R x x)
```

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 O)=O.
foo is assumed
```

We want to prove the following conditional equality:

```
Coq < Lemma L1 : (k:nat)k=O->(f k)=k.
```

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

```
Coq < Intros k E.
1 subgoal
  k : nat
  E : k=O
  ============================
   (f k)=k
```

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

```
Coq < Rewrite E.
1 subgoal
  k : nat
  E : k=O
  ============================
   (f O)=O
```

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

Warning to users of Coq old versions: In CoqV5.8 only the second occurrence of `k` would have been replaced, and we would have had to use `Rewrite` twice in order to get the same effect.

Now `Apply foo` will finish the proof:

```
Coq < Apply foo.
Subtree proved!

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

Coq < Replace (f O) with O.
```

```
2 subgoals
  ============================
   (f O)=O
subgoal 2 is:
 O=(f O)
```

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 appying lemma `foo`; the second one too, provided we apply first symmetry of equality, for instance with tactic `Symmetry`:

```
Coq < Apply foo.
1 subgoal
  ============================
   O=(f O)

Coq < Symmetry; Apply foo.
Subtree proved!

Coq < Save.
Replace (f O) with O.
Apply foo.
(Symmetry;Apply foo).
L2 is defined
```

1.4.6 Predicate calculus over Type

We just explained the basis of first-order reasoning in the universe of mathematical Sets. Similar reasoning is available at the level of abstract Types. In order to develop such abtract reasoning, one must load the library `Logic_Type`.

```
Coq < Require Logic_Type.
Warning: Logic_Type already imported
```

New proof combinators are now available, such as the existential quantification `exT` over a Type, available with syntax `(ExT P)`. The corresponding introduction combinator may be invoked by the tactic `Exists` as above.

```
Coq < Check exT_intro.
exT_intro
     : (A:Type)(P:A->Prop)(x:A)(P x)->(ExT P)
```

Similarly, equality over Type is available, with notation `M==N`. The equality tactics process `==` in the same way as `=`.

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 instanciates 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](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 <                    (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
  ============================
   (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
  ============================
   (x:set)(y:set)(z:set)(subset x y)->(subset y z)->(subset x z)
```

Now, we must unfold `subset`:

```
Coq < Unfold subset.
1 subgoal
  ============================
   (x:set)
    (y:set)
     (z:set)
      ((x0:U)(element x0 x)->(element x0 y))
       ->((x0:U)(element x0 y)->(element x0 z))
          ->(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.
Subtree proved!
```

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

```
Coq < Undo 2.
1 subgoal
  ============================
   (x:set)(y:set)(z:set)(subset x y)->(subset y z)->(subset x z)

Coq < Unfold 2 subset.
1 subgoal
  ============================
   (x:set)
    (y:set)
     (z:set)
      (subset x y)->((x0:U)(element x0 y)->(element x0 z))->(subset x z)
```

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

```
Coq < Intros.
1 subgoal
  x : set
  y : set
  z : set
  H : (subset x y)
  H0 : (x:U)(element x y)->(element x z)
  ============================
   (subset x z)

Coq < Unfold subset in H.
1 subgoal
  x : set
  y : set
  z : set
```

```
  H : (x0:U)(element x0 x)->(element x0 y)
  H0 : (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
  x : set
  y : set
  z : set
  H : (x0:U)(element x0 x)->(element x0 y)
  H0 : (x:U)(element x y)->(element x z)
  ============================
   (x0:U)(element x0 x)->(element x0 z)

Coq < Auto. Save.
Subtree proved!
Unfold transitive .
Unfold 2 subset .
Intros.
Unfold subset  in H.
Red.
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*. It is possible to enforce the reverse convention by declaring a definition as *opaque* or a lemma as *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 : bool | false : bool.
bool_ind is defined
bool_rec is defined
bool_rect is defined
bool 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
     : (P:bool->Prop)(P true)->(P false)->(b:bool)(P b)

Coq < Check bool_rec.
bool_rec
     : (P:bool->Set)(P true)->(P false)->(b:bool)(P b)

Coq < Check bool_rect.
```

```
bool_rect
     : (P:bool->Type)(P true)->(P false)->(b:bool)(P b)
```

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

```
Coq < Lemma duality : (b:bool)(b=true \/ b=false).
1 subgoal
  ============================
   (b:bool)b=true\/b=false

Coq < Intro b.
1 subgoal
  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
  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
  b : bool
  ============================
   false=true\/false=false

Coq < Right; Trivial.
Subtree proved!
```

Indeed, the whole proof can be done with the combination of the `Induction` tactic, which combines `Intro` and `Elim`, with good old `Auto`:

```
Coq < Restart.
1 subgoal
  ============================
   (b:bool)b=true\/b=false

Coq < Induction b; Auto.
Subtree proved!

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

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

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

Coq < Check nat_rec.
nat_rec
     : (P:nat->Set)(P O)->((n:nat)(P n)->(P (S n)))->(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 [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
     : ([_:nat]nat O)
        ->((n:nat)([_:nat]nat n)->([_:nat]nat (S n)))
           ->(n:nat)([_: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`, which β-reduces an expression to its *normal form*:

```
Coq < Eval ([_:nat]nat O)
Coq <          ->((y:nat)([_:nat]nat y)->([_:nat]nat (S y)))
Coq <             ->(n:nat)([_: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 [p:nat][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 < Compute (addition (S (S O)) (S (S (S O)))).
     = (S (S (S (S (S O)))))
     : nat
```

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

```
Coq < Fixpoint plus [n:nat] : nat -> nat :=
Coq <     [m:nat]<nat>Case n of
Coq <        (* O *) m
Coq <      (* S p *) [p:nat](S (plus p m)) end.
plus is recursively defined
```

Here the strings `(* ... *)` are just comments to help understand the cases.

Indeed in `Coq` version V5.10 an easier syntax is available to describe directly simple recursions, as follows:

```
Coq < Recursive Definition pluss : nat->nat->nat :=
Coq <     O      m => m
Coq < |   (S p) m => (S (pluss p m)).
pluss_eq1 is defined
pluss_eq2 is defined
pluss is recursively defined.
```

Warning This facility was NOT available in earlier versions of `Coq`.

The names `pluss_eq1` and `pluss_eq2` are the defining equations of `pluss`, entered as equalities. For instance:

```
Coq < Check pluss_eq1.
pluss_eq1
     : (m:nat)(pluss O m)=m
```

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.
Current goals aborted
```

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 : (n:nat)n=(plus n O).
1 subgoal
  ============================
   (n:nat)n=(plus n O)

Coq < Intro n; Elim n.
2 subgoals
  n : nat
  ============================
   O=(plus O O)
subgoal 2 is:
 (n:nat)n=(plus n O)->(S n)=(plus (S n) O)
```

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 excactly Peano's induction scheme. Pattern-matching instanciated the corresponding predicate `P` to `[n:nat]n=(plus n O)`, and we get as subgoals the corresponding instanciations of the base case `(P O)` , and of the inductive step `(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
  ============================
   O=O
subgoal 2 is:
 (n:nat)n=(plus n O)->(S n)=(plus (S n) O)

Coq < Auto.
1 subgoal
  n : nat
  ============================
   (n:nat)n=(plus n O)->(S n)=(plus (S n) O)
```

We proceed in the same way for the base step:

```
Coq < Simpl; Auto.
Subtree proved!

Coq < Save.
(Intros n;Elim n).
Simpl.
Auto.
(Simpl;Auto).
plus_n_O is defined
```

Here Auto succeded, because it used as a hint lemma eq_S, which say that successor preserves equality:

```
Coq < Check eq_S.
eq_S
     : (n:nat)(m:nat)n=m->(S n)=(S m)
```

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

```
Coq < Hint plus_n_O.
```

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

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

We now go faster, remembering that tactic 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 < Induction n; Simpl; Auto.
Subtree proved!

Coq < Save.
(Induction n;Simpl;Auto).
plus_n_S is defined

Coq < Hint plus_n_S.
```

Let us end this exercise with the commutativity of plus:

```
Coq < Lemma plus_com : (n,m:nat)(plus n m)=(plus m n).
1 subgoal
  ============================
   (n:nat)(m:nat)(plus n m)=(plus m n)
```

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

```
Coq < Induction m; Simpl; Auto.
1 subgoal
  n : nat
  m : nat
  ============================
   (n0:nat)(plus n n0)=(plus n0 n)->(plus n (S n0))=(S (plus n0 n))
```

Here Auto succeded 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.
Subtree proved!

Coq < Save.
(Induction m;Simpl;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
Coq <        := [n:nat]<Prop>Case n of (* O *)   False
Coq <                                  (* S p *)  [p:nat]True 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 : (n:nat)(Is_S (S n)).
1 subgoal
  ============================
   (n:nat)(Is_S (S n))

Coq < Simpl; Trivial.
Subtree proved!

Coq < Save.
(Simpl;Try 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 : (n:nat)~(O=(S n)).
1 subgoal
  ============================
   (n:nat)~O=(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 : O=(S n)
  ============================
   False
```

Now we use our trick:

```
Coq < Change (Is_S O).
1 subgoal
  n : nat
  H : O=(S n)
  ============================
   (Is_S O)
```

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 : O=(S n)
  ============================
   (Is_S (S n))

Coq < Simpl; Trivial.
Subtree proved!
```

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
  ============================
   (n:nat)~O=(S n)

Coq < Intro n; Discriminate.
Subtree proved!

Coq < Save.
(Intros 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 : (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
     : (n:nat)
        (P:nat->Prop)
         (P n)
          ->((m:nat)(le n m)->(P m)->(P (S m)))
             ->(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 : (n,m:nat)(le n m)->(le (S n) (S m)).
1 subgoal
  ============================
   (n:nat)(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:
 (m:nat)(le n m)->(le (S n) (S m))->(le (S n) (S (S m)))
```

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

Coq < Intros; Apply le_S; Trivial.
Subtree proved!
```

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

Coq < Hint 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 < Induction 1; Auto.
Subtree proved!

Coq < Save.
(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 : (n:nat)(le n O)->n=O.
1 subgoal
  ============================
   (n:nat)(le n O)->n=O
```

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 `CompInv` as follows:

```
Coq < Intros n H; CompInv H.
Discarding CompInv H .
Syntax error, char 374

Coq < Trivial.
```

```
1 subgoal
  ============================
   (n:nat)(le n O)->n=O

Coq < Save.
Try Trivial.
<Your Tactic Text here>
Error: Attempt to save an incomplete proof
during command Save.
```

Warning This important facility was NOT available in earlier versions of Coq. More information on inversion tactics is provided in the document "Tactics-Inversion".

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 Arith.
[Reinterning Arith ...done]
[Reinterning Le ...done]
[Reinterning Lt ...done]
[Reinterning Plus ...done]
[Reinterning Gt ...done]
[Reinterning Minus ...done]
[Reinterning Mult ...done]
[Reinterning Between ...done]
```

Such a command looks for a (compiled) module file `Arith.vi` on the current `LoadPath`. This loadpath may be changed with the commands `AddPath` and `DelPath`.

The loading of such a compiled file is quick, because the corresponding development is not typechecked again. This is a great saving compared to previous versions of our proof assistant.

If you want to recursively import modules which are required for module M, you should use `Require Export M`.

Warning Coq does not yet provides parametric modules.

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, using the command `Compile Module my_module` directly at the Coq toplevel, or else 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 my_module`.

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 `Search` 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, juste ask:

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

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.

Unité de recherche Inria Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 Villers Lès Nancy
Unité de recherche Inria Rennes, Irisa, Campus universitaire de Beaulieu, 35042 Rennes Cedex
Unité de recherche Inria Rhône-Alpes, 46 avenue Félix Viallet, 38031 Grenoble Cedex 1
Unité de recherche Inria Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105,
78153 Le Chesnay Cedex
Unité de recherche Inria Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 Sophia-Antipolis Cedex

Éditeur

Inria, Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex (France)

ISSN 0249-6399