A Tutorial on Recursive Types in Coq

Eduardo Giménez

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

A Tutorial on Recursive Types in Coq

Eduardo Giménez

No 0221

May 1998

THÈME 2

ISSN 0249-0803

A Tutorial on Recursive Types in Coq

Eduardo Giménez*

Thème 2 — Génie logiciel
et calcul symbolique
Projet COQ

Rapport technique n˚0221 — May 1998 — 42 pages

Abstract: This document is an introduction to the definition and use of recursive types in the Coq proof environment. It explains how recursive types like natural numbers and infinite streams are defined in Coq, and the kind of proof techniques that can be used to reason about them (case analysis, induction, inversion of predicates, co-induction, etc). Each technique is illustrated through an executable and self-contained Coq script.

Key-words: Proof environments, recursive types.

(Résumé : tsvp)

* Eduardo.Gimenez@inria.fr

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
Téléphone : 01 39 63 55 11 - International : +33 1 39 63 55 11
Télécopie : (33) 01 39 63 53 30 - International : +33 1 39 63 53 30

Manuel d’introduction aux types récursifs de Coq

Résumé : Ce document est une introduction à la définition et l’usage des types récursifs dans l’environnement de preuves Coq. Il explique comment des types tels que les entiers ou les listes infinies sont définis en Coq, ainsi que la classe de techniques de preuve qui peuvent être utilisées pour raisonner sur ces types (analyse par cas, récurrence, inversion de prédicats, co-induction, etc.). Chaque technique est illustrée par un script de preuve complet et exécutable.

Mots-clé : Environnements de preuve, types récursifs.

Contents

1 About this Article

This document is an introduction to the definition and use of recursive types in the Coq proof environment. It was born from the notes written for the course about the version V5.10 of Coq, held at the Ecole Normale Supérieure de Lyon in March 1996. This article is a revised and improved version of that notes for the version V6.2 of the system.

We assume that the reader has some familiarity with the proofs-as-programs paradigm of Logic [4] and the generalities of the Coq system [2]. You would take a greater advantage of this document if you first read the general tutorial about Coq [8], and take a look to Coq's reference manual [2]. If you are familiar with other proof environments based on type theory and the LCF style –like PVS, LEGO, Isabelle, etc– then you will find not difficulty to guess the unexplained details.

The better way to read this document is to start up the Coq system, type by yourself the examples and exercises proposed, and observe the behavior of the system. Along the document, the input to be typed is always written in italic verbatim font, and preceded by Coq's prompt `Coq <`. For the lack of space and readability, Coq's answer will be omitted when there is no special comment to make about it. The commands and answers appearing in this document have been automatically checked by Coq using the coq-tex tool, so you can be confident about them.

The tutorial is organised as follows. The next section describes how recursive types are defined in Coq, and introduces some useful ones, like natural numbers, the empty type, the propositional equality type, and the logic connectives. Section 3 explains definitions by pattern-matching and their connection with the principle of case analysis. This principle is the most basic elimination rule associated with recursive types, and follows a general scheme that, we illustrate for some of the types introduced in Section 2. Section 4 illustrates the pragmatics of this principle, showing different proof techniques based on it. Section 5 introduces definitions by structural recursion and proofs by induction. Finally, Section 6 is a brief introduction to co-inductive types –i.e., types containing infinite objects– and the principle of co-induction.

2 Introducing Recursive Types

Recursive types are types closed with respect to their introduction rules. These rules explain the more basic or *canonical* ways of constructing an element of that type. In this sense, they characterize the recursive type. Different rules must be considered as introducing different objects. In order to fix ideas, let us introduce in Coq the most well-known example of a recursive type: the type of natural numbers.

```
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 definition of a recursive type has two main parts. First, we establish what kind of recursive type we will characterize (a set, in this case). Second, we present the introduction rules that defines the type (*O* and *S*), also called its *constructors*. To say that *nat* is *closed* under these introduction rules means that *O* and *S* determine all the elements of this type. In other words, if n:*nat*, then n must have been introduced either by the rule *O* or by an application of the rule *S* to a previously constructed natural number. On the contrary, the type *Set* is an *open* type, since we do not know *a priori* all the possible ways of introducing an object of type set.

After entering this command, the constant *nat*, *O* and *S* are available in the current context. We can see their types using the command `Check` [2, Section 5.2.1]:

```
Coq < Check nat.
nat
     : Set

Coq < Check O.
O
     : nat

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

The other constants *nat_ind*, *nat_rec* and *nat_rect* added to the context correspond to different principles of structural induction on natural numbers that Coq infers automatically from the definition. We will come back to them in Section 5.

In fact, the type of natural numbers as well as several useful theorems about them are already defined in the basic library of Coq, so there is no need to introduce them. Therefore, let us throw away the former (re)definition of *nat*, and take a look to some other recursive types contained in this basic library of Coq.

```
Coq < Reset nat.

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

The empty type. Another example of an inductive type is the contradictory proposition. This type inhabits the universe of propositions, and have no element at all.

```
Coq < Print False.
Inductive False : Prop :=
```

Note that no constructor is given in the definition.

The singleton type. Similarly, the tautological proposition *True* is defined as a recursive type with only one element *I*:

```
Coq < Print True.
Inductive True : Prop :=  I : True
```

Relations as recursive types. Relations can be also introduced in a smart way as a recursive family of propositions. Let us take as example the order $n < m$ on natural numbers, that we called *Lth*. This relation may be introduced through the following declaration:

```
Coq < Inductive Lth [n:nat] : nat->Prop :=
Coq <   Lth_intro1: (Lth n (S n)) |
Coq <   Lth_intro2: (m:nat)(Lth n m)->(Lth n (S m)).
```

Remark the difference between the two arguments of this predicate. The first one is a *general parameter*, global to all the introduction rules,while the second one is an *index*, which is instantiated differently in the introduction rules. This declaration introduces the binary relation $n < m$ as the family of unary predicates "*to be greater than a given* n", parameterized by n.

The introduction rules of this type can be seen as a sort of Prolog rules for proving that a given integer n is less than another one. In fact, an object of type $n < m$ is nothing but a proof built up using the constructors Lth_intro_1 and Lth_intro_2 of this type. As an example, let us construct a proof that zero is less than three using Coq's proof assistant. Such an object can be obtained applying two times the second introduction rule of *Lth*, to a proof that zero is less than one, which is provided by the first constructor of `Lth`:

```
Coq < Theorem zero_less_than_three: (Lth 0 (S (S (S 0)))).
Coq < Constructor 2.
Coq < Constructor 2.
Coq < Constructor 1.
Coq < Qed.
```

We write the subgoal solved by each tactic beside it. When the current goal is a recursive type, the tactic `Constructor i` [2, Section 7.6.1] applies the i-th constructor in the definition of the type. We can take a look to the proof constructed using the command print:

```
Coq < Print zero_less_than_three.
zero_less_than_three =
(Lth_intro2 0 (S (S 0)) (Lth_intro2 0 (S 0) (Lth_intro1 0)))
     : (Lth 0 (S (S (S 0))))
```

If no name is supplied, then the tactic looks for the first constructor that solves the goal. Hence, a different proof of this lemma can be obtained iterating the tactic `Constructor` until it fails:

```
Coq < Reset zero_less_than_three.
Coq < Lemma zero_less_than_three: (Lth 0 (S (S (S 0)))).
Coq < Repeat Constructor.
Coq < Qed.
```

The propositional equality type. In Coq, the propositional equality between two elements of a set, noted $a = b$, is introduced as a family of recursive predicates *to be equal to* a parameterised by both a and its type A. This family of types has only one introduction rule, which corresponds to reflexivity.

```
Coq < Print eq.
Inductive eq [A:Set; x:A] : A->Prop :=  refl_equal : (eq A x x)
```

Two other similar equality types called *eqT* and *identityT* can be used to state the equality of two sets (notation A==B) and of two types (notation T_1===T_2), respectively. The tactic `Reflexivity` [2, Section 7.8.4] working on all these types is just a shorthand for `Constructor 1`.

Logical connectives. The conjunction and disjunction of two propositions are also examples of recursive types.

```
Coq < Print or.
Inductive or [A:Prop; B:Prop] : Prop :=
      or_introl : A->(or A B) | or_intror : B->(or A B)
Coq < Print and.
Inductive and [A:Prop; B:Prop] : Prop :=  conj : A->B->(and A B)
```

The propositions A and B are general parameters of these connectives. Choosing different universes for the general parameters A and B and for the recursive type itself gives rise to different type constructors. For example, the type *sumbool* is a disjunction but with computational contents.

```
Coq < Print sumbool.
Inductive sumbool [A:Prop; B:Prop] : Set :=
      left : A->(sumbool A B) | right : B->(sumbool A B)
```

This type –noted {A}+{B} in Coq– can be used in Coq programs as a sort of boolean type, to check whether it is A or B that is true. The objects (*left* p) and (*right* q) replace the boolean values *true* and *false*, respectively. The advantage of this type instead of *bool* is that it makes available the proofs p of : A or q of B, that could be necessary to construct a verification proof about the program (cf. the example in Section 5.2). Once the program verified, the proofs are erased by the extraction procedure.

The existential quantifier is yet another example of a logical connective introduced as a recursive type.

```
Coq < Print ex.
Inductive ex [A:Set; P:A->Prop] : Prop :=
      ex_intro : (x:A)(P x)->(ex A P)
```

The former quantifier inhabits the universe of propositions. As for conjuntion and disjunction connective, there is also another version of existential quantification inhabiting the universe *Set*, which is noted $\{x : A|(P\ x)\}$.

2.1 Mutually Dependent Declarations

Mutually dependent declarations of recursive types are also allowed in Coq. A typical example of these kind of declaration is the introduction of the trees of unbounded (but finite) width:

```
Coq < Mutual [A: Set] Inductive
Coq <   Tree   : Set :=
Coq <     node : (Forest A) -> (Tree A)
Coq < with
Coq <   Forest     : Set :=
Coq <     nochild  : (Forest A) |
Coq <     addchild : (Tree A)->(Forest A)->(Forest A).
```

Note that the parameter A is general to the whole block of declarations. Yet another example of mutually dependent types are the predicates *Even* and *Odd* on natural numbers:

```
Coq < Mutual Inductive
Coq <   Even    : nat->Prop :=
Coq <     even0 : (Even 0) |
Coq <     evenS : (n:nat)(Odd n)->(Even (S n))
Coq < with
Coq <   Odd     : nat->Prop :=
Coq <     odd1 : (Odd (S 0)) |
Coq <     oddS : (n:nat)(Even n)->(Odd (S n)).
```

3 Case Analysis and Pattern-matching

An *elimination rule* for the type A is some way to use an object $a : A$ in order to define an object in another type. From the explanation given in the previous section, it seems natural to consider *case analysis* as the natural elimination rule for a recursive type. If $n : nat$ means that n was introduced using either O or S —which are different introduction rules— then we may define an object $\langle Q\rangle \mathsf{Case}\ n\ \mathsf{of}\ g_1\ g_2\ \mathsf{end}$ in another type Q depending on which constructor was used to introduce n. A first possible typing rule for this construction is the following:

$$\frac{\begin{array}{cccc} & & [\text{case 1: } n = O] & [\text{case 2: } n = (S\ m)] \\ & & \Downarrow & \Downarrow \\ Q : Set & n : nat & g_0 : Q & g_1 : (m : nat)Q \end{array}}{\langle Q\rangle \mathsf{Case}\ n\ \mathsf{of}\ g_0\ \ g_1\ \mathsf{end} : Q}$$

If the rule applied was S, then we may also use its argument $m : nat$ in the definition. For this reason the case associated to S is represented as a function, whose argument denotes the natural number m. The computing rules associated with the definition are the expected ones:

$$\langle Q\rangle \mathsf{Case}\ O\ \mathsf{of}\ g_0\ g_1\ \mathsf{end} \Longrightarrow g_0 \qquad \langle Q\rangle \mathsf{Case}\ (S\ n)\ \mathsf{of}\ g_0\ g_1\ \mathsf{end} \Longrightarrow (g_1\ n)$$

In order to improve readability, the functions in the branches may be expressed as patterns, as in the pattern-matching construction used in functional programming languages. Furthermore, in most of the cases Coq is able to infer the type Q of the object defined, so this part of the expression can be omitted.

Example: the predecessor function. An example of a definition by case analysis is the function which computes the predecessor of a given natural number:

```
Coq < Definition pred := [n:nat]Cases n of 0 => 0 | (S m) => m end.
```

As in functional programming, tuples and wild-cards can be used in patterns [2, Section 2.2]. Such definitions are automatically compiled by Coq into an expression which may contain several nested case expressions. For example, the or function on booleans can be defined as follows:

```
Coq < Definition boolOr :=
Coq <   [b1,b2:bool]
Coq <        Cases b1 b2 of
Coq <              true  _     => true
Coq <            | _     true  => true
Coq <            | false false => false
Coq <        end.
```

and its definition is compiled by Coq into the following term:

```
Coq < Print boolOr.
boolOr =
[b1,b2:bool]
 if b1 then if b2 then true else true else if b2 then true else false
     : bool->bool->bool
```

3.1 Dependent Case Analysis

A more general typing rule for case expressions is obtained considering that not only the object defined may depend on n, but also its type. This gives rise to the following typing rule:

$$\frac{\begin{array}{cccc} & & [\text{case 1: } n = O] & [\text{case 2: } n = (S\ m)] \\ & & \Downarrow & \Downarrow \\ Q : nat \to Set & n : nat & g_0 : (Q\ n) & g_1 : (m : nat)(Q\ n) \end{array}}{\mathsf{Case}\ n\ \mathsf{of}\ g_0\ \ g_1\ \mathsf{end} : (Q\ n)}$$

Furthermore, if n is replaced by the respective right-hand sides in the type of the branches, then the informal equalities $[n = O]$ and $[n = (S\ m)]$ can be internalised into the rule:

$$\frac{Q : nat \to Set \quad n : nat \quad g_0 : (Q\ O) \quad g_1 : (m : nat)(Q\ (S\ m))}{\mathsf{Case}\ n\ \mathsf{of}\ g_0\ \ g_1\ \mathsf{end} : (Q\ n)}$$

The interest of this rule of *dependent* pattern-matching is that it can be also read as the logical principle: in order to prove that a property Q holds for all n, it is sufficient to prove that Q holds for O and that for all $m : nat$, Q holds for $(S\ m)$. The former, non-dependent version of case analysis can be obtained from this latter rule just taking Q as a constant function on n.

Example: specification of the predecessor function. Instead of introducing the predecessor function directly as a definition, we can extract it from a proof of its specification. A reasonable specification for *pred* is to say that for all n there exists another m such that either m is zero, or $(S\ m)$ is the input natural n. The function *pred* is just the way to compute such an m.

```
Coq < Theorem predspec : (n:nat){m:nat | m=0\/n=(S m)  }.

Coq < Intro n.

Coq < Case n.
2 subgoals

  n : nat
  ============================
   {m:nat | (m=0\/0=(S m))}
subgoal 2 is:
 (n:nat){m:nat | (m=0\/(S n)=(S m))}

Coq < (* n = 0     *) Exists 0; Left; Reflexivity.

Coq < (* n = (S m) *) Intro  m; Exists m ; Right; Reflexivity.
```

The tactic `Case t` [2, Section 7.7.2] allows to define a proof by case analysis on the term t. The latter tactic `Destruct x` [2, Section 7.7.2] can provide an even more direct proof. This tactic introduces all the universally quantified variables of the goal until x, and then applies the `Case` tactic to this one.

```
Coq < Restart.

Coq < Destruct n.

Coq <      Exists 0; Left; Reflexivity.

Coq <      Intro  m; Exists m; Right; Reflexivity.

Coq < Defined.
```

The command `Extraction` [2, Section 5.2.3] can be used to see the computational contents associated to the proof of the theorem *predspec*:

```
Coq < Extraction predspec.
predspec ==> [n:nat]Cases n of
                      0 => 0
                    | (S m) => m
                    end
 :  nat->nat
```

Exercise 3.1 *Prove the following theorem:*

```
Theorem identity : (x:nat)x=(Cases x of 0 => 0 | (S m) => (S m) end).
```

3.2 Some Examples of Case Analysis

Case analysis is then the most basic elimination rule that Coq provides for recursive types. This rule follows a general schema, valid for any positive recursive type R. First, if R has type $(z_1 : A_1)\ldots(z_p : A_r)S$, with S either *Set*, *Prop* or *Type*, then a case expression on $p : (R\ a_1 \ldots a_r)$ inhabits $(Q\ a_1 \ldots a_r\ p)$. The types of the branches of the case expression are obtained form the definition of the type in this way: if the type of the i-th constructor of R is $(x_1 : T_1)\ldots(x_n : T_n)(R\ q_1 \ldots q_m)$, then the type of the $i-th$ branch is $(x_1 : T_1)\ldots(x_n : T_n)(Q\ q_1 \ldots q_m)$ for non-dependent case analysis, and $(x_1 : T_1)\ldots(x_n : T_n)(Q\ q_1 \ldots q_m, (c_i\ x_1\ \ldots x_n))$ for dependent one. In the following section, we illustrate this general scheme for different recursive types.

3.2.1 The Empty Type

In a definition by case analysis, there is one branch for each introduction rule of the type. Hence, in a definition by case analysis on p : *False* there are no cases to be considered. In other words, the rule of (non-dependent) case analysis for the type *False* is:

$$\frac{Q : \mathit{Prop} \qquad p : \mathit{False}}{\mathsf{Case}\ p\ \mathsf{of}\ \mathsf{end} : Q}$$

As a corollary, if we could construct an object in *False*, then it could be possible to define an object in any type. The tactic `Contradiction` [2, Section 7.4.2] corresponds to the application of the elimination rule above. It searches in the context for an absurd hypothesis (this is, a hypothesis whose type is *False*) and then proves the goal by a case analysis of it.

```
Coq < Theorem fromFalse : False -> 0=(S 0).

Coq < Intro absurd. Contradiction.

Coq < Qed.
```

The tactic `Absurd P` also proves any goal by elimination on the proposition *False*, provided that P and $\neg P$ can be derived from the context. In Coq, the proposition $\neg P$ is just an abbreviation for $(P \rightarrow \mathit{False})$. So, if it is possible to find $f : \neg P$ and $a : P$ then $(f\ a) : \mathit{False}$, and the goal can be proved by case analysis of this object.

```
Coq < Theorem  nosense : (Q:Prop)~0=0->Q.

Coq < Intros Q H.
```

```
Coq < Absurd 0=0.
2 subgoals

  Q : Prop
  H : ~0=0
  ============================
   ~0=0
subgoal 2 is:
 0=0

Coq <  (* Goal  ~0=0 *)  Assumption.

Coq <  (* Goal   0=0 *)  Reflexivity.

Coq < Qed.
```

3.2.2 The Equality Type

Following the general scheme mentioned above, the rule for defining a proof of C by non-dependent case analysis of $p : a = b$ is:

$$\frac{A : Set \quad a : A \qquad b : A \quad Q : A \to Prop \quad p : a \stackrel{A}{=} b \quad g : (Q\ a)}{\mathsf{Case}\ p\ \mathsf{of}\ g\ \mathsf{end} : (Q\ b)}$$

Therefore, doing case analysis on a proof of the equality $a = b$ amounts to replace all the occurrences of the term b with the term a in the goal to be proven. Let us illustrate this through an example: the transitivity property of this equality.

```
Coq < Theorem trans : (n,m,p:nat)n=m->m=p->n=p.

Coq < Intros n m p eqnm.

Coq < Case eqnm.
1 subgoal

  n : nat
  m : nat
  p : nat
  eqnm : n=m
  ============================
   n=p->n=p

Coq < Trivial.
```

Exercise 3.2 *Prove the symmetry property of propositional equality.*

Instead of using `Case`, we can use the tactic `Rewrite` H [2, Section 7.8.1]. If H : a = b, then this tactic performs a case analysis on a proof obtained by applying a symmetry theorem to H. The application of symmetry allows to rewrite the equality from left to right, which looks more natural. An optional parameter (either `->` or `<-`) can be used to precise in which sense the equality must be rewritten. By default, `Rewrite` H corresponds to `Rewrite -> H`

```
Coq < Undo 2.
1 subgoal

  n : nat
  m : nat
  p : nat
  eqnm : n=m
  ============================
   m=p->n=p

Coq < Rewrite eqnm.
1 subgoal

  n : nat
  m : nat
  p : nat
  eqnm : n=m
  ============================
   m=p->m=p

Coq < Trivial.

Coq < Qed.
```

If H : a = b, then the tactic `Rewrite` replaces *all* the occurrences of a by b. However, in certain situations we could be interested in rewriting some of the occurrences, but not all of them. This can be done using the tactic `Pattern` [2, Section 7.5.8]. Let us consider yet another example to illustrate this.

```
Coq < Theorem lt_if_circular : (n:nat)n=(S n)->(lt n n).

Coq < Intros n eqnSn.
```

A direct proof of this theorem can be obtained by first rewriting the second occurrence of n in the conclusion, and then applying the first introduction rule of *Lth.* However, in

order to be able to apply this introduction rule, the first occurrence must remain unchanged. Applying the tactic `Pattern 2 n` before rewriting explicitly abstracts the second occurrence of `n` from the goal, pointing to `Rewrite` the particular predicate on n that we search to prove.

```
Coq < Pattern 2 n.
1 subgoal

  n : nat
  eqnSn : n=(S n)
  ============================
   ([n0:nat](lt n n0) n)

Coq < Rewrite eqnSn.
1 subgoal

  n : nat
  eqnSn : n=(S n)
  ============================
   (lt n (S n))

Coq < Constructor.

Coq < Qed.
```

3.2.3 The Predicate $n < m$

The last instance of the elimination schema that we will illustrate is the rule of case analysis for the predicate $n <$m:

$$\frac{\begin{array}{c} n : \mathit{nat} \\ Q : \mathit{nat} \to \mathit{Prop} \quad p : (\mathit{Lth}\ n\ m) \quad g_0 : (Q\ (S\ n)) \quad g_1 : (m : \mathit{nat})(\mathit{Lth}\ n\ m) \to (Q\ (S\ m)) \end{array}}{\mathsf{Case}\ p\ \mathsf{of}\ g_0\ \ g_1\ \mathsf{end} : (Q\ m)}$$

Notice that the choice of introducing some of the arguments of the predicate as being general parameters in its definition has consequences on the rule of case analysis that is derived. In particular, the type Q of the object defined by the case expression only depends on the indexes of the predicate, and not on the general parameters. In the definition of the predicate *Lth*, the first argument of this relation is a general parameter of the definition. Hence, the predicate Q to be proven only depends on the second argument of the relation. In other words, the integer n is also a general parameter of the rule of case analysis.

An example of an application of this rule is the following theorem, showing that any integer greater than zero is the successor of another integer:

```
Coq < Theorem notZero : (n:nat)(Lth 0 n)->(EX p:nat | n=(S p)).
Coq < Intros n H.
Coq < Case H.
Coq <    Exists 0; Reflexivity.
Coq <    Intros m H1; Exists m; Reflexivity.
Coq < Qed.
```

3.3 Case Analysis and Logical Paradoxes

In the previous section we have illustrated the general scheme for generating the rule of case analysis associated to some recursive type from the definition of the type. However, if the logical soundness is to be preserved, certain restrictions to this schema are necessary. This section provides a briefly explanation of these restrictions.

3.3.1 The Positiveness Condition

In order to make sense of recursive types as types closed under their introduction rules, a constraint has to be imposed on the possible forms of such rules. This constraint, known as the *positiveness condition*, is necessary to prevent the user from naively introducing some recursive types which would open the door to logical paradoxes. An example of such a dangerous type is the "recursive type" Λ, whose only constructor is $\lambda : (\Lambda \to \mathit{False}) \to \Lambda$. Following the pattern given in Section 3.2, the rule of (non dependent) case analysis for Λ would be the following:

$$\frac{Q : \mathit{Prop} \qquad p : \Lambda \qquad g : (\Lambda \to \mathit{False}) \to Q :}{\mathsf{Case}\ p\ \mathsf{of}\ g\ \mathsf{end} : Q}$$

In order to explain why this rule leads to a paradox, let us assume a set Λ, a function λ standing for its constructor, and another function $Case_\Lambda$ standing for the rule of case analysis on Λ.

```
Coq < Section Paradox.
Coq < Variable Lambda : Set.
Coq < Variable lambda : (Lambda->False)->Lambda.
Coq < Variable CaseL  : (Q:Prop)Lambda->((Lambda->False)->Q)->Q.
```

From this constants, it is possible to define application by case analysis. Then, through auto-application, the well-known looping term $(\lambda x.(x\ x)\ \lambda x.(x\ x))$ provides a proof of falsehood.

```
Coq < Definition app  : Lambda->Lambda->False :=
Coq <             [f,x:Lambda](CaseL False f [h:Lambda->False](h x)).

Coq < Definition Delta : Lambda :=  (lambda [x:Lambda](app x x)).

Coq < Definition loop  : False  :=  (app Delta Delta).

Coq < End Paradox.
```

This example can be seen as a formulation of Russel's paradox in type theory: just read $(app\ x\ x)$ as $x \notin x$, and the constructor $(\lambda\ [x : \Lambda](P\ x))$ as the definition by set comprehension $\{x \mid (P\ x)\}$. If $\Lambda Case$ would satisfy the reduction rule associated to case analysis, that is, $(\Lambda Case\ Q\ (\lambda\ f)\ h) \Longrightarrow (h\ f)$, then such a term would compute into itself. This is not actually surprising, since the proof of the logical soundness of Coq strongly lays on the property that any well-typed term must terminate. Hence, non-termination is usually a synonymous of inconsistency.

In this case, the construction of a non-terminating program comes from the so-called *negative occurrence* of Λ in the argument of the constructor λ. In order to be admissible for Coq, the type R must be positive in the types of the arguments of its own introduction rules, in the sense on the following definition:

1. R is positive in T if R does not occur in T;
2. R is positive in $(R\ \vec{t})$ if R does not occur in $\vec{t}$;
3. R is positive in $(x : A)C$ if it does not occur in A and R is positive in C;
4. R is positive in $(J\ \vec{t})$, if J is a recursive type, and for any term t_i either :
   (a) R does not occur in t_i, or
   (b) R is positive in t_i, t_i instantiates a general parameter of J, and this parameter is positive in the arguments of the constructors of J.

When we can show that R is positive without using the item (4) of the definition above, then we say that R is *strictly positive*.

Note that the positiveness condition does not avoid functional recursive arguments in the constructors. The type of ordinal numbers is an example of a recursive type with a functional recursive argument:

```
Coq < Inductive Ord:Set :=
Coq <   Ord0  : Ord |
Coq <   OrdS  : Ord->Ord |
Coq <   Limit : (nat->Ord)->Ord.
```

In this representation, a limit ordinal $(Limit\ h)$ is a sort of tree with an infinite width, whose nth child is obtained applying the function h to n.

An example of a positive type that is not strictly positive is the inclusion relation for ordinal numbers, defined as an inductive family of types. In the case of limit ordinals, it is necessary to compare each of their children without taking care of the order in which they appear. This restriction leads to the following definition:

```
Coq < Inductive Olt : Ord->Ord->Prop :=
Coq <    Olt0 : (o:Ord)(Olt Ord0 (OrdS o)) |
Coq <    OltS : (o1,o2:Ord)(Olt o1 o2)->(Olt (OrdS o1) (OrdS o2)) |
Coq <    OltL : (f,g:nat->Ord)
Coq <             ((n:nat)(EX m:nat | (Olt (f n) (g m))))->
Coq <               (Olt (Limit f) (Limit g)).
```

The occurrences of *Olt* in the premises of the rule *OltL* are positive in the large sense, because it appears instantiating a general parameter of the type *EX* (cf. Section 2). An alternative, strictly positive definition of this predicate can be obtained skolemising the proposition $\forall n.\exists m.(Olt\ (f\ n)\ (g\ n))$ in the type of *OltL*.

```
Coq < Reset Olt.
```

```
Coq < Inductive Olt : Ord->Ord->Prop :=
Coq <    Olt0 : (o:Ord)(Olt Ord0 (OrdS o)) |
Coq <    OltS : (o1,o2:Ord)(Olt o1 o2)->(Olt (OrdS o1) (OrdS o2)) |
Coq <    OltL : (f,g:nat->Ord)(h:nat->nat)
Coq <                ((n:nat)(Olt (f n) (g (h n))))->
Coq <                   (Olt (Limit f) (Limit g)).
```

In general, strictly positive definitions are preferable to only positive ones. The reason is that it is sometimes difficult to derive structural induction combinators for the latter ones. Such combinators are automatically generated for strictly positive types, but not for the only positive ones. Nevertheless, sometimes non-strictly positive definitions provide a smarter or shorter way of declaring a recursive type.

Another way of transforming a positive definition into a strictly positive one is using mutually dependent declarations. For example, the type of trees of unbounded width can be introduced as a positive type using the type of polymorphic lists:

```
Coq < Require PolyList.

Coq < Inductive Tree   [A:Set] : Set :=
Coq <           node   : A-> (list (Tree A)) -> (Tree A).
```

This declaration can be transformed into a strictly positive one adding an extra type to the definition, as was done in Section 2.1.

3.3.2 Impredicative Recursive Types

A recursive type R inhabiting a universe S is *predicative* if the introduction rules of R do not make a universal quantification on a universe containing S. All the recursive types previously introduced are examples of predicative types. An example of an impredicative one is the type *exT*, the dependent product of a certain set (or proposition) X, and a proof of a property P about X.

```
Coq < Print exT.
Inductive exT [A:Type; P:A->Prop] : Prop :=
      exT_intro : (x:A)(P x)->(exT A P)
```

This type is useful for expressing existential quantification over types, like "there exists a proposition A such that (P A)" —written $(EXT\ A : \mathtt{Prop} \mid (\mathtt{P}\ A))$ in Coq. However, note that the constructor of this type can be used to inject any proposition –even itself!– into the type. A careless use of such a self-contained objects may to a variant of Burali-Forti's paradox. The construction of Burali-Forti's paradox is more complicated than Russel's one, so we will not describe it here, and point the reader interested in it to [3, 1].

One possible way of avoiding this new source of paradoxes is to restrict the kind of eliminations by case analysis that can be done on impredicative types. In particular, projections on those universes equal or bigger than the one inhabited by the impredicative type must be forbidden [3]. A consequence of this restriction is that it is not possible to define the first projection of the type $(EXT\ A : Prop \mid (\mathtt{P}\ A))$.

3.3.3 Extraction Constraints

There is a final constraint on case analysis that is not motivated by the potential introduction of paradoxes, but for compatibility reasons with Coq's extraction mechanism [2, Appendix 15]. This mechanism is based on the classification of basic types into the universe *Set* of sets and the universe *Prop* of propositions. The objects of a type in the universe *Set* are considered as relevant for computation purposes. The objects of a type in *Prop* are considered just as formalised comments, not necessary for execution. The extraction mechanism

consists in erasing such formal comments in order to obtain an executable program. Hence, in general, it is not possible to define an object in a set (that should be kept by the extraction mechanism) by case analysis of a proof (which will be through away).

Nevertheless, this general rule has an exception which is important in practice: if the definition proceeds by case analysis on a proof of a *singleton proposition*, then it is allowed. A singleton proposition is a non-recursive proposition with a single constructor c, all whose arguments are proofs. For example, the propositional equality and the conjunction of two propositions are examples of singleton propositions.

3.3.4 Strong Case Analysis on Proofs

In plain Coq, it is possible to define a proposition Q by case analysis on the proofs of another recursive proposition R. As we will see in Section 4.1, this enables to prove that different introduction rules of R construct different objects. However, this property is in contradiction with the principle of excluded middle of classical logic, because this principle entails that the proofs of a proposition cannot be distinguished. This principle is not provable in Coq, but it is frequently introduced by the users as an axiom, for reasoning in classical logic. For this reason, the definition of propositions by case analysis on proofs is currently not allowed in Coq.

3.3.5 Summary of Constraints

To end with this section, the following table summarizes which universe U_1 may inhabit an object of type Q defined by case analysis on $x : R$, depending on the universe U_2 inhabited by the recursive type R.

	$Q : U_1$			
$n : R : U_2$		*Set*	*Prop*	*Type*
	Set	yes	yes	if R predicative
	Prop	if R singleton	yes	no
	Type	yes	yes	yes

4 Some Proof Techniques Based on Case Analysis

In this section we illustrate the use of case analysis as a proof principle, explaining the proof technique behind three very useful Coq tactics, called `Discriminate`, `Inject` and `Inversion`.

4.1 Discrimination of introduction rules

In the informal semantics of recursive types described in Section 2 it was said that each of the introduction rules of a recursive type is considered as being different from all the others. It is possible to capture this fact inside the logical system using the propositional equality. We take as example the following theorem, stating that *O* constructs a natural number different from any of those constructed with *S*.

```
Coq < Theorem S_is_not_O : (n:nat)~((S n)=O).
```

In order to prove this theorem, we first define a proposition by case analysis on natural numbers, so that the proposition is true for *O* and false for any natural number constructed with *S*. This uses the empty and singleton type introduced in Sections 2.

```
Coq < Definition Is_zero := [x:nat]Cases x of O => True | _ => False end.
```

Then, we prove the following lemma:

```
Coq < Lemma  disc : (m:nat)m=O->(Is_zero m).

Coq < Intros m eqmO; Rewrite  eqmO.
1 subgoal

  m : nat
  eqmO : m=O
  ============================
   (Is_zero O)

Coq < Simpl.
1 subgoal

  m : nat
  eqmO : m=O
  ============================
   True

Coq < Constructor.

Coq < Qed.
```

Finally, the proof of *O_is_not_S* follows by the application of the previous lemma to *O*.

```
Coq < Red.
Coq < Intros n eqSn0.
Coq < Apply disc with m:=(S n).
Coq < Assumption.
Coq < Qed.
```

`Discriminate` [2, Section 7.9.3] is a special-purpose tactic for proving disequalities between two elements of a recursive type introduced by different constructors. It generalizes the proof method described here for natural numbers to any recursive type. This tactic is also capable of proving disequalities where the difference is not in the constructors at the head of the terms, but deeper inside them. For example, it can be used to prove the following theorem:

```
Coq < Theorem disc2 : (n:nat)~((S (S n))=(S 0)).
Coq < Intro n.
Coq < Discriminate.
Coq < Qed.
```

When there is an assumption H in the context stating a false equality $t_1 = t_2$, `Discriminate` solves the goal by first proving $(t_1 \neq t_2)$ and then reasoning by absurdity with respect to H:

```
Coq < Theorem disc3 : (Q:Prop)(n:nat)((S (S n))=(S 0))->Q.
Coq < Intros Q n false_eq.
Coq < Discriminate.
Coq < Qed.
```

In this case, the proof proceeds by absurdity with respect to the false equality assumed, whose negation is proved by discrimination.

4.2 Injectiveness of introduction rules

Another useful property about recursive types is the injectiveness of introduction rules, this is, that whenever two objects were built using the same introduction rule, then this rule should have been applied to the same element. This can be stated formally using the propositional equality:

```
Coq < Theorem inj : (n,m:nat)(S n)=(S m)->n=m.
```

This theorem is just a corollary of a lemma about the predecessor function:

```
Coq < Lemma injpred : (n,m:nat)n=m->(pred n)=(pred m).
Coq < Intros n m eqnm.
Coq < Rewrite    eqnm.
Coq < Reflexivity.
Coq < Qed.
```

Once this lemma is proven, the theorem follows directly from it:

```
Coq < Intros n m eqSnSm.
Coq < Apply injpred with n:=(S n) m:=(S m);Assumption.
Coq < Qed.
```

This proof method is implemented by the tactic `Injection` [2, Section 7.9.4]. This tactic can be used to prove an equality constraint $x = t$ about a variable x from a hypothesis involving x. Similarly to the case of `Discriminate`, the tactic can be also applied if x does not occur in a direct sub-term, but somewhere deeper inside it. Its application may leave some trivial goals that can be easily solved using the tactic `Trivial`.

```
Coq < Theorem inj : (n,m:nat)(S (S n))=(S (S m))->n=m.
Coq < Intros n m eqSSnSSm.

Coq < Injection eqSSnSSm.
1 subgoal

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

Coq < Trivial.
Coq < Abort.
```

4.3 Inversion Techniques

In section 3.1, we motivated the rule of dependent case analysis as a way of internalizing the informal equalities $[n = O]$ and $[n = (S\ m)]$ associated to each case. This internalisation consisted in instantiating n with the corresponding term in the type of each branch. However, sometimes it could be better to internalise these equalities as extra hypotheses –for example, in order to use the tactics `Rewrite`, `Discriminate` or `Injection` presented in the previous sections. This is frequently the case were the element analysed is denoted by a term which is not a variable, or when it is an object of a particular instance of a recursive family of types. Consider for example the following theorem:

```
Coq < Theorem no_lt_than_zero : (n:nat)~(Lth n 0).
```

Intuitively, this theorem should follow by case analysis on the hypothesis $H : (Lth\ n\ O)$, because no introduction rule allows to instantiate the second argument of *Lth* with zero. However, there is no way of capturing this with the rule of case analysis presented in section 2, because it does not take into account what particular instance of the family the type of H is. Let us try it:

```
Coq < Red.
Coq < Intros n H.

Coq < Case H.
2 subgoals

  n : nat
  H : (Lth n 0)
  ============================
   False
subgoal 2 is:
 (m:nat)(Lth n m)->False
```

What is necessary here is to make available the equalities $[O = (S\ n))]$ and $[O = (S\ m)]$ as extra hypotheses of the branches, so that the goal can be solved using the `Discriminate` tactic. In order to obtain the desired equalities as hypotheses, let us prove a more general lemma, that our theorem is a corollary of:

```
Coq < Undo.
Coq < Lemma  no_lt_than_m : (m,n:nat)(Lth n m)->(m=0)->False.
Coq < Intros n m H.
```

```
Coq < Case H.
2 subgoals

  n : nat
  m : nat
  H : (Lth m n)
  ============================
   (S m)=0->False
subgoal 2 is:
 (m0:nat)(Lth m m0)->(S m0)=0->False

Coq < Intros;Discriminate.

Coq < Intros;Discriminate.

Coq < Qed.
```

The theorem can be now solved by an application of this lemma:

```
Coq < Show.
1 subgoal

  n : nat
  H : (Lth n 0)
  ============================
   False

Coq < Apply no_lt_than_m with m:=0 n:=n.
2 subgoals

  n : nat
  H : (Lth n 0)
  ============================
   (Lth n 0)
subgoal 2 is:
 0=0

Coq < Assumption.

Coq < Reflexivity.

Coq < Abort.
```

The general method to address such situations consists in changing the goal to be proven into an implication, introducing as preconditions the equalities needed to eliminate those

cases that make no sense. This proof technique is implemented by the tactic `Inversion` [2, Section 7.10]. In order to prove a goal $(G\ \vec{q})$ from an object of type $(R\ \vec{t})$, this tactic automatically generates a lemma $\forall C.\forall \vec{x}.(R\ \vec{x}) \rightarrow \vec{x} = \vec{t} \rightarrow \vec{B} \rightarrow (G\ \vec{q})$, where the list of propositions $\vec{B}$ correspond to those sub-goals that cannot be directly proven using `Discriminate`. This lemma can be either stocked for further use, or generated interactively. In this latter case, the subgoals yield by the tactic are the hypotheses $\vec{B}$ of the lemma. If the lemma has been stocked, then the tactic `Use Inversion` can be used to apply it. Let us show both approaches on the previous example:

Interactive mode.

```
Coq < Theorem no_lt_than_zero : (n:nat)~(Lth n 0).
Coq < Red.
Coq < Intros n H.
Coq < Inversion H.
Coq < Qed.
```

Static mode.

```
Coq < Reset no_lt_than_zero.
Coq < Derive Inversion inversion_lemma with (n:nat)(Lth n 0).
Coq < Theorem no_lt_than_zero : (n:nat)~(Lth n 0).
Coq < Red.
Coq < Intros n H.
Coq < Inversion H using inversion_lemma.
Coq < Qed.
```

In the example above, all the cases are solved using discriminate, so it remains no subgoal to be proven (i.e. the list $\vec{B}$ is empty). Let us present a second example, where this list is not empty:

```
Coq < Theorem reverse_rules : (n,m:nat)(Lth n (S m))->n=m\/(Lth n m).
Coq < Intros n m H.
```

```
Coq < Inversion H.
2 subgoals

  n : nat
  m : nat
  H : (Lth n (S m))
  H1 : n=m
  ============================
   m=m\/(Lth m m)
subgoal 2 is:
 n=m\/(Lth n m)

Coq < Left;Reflexivity.
1 subgoal

  n : nat
  m : nat
  H : (Lth n (S m))
  m0 : nat
  H0 : m0=m
  H1 : (Lth n m)
  ============================
   n=m\/(Lth n m)

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

This example shows how this tactic can be used to "reverse" the introduction rules of a recursive type, deriving the possible premises that could lead to prove a given instance of the predicate. This is why these tactics are called `Inversion` tactics: they go back from conclusions to premises.

The hypothesis corresponding to the propositional equalities are not needed in this example, since the tactic does the rewriting necessary to solve the subgoals. When the equalities are no longer needed after the inversion, it is better to use the tactic `Inversion_clear`. This variant of the tactic clears from the context all the equalities introduced.

```
Coq < Restart.

Coq < Intros n m H.
```

```
Coq < Inversion_clear H.
2 subgoals

  n : nat
  m : nat
  ============================
   m=m\/(Lth m m)
subgoal 2 is:
 n=m\/(Lth n m)
Coq < Left;Reflexivity.
1 subgoal

  n : nat
  m : nat
  H0 : (Lth n m)
  ============================
   n=m\/(Lth n m)

Coq < Right;Assumption.
Coq < Abort.
```

Exercise 4.1 *Consider the following language of arithmetic expression, and its operational semantics:*

```
Coq < Inductive ArithExp : Set :=
Coq <      Zero : ArithExp
Coq <    | Succ : ArithExp -> ArithExp
Coq <    | Plus : ArithExp -> ArithExp -> ArithExp.
Coq < Inductive RewriteRel : ArithExp -> ArithExp -> Prop :=
Coq <      RewSucc  : (e1,e2:ArithExp)
Coq <                   (RewriteRel e1 e2)->(RewriteRel (Succ e1) (Succ e2))
Coq <   |  RewPlusZ : (e:ArithExp)
Coq <                   (RewriteRel (Plus Zero e) e)
Coq <   |  RewPlusS : (e1,e2:ArithExp)
Coq <                   (RewriteRel e1 e2)->
Coq <                     (RewriteRel (Plus (Succ e1) e2) (Succ (Plus e1 e2)))
Coq <   |  RewComm  :  (e1,e2,e3:ArithExp)
Coq <                    (RewriteRel (Plus e1 e2) (Plus e2 e1)).
```

1. *Prove that* `Zero` *cannot be computed any further.*
2. *Prove that an expression* (`Succ` e) *is always computed into an expression having the constructor* `Succ` *at the head.*

5 Inductive Types and Structural Induction

All the examples we have presented upon here correspond to a special class of recursive types called *inductive* types. Inductive types are those recursive types whose elements are well-founded with respect to the structural order induced by the constructors of the type. In addition to case analysis, this extra hypothesis about well-foundness justifies a stronger elimination rule for them, called *structural induction*. This form of elimination consists in defining a value $(f\ x)$ from some element x of the inductive type I, assuming that values have been already associated in the same way to the sub-parts of x of type I.

Definitions by structural induction are expressed through the `FixPoint` command [2, Section 1.3.4]. This command is quite close to the `let-rec` construction of functional programming languages. For example, the following definition introduces the addition of two natural numbers:

```
Coq < Fixpoint add [n:nat] : nat->nat :=
Coq <   [x:nat]Cases n of
Coq <                  O     => x
Coq <                | (S m) => (S (add m x))
Coq <             end.
```

The definition is by structural induction on the first argument of the function. This is indicated enclosing n into square brackets. In order to be accepted, the definition must satisfy a syntactical condition, called the *guardedness condition*. Roughly speaking, this condition constrains the arguments of a recursive call to be pattern variables, issued from a case analysis of the formal argument of the function enclosed into brackets. In the case of the function *add*, the argument m in the recursive call is a pattern variable issued from a case analysis of n. Therefore, the definition is accepted.

If a list of arguments appear enclosed into the square brackets, then the last argument of the list is assumed to be the decreasing one. For example, addition can be also defined by induction on the second argument, like this:

```
Coq < Reset add.
```

```
Coq < Fixpoint add [x:nat;n:nat] : nat :=
Coq <    Cases n of
Coq <          O    => x
Coq <       | (S m) => (S (add x m))
Coq <             end.
```

The guardedness condition must be satisfied only by the last argument of the enclosed list. For example, the following declaration is an alternative way of defining addition:

```
Coq < Reset add.

Coq < Fixpoint add [n:nat] : nat->nat :=
Coq <   Cases n of
Coq <         0    => [x:nat]x
Coq <     | (S m) => [x:nat](add m (S x))
Coq <   end.
```

In this definition, the second argument of *add* grows at each recursive call. However, as the first one always decreases, the definition is sound. Moreover, the argument in the recursive call could be a deeper component of n. This is the case in the following definition of a boolean function determining whether a number is even or odd:

```
Coq < Fixpoint even [n:nat] : bool :=
Coq <   Cases n of
Coq <         0         => true
Coq <       | (S 0)     => false
Coq <       | (S (S m)) => (even m)
Coq <   end.
```

Mutually dependent definitions by structural induction are also allowed. For example, the previous function *even* could be alternatively defined using an auxiliary function *odd*:

```
Coq < Reset even.

Coq < Fixpoint
Coq < even [n:nat] : bool :=
Coq <   Cases n of
Coq <         0     => true
Coq <       | (S m) => (odd m)
Coq <   end
Coq < with
Coq < odd [n:nat] : bool :=
Coq <   Cases n of
Coq <         0     => false
Coq <       | (S m) => (even m)
Coq <   end.
```

Definitions by structural induction are computed lazily, i.e. they are expanded only when they are applied, and the decreasing argument is a term having a constructor at the head. We can check this using the `Eval` command, which computes the normal form of a well typed term.

```
Coq < Eval Simpl in even.
     = even
     : nat->bool

Coq < Eval Simpl in [x:nat](even x).
     = [x:nat](even x)
     : nat->bool

Coq < Eval Simpl in (even O).
     = true
     : bool

Coq < Eval Simpl in (even (S O)).
     = false
     : bool
```

5.1 Proofs by Structural Induction

The principle of structural induction can be also used in order to define proofs, that is, to prove theorems. Let us call an *elimination combinator* any function that, given a predicate `P`, defines a proof of `(P x)` by structural induction on `x`. In Coq, the principle of proof by induction on natural numbers is a particular case of an elimination combinator. The definition of this combinator depends on three general parameters: the predicate to be proven, the base case, and inductive hypothesis:

```
Coq < Section Principle_of_Induction.

Coq < Variable    P               : nat->Prop.

Coq < Hypothesis  base_case       : (P O).

Coq < Hypothesis  inductive_hyp   : (n:nat)(P n)->(P (S n)).

Coq < Fixpoint natind  [n:nat]    : (P n) :=
Coq <    <P>Cases n of
Coq <            O     => base_case
Coq <          | (S m) => (inductive_hyp m (natind m))
Coq <    end.

Coq < End Principle_of_Induction.
```

As this proof principle is very used, Coq automatically generates it when an inductive type is introduced. Similar principles `nat_rec` and `nat_rect` for defining objects in the universes *Set* and *Type* are also automatically generated. The command `Scheme` [2, Section 7.15] can be used to generate an elimination combinators from certain parameters, like the

universe that must inhabit the object defined, whether the case analysis in the definitions must be dependent or not, etc. For example, it can be used to generate an elimination combinator for reasoning on even natural numbers from the mutually dependent predicates introduced in page 6. We do not display the combinators here by lack of space, but you can see them using the `Print` command.

```
Coq < Scheme Even_induction := Minimality for Even Sort Prop
Coq < with   Odd_induction  := Minimality for Odd  Sort Prop.
```

Another example of an elimination combinator is the principle of double induction on nataural numbers, introduced by the following definition:

```
Coq < Section Principle_of_Double_Induction.

Coq < Variable    P               : nat->nat->Prop.

Coq < Hypothesis  base_case1      : (x:nat)(P O x).

Coq < Hypothesis  base_case2      : (x:nat)(P x O).

Coq < Hypothesis  inductive_hyp   : (n,m:nat)(P n m)->(P (S n) (S m)).

Coq < Fixpoint nat_double_ind [n:nat] : (m:nat)(P n m) :=
Coq <  [m:nat]
Coq <   <P>Cases n m of
Coq <              O      x    => (base_case1 x)
Coq <            | x      O    => (base_case2 x)
Coq <            | (S x) (S y) => (inductive_hyp x y (nat_double_ind x y))
Coq <      end.

Coq < End Principle_of_Double_Induction.
```

Changing the type of `P` into $nat \rightarrow nat \rightarrow Set$, another combinator *nat_double_rec* for constructing proofs with computational contents can be defined in exactly the same way.

Exercise 5.1 *Define the combinator nat_double_rec, for proving theorems with computational contents by double induction.*

Using Elimination Combinators. The tactic `Apply` can be used to apply one of these proof principles during the development of a proof. Consider for example the proposition $\forall n.n \neq (S\ n)$.

```
Coq < Lemma not_circular : (n:nat)~n=(S n).
Coq < Intro n.
Coq < Apply nat_ind with P := [n:nat]~n=(S n).
Coq <  Discriminate.
Coq <  Red;Intros m hypind eqmSm;Apply hypind;Injection eqmSm;Trivial.
```

The tactic `Elim` [2, Section 7.7.1] is a refinement of `Apply`, specially conceived for the application of elimination combinators. If `t` is an object of an inductive type `I`, then `Elim t` tries to find an abstraction `P` of the current goal `G` such that $(\texttt{P}\ \texttt{t}) \equiv \texttt{G}$. Then it solves the goal applying $(\texttt{I\_ind}\ \texttt{P})$, where `I_ind` is the combinator associated to `I`. The different cases of the induction appears then as sub-goals that remain to be solved.

```
Coq < Restart.
Coq < Intro n.
Coq < Elim n.
Coq <  Discriminate.
Coq <  Red;Intros m hypind eqmSm;Apply hypind;Injection eqmSm;Trivial.
Coq < Qed.
```

The option `Elim t using C` allows to use a derived combinator `C` instead of the default one. Consider the following theorem, stating that equality is decidable on natural numbers:

```
Coq < Lemma iseq : (x,y:nat){x=y}+{~x=y}.
Coq < Intros x y.
```

Let us prove this theorem using the combinator `nat_double_rec` of exercise 5.1. The example also illustrates how `Elim` may sometimes fail in finding a suitable abstraction `P` of the goal. Note that if `Elim x` is used directly on the goal, the result is not the expected one.

```
Coq < Elim x using nat_double_rec.
4 subgoals

  x : nat
  y : nat
  ============================
   (x:nat){x=y}+{~x=y}
```

```
subgoal 2 is:
 nat->{0=y}+{~0=y}
subgoal 3 is:
 nat->(m:nat){m=y}+{~m=y}->{(S m)=y}+{~(S m)=y}
subgoal 4 is:
 nat
```

The four sub-goals obtained do not correspond to the premises that would be expected for the principle `nat_double_rec`. The problem comes from the fact that this principle for eliminating χ has a universally quantified formula as conclusion, which confuses `Elim` about the right way of abstracting the goal. Therefore, in this case the abstraction must be explicited using the tactic `Pattern`. Once the right abstraction is provided, the rest of the proof is immediate, and can be solved automatically by the tactic `Auto`.

```
Coq < Undo.

Coq < Pattern y x.
1 subgoal

  x : nat
  y : nat
  ============================
   ([n,n0:nat]{n0=n}+{~n0=n} y x)

Coq < Elim x using nat_double_rec.
3 subgoals

  x : nat
  y : nat
  ============================
   (x:nat){x=0}+{~x=0}
subgoal 2 is:
 (x:nat){0=x}+{~0=x}
subgoal 3 is:
 (n,m:nat){m=n}+{~m=n}->{(S m)=(S n)}+{~(S m)=(S n)}

Coq < Destruct x; Auto.

Coq < Destruct x; Auto.

Coq < Intros n m H;Case H;
Coq <  [Intro eq;    Rewrite eq; Auto|
Coq <   Intro diseq; Right; Red; Intro eq; Apply diseq;Injection eq; Trivial].

Coq < Defined.
```

The tactic `Decide Equality` [2, Section 7.9.1] generalises the proof above to a large class of inductive types. It can be used for proving a proposition of the form $(x, y : R)\{x = y\} + \{ x = y\}$, where R is an inductive datatype all whose constructors take informative arguments —like for example the type *nat*:

```
Coq < Reset iseq.
Coq < Theorem iseq : (x,y:nat){x=y}+{~x=y}.
Coq < Require EqDecide.
Coq < Decide Equality.
Coq < Defined.
```

Exercise 5.2

1. *Define an inductive function* `nat2ord` *injecting natural numbers into ordinals.*
2. *Provide an elimination combinator for the ordinals.*
3. *Define the equality between ordinals as mutual inclusion, and prove that it is an equivalence relation.*

Exercise 5.3 *Define the type of lists, and a predicate "being an ordered list" using an inductive family. Then, define the function* $(\mathtt{from}\ n) = [0, 1, \ldots n]$ *and prove that it always generates an ordered list.*

5.2 Well-founded Recursion

Structural induction is a strong elimination rule for inductive types. This method can be used to define any function whose termination is based on the well-foundedness of certain order relation R decreasing at each recursive call. What makes this principle so strong is the possibility of reasoning by structural induction on the proof that certain R is well-founded. In order to illustrate this we have first to introduce the predicate of accessibility.

```
Coq < Print Acc.
Inductive Acc [A:Set; R:A->A->Prop] : A->Prop :=
      Acc_intro : (x:A)((y:A)(R y x)->(Acc A R y))->(Acc A R x)
```

This inductive predicate characterize those elements x of A such that any descending R-chain $\ldots x_2\ R\ x_1\ R\ x$ starting from x is finite. A well-founded relation is a relation such that all the elements of A are accessible.

Consider now the problem of representing in Coq the following ML function computing the integer division $div(x, y) = \frac{x}{y+1}$:

```
let rec div x y =
  if x = 0 then 0
  else if y = 0 then x
       else (div (x-y) y)+1;;
```

The representation of if-then-else expression does not pose any problem: it is just a shorthand for a case expression on a type with only two constructors. The following *grammar rule* [2, Chapter 9] extends Coq's grammar with the new syntactic construction:

```
Coq < Grammar command command1 :=
Coq <   ruleIf [ "If"     command($c1)
Coq <            "then"   command($c2)
Coq <            "else"   command($c3) ]
Coq <       -> [<<Cases $c1 of (left eqxy) => $c2 | (right diseq) => $c3 end>>].
```

The equality function on natural numbers can be represented as the function *iseq* defined page 31. Giving x and y, this function yields either the value $(left\ p)$ if there exists a proof $p : x = y$, or the value $(right\ q)$ if there exists $q : a \neq b$. The subtraction function is already defined in the library Minus.

Hence, direct translation of the ML function *div* would be:

```
Coq < Require Minus.

Coq < Fixpoint div  [x:nat] : nat->nat :=
Coq <  [y:nat]
Coq <      If  (iseq x 0) then 0
Coq <      else If (iseq y 0) then x
Coq <           else (S (div (minus x y) y)).
Coq < Error: Recursive call applied to an illegal term
Coq <        The recursive definition [...] is not well-formed
```

The program *div* is rejected by Coq because it does not verify the syntactical condition to ensure termination. In particular, the argument of the recursive call is not a pattern variable issued from a case analysis on x. However, we know that this program always stops. One way to justify its termination is to define it by structural induction on a proof of that n is accessible trough the relation `lt`. Note that any natural number n is accessible for this relation. In order to do this, it is first necessary to prove some auxiliary lemmas, justifying that the first argument of `div` decreases at each recursive call.

```
Coq < Lemma smaller : (x,y:nat)(Lth (minus x y) (S x)).

Coq < Intros x y; Pattern y x;
Coq <  Elim x using nat_double_ind;
Coq <  ((Destruct x;Constructor) Orelse (Simpl;Constructor;Assumption)).

Coq < Qed.

Coq < Lemma smallerSS : (x,y:nat)~x=O->~y=O->(Lth (minus x y) x).

Coq < Destruct x; Destruct y;
Coq <  ((Simpl; Intros; Apply smaller; Assumption) Orelse
Coq <   (Intros; Absurd O=O; Auto)).

Coq < Qed.
```

The last two lemmas are necessary to prove that for any pair of positive natural numbers x and y, if x is accessible with respect to *Lth*, then so is $x - y$.

```
Coq < Theorem decrease:
Coq <   (x,y:nat)(Acc nat Lth x)->~x=O->~y=O->(Acc nat Lth (minus x y)).

Coq < Intros x m H.

Coq <  Case H;Intros n h diseqx diseqy.

Coq <   Apply h; (Apply smallerSS; Assumption).

Coq < Defined.
```

Let us take a look to the proof of the lemma *decrease*, since the way in which it has been proven is crucial for what follows.

```
Coq < Print decrease.
decrease =
[x,m:nat]
 [H:(Acc nat Lth x)]
  <[n:nat]~n=O->~m=O->(Acc nat Lth (minus n m))>
    Cases H of
      (Acc_intro n h) =>
          [diseqx:~n=O]
           [diseqy:~m=O](h (minus n m) (smallerSS n m diseqx diseqy))
    end
     : (x,y:nat)(Acc nat Lth x)->~x=O->~y=O->(Acc nat Lth (minus x y))
```

Remark that the function (`decrease` n m H) indeed yields an accessibility proof that is *structurally smaller* than its argument H, because is (an application of) a its recursive component h. This enables to justify the following definition of *div*:

```
Coq < Require Refine.

Coq < Theorem  div : (x:nat)(y:nat)(Acc nat Lth x)->nat.

Coq < Fix 3;Intros x y H.
1 subgoal

  div : (x,_:nat)(Acc nat Lth x)->nat
  x : nat
  y : nat
  H : (Acc nat Lth x)
  ============================
   nat

Coq < Refine
Coq <     If (iseq x O) then O
Coq <     else If   (iseq y O) then x
Coq <          else (S (div (minus x y) y ?)).
1 subgoal

  div : (x,_:nat)(Acc nat Lth x)->nat
  x : nat
  y : nat
  H : (Acc nat Lth x)
  diseq : ~x=O
  diseq0 : ~y=O
  ============================
   (Acc nat Lth (minus x y))

Coq < Apply (decrease x y); Assumption.

Coq < Defined.
```

Let us explain the proof above. In this new definition of the program, what decreases is not n but the *proof* of the accessibility of n. The tactic `Fix 3` is used to indicate that the proof proceeds by structural induction on the third argument of the theorem –that is, on the accessibility proof. It also introduces a new hypothesis in the context, named as the current theorem, and with the same type as the goal. Then, the proof is refined with an incomplete proof term, containing a hole `?`. This hole corresponds to the proof of accessibility for $x-y$, and is filled up with the (smaller!) accessibility proof provided by the function *decrease*.

Lets take a look to the term *div* defined:

```
Coq < Print div.
div =
Fix div{div [x:nat;y:nat;H:(Acc nat Lth x)] : nat :=
          Cases (iseq x 0) of
            (left _) => 0
          | (right diseq) =>
                Cases (iseq y 0) of
                  (left _) => x
                | (right diseq0) =>
                      (S
                        (div (minus x y) y
                          (decrease x y H diseq diseq0)))
                end
          end}
     : (x,_:nat)(Acc nat Lth x)->nat
```

If the non-informative parts from this proof –that is, the accessibility proof– are erased, then we obtain exactly the program that we were looking for. This methodology enables the representation of any program whose termination can be proved in Coq. Once the expected properties from this program have been verified, the justification of its termination can be through away, keeping just the desired computational behavior for it.

6 CoInductive Types and Non-ending Constructions

Co-inductive types are those recursive types which are not inductive, i.e. that may contain non-well-founded objects [7, 6]. An example of a co-inductive type is the type of (possibly) infinite sequences formed with elements of type A, also called streams. This type can be introduced through the following definition:

```
Coq < Section Streams.
Coq < Variable A : Set.
Coq < CoInductive Stream :Set   := Cons : A->Stream->Stream.
```

Structural induction is the way of expressing that inductive types only contain well-founded objects. Hence, this elimination principle is not valid for co-inductive types, and the only elimination rule for streams is case analysis. This principle can be used, for example, to define the destructors *head* and *tail*.

```
Coq < Definition head  : Stream->A
Coq <                  := [x:Stream]Cases x of (Cons a s) => a end.

Coq < Definition tail  : Stream->Stream
Coq <                  := [x:Stream]Cases x of (Cons a s) => s end.
```

Infinite objects are defined by means of (non-ending) methods of construction, like in lazy functional programming languages. Such methods can be defined using the `CoFixpoint` command [2, Section 1.3.4]. For example, the following definition introduces the infinite list $[a, a, a, \ldots]$:

```
Coq < CoFixpoint repeat  : A->Stream  := [a:A](Cons a (repeat a)).
```

However, not any recursive definition is an admissible method of construction. Similarly to the case of structural induction, the definition must verify a *guardedness* condition to be accepted. This condition states that any recursive call in the definition must be protected –i.e, be an argument of– some constructor, and only an argument of constructors [5]. The following definitions are examples of valid methods of construction:

```
Coq < Variable   f        : A->A.

Coq < CoFixpoint iterate  : A->Stream  := [a:A](Cons a (iterate (f a))).

Coq < CoFixpoint map : Stream->Stream :=
Coq <   [s:Stream]
Coq <     Cases s of
Coq <        (Cons a s) => (Cons (f a) (map s))
Coq <     end.
```

Exercise 6.1 *Define two different methods for constructing the stream which infinitely alternates the booleans* `true` *and* `false`*.*

Exercise 6.2 *Using the destructors hd and tl, define a function which takes the n-th element of an infinite stream.*

A non-ending method of construction is computed lazily. This means that its definition is unfolded only when the object that it introduces is eliminated, that is, when it appears as the argument of a case expression. We can check this using the command `Eval`.

```
Coq < Eval Simpl in  [a:A](repeat a).
     = [a:A](repeat a)
     : A->Stream

Coq < Eval Simpl in  [a:A](head (repeat a)).
     = [a:A]a
     : A->A
```

6.1 Extensional Properties

Case analysis is also a valid proof principle for infinite objects. However, this principle is not sufficient to prove *extensional* properties, that is, properties concerning the whole infinite object [6]. A typical example of an extensional property is the predicate expressing that two streams have the same elements. In many cases, the minimal reflexive relation $a = b$ that is used as equality for inductive types is too small to capture equality between streams. Consider for example the streams $(\mathit{iterate}\ (f\ x))$ and $(\mathit{map}\ (\mathit{iterates}\ x))$. Even though these two streams have the same elements, no finite expansion of their definitions lead to equal terms. In other words, in order to deal with extensional properties, it is necessary to construct infinite proofs. The type of infinite proofs of equality can be introduced as a co-inductive predicate, as follows:

```
Coq < CoInductive EqSt  : Stream->Stream->Prop :=
Coq <   eqst : (s1,s2:Stream)
Coq <             ((head s1)=(head s2))->
Coq <                 (EqSt (tail s1) (tail s2))
Coq <                     ->(EqSt s1 s2).
```

It is possible to introduce proof principles for reasoning about infinite objects as combinators defined through `CoFixpoint`. However, oppositely to the case of inductive types, proof principles associated to co-inductive types are not elimination but *introduction* combinators. An example of such a combinator is Park's principle for proving the equality of two streams, usually called the *principle of co-induction*. It states that two streams are equal if they satisfy a *bisimulation*. A bisimulation is a binary relation R such that any pair of streams s_1 ad s_2 satisfying R have equal heads, and tails also satisfying R. This principle is in fact a method for constructing an infinite proof:

```
Coq < Section Parks_Principle.

Coq < Variable    R      : Stream->Stream->Prop.

Coq < Hypothesis  bisim1 : (s1,s2:Stream)(R s1 s2)->(head s1)=(head s2).

Coq < Hypothesis  bisim2 : (s1,s2:Stream)(R s1 s2)->(R (tail s1) (tail s2)).

Coq < CoFixpoint park_ppl     : (s1,s2:Stream)(R s1 s2)->(EqSt s1 s2) :=
Coq <  [s1,s2:Stream]
Coq <   [p:(R s1 s2)]
Coq <      (eqst s1 s2 (bisim1 s1 s2 p)
Coq <                  (park_ppl (tail s1) (tail s2) (bisim2 s1 s2 p) )).

Coq < End Parks_Principle.
```

Let us use the principle of co-induction to prove the extensional equality mentioned above.

```
Coq < Theorem equality : (x:A)(EqSt (iterate (f x)) (map (iterate x))).

Coq < Intro x.

Coq < Apply park_ppl with
Coq <    R:=  [s1,s2:Stream]
Coq <           (EX x | s1=(iterate (f x))/\s2=(map (iterate x))).

Coq <   Intros s1 s2 (x0,(eqs1,eqs2));Rewrite eqs1;Rewrite eqs2;Reflexivity.

Coq <   Intros s1 s2 (x0,(eqs1,eqs2)).

Coq <   Exists (f x0);Split;[Rewrite eqs1|Rewrite eqs2]; Reflexivity.

Coq <   Exists x;Split; Reflexivity.

Coq < Qed.
```

The use of Park's principle is sometimes annoying, because it requires to find an invariant relation and prove that it is indeed a bisimulation. In many cases, a shorter proof can be obtained trying to construct an ad-hoc infinite proof, defined by a guarded declaration. The tactic `Cofix f` can be used to do that. Similarly to the tactic `Fix` described in Section 5.2, this tactic introduces an extra hypothesis `f` into the context, whose type is the same as the current goal. Note that the applications of `f` in the proof *must be guarded.* In order to prevent us from doing unguarded calls, we can define a tactic that always apply a constructor before using `f` [2, Chapter 10] :

```
Coq < Reset equality.

Coq < Tactic Definition InfiniteProof [$f] :=
Coq <  [ <:tactic:<Cofix $f; Constructor;
Coq <                   [Clear $f |
Coq <                    Simpl; Try (Apply $f;Clear $f)]>> ].
```

In the example above, this tactic produces a much more simpler proof that the former one:

```
Coq < Theorem equality : (x:A)(EqSt (iterate (f x)) (map (iterate x))).

Coq < InfiniteProof equality.
1 subgoal

  A : Set
  f : A->A
  x : A
  ============================
   (head (iterate (f x)))=(head (map (iterate x)))
```

```
Coq < Reflexivity.
Coq < Qed.

Coq < Print equality.
equality =
CoFix equality{equality : (x:A)(EqSt (iterate (f x)) (map (iterate x))) :=
                  [x:A]
                   (eqst (iterate (f x)) (map (iterate x))
                     (refl_equal A (head (map (iterate x))))
                     (equality (f x)))}
     : (x:A)(EqSt (iterate (f x)) (map (iterate x)))
```

Exercise 6.3 *Define a co-inductive type* Nat *containing non-standard natural numbers – this is, verifying* $\exists m \in Nat. \forall n \in Nat. n < m$.

Exercise 6.4 *Prove that the extensional equality of streams is an equivalence relation using Park's co-induction principle.*

Exercise 6.5 *Provide a suitable definition of "being an ordered list" for infinite lists and define a principle for proving that an infinite list is ordered. Apply this method to the list* $[0, 1, \ldots]$. *Compare the result with exercise 5.3.*

References

[1] B. Barras. A formalisation of Burali-Forti's paradox in coq. Distributed within the bunch of contribution to the Coq system, March 1998. `http://pauillac.inria.fr/coq`.

[2] B. Barras, S. Boutin, C. Cornes, J. Courant, Y. Coscoy, D. Delahaye, D. de Rauglaudre, J.-C. Filliâtre, E. Giménez, H. Herbelin, G. Huet, H. Laulhère, C. Muñoz, C. Murthy, C. Parent-Vigouroux, P. Loiseleur, C. Paulin-Mohring, A. Saïbi, and B. Werner. The Coq Proof Assistant Reference Manual – Version V6.2. Technical report, INRIA, 1998.

[3] T. Coquand. An analysis of girard's paradox. In *Symposium on Logic in Computer Science*, Cambridge, MA, 1986. IEEE Computer Society Press.

[4] T. Coquand. Metamathematical Investigations of a Calculus of Constructions. In P. Odifreddi, editor, *Logic and Computer Science*, volume 31 of *The APIC series*, pages 91–122. Academic Press, 1990.

[5] E. Giménez. Codifying guarded definitions with recursive schemes. In *Workshop on Types for Proofs and Programs*, number 996 in LNCS, pages 39–59. Springer-Verlag, 1994.

[6] E. Giménez. An application of co-inductive types in coq: verification of the alternating bit protocol. In *Workshop on Types for Proofs and Programs*, number 1158 in LNCS, pages 135–152. Springer-Verlag, 1995.

[7] E. Giménez. *A Calculus of Infinite Constructions and its application to the verification of communicating systems*. PhD thesis, Ecole Normale Supérieure de Lyon, 1996.

[8] G. Huet, G. Kahn, and C. Paulin-Mohring. The Coq proof assistant - a tutorial, version 6.1. rapport technique 204, INRIA, Août 1997. Version révisée distribuée avec Coq.

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifi que,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e 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)
http://www.inria.fr
ISSN 0249-6399