A Tutorial on [Co-]Inductive Types in Coq

Eduardo Giménez‡, Pierre Castéran†

May 1998 — February 11, 2009

Abstract

This document is an introduction to the definition and use of inductive and co-inductive types in the Coq proof environment. It explains how 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.

‡Eduardo.Gimenez@inria.fr
†Pierre.Casteran@labri.fr

The first versions of this document were entirely written by Eduardo Gimenez. Pierre Castéran wrote the 2004 and 2006 revisions.
# Contents

1 About this document 4

2 Introducing Inductive Types 5
  2.1 Lists 6
  2.2 Vectors 8
  2.3 The contradictory proposition 9
  2.4 The tautological proposition 9
  2.5 Relations as inductive types 9
  2.6 About general parameters (Coq version ≥ 8.1) 11
  2.7 The propositional equality type 12
  2.8 Logical connectives 13
  2.9 The existential quantifier 15
  2.10 Mutually Dependent Definitions 16

3 Case Analysis and Pattern-matching 16
  3.1 Non-dependent Case Analysis 16
    3.1.1 Example: the predecessor function 17
  3.2 Dependent Case Analysis 18
    3.2.1 Example: strong specification of the predecessor function 18
  3.3 Some Examples of Case Analysis 20
    3.3.1 The Empty Type 20
    3.3.2 The Equality Type 21
    3.3.3 The Predicate $n \leq m$ 24
    3.3.4 Vectors 26
  3.4 Case Analysis and Logical Paradoxes 27
    3.4.1 The Positivity Condition 27
    3.4.2 Impredicative Inductive Types 30
    3.4.3 Extraction Constraints 32
    3.4.4 Strong Case Analysis on Proofs 32
    3.4.5 Summary of Constraints 33

4 Some Proof Techniques Based on Case Analysis 34
  4.1 Discrimination of introduction rules 34
  4.2 Injectiveness of introduction rules 36
  4.3 Inversion Techniques 37
    4.3.1 Interactive mode 39
    4.3.2 Static mode 39
5 **Inductive Types and Structural Induction** 42
  5.1 Proofs by Structural Induction 44
  5.2 Using Elimination Combinators 46
  5.3 Well-founded Recursion 49

6 **A case study in dependent elimination** 55

7 **Co-inductive Types and Non-ending Constructions** 60
  7.1 Extensional Properties 62
  7.2 About injection, discriminate, and inversion 64
1 About this document

This document is an introduction to the definition and use of inductive and co-inductive types in the Coq proof environment. It was born from the notes written for the course about the version V5.10 of Coq, given by Eduardo Gimenez at the École Normale Supérieure de Lyon in March 1996. This article is a revised and improved version of these notes for the version V8.0 of the system.

We assume that the reader has some familiarity with the proofs-as-programs paradigm of Logic [7] and the generalities of the Coq system [4]. You would take a greater advantage of this document if you first read the general tutorial about Coq and Coq’s FAQ, both available on [5]. A text book [3], accompanied with a lot of examples and exercises [2], presents a detailed description of the Coq system and its underlying formalism: the Calculus of Inductive Construction. Finally, the complete description of Coq is given in the reference manual [4]. Most of the tactics and commands we describe have several options, which we do not present exhaustively. If some script herein uses a non described feature, please refer to the Reference Manual.

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, and observe the behavior of the system. All the examples proposed in this tutorial can be downloaded from the same site as the present document.

The tutorial is organised as follows. The next section describes how inductive types are defined in Coq, and introduces some useful ones, like natural numbers, the empty type, the propositional equality type, and the logical 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 inductive or co-inductive 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. Section 6 presents some elaborate techniques about dependent case analysis. Finally, Section 7 is a brief introduction to co-inductive types –i.e., types containing infinite objects– and the principle of co-induction.

Thanks to Bruno Barras, Yves Bertot, Hugo Herbelin, Jean-François Monin and Michel Lévy for their help.
Lexical conventions

The *typewriter* font is used to represent text input by the user, while the *italic* font is used to represent the text output by the system as answers.

Moreover, the mathematical symbols ≤, ≠, ∃, ∀, →, ∀, ∧, and ⇒ stand for the character strings ≤, ≠, exists, forall, →, ≤, ∨, ∧, and ⇒, respectively. For instance, the *Coq* statement

forall A:Type,(exists x : A, forall (y:A), x <> y) -> 2 = 3

is written as follows in this tutorial:

∀A:Type,(∃x:A, ∀y:A, x ≠ y) → 2 = 3

When a fragment of *Coq* input text appears in the middle of regular text, we often place this fragment between double quotes “…” These double quotes do not belong to the *Coq* syntax.

Finally, any string enclosed between (* and *) is a comment and is ignored by the *Coq* system.

2 Introducing Inductive Types

Inductive types are types closed with respect to their introduction rules. These rules explain the most basic or *canonical* ways of constructing an element of the 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.

\[
\text{Inductive nat : Set :=} \\
| \mathsf{O} : \text{nat} \\
| \mathsf{S} : \text{nat \rightarrow nat}. \\
\]

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 define the type (0 and S), also called its *constructors*. The constructors 0 and S determine all the elements of this type. In other words, if \(n:\text{nat}\), then \(n\) must have been introduced either by the rule \(0\) or by an application of the rule \(S\) to a previously constructed natural number. In this sense, we can say that \(\text{nat}\) is *closed*. On the contrary, the type \(\text{Set}\) is an *open* type, since we do not know *a priori* all the possible ways of introducing an object of type \(\text{Set}\).

After entering this command, the constants \(\text{nat}, \mathsf{O}\) and \(\mathsf{S}\) are available in the current context. We can see their types using the *Check* command:
Check nat.

nat : Set

Check O.

O : nat

Check S.

S : nat → nat

Moreover, Coq adds to the context three constants named nat_ind, nat_rec and nat_rect, which 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 our (re)definition of nat, using the command Reset.

Reset nat.

Print nat.

Inductive nat : Set := O : nat | S : nat → nat
For S: Argument scope is [nat_scope]

Notice that Coq’s interpretation scope for natural numbers (called nat_scope) allows us to read and write natural numbers in decimal form (see [4]). For instance, the constructor 0 can be read or written as the digit 0, and the term “S (S (S 0))” as 3.

Check O.

0 : nat.

Check (S (S (S 0))).

3 : nat

Let us now take a look to some other recursive types contained in the standard library of Coq.

2.1 Lists

Lists are defined in library List\textsuperscript{2}

\textsuperscript{2}Notice that in versions of Coq prior to 8.1, the parameter A had sort Set instead of Type; the constant list was thus of type Set → Set.
Require Import List.
Print list.

Inductive list (A : Type) : Type :=
  nil : list A \ cons : A \rightarrow list A \rightarrow list A

For nil: Argument A is implicit
For cons: Argument A is implicit
For list: Argument scope is [type_scope]
For nil: Argument scope is [type_scope]
For cons: Argument scopes are [type_scope _ _]

In this definition, A is a general parameter, global to both constructors. This kind of definition allows us to build a whole family of inductive types, indexed over the sort Type. This can be observed if we consider the type of identifiers list, cons and nil. Notice the notation (A := ...) which must be used when Coq’s type inference algorithm cannot infer the implicit parameter A.

Check list.
list
  : Type \rightarrow Type

Check (nil (A:=nat)).
nil
  : list nat

Check (nil (A:= nat \rightarrow nat)).
nil
  : list (nat \rightarrow nat)

Check (fun A : Type \Rightarrow (cons (A:=A))).
fun A : Type \Rightarrow cons (A:=A)
  : \forall A : Type, A \rightarrow list A \rightarrow list A

Check (cons 3 (cons 2 nil)).
3 :: 2 :: nil
  : list nat

Check (nat :: bool ::nil).
nat :: bool :: nil
  : list Set
Check ((3<=4) :: True :: nil).
(3<=4) :: True :: nil
   : list Prop

Check (Prop::Set::nil).
Prop::Set::nil
   : list Type

2.2 Vectors.

Like list, vector is a polymorphic type: if \( A \) is a type, and \( n \) a natural number, "vector \( A \ n \)" is the type of vectors of elements of \( A \) and size \( n \).

Require Import Bvector.

Print vector.

```
Inductive vector (A : Type) : nat → Type :=
  Vnil : vector A 0
| Vcons : A → ∀ n : nat, vector A n → vector A (S n)
For vector: Argument scopes are [type_scope nat_scope]
For Vnil: Argument scope is [type_scope]
For Vcons: Argument scopes are [type_scope _ nat_scope _]
```

Remark the difference between the two parameters \( A \) and \( n \): 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. Such types parameterized by regular values are called dependent types.

Check (Vnil nat).
Vnil nat
   : vector nat 0

Check (fun (A:Type)(a:A) ⇒ Vcons _ a _ (Vnil _)).
fun (A : Type) (a : A) ⇒ Vcons A a 0 (Vnil A)
   : ∀ A : Type, A → vector A 1

Check (Vcons _ 5 _ (Vcons _ 3 _ (Vnil _))).
Vcons nat 5 1 (Vcons nat 3 0 (Vnil nat))
2.3 The contradictory proposition.

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

Print False.
Inductive False : Prop :=

Notice that no constructor is given in this definition.

2.4 The tautological proposition.

Similarly, the tautological proposition True is defined as an inductive type with only one element I:

Print True.
Inductive True : Prop := I : True

2.5 Relations as inductive types.

Some relations can also be introduced in a smart way as an inductive family of propositions. Let us take as example the order \( n \leq m \) on natural numbers, called le in Coq. This relation is introduced through the following definition, quoted from the standard library\(^3\):

Print le.
Inductive le (n:nat) : nat → Prop :=
| le_n: n ≤ n |
| le_S: ∀ m, n ≤ m → n ≤ S m.

Notice that in this definition \( n \) is a general parameter, while the second argument of le is an index (see section 2.2). This definition introduces the binary relation \( n \leq m \) as the family of unary predicates “to be greater or equal 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 or equal than another one. In fact, an object of type \( n \leq m \) is nothing but a proof built up using the constructors le_n and le_S

\(^3\)In the interpretation scope for Peano arithmetic: nat_scope, “\( n \leq m \)” is equivalent to “le _n m”.

: vector nat 2
of this type. As an example, let us construct a proof that zero is less or equal than three using *Coq*’s interactive proof mode. Such an object can be obtained applying three times the second introduction rule of \( \leq \), to a proof that zero is less or equal than itself, which is provided by the first constructor of \( \leq \):

Theorem zero_leq_three : 0 ≤ 3.
Proof.
\[ 1 \text{ subgoal} \]

\[ \begin{align*}
0 \leq 3
\end{align*} \]

Proof.
constructor 2.
\[ 1 \text{ subgoal} \]

\[ \begin{align*}
0 \leq 2
\end{align*} \]

constructor 2.
\[ 1 \text{ subgoal} \]

\[ \begin{align*}
0 \leq 1
\end{align*} \]

constructor 2
\[ 1 \text{ subgoal} \]

\[ \begin{align*}
0 \leq 0
\end{align*} \]

constructor 1.
Proof completed
Qed.

When the current goal is an inductive type, the tactic “constructor i” applies the \( i \)-th constructor in the definition of the type. We can take a look at the proof constructed using the command *Print*:

Print Print zero_leq_three.
zero_leq_three =
When the parameter $i$ is not supplied, the tactic constructor tries to apply "constructor 1", "constructor 2",..., "constructor $n" where $n$ is the number of constructors of the inductive type $\text{2 in our example}$ of the conclusion of the goal. Our little proof can thus be obtained iterating the tactic constructor until it fails:

```
Lemma zero_leq_three': 0 ≤ 3.
  repeat constructor.
Qed.
```

Notice that the strict order on $\text{nat}$, called $\text{lt}$ is not inductively defined: the proposition $n < p$ (notation for $\text{lt n p}$) is reducible to $(S n) ≤ p$.

```
Print lt.

lt = fun n m : nat ⇒ S n ≤ m : nat → nat → Prop
```

```
Lemma zero_lt_three : 0 < 3.
  Proof.
  repeat constructor.
  Qed.
```

```
Print zero_lt_three.

zero_lt_three = le_S 1 2 (le_S 1 1 (le_n 1)): 0 < 3
```

### 2.6 About general parameters ($\text{Coq version ≥ 8.1}$)

Since version 8.1, it is possible to write more compact inductive definitions than in earlier versions.

Consider the following alternative definition of the relation $\leq$ on type $\text{nat}$:

```
Inductive le'(n:nat):nat → Prop :=
  le'_n : le' n n
  le'_S : forall p, le' (S n) p -> le' n p.
```

```
Hint Constructors le'.
```
We notice that the type of the second constructor of \( \text{le}' \) has an argument whose type is \( \text{le}' (S \ n) \ p \). This contrasts with earlier versions of Coq, in which a general parameter \( a \) of an inductive type \( I \) had to appear only in applications of the form \( I \ldots a \).

Since version 8.1, if \( a \) is a general parameter of an inductive type \( I \), the type of an argument of a constructor of \( I \) may be of the form \( I \ldots t_a \), where \( t_a \) is any term. Notice that the final type of the constructors must be of the form \( I \ldots a \), since these constructors describe how to form inhabitants of type \( I \ldots a \) (this is the role of parameter \( a \)).

Another example of this new feature is Coq’s definition of accessibility (see Section 5.3), which has a general parameter \( x \); the constructor for the predicate “\( x \) is accessible” takes an argument of type “\( y \) is accessible”.

In earlier versions of Coq, a relation like \( \text{le}' \) would have to be defined without \( n \) being a general parameter.

```
Reset le'.

Inductive le': nat -> nat -> Prop :=
  | le'_n : forall n, le' n n
  | le'_S : forall n p, le' (S n) p -> le' n p.
```

### 2.7 The propositional equality type.

In Coq, the propositional equality between two inhabitants \( a \) and \( b \) of the same type \( A \), noted \( a = b \), is introduced as a family of recursive predicates “\( \text{to be equal to} \ a \)”, parameterised by both \( a \) and its type \( A \). This family of types has only one introduction rule, which corresponds to reflexivity. Notice that the syntax “\( a = b \)” is an abbreviation for “\( \text{eq} \ a \ b \)”, and that the parameter \( A \) is implicit, as it can be inferred from \( a \).

```
Print eq.
Inductive eq (A : Type) (x : A) : A -> Prop :=
  refl_equal : x = x
For eq: Argument A is implicit
For refl_equal: Argument A is implicit
For eq: Argument scopes are [type_scope _ _]
For refl_equal: Argument scopes are [type_scope _]
```

Notice also that the first parameter \( A \) of \( \text{eq} \) has type Type. The type system of Coq allows us to consider equality between various kinds of terms: elements of a set, proofs, propositions, types, and so on. Look at [4, 3] to get more details on Coq’s type system, as well as implicit arguments and argument scopes.
Lemma eq_3_3 : 2 + 1 = 3.
Proof.
  reflexivity.
Qed.

Lemma eq_proof_proof : refl_equal (2*6) = refl_equal (3*4).
Proof.
  reflexivity.
Qed.

Print eq_proof_proof.
  eq_proof_proof =
  refl_equal (refl_equal (3 * 4))
  : refl_equal (2 * 6) = refl_equal (3 * 4)

Lemma eq_lt_le : (2 < 4) = (3 ≤ 4).
Proof.
  reflexivity.
Qed.

Lemma eq_nat_nat : nat = nat.
Proof.
  reflexivity.
Qed.

Lemma eq_Set_Set : Set = Set.
Proof.
  reflexivity.
Qed.

2.8 Logical connectives.

The conjunction and disjunction of two propositions are also examples of recursive

types:

Inductive or (A B : Prop) : Prop :=
  or_introl : A → A ∨ B | or_intror : B → A ∨ B

Inductive and (A B : Prop) : Prop :=
The propositions $A$ and $B$ are general parameters of these connectives. Choosing different universes for $A$ and $B$ and for the inductive type itself gives rise to different type constructors. For example, the type `sumbool` is a disjunction but with computational contents.

```coq
Inductive sumbool (A B : Prop) : Set :=
  left : A → {A} + {B} | right : B → {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 values “left $p$” and “right $q$” replace the boolean values `true` and `false`, respectively. The advantage of this type over `bool` is that it makes available the proofs $p$ of $A$ or $q$ of $B$, which could be necessary to construct a verification proof about the program. For instance, let us consider the certified program `le_lt_dec` of the Standard Library.

```coq
Require Import Compare_dec.
Check le_lt_dec.

``le_lt_dec``

````

We use `le_lt_dec` to build a function for computing the max of two natural numbers:

```coq
Definition max (n p : nat) := match le_lt_dec n p with
  | left _ ⇒ p
  | right _ ⇒ n
end.
```

In the following proof, the case analysis on the term “`le_lt_dec n p`” gives us an access to proofs of $n \leq p$ in the first case, $p < n$ in the other.

```coq
Theorem le_max : \forall n p, n \leq p → max n p = p.
Proof.
  intros n p ; unfold max ; case (le_lt_dec n p) ; simpl.

2 subgoals
```
\[ n : \text{nat} \]
\[ p : \text{nat} \]

\[ \text{============================} \]
\[ n \leq p \rightarrow n \leq p \rightarrow p = p \]

**subgoal 2 is:**
\[ p < n \rightarrow n \leq p \rightarrow n = p \]

trivial.
intros; absurd (p < p); eauto with arith.
Qed.

Once the program verified, the proofs are erased by the extraction procedure:

Extraction max.

(** val max : nat \rightarrow nat \rightarrow nat **) 

\[
\begin{aligned}
\text{let max n p =} \\
\text{match le_lt_dec n p with} \\
\text{ \ | Left \rightarrow p} \\
\text{ \ | Right \rightarrow n}
\end{aligned}
\]

Another example of use of \texttt{sumbool} is given in Section 5.3: the theorem \texttt{eq_nat_dec} of library \texttt{Coq.Arith.Peano_dec} is used in an euclidean division algorithm.

### 2.9 The existential quantifier.

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

\[
\begin{aligned}
\text{Inductive ex (A : Type) (P : A \rightarrow Prop) : Prop :=} \\
\text{ \ ex_intro : \forall x : A, P x \rightarrow ex P}
\end{aligned}
\]

Notice that \texttt{Coq} uses the abbreviation \( \exists x : A, B \) for \( \text{ex (fun x : A \Rightarrow B)} \).

The former quantifier inhabits the universe of propositions. As for the conjunction and disjunction connectives, there is also another version of existential quantification inhabiting the universes \texttt{Type}, which is written \texttt{sig P}. The syntax \( \{ x : A \mid B \} \) is an abreviation for \( \text{sig (fun x : A \Rightarrow B)} \).
2.10 Mutually Dependent Definitions

Mutually dependent definitions 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:

\[
\text{Inductive tree}(A:\text{Type}) : \text{Type} := \\
\text{node} : A \to \text{forest} A \to \text{tree} A \\
\text{with forest}(A:\text{Set}) : \text{Type} := \\
\text{nochild} : \text{forest} A | \\
\text{addchild} : \text{tree} A \to \text{forest} A \to \text{forest} A.
\]

Yet another example of mutually dependent types are the predicates \text{even} and \text{odd} on natural numbers:

\[
\text{Inductive even : nat}\to\text{Prop} := \\
\text{evenO} : \text{even} 0 | \\
\text{evenS} : \forall n, \text{odd} n \to \text{even} (S n) \\
\text{with odd : nat}\to\text{Prop} := \\
\text{oddS} : \forall n, \text{even} n \to \text{odd} (S n).
\]

Lemma odd_49 : odd (7 * 7).
\text{simpl;} \text{repeat constructor.}
\text{Qed.}

3 Case Analysis and Pattern-matching

3.1 Non-dependent Case Analysis

An elimination rule for the type \( A \) is some way to use an object \( a : A \) in order to define an object in some type \( B \). A natural elimination rule for an inductive type is case analysis.

For instance, any value of type \text{nat} is built using either \text{O} or \text{S}. Thus, a systematic way of building a value of type \( B \) from any value of type \text{nat} is to associate to \( 0 \) a constant \( t_0 : B \) and to every term of the form “ \( S \ p \)” a term \( t_S : B \). The following construction has type \( B \):

\[
\text{match } n \text{ return } B \text{ with } 0 \Rightarrow t_0 | S \ p \Rightarrow t_S \text{ end}
\]
In most of the cases, Coq is able to infer the type $B$ of the object defined, so the "return $B$" part can be omitted.

The computing rules associated with this construct are the expected ones (the notation $t_S\{q/p\}$ stands for the substitution of $p$ by $q$ in $t_S$):

\[
\begin{align*}
\text{match } O \text{ return } b \text{ with } 0 & \Rightarrow t_O \mid S \ p \Rightarrow t_S \text{ end } \Rightarrow t_O \\
\text{match } S \ q \text{ return } b \text{ with } 0 & \Rightarrow t_O \mid S \ p \Rightarrow t_S \text{ end } \Rightarrow t_S\{q/p\}
\end{align*}
\]

3.1.1 Example: the predecessor function.

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

\[
\text{Definition pred } (n:\text{nat}) := \text{match } n \text{ with} \\
| O & \Rightarrow O \\
| S \ m & \Rightarrow m \\
\text{end.}
\]

Eval simpl in pred 56.
\[
= 55 \\
: \text{nat}
\]

Eval simpl in pred 0.
\[
= 0 \\
: \text{nat}
\]

Eval simpl in fun $p \Rightarrow$ pred ($S \ p$).
\[
= \text{fun } p : \text{nat} \Rightarrow p \\
: \text{nat} \rightarrow \text{nat}
\]

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

\[
\text{Definition xorb } (b_1 \ b_2:\text{bool}) := \\
\text{match } b_1, b_2 \text{ with} \\
| \text{false, true } & \Rightarrow \text{true} \\
| \text{true, false } & \Rightarrow \text{true} \\
| \_, \_ & \Rightarrow \text{false} \\
\text{end.}
\]
This kind of definition is compiled in Coq as follows:\footnote{Coq uses the conditional \texttt{if \ b then \ a \ else \ b} as an abbreviation to \texttt{match \ b \ with \ true \ ⇒ \ a \ | \ false \ ⇒ \ b \ end}}: 

```
Print xorb.
xorb = 
  fun b1 b2 : bool ⇒
    if b1 then if b2 then false else true 
    else if b2 then true else false
 : bool → bool → bool
```

3.2 Dependent Case Analysis

For a pattern matching construct of the form \texttt{match \ n \ with \ ... \ end} a more general typing rule is obtained considering that the type of the whole expression may also depend on \( n \). For instance, let us consider some function \( Q : \text{nat} \rightarrow \text{Type} \), and \( n : \text{nat} \). In order to build a term of type \( Q \ n \), we can associate to the constructor \( O \) some term \( t_O : Q \ O \) and to the pattern \( \text{S} \ p \) some term \( t_S : Q \ (\text{S} \ p) \). Notice that the terms \( t_O \) and \( t_S \) do not have the same type.

The syntax of the dependent case analysis and its associated typing rule make precise how the resulting type depends on the argument of the pattern matching, and which constraint holds on the branches of the pattern matching:

\[
\frac{Q : \text{nat} \rightarrow \text{Type} \quad t_O : Q \ O \quad p : \text{nat} \vdash t_p : Q \ (\text{S} \ p) \quad n : \text{nat}}{\text{match} \ n \ \text{as} \ n_0 \ \text{return} \ Q \ n_0 \ \text{with} \ | \ 0 \ ⇒ t_O \ | \ \text{S} \ p \ ⇒ t_S \ \text{end} : Q \ n}
\]

The interest of this rule of dependent pattern-matching is that it can also be read as the following logical principle (when \( Q \) has type \( \text{nat} \rightarrow \text{Prop} \) by Prop in the type of \( Q \)): 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 \( p : \text{nat} \), \( Q \) holds for \( (\text{S} \ p) \). The former, non-dependent version of case analysis can be obtained from this latter rule just taking \( Q \) as a constant function on \( n \).

Notice that destructuring \( n \) into \( O \) or \( \text{S} \ p \) doesn’t make appear in the goal the equalities \( n = O \) and \( n = \text{S} \ p \). They are “internalized” in the rules above (see section 4.3.)

3.2.1 Example: strong specification of the predecessor function.

In Section 3.1.1, the predecessor function was defined directly as a function from \( \text{nat} \) to \( \text{nat} \). It remains to prove that this function has some desired properties. Another way to proceed is to, first introduce a specification of what is the predecessor.
of a natural number, under the form of a \textit{Coq} type, then build an inhabitant of this type: in other words, a realization of this specification. This way, the correctness of this realization is ensured by \textit{Coq}'s type system.

A reasonable specification for \texttt{pred} is to say that for all \( n \) there exists another \( m \) such that either \( m = n = 0 \), or \((S \ m)\) is equal to \( n \). The function \texttt{pred} should be just the way to compute such an \( m \).

\begin{verbatim}
Definition pred_spec (n:nat) :=
  \{m:nat | n=0 \land m=0 \lor n = S m\}.

Definition predecessor : \forall n:nat, pred_spec n.
  intro n; case n.

n : nat

\end{verbatim}

\begin{verbatim}
unfold pred_spec;exists 0;auto.
\end{verbatim}


\begin{verbatim}
\forall n0 : nat, pred_spec (S n0)

unfold pred_spec; intro n0; exists n0; auto.

Defined.
\end{verbatim}

If we print the term built by \textit{Coq}, its dependent pattern-matching structure can be observed:

\begin{verbatim}
predecessor = fun n : nat ⇒
match n as n0 return (pred_spec n0) with
| 0 ⇒
  exist (fun m : nat ⇒ 0 = 0 \land m = 0 \lor 0 = S m) 0
  (or_introl (0 = 1)
   (conj (refl_equal 0) (refl_equal 0)))
| S n0 ⇒
  exist (fun m : nat ⇒ S n0 = 0 \land m = 0 \lor S n0 = S m) n0
  (or_intror (S n0 = 0 \land n0 = 0) (refl_equal (S n0)))
end : \forall n : nat, pred_spec n
\end{verbatim}

Notice that there are many variants to the pattern "\texttt{intros ...; case ...}". Look at for tactics "\texttt{destruct}", "\texttt{intro pattern}", etc. in the reference manual and/or the book.
The command \texttt{Extraction} can be used to see the computational contents associated to the \textit{certified} function \texttt{predecessor}:

\texttt{Extraction predecessor.}

\((**\texttt{val predecessor : nat \rightarrow pred_spec ***)}

\texttt{let predecessor = function}
\begin{itemize}
  \item \texttt{O \rightarrow O}
  \item \texttt{S n0 \rightarrow n0}
\end{itemize}

\textbf{Exercise 3.1} \textit{Prove the following theorem:}

\texttt{Theorem nat\_expand : \forall n:nat,}
\begin{align*}
  n &= \texttt{match n with} \\
    &| 0 \Rightarrow 0 \\
    &| S p \Rightarrow S p
\end{align*}

\texttt{end.}

\textbf{3.3 Some Examples of Case Analysis}

The reader will find in the Reference manual all details about typing case analysis (chapter 4: Calculus of Inductive Constructions, and chapter 15: Extended Pattern-Matching).

The following commented examples will show the different situations to consider.

\textbf{3.3.1 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 \(p : \text{False}\) there are no cases to be considered. In other words, the rule of (non-dependent) case analysis for the type \text{False} is (for \(s \) in \text{Prop, Set or Type)}:

\[ \begin{align*}
  Q : s & \quad p : \text{False} \\
  \text{match } p \text{ return } Q \text{ with end : } Q
\end{align*} \]

As a corollary, if we could construct an object in \text{False}, then it could be possible to define an object in any type. The tactic \texttt{contradiction} 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 \text{False}) and then proves the goal by a case analysis of it.

20
Theorem fromFalse : False \rightarrow \text{0}=1.
Proof.
  intro H.
  contradiction.
Qed.

In Coq the negation is defined as follows :

Definition not (P:Prop) := P \rightarrow \text{False}

The proposition “not A” is also written “\sim A”.
If A and B are propositions, a is a proof of A and H is a proof of \sim A, the
term “match H a return B with end” is a proof term of B. Thus, if your
goal is B and you have some hypothesis H :\sim A, the tactic “case H” generates
a new subgoal with statement A, as shown by the following example\(^5\).

Fact Nosense : 0 \neq 0 \rightarrow 2 = 3.
Proof.
  intro H; case H.

\begin{verbatim}
  0 = 0
  reflexivity.
Qed.
\end{verbatim}

The tactic “absurd A” (where A is any proposition), is based on the same
principle, but generates two subgoals: A and \sim A, for solving B.

3.3.2 The Equality Type

Let A : Type, a, b of type A, and \pi a proof of a = b. Non dependent case
analysis of \pi allows us to associate to any proof of “Q a” a proof of “Q b”,
where Q : A\rightarrow s (where s \in \{Prop, Set, Type\}). The following term is a proof of
“Q a \rightarrow Q b”.

\begin{verbatim}
fun H : Q a =>
  match \pi in (_, y) return Q y with
    refl_equal => H
  end
\end{verbatim}

\(^5\)Notice that a\neq b is just an abbreviation for “\sim a= b”
Notice the header of the match construct. It expresses how the resulting type "Q y" depends on the type of p. Notice also that in the pattern introduced by the keyword in, the parameter a in the type "a = y" must be implicit, and replaced by a wildcard '_'.

Therefore, case analysis on a proof of the equality a = b amounts to replacing 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.

Theorem trans : \forall n m p : nat, n = m → m = p → n = p.
Proof.
intros n m p eqnm.

n : nat
m : nat
p : nat
eqnm : n = m

============================

m = p → n = p

case eqnm.

n : nat
m : nat
p : nat
eqnm : n = m

============================

n = p → n = p

trivial.
Qed.

Exercise 3.2 Prove the symmetry property of equality.

Instead of using case, we can use the tactic rewrite. If H is a proof of a = b, then "rewrite H" performs a case analysis on a proof of b = a, obtained by applying a symmetry theorem to H. This application of symmetry allows us 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"

Lemma Rw : \forall x y : nat, y = y * x → y * x * x = y.
intros x y e; do 2 rewrite <- e.
Notice that, if $H : a = b$, then the tactic \texttt{``rewrite } H \texttt{''} replaces \textit{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 \texttt{pattern}. Let us consider yet another example to illustrate this.

Let us start with some simple theorems of arithmetic; two of them are already proven in the Standard Library, the last is left as an exercise.

\begin{align*}
mult_1_l & : \forall n : \text{nat}, 1 * n = n \\
mult_plus_distr_r & : \forall n m p : \text{nat}, (n + m) * p = n * p + m * p \\
mult_distr_S & : \forall n p : \text{nat}, n * p + p = (S n) * p.
\end{align*}

Let us now prove a simple result:

\textbf{Lemma four_n : } $\forall n : \text{nat}, n+n+n+n = 4*n$.
\textbf{Proof.} \\
intro n;rewrite <- (mult_1_l n).

\begin{align*}
n & : \text{nat} \\
\quad & \text{-----------------------------------------------} \\
1 * n + 1 * n + 1 * n + 1 * n & = 4 * (1 * n)
\end{align*}

We can see that the \texttt{rewrite} tactic call replaced \textit{all} the occurrences of \texttt{n} by the term \texttt{``1 * n''}. If we want to do the rewriting only on the leftmost occurrence of \texttt{n}, we can mark this occurrence using the \texttt{pattern} tactic:
Undo.
intro n; pattern n at 1.

n : nat

============================================
(fun n0 : nat ⇒ n0 + n + n + n = 4 * n) n

Applying the tactic "pattern n at 1" allowed us to explicitly abstract the first occurrence of n from the goal, putting this goal under the form " Q n ", thus pointing to rewrite the particular predicate on n that we search to prove.

rewrite <- mult_1_l.

1 subgoal

n : nat

============================================
1 * n + n + n + n = 4 * n

repeat rewrite mult_distr_S.

n : nat

============================================
4 * n = 4 * n

trivial.
Qed.

3.3.3 The Predicate n ≤ m

The last but one instance of the elimination schema that we will illustrate is case analysis for the predicate n ≤ m:

Let n and p be terms of type nat, and Q a predicate of type nat → Prop. If H is a proof of " n ≤ p ", H₀ a proof of " Q n " and Hₖ a proof of the statement " ∀ m : nat, n ≤ m → Q (S m) " , then the term

match H in (_ ≤ q) return (Q q) with
| le_n ⇒ H₀
| le_S m Hm ⇒ HS m Hm
end
is a proof term of “\( Q \ p \)”. The two patterns of this match construct describe all possible forms of proofs of “\( n \leq m \)” (notice again that the general parameter \( n \) is implicit in the “in . . .” clause and is absent from the match patterns.

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 \( \leq \), 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 or equal than 1 is the successor of another natural number:

**Lemma predecessor_of_positive** :

\[
\forall n, 1 \leq n \rightarrow \exists p : \text{nat}, n = S p.
\]

**Proof**.

\[
\begin{align*}
\text{intros } n \ H; \text{case } H. \\
\quad n : \text{nat} \\
\quad H : l \leq n \\
\quad \text{exists 0; trivial.}
\end{align*}
\]

\[
\begin{align*}
\text{intros } m _ \ . \\
\quad \text{exists m} . \\
\quad \text{trivial}. \\
\text{Qed.}
\end{align*}
\]
3.3.4 Vectors

The vector polymorphic and dependent family of types will give an idea of the most general scheme of pattern-matching.

For instance, let us define a function for computing the tail of any vector. Notice that we shall build a total function, by considering that the tail of an empty vector is this vector itself. In that sense, it will be slightly different from the \texttt{Vtail} function of the Standard Library, which is defined only for vectors of type \texttt{"vector A (S n)"}.

The header of the function we want to build is the following:

\begin{verbatim}
Definition Vtail_total
  (A : Type) (n : nat) (v : vector A n) : vector A (pred n):=

Since the branches will not have the same type (depending on the parameter \texttt{n}), the body of this function is a dependent pattern matching on \texttt{v}. So we will have:

\begin{verbatim}
m\texttt{atch} v \texttt{in} (vector \_ n0) \texttt{return} (vector A (pred n0)) \texttt{with}

The first branch deals with the constructor \texttt{Vnil} and must return a value in \texttt{"vector A (pred 0)"}, convertible to \texttt{"vector A 0"}. So, we propose:
\begin{verbatim}
| Vnil \Rightarrow Vnil A
\end{verbatim}

The second branch considers a vector in \texttt{"vector A (S n0)"} of the form \texttt{"Vcons A n0 v0"}, with \texttt{"v0:vector A n0"}, and must return a value of type \texttt{"vector A (pred (S n0))"}, which is convertible to \texttt{"vector A n0"}. This second branch is thus:
\begin{verbatim}
| Vcons _ n0 v0 \Rightarrow v0
\end{verbatim}

Here is the full definition:

\begin{verbatim}
Definition Vtail_total
  (A : Type) (n : nat) (v : vector A n) : vector A (pred n):=
  match v in (vector _ n0) return (vector A (pred n0)) with
  | Vnil \Rightarrow Vnil A
  | Vcons _ n0 v0 \Rightarrow v0
end.
\end{verbatim}
3.4 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 brief explanation of these restrictions.

3.4.1 The Positivity 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 positivity 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 “inductive type” Lambda, whose only constructor is lambda of type (Lambda → False) → Lambda. Following the pattern given in Section 3.3, the rule of (non dependent) case analysis for Lambda would be the following:

\[ \begin{array}{c}
Q : \text{Prop} \\
p : \text{Lambda} \\
h : \text{Lambda} \rightarrow \text{False} \\
\end{array} \]

\[ \frac{t : Q}{\text{match } p \text{ return } Q \text{ with } \lambda h \Rightarrow t \text{ end : } Q} \]

In order to avoid paradoxes, it is impossible to construct the type Lambda in Coq:

\[
\text{Inductive Lambda : Set :=} \\
\quad \lambda \text{ : (Lambda } \rightarrow \text{ False) } \rightarrow \text{ Lambda.}
\]

\textbf{Error: Non strictly positive occurrence of "Lambda" in "(Lambda } \rightarrow \text{ False) } \rightarrow \text{ Lambda"}

In order to explain this danger, we will declare some constants for simulating the construction of Lambda as an inductive type.

Let us open some section, and declare two variables, the first one for Lambda, the other for the constructor lambda.

\textbf{Section Paradox.}
\textbf{Variable Lambda : Set.}
\textbf{Variable lambda : (Lambda } \rightarrow \text{ False) } \rightarrow \text{ Lambda.}

Since Lambda is not a truely inductive type, we can’t use the \texttt{match} construct. Nevertheless, we can simulate it by a variable matchL such that the term
“matchL $l \ Q$ (fun $h : \Lambda \rightarrow \text{False} \Rightarrow t$)” should be understood as
“match $l$ return $Q$ with | lambda $h \Rightarrow t$)”

Variable matchL : $\Lambda \rightarrow \forall Q : \text{Prop}, \((\Lambda \rightarrow \text{False}) \rightarrow Q) \rightarrow Q$.

>From these 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.

Definition application (f x : $\Lambda$) : $\text{False} :=$
matchL f $\text{False}$ (fun $h \Rightarrow h \ x$).

Definition Delta : $\Lambda :=$
lambda (fun $x : \Lambda \Rightarrow \text{application} \ x \ x$).

Definition loop : $\text{False} := \text{application} \ Delta \ Delta$.

Theorem two_is_three : $2 = 3$.
Proof.
elim loop.
Qed.

End Paradox.

This example can be seen as a formulation of Russell’s paradox in type theory associating $(\text{application} \ x \ x)$ to the formula $x \not\in x$, and $\text{Delta}$ to the set $\{x \mid x \not\in x\}$. If matchL would satisfy the reduction rule associated to case analysis, that is,

matchL (lambda $f$) $Q \ h \Rightarrow h \ f$

then the term loop 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.

Remark

In this case, the construction of a non-terminating program comes from the so-called negative occurrence of $\Lambda$ in the argument of the constructor lambda.
The reader will find in the Reference Manual a complete formal definition of the notions of *positivity condition* and *strict positivity* that an inductive definition must satisfy.

Notice that the positivity condition does not forbid us to put functional recursive arguments in the constructors.

For instance, let us consider the type of infinitely branching trees, with labels in $\mathbb{Z}$.

Require Import ZArith.

Inductive itree : Set :=
| ileaf : itree
| inode : $\mathbb{Z}$ $\rightarrow$ (nat $\rightarrow$ itree) $\rightarrow$ itree.

In this representation, the $i$-th child of a tree represented by "inode $z\ s$" is obtained by applying the function $s$ to $i$. The following definitions show how to construct a tree with a single node, a tree of height 1 and a tree of height 2:

Definition isingle l := inode l (fun i ⇒ ileaf).

Definition t1 := inode 0 (fun n ⇒ isingle (Z_of_nat n)).

Definition t2 :=
inode 0
(fun n : nat ⇒
inode (Z_of_nat n)
(fun p ⇒ isingle (Z_of_nat (n*p)))).

Let us define a preorder on infinitely branching trees. In order to compare two non-leaf trees, it is necessary to compare each of their children without taking care of the order in which they appear:

Inductive itree_le : itree $\rightarrow$ itree $\rightarrow$ Prop :=
| le_leaf : $\forall$ t, itree_le ileaf t
| le_node : $\forall$ l l' s s',
\[ Zle l l' \rightarrow \\
(\forall i, \exists j: \text{nat}, \text{itree}_\le (s i) (s' j)) \rightarrow \\
\text{itree}_\le (\text{inode } l s) (\text{inode } l' s'). \]

Notice that a call to the predicate $\text{itree}_\le$ appears as a general parameter of the inductive type $\text{ex}$ (see Sect.2.9). This kind of definition is accepted by Coq, but
may lead to some difficulties, since the induction principle automatically generated by the system is not the most appropriate (see chapter 14 of [3] for a detailed explanation).

The following definition, obtained by skolemising the proposition \( \forall i, \exists j, (\text{itree}_\le (s \ i) (s' \ j)) \) in the type of \( \text{itree}_\le \), does not present this problem:

\[
\text{Inductive itree}_\le' : \text{itree} \to \text{itree} \to \text{Prop} := \\
| \text{le_leaf'} : \forall t, \text{itree}_\le' \ \text{ileaf} t \\
| \text{le_node'} : \forall l \ l' \ s \ s' \ g, \\
\hspace{1em} \text{Zle} \ l \ l' \ \to \\
\hspace{2em} (\forall i, \text{itree}_\le' (s \ i) (s' (g \ i))) \ \to \\
\hspace{2em} \text{itree}_\le' (\text{inode} \ l \ s) (\text{inode} \ l' \ s').
\]

Another example is the type of trees of unbounded width, in which a recursive subterm \( (\text{ltree} \ A) \) instantiates the type of polymorphic lists:

\[
\text{Require Import List.}
\]

\[
\text{Inductive ltree} \ (A:\text{Set}) : \text{Set} := \\
\hspace{1em} \text{lnode} : A \to \text{list} (\text{ltree} \ A) \to \text{ltree} \ A.
\]

This declaration can be transformed adding an extra type to the definition, as was done in Section 2.10.

### 3.4.2 Impredicative Inductive Types

An inductive type \( I \) inhabiting a universe \( U \) is predicative if the introduction rules of \( I \) do not make a universal quantification on a universe containing \( U \). All the recursive types previously introduced are examples of predicative types. An example of an impredicative one is the following type:

\[
\text{Inductive prop} : \text{Prop} := \\
\hspace{1em} \text{prop_intro} : \text{Prop} \to \text{prop}.
\]

Notice that the constructor of this type can be used to inject any proposition –even itself!– into the type.

\[
\text{Check (prop_intro prop).}
\]

\[
\text{prop_intro prop}
\]

\[
: \text{prop}
\]
A careless use of such a self-contained objects may lead 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 interested reader to [1, 6].

Another example is the second order existential quantifier for propositions:

\[
\text{Inductive} \; \text{ex}_\text{Prop} \; (P : \text{Prop} \rightarrow \text{Prop}) : \text{Prop} := \\
\quad \text{exP_intro} : \forall X : \text{Prop}, P X \rightarrow \text{ex}_\text{Prop} P.
\]

Notice that predicativity on sort \(\text{Set}\) forbids us to build the following definitions.

\[
\text{Inductive} \; \text{aSet} : \text{Set} := \\
\quad \text{aSet_intro} : \text{Set} \rightarrow \text{aSet}.
\]

*User error: Large non-propositional inductive types must be in Type*

\[
\text{Inductive} \; \text{ex}_\text{Set} \; (P : \text{Set} \rightarrow \text{Prop}) : \text{Set} := \\
\quad \text{exS_intro} : \forall X : \text{Set}, P X \rightarrow \text{ex}_\text{Set} P.
\]

*User error: Large non-propositional inductive types must be in Type*

Nevertheless, one can define types like \(\text{aSet}\) and \(\text{ex}_\text{Set}\), as inhabitants of \(\text{Type}\).

\[
\text{Inductive} \; \text{ex}_\text{Set} \; (P : \text{Set} \rightarrow \text{Prop}) : \text{Type} := \\
\quad \text{exS_intro} : \forall X : \text{Set}, P X \rightarrow \text{ex}_\text{Set} P.
\]

In the following example, the inductive type \(\text{typ}\) can be defined, but the term associated with the interactive Definition of \(\text{typ_inject}\) is incompatible with \(\text{Coq}\)’s hierarchy of universes:

\[
\text{Inductive} \; \text{typ} : \text{Type} := \\
\quad \text{typ_intro} : \text{Type} \rightarrow \text{typ}.
\]

\[
\text{Definition} \; \text{typ_inject} : \text{typ}.
\]

\[
\text{split; exact typ.}
\]

*Proof completed*

\[
\text{Defined.}
\]

*Error: Universe Inconsistency.*

Abort.
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 [6]. A consequence of this restriction is that it is not possible to define the first projection of the type “ex\_Prop P”:

Check (fun (P:Prop→Prop)(p: ex\_Prop P) ⇒
match p with exP\_intro X HX ⇒ X end).

Error:
Incorrect elimination of "p" in the inductive type
"ex\_Prop", the return type has sort "Type" while it should be
"Prop"

Elimination of an inductive object of sort "Prop"
is not allowed on a predicate in sort "Type"
because proofs can be eliminated only to build proofs.

### 3.4.3 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. This mechanism is based on the classification of basic types into the universe \( \text{Set} \) of sets and the universe \( \text{Prop} \) of propositions. The objects of a type in the universe \( \text{Set} \) are considered as relevant for computation purposes. The objects of a type in \( \text{Prop} \) 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 thrown 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 or an empty type (e.g. \text{False}), 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.

### 3.4.4 Strong Case Analysis on Proofs

One could consider allowing 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
would enable one to prove that different introduction rules of $R$ construct different objects. However, this property would be 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 not allowed in Coq.

Definition comes_from_the_left (P Q:Prop)(H:P ∨ Q): Prop :=
match H with
  | or_introl p ⇒ True
  | or_intror q ⇒ False
end.

Error:
Incorrect elimination of "H" in the inductive type "or", the return type has sort "Type" while it should be "Prop"

Elimination of an inductive object of sort "Prop"
is not allowed on a predicate in sort "Type"
because proofs can be eliminated only to build proofs.

On the other hand, if we replace the proposition $P ∨ Q$ with the informative type $\{P\} + \{Q\}$, the elimination is accepted:

Definition comes_from_the_left_sumbool (P Q:Prop)(x:{P} + {Q}): Prop :=
match x with
  | left p ⇒ True
  | right q ⇒ False
end.

3.4.5 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 inductive types $R$.

<table>
<thead>
<tr>
<th>$x : R : U_2$</th>
<th>$Q : U_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$Set$</td>
</tr>
<tr>
<td></td>
<td>$Prop$</td>
</tr>
<tr>
<td></td>
<td>$Type$</td>
</tr>
<tr>
<td>$Set$</td>
<td>yes</td>
</tr>
<tr>
<td>$Prop$</td>
<td>yes</td>
</tr>
<tr>
<td>$Type$</td>
<td>yes</td>
</tr>
</tbody>
</table>

| $Set$         | yes      |
| $Prop$        | yes      |
| $Type$        | 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 techniques behind three very useful Coq tactics, called discriminate, injection 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 $S$.

Theorem S_is_not_O : $\forall$ n, $S \ n \neq 0$.

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.

Definition Is_zero (x:nat):= match x with
        | 0  ⇒ True
        | _  ⇒ False
end.

Then, we prove the following lemma:

---

6In the box indexed by $U_1 = \text{Type}$ and $U_2 = \text{Set}$, the answer “yes” takes into account the predicativity of sort $\text{Set}$. If you are working with the option “impredicative-set”, you must put in this box the condition “if $R$ is predicative”. 

34
Lemma 0_is_zero : \( \forall m, m = 0 \rightarrow Is_zero m \).
Proof.
  intros m H; subst m.

===============
Is_zero 0
simpl; trivial.
Qed.

Finally, the proof of S_is_not_0 follows by the application of the previous lemma to \( S n \).

red; intros n Hn.

\[ \begin{array}{l}
n : \text{nat} \\
Hn : S n = 0 \\
\hline
\end{array} \]

===============
False
apply 0_is_zero with (m := S n).
assumption.
Qed.

The tactic discriminate 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 [co]-inductive 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:

Theorem disc2 : \( \forall n, S (S n) \neq 1 \).
Proof.
  intros n Hn; discriminate.
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 \( H \):
Theorem disc3 : ∀ n, S (S n) = 0 → ∀ Q:Prop, Q.
Proof.
  intros n Hn Q.
  discriminate.
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, i.e., 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:

Theorem inj : ∀ n m, S n = S m → n = m.
Proof.

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

Lemma inj_pred : ∀ n m, n = m → pred n = pred m.
Proof.
  intros n m eq_n_m.
  rewrite eq_n_m.
  trivial.
Qed.

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

intros n m eq_Sn_Sm.
apply inj_pred with (n:= S n) (m := S m); assumption.
Qed.

This proof method is implemented by the tactic injection. This tactic is applied to a term \( t \) of type “ \( c t_1 \ldots t_n = c t'_1 \ldots t'_{n'} \)”, where \( c \) is some constructor of an inductive type. The tactic injection is applied as deep as possible to derive the equality of all pairs of subterms of \( t_i \) and \( t'_i \) placed in the same position. All these equalities are put as antecedents of the current goal.

Like discriminate, the tactic injection 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.
Lemma list_inject : ∀ (A:Type)(a b :A)(l l’:list A),
    a :: b :: l = b :: a :: l’ → a = b ∧ l = l’.
Proof.
intros A a b l l’ e.

  e : a :: b :: l = b :: a :: l’
  ===============
  a = b ∧ l = l’

injection e.

  ===============
  l = l’ → b = a → a = b → a = b ∧ l = l’

auto.
Qed.

4.3 Inversion Techniques

In section 3.2, we motivated the rule of dependent case analysis as a way of
internalizing the informal equalities \( n = 0 \) and \( n = S p \) 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 fre-
quently the case when 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:

Theorem not_le_Sn_0 : ∀ n:nat, ¬ (S n ≤ 0).

Intuitively, this theorem should follow by case analysis on the hypothesis \( H : (S n ≤ 0) \), because no introduction rule allows to instantiate the arguments of
\( \le \) with respectively a successor and zero. However, there is no way of capturing
this with the typing rule for 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:
Proof.
red; intros n H; case H.
2 subgoals

\( n : \text{nat} \)
\( H : S n \leq 0 \)

\[
\begin{align*}
& \text{False} \\
& \text{subgoal 2 is:} \\
& \forall m : \text{nat}, S n \leq m \rightarrow \text{False}
\end{align*}
\]

Undo.

What is necessary here is to make available the equalities “\( S n = 0 \)” and “\( S m = 0 \)” as extra hypotheses of the branches, so that the goal can be solved using the \texttt{Discriminate} tactic. In order to obtain the desired equalities as hypotheses, let us prove an auxiliary lemma, that our theorem is a corollary of:

\begin{verbatim}
Lemma not_le_Sn_0_with_constraints :
  \forall n p , S n \leq p \rightarrow p = 0 \rightarrow False.
Proof.
  intros n p H; case H .
\end{verbatim}

2 subgoals

\( n : \text{nat} \)
\( p : \text{nat} \)
\( H : S n \leq p \)

\[
\begin{align*}
& S n = 0 \rightarrow \text{False} \\
& \text{subgoal 2 is:} \\
& \forall m : \text{nat}, S n \leq m \rightarrow S m = 0 \rightarrow \text{False}
\end{align*}
\]

\begin{verbatim}
intros; discriminate.
intros; discriminate.
Qed.
\end{verbatim}

Our main theorem can now be solved by an application of this lemma:
Show.

2 subgoals

\( n : \text{nat} \)
\( p : \text{nat} \)
\( H : S \ n \leq \ p \)

\[ \text{-----------------------------} \]
\( S \ n = 0 \to \text{False} \)

subgoal 2 is:
\( \forall \ m : \text{nat}, S \ n \leq m \to S \ m = 0 \to \text{False} \)

\text{eapply not_le_Sn_0_with_constraints;} \text{ eauto}.
\text{Qed.}

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 the cases that make no sense. This proof technique is implemented by the tactic \text{inversion}. In order to prove a goal \( G \vec{q} \) from an object of type \( R \vec{t} \), this tactic automatically generates a lemma \( \forall \vec{x}.(R \vec{x}) \to \vec{x} = \vec{t} \to \vec{B} \to (G \vec{q}) \), where the list of propositions \( \vec{B} \) correspond to the subgoals that cannot be directly proven using \text{discriminate}. This lemma can either be saved for later use, or generated interactively. In this latter case, the subgoals yielded by the tactic are the hypotheses \( \vec{B} \) of the lemma. If the lemma has been stored, then the tactic “inversion ...using ...” can be used to apply it.

Let us show both techniques on our previous example:

### 4.3.1 Interactive mode

\text{Theorem not_le_Sn_0'} : \forall \ n : \text{nat}, \neg (S \ n \leq 0).
\text{Proof}.
\text{red; intros n H ; inversion H.}
\text{Qed.}

### 4.3.2 Static mode

\text{Derive Inversion le_Sn_0_inv with (} \forall \ n : \text{nat}, S \ n \leq 0).\text{)}
\text{Theorem le_Sn_0'' : \forall \ n \ p : \text{nat}, \neg S \ n \leq 0 .}
\text{Proof}.
intros n p H;
inversion H using le_Sn_0_inv.
Qed.

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

**Theorem le_reverse_rules :**
\[ \forall n m : \text{nat}, n \leq m \rightarrow n = m \lor \exists p : \text{nat}, n \leq p \land m = S p. \]

**Proof.**
intros n m H; inversion H.

2 subgoals

\[ n : \text{nat} \]
\[ m : \text{nat} \]
\[ H : n \leq m \]
\[ H0 : n = m \]

\[ \equiv \]
\[ m = m \lor (\exists p : \text{nat}, m \leq p \land m = S p) \]

**subgoal 2 is:**
\[ n = S m0 \lor (\exists p : \text{nat}, n \leq p \land S m0 = S p) \]

left; trivial.
right; exists m0; split; trivial.

**Proof completed**

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 hypotheses corresponding to the propositional equalities are not needed in this example, since the tactic does the necessary rewriting to solve the subgoals.
When the equalities are no longer needed after the inversion, it is better to use the tactic \texttt{Inversion\_clear}. This variant of the tactic clears from the context all the equalities introduced.

Restart.
\begin{verbatim}
intros n m H; inversion\_clear H.
\end{verbatim}

\begin{verbatim}
 n : nat
 m : nat

============================
m = m \lor (\exists p : nat, m \leq p \land m = S p)
\end{verbatim}

\texttt{left; trivial.}

\begin{verbatim}
 n : nat
 m : nat
 m0 : nat
 H0 : n \leq m0

============================
n = S m0 \lor (\exists p : nat, n \leq p \land S m0 = S p)
\end{verbatim}

\texttt{right; exists m0; split; trivial. Qed.}

\textbf{Exercise 4.1} Consider the following language of arithmetic expression, and its operational semantics, described by a set of rewriting rules.

\begin{verbatim}
Inductive ArithExp : Set :=
 | Zero : ArithExp
 | Succ : ArithExp \rightarrow ArithExp
 | Plus : ArithExp \rightarrow ArithExp \rightarrow ArithExp.
\end{verbatim}

\begin{verbatim}
Inductive RewriteRel : ArithExp \rightarrow ArithExp \rightarrow Prop :=
 | RewSucc : \forall e1 e2 : ArithExp,
 RewriteRel e1 e2 \rightarrow
 RewriteRel (Succ e1) (Succ e2)
 | RewPlus0 : \forall e:ArithExp,
 RewriteRel (Plus Zero e) e
\end{verbatim}
\[ RewPlusS : \forall e1 \, e2:\text{ArithExp}, \]
\[ \text{RewriteRel} \, e1 \, e2 \rightarrow \]
\[ \text{RewriteRel} \, (\text{Plus} \, (\text{Succ} \, e1) \, e2) \]
\[ (\text{Succ} \, (\text{Plus} \, e1 \, e2)). \]

1. Prove that Zero cannot be rewritten any further.

2. Prove that an expression of the form “Succ e” is always rewritten into an expression of the same form.

5 Inductive Types and Structural Induction

Elements of inductive types 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-foundedness 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. 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 (already defined in the Standard Library:)

\[
\text{Fixpoint plus (n p:nat) \{struct n\} : nat :=}
\]
\[
\begin{align*}
\text{match n with} \\
| 0 & \Rightarrow & p \\
| S m & \Rightarrow & S (\text{plus m p})
\end{align*}
\]

The definition is by structural induction on the first argument of the function. This is indicated by the “\{struct n\)” directive in the function’s header\(^7\). 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 pointed by the struct directive. In the case of the function plus, the argument m in the recursive call is a pattern variable issued from a case analysis of n. Therefore, the definition is accepted.

Notice that we could have defined the addition with structural induction on its second argument:

\(^7\)This directive is optional in the case of a function of a single argument.
Fixpoint plus' (n p:nat) {struct p} : nat :=
  match p with
  | 0 ⇒ n
  | S q ⇒ S (plus' n q)
  end.

In the following definition of addition, the second argument of plus'' grows at each recursive call. However, as the first one always decreases, the definition is sound.

Fixpoint plus'' (n p:nat) {struct n} : nat :=
  match n with
  | 0 ⇒ p
  | S m ⇒ plus'' m (S p)
  end.

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:

Fixpoint even_test (n:nat) : bool :=
  match n with
  | 0 ⇒ true
  | 1 ⇒ false
  | S (S p) ⇒ even_test p
  end.

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

Reset even_test.

Fixpoint even_test (n:nat) : bool :=
  match n with
  | 0 ⇒ true
  | S p ⇒ odd_test p
  end
with odd_test (n:nat) : bool :=
  match n
  with
  | 0 ⇒ false
  | S p ⇒ even_test p
end.

Definitions by structural induction are computed 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.

Eval simpl in even_test.

= even_test
  : nat → bool

Eval simpl in (fun x : nat ⇒ even x).

= fun x : nat ⇒ even x
  : nat → Prop

Eval simpl in (fun x : nat ⇒ plus 5 x).

= fun x : nat ⇒ S (S (S (S (S x))))

Eval simpl in (fun x : nat ⇒ even_test (plus 5 x)).

= fun x : nat ⇒ odd_test x
  : nat → bool

Eval simpl in (fun x : nat ⇒ even_test (plus x 5)).

= fun x : nat ⇒ even_test (x + 5)
  : nat → 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 the inductive
step:

Section Principle_of_Induction.
Variable $P : \text{nat} \rightarrow \text{Prop}$.
Hypothesis base_case : $P\ 0$.
Hypothesis inductive_case : $\forall\ n: \text{nat},\ P\ n \rightarrow P\ (S\ n)$.
Fixpoint nat_ind (n: nat) : (P n) :=
  match n return P n with
  | 0 => base_case
  | S m => inductive_step m (nat_ind m)
end.
End Principle_of_Induction.

As this proof principle is used very often, 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 can be used to generate an elimination combinator from
certain parameters, like the universe that the defined objects must inhabit, 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 16. We do
not display the combinators here by lack of space, but you can see them using the
Print command.

Scheme Even_induction := Minimality for even Sort Prop
with Odd_induction := Minimality for odd Sort Prop.

Theorem even_plus_four : $\forall\ n: \text{nat},\ \text{even}\ n \rightarrow \text{even}\ (4+n)$.
Proof.
  intros n H.
  elim H using Even_induction with (P0 := fun n => odd (4+n));
  simpl; repeat constructor; assumption.
Qed.

8In fact, whenever possible, Coq generates the principle $I\_\text{rect}$, then derives from it the weaker
principles $I\_\text{ind}$ and $I\_\text{rec}$. If some principle has to be defined by hand, the user may try to build
$I\_\text{rec}$ (if possible). Thanks to Coq’s conversion rule, this principle can be used directly to build
proofs and/or programs.

45
Another example of an elimination combinator is the principle of double induction on natural numbers, introduced by the following definition:

Section Principle_of_Double_Induction.
Variable P : nat → nat → Prop.
Hypothesis base_case1 : ∀ m:nat, P 0 m.
Hypothesis base_case2 : ∀ n:nat, P (S n) 0.
Hypothesis inductive_step : ∀ n m:nat, P n m → P (S n) (S m).

Fixpoint nat_double_ind (n m:nat){struct n} : P n m :=
match n, m return P n m with
| 0 , x ⇒ base_case1 x
| (S x), 0 ⇒ base_case2 x
| (S x), (S y) ⇒ inductive_step x y (nat_double_ind x y)
end.
End Principle_of_Double_Induction.

Changing the type of \( P \) into \( nat → nat → Type \), another combinator for constructing (certified) programs, \( nat\_double\_rect \), can be defined in exactly the same way. This definition is left as an exercise.

For instance the function computing the minimum of two natural numbers can be defined in the following way:

Definition min : nat → nat → nat :=
nat_double_rect (fun (x y:nat) ⇒ nat)
  (fun (x:nat) ⇒ 0)
  (fun (y:nat) ⇒ 0)
  (fun (x y r:nat) ⇒ S r).

Eval compute in (min 5 8).
= 5 : nat

5.2 Using Elimination Combinators.

The tactic apply can be used to apply one of these proof principles during the development of a proof.

Lemma not_circular : ∀ n:nat, n ≠ S n.
Proof.
  intro n.
apply nat_ind with (P:= fun n ⇒ n ≠ S n).

2 subgoals

n : nat

0 ≠ 1

subgoal 2 is:
∀ n0 : nat, n0 ≠ S n0 → S n0 ≠ S (S n0)

The tactic elim is a refinement of apply, specially designed 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 (P t) ≡ G. Then it solves the goal applying "I_ind P", where I_ind is the combinator associated to I. The different cases of the induction then appear as subgoals that remain to be solved. In the previous proof, the tactic call "apply nat_ind with (P:= fun n ⇒ n ≠ S n)" can simply be replaced with "elim n".

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:

Lemma eq_nat_dec : ∀ n p:nat, {n=p}+{n ≠ p}.
Proof.
intros n p.

Let us prove this theorem using the combinator nat_double_rect of section 5.1. The example also illustrates how elim may sometimes fail in finding a suitable abstraction P of the goal. Note that if "elim n" is used directly on the goal, the result is not the expected one.
elim n using nat_double_rect.

4 subgoals

n : nat
p : nat

∀ x : nat, {x = p} + {x ≠ p}

subgoal 2 is:
nat → {0 = p} + {0 ≠ p}

subgoal 3 is:
nat → ∀ m : nat, {m = p} + {m ≠ p} → {S m = p} + {S m ≠ p}

subgoal 4 is:
nat

The four sub-goals obtained do not correspond to the premises that would be expected for the principle nat_double_rect. The problem comes from the fact that this principle for eliminating n 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 pattern tactic. Once the right abstraction is provided, the rest of the proof is immediate:

Undo.

pattern p,n.

n : nat
p : nat

(fun n0 n1 : nat ⇒ {n1 = n0} + {n1 ≠ n0}) p n

elim n using nat_double_rect.

3 subgoals

n : nat
p : nat

---

48
∀ x : nat, {x = 0} + {x ≠ 0}

subgoal 2 is:
∀ x : nat, {0 = S x} + {0 ≠ S x}

subgoal 3 is:
∀ n0 m : nat, {m = n0} + {m ≠ n0} → {S m = S n0} + {S m ≠ S n0}

destruct x; auto.
destruct x; auto.
intros n0 m H; case H.
intro eq; rewrite eq; auto.
intro neg; right; red; injection 1; auto.
Defined.

Notice that the tactic “decide equality” 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:

Definition eq_nat_dec' : ∀ n p:nat, {n=p} + {n≠p}.
decide equality.
Defined.

Exercise 5.1 1. Define a recursive function of name nat2itree that maps any natural number n into an infinitely branching tree of height n.

2. Provide an elimination combinator for these trees.

3. Prove that the relation itree_le is a preorder (i.e. reflexive and transitive).

Exercise 5.2 Define the type of lists, and a predicate “being an ordered list” using an inductive family. Then, define the function (from n) = 0 :: 1 ... n :: nil and prove that it always generates an ordered list.

Exercise 5.3 Prove that le' n p and n ≤ p are logically equivalent for all n and p. (le' is defined in section 2.6).

5.3 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 a consequence of the well-foundedness of a 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.

\textbf{Print Acc.}

\textit{Inductive Acc} $(A : \text{Type}) (R : A \rightarrow A \rightarrow \text{Prop}) (x:A) : \text{Prop} :=$
\begin{align*}
\text{Acc_intro} & : (\forall y : A, R y x \rightarrow \text{Acc} R y) \rightarrow \text{Acc} R x
\end{align*}
\textit{For Acc: Argument A is implicit}
\textit{For Acc_intro: Arguments A, R are implicit}

\dots

This inductive predicate characterizes those elements $x$ of $A$ such that any descend-
ing $R$-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. \textit{Notice the use of parameter}
$x$ (see Section 2.6, page 11).

Consider now the problem of representing in Coq the following ML function
$\text{div}(x, y)$ on natural numbers, which computes $\lceil \frac{x}{y} \rceil$ if $y > 0$ and yields $x$ otherwise.

\begin{verbatim}
let rec div x y =
  if x = 0 then 0
  else if y = 0 then x
  else (div (x-y) y)+1;;
\end{verbatim}

The equality test on natural numbers can be implemented using the function
$\text{eq_nat_dec}$ that is defined page 47. Giving $x$ and $y$, this function yields either the
value \textit{(left p)} if there exists a proof $p : x = y$, or the value \textit{(right q)} if there exists
$q : a \neq b$. The subtraction function is already defined in the library \textit{Minus}.

Hence, direct translation of the ML function $\text{div}$ would be:

\textbf{Require Import Minus.}

\textbf{Fixpoint div} $(x y : \text{nat}) \{\text{struct} x\} : \text{nat} :=$
\begin{verbatim}
if eq_nat_dec x 0
  then 0
else if eq_nat_dec y 0
  then x
  else S (div (x-y) y).
\end{verbatim}

\textit{Error:}
Recursive definition of div is ill-formed.
In environment
div : nat → nat → nat
x : nat
y : nat
_: x ≠ 0
_: y ≠ 0

Recursive call to div has principal argument equal to
"x - y"
instead of a subterm of x

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. We would have the same problem if we had the directive \{struct y\} instead of \{struct 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 that x is accessible through the relation <. Notice that any natural number x 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.

Lemma minus_smaller_S : ∀ x y:nat, x - y < S x.
Proof.
  intros x y; pattern y, x;
  elim x using nat_double_ind.
  destruct x0; auto with arith.
  simpl; auto with arith.
  simpl; auto with arith.
Qed.

Lemma minus_smaller_positive :
∀ x y:nat, x ≠ 0 → y ≠ 0 → x - y < x.
Proof.
  destruct x; destruct y;
  ( simpl;intros; apply minus_smaller ||
    intros; absurd (0=0); auto).
Qed.

51
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 \( \text{lt} \), then so is \( x - y \).

Definition \text{minus\_decrease} : \forall \ x \ y : \text{nat}, \text{Acc \ l t \ x} \rightarrow
\begin{align*}
    \quad & x \neq 0 \rightarrow \\
    \quad & y \neq 0 \rightarrow \\
    \quad & \text{Acc \ l t \ (x-y)}.
\end{align*}

Proof.
intros \( x \ y \ H \); case \( H \).
iintros \( Hz \) posz posy.
apply \( Hz \); apply \text{minus\_smaller\_positive}; assumption.
Defined.

Let us take a look at the proof of the lemma \text{minus\_decrease}, since the way in
which it has been proven is crucial for what follows.

Print \text{minus\_decrease}.

\[
\text{minus\_decrease} = \\
\quad \text{fun (x y : nat) (H : \text{Acc \ l t x}) \Rightarrow} \\
\quad \text{match H in (\text{Acc \_ y0}) return (y0 \neq 0 \rightarrow y \neq 0 \rightarrow \text{Acc \ l t \ (y0 - y)}) with} \\
\quad \text{| \text{Acc\_intro \ z \ Hz \Rightarrow} \\
\quad \quad \text{fun (posz : z \neq 0) (posy : y \neq 0) \Rightarrow} \\
\quad \quad \quad \text{Hz (z - y) (\text{minus\_smaller\_positive \ z \ y \ posz \ posy})}}
\end{align*}

end

\begin{align*}
    \forall \ x \ y : \text{nat}, \text{Acc \ l t \ x} \rightarrow x \neq 0 \rightarrow y \neq 0 \rightarrow \text{Acc \ l t \ (x - y)}
\end{align*}

Notice that the function call \((\text{minus\_decrease \ n \ m \ H})\) indeed yields an acces-
sibility proof that is \textit{structurally smaller} than its argument \( H \), because it is (an
application of) its recursive component \( Hz \). This enables to justify the following
definition of \text{div\_aux}:

Definition \text{div\_aux} (x y: nat)(H: \text{Acc \ l t \ x}) : \text{nat}.
fix 3.
iintros.
refine (if \text{eq\_nat\_dec \ x \ 0}
then 0
else if \text{eq\_nat\_dec \ y \ 0}
then y
else \text{div\_aux \ (x-y) \ y \_}).
\[ \text{div\_aux} : \forall x : \text{nat}, \text{nat} \rightarrow \text{Acc lt} x \rightarrow \text{nat} \]
\[ x : \text{nat} \]
\[ y : \text{nat} \]
\[ H : \text{Acc lt} x \]
\[ _ : x \neq 0 \]
\[ _0 : y \neq 0 \]
\[ \text{-----------------------------} \]
\[ \text{Acc lt} (x - y) \]

```
apply (minus_decrease x y H);auto.
Defined.
```

The main division function is easily defined, using the theorem \text{lt\_wf} of the library \text{Wf\_nat}. This theorem asserts that \text{nat} is well founded w.r.t. \text{lt}, thus any natural number is accessible.

**Definition** \text{div} x y := \text{div\_aux} x y (\text{lt\_wf} x).

Let us explain the proof above. In the definition of \text{div\_aux}, what decreases is not \text{x} but the proof of the accessibility of \text{x}. 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 \text{x} \text{- y}, and is filled up with the (smaller!) accessibility proof provided by the function \text{minus\_decrease}.

Let us take a look to the term \text{div\_aux} defined:
Print div_aux.

div_aux =
(fix divAux \ (x : nat) \ (y : nat) \ (H : Acc lt x) \ (struct H) : nat :=
  match eq_nat_dec x 0 with
  | left _, ⇒ 0
  | right _, ⇒
    match eq_nat_dec y 0 with
    | left _, ⇒ y
    | right _, ⇒ divAux (x - y) y (minus_decrease x y H _ _ 0)
  end
end
  : ∀ x : nat, nat → Acc lt 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.

Extraction div.

let div x y =
  divAux x y

Extraction div_aux.

let rec divAux x y =
  match eq_nat_dec x O with
  | Left _, ⇒ O
  | Right _, ⇒
    (match eq_nat_dec y O with
    | Left _, ⇒ y
    | Right _, ⇒ divAux (minus x y) y)

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 thrown away, keeping just the desired computational behavior for it.
6 A case study in dependent elimination

Dependent types are very expressive, but ignoring some useful techniques can cause some problems to the beginner. Let us consider again the type of vectors (see section 2.2). We want to prove a quite trivial property: the only value of type "vector A 0" is "Vnil A".

Our first naive attempt leads to a cul-de-sac.

Lemma vector0_is_vnil :
\forall (A:Type)(v:vector A 0), v = Vnil A.
Proof.
intros A v; inversion v.

1 subgoal

A : Set
v : vector A 0
=================================
\forall (A:Type)(v:vector A 0), v = Vnil A.

Abort.

Another attempt is to do a case analysis on a vector of any length n, under an explicit hypothesis n = 0. The tactic discriminate will help us to get rid of the case n = S p. Unfortunately, even the statement of our lemma is refused!

Lemma vector0_is_vnil_aux :
\forall (A:Type)(n:nat)(v:vector A n), n = 0 \rightarrow v = Vnil A.

Error: In environment
A : Type
n : nat
v : vector A n
e : n = 0
The term "Vnil A" has type "vector A 0" while it is expected to have type "vector A n".

In effect, the equality "v = Vnil A" is ill-typed and this is because the type "vector A n" is not convertible with "vector A 0".

This problem can be solved if we consider the heterogeneous equality JMeq [11] which allows us to consider terms of different types, even if this equality can
Lemma vector0_is_vnil_aux :
  \forall (A:Type)(n:nat)(v:vector A n),
  n= 0 → JMeq v (Vnil A).
Proof.
destruct v.
auto.
intro; discriminate.
Qed.

Our property of vectors of null length can be easily proven:

Lemma vector0_is_vnil :
  \forall (A:Type)(v:vector A 0), v = Vnil A.
intros a v;apply JMeq_eq.
apply vector0_is_vnil_aux.
trivial.
Qed.

It is interesting to look at another proof of vector0_is_vnil, which illustrates a technique developed and used by various people (consult in the Coq-club mailing list archive the contributions by Yves Bertot, Pierre Letouzey, Laurent Théry, Jean Duprat, and Nicolas Magaud, Venanzio Capretta and Conor McBride). This technique is also used for unfolding infinite list definitions (see chapter13 of [3]). Notice that this definition does not rely on any axiom (e.g. JMeq_eq).

We first give a new definition of the identity on vectors. Before that, we make the use of constructors and selectors lighter thanks to the implicit arguments feature:

Implicit Arguments Vcons [A n].
Implicit Arguments Vnil [A].
Implicit Arguments Vhead [A n].
Implicit Arguments Vtail [A n].

Definition Vid :
  \forall (A : Type)(n:nat), vector A n → vector A n.
Proof.
destruct n; intro v.
exact Vnil.
exact (Vcons (Vhead v) (Vtail v)).
Defined.
Then we prove that $\text{Vid}$ is the identity on vectors:

Lemma $\text{Vid}_\text{eq}$ : $\forall (n:\text{nat)} (A:\text{Type})(v:\text{vector } A \ n), v = (\text{Vid } _n \ n\ v)$.
Proof.
destruct $v$.

$A : \text{Type}$

$\text{Vnil} = \text{Vid } A \ 0 \ \text{Vnil}$

subgoal 2 is:
$\text{Vcons } a \ v = \text{Vid } (S n) \ (\text{Vcons } a \ v)$

reflexivity.
reflexivity.
Defined.

Why defining a new identity function on vectors? The following dialogue shows that $\text{Vid}$ has some interesting computational properties:

Eval simpl in (fun (A:Type)(v:vector A 0) ⇒ (Vid _ _ v)).
= fun (A : Type) (_ : vector A 0) ⇒ Vnil
  : $\forall A : \text{Type}, \text{vector } A \ 0 \ \rightarrow \text{vector } A \ 0$

Notice that the plain identity on vectors doesn’t convert $v$ into $\text{Vnil}$.

Eval simpl in (fun (A:Type)(v:vector A 0) ⇒ v).
= fun (A : Type) (v : vector A 0) ⇒ v
  : $\forall A : \text{Type}, \text{vector } A \ 0 \ \rightarrow \text{vector } A \ 0$

Then we prove easily that any vector of length 0 is $\text{Vnil}$:

Theorem zero_nil : $\forall A (v:\text{vector } A \ 0), v = \text{Vnil}$.
Proof.
intros.
change ($\text{Vnil } (A :=A))$ with ($\text{Vid } _0 \ v$).

1 subgoal
\( A : \text{Type} \)
\( v : \text{vector} A \ 0 \)

\[ v \= \text{Vid} A \ 0 \ v \]

apply \text{Vid\_eq}.
Defined.

A similar result can be proven about vectors of strictly positive length\(^9\).

**Theorem decomp** :
\( \forall \ (A : \text{Type}) \ (n : \text{nat}) \ (v : \text{vector} \ A \ (S \ n)) , \)
\( v \= \text{Vcons} \ (\text{Vhead} \ v) \ (\text{Vtail} \ v) . \)

**Proof** .
intros.
change \((\text{Vcons} \ (\text{Vhead} \ v) \ (\text{Vtail} \ v))\) with \((\text{Vid} \ (S \ n) \ v)\).

1 subgoal

\( A : \text{Type} \)
\( n : \text{nat} \)
\( v : \text{vector} \ A \ (S \ n) \)

\[ v \= \text{Vid} A \ (S \ n) \ v \]

apply \text{Vid\_eq}.
Defined.

Both lemmas: \text{zero\_nil} and \text{decomp}, can be used to easily derive a double recursion principle on vectors of same length:

**Definition vector\_double\_rect** :
\( \forall \ (A: \text{Type}) \ (P : \forall \ (n:\text{nat}), (\text{vector} \ A \ n) \rightarrow (\text{vector} \ A \ n) \rightarrow \text{Type}) , \)
\( P \ 0 \ \text{Vnil} \ \text{Vnil} \rightarrow \)
\( (\forall \ n \ (v1 \ v2 : \text{vector} \ A \ n) \ a \ b , \ P \ n \ v1 \ v2 \rightarrow \)
\( P \ (S \ n) \ (\text{Vcons} \ a \ v1) \ (\text{Vcons} \ b \ v2)) \rightarrow \)
\( \forall \ n \ (v1 \ v2 : \text{vector} \ A \ n) , \ P \ n \ v1 \ v2 . \)

\(^9\)As for \text{Vid} and \text{Vid\_eq}, this definition is from Jean Duprat.
induction n.
intros; rewrite (zero_nil _ v1); rewrite (zero_nil _ v2).
auto.
intros v1 v2; rewrite (decomp _ v1); rewrite (decomp _ v2).
apply X0; auto.
Defined.

Notice that, due to the conversion rule of Coq’s type system, this function can be used directly with Prop or Type instead of type (thus it is useless to build vector_double_ind and vector_double_rec) from scratch.

We finish this example with showing how to define the bitwise or on boolean vectors of the same length, and proving a little property about this operation.

Definition bitwise_or n v1 v2 : vector bool n :=
vector_double_rect
  bool
  (fun n v1 v2 ⇒ vector bool n)
Vnil
  (fun n v1 v2 a b r ⇒ Vcons (orb a b) r) n v1 v2.

Let us define recursively the $n$-th element of a vector. Notice that it must be a partial function, in case $n$ is greater or equal than the length of the vector. Since Coq only considers total functions, the function returns a value in an option type.

  option A :=
  match n,v with
  _ , Vnil ⇒ None
  | 0 , Vcons b _ _ ⇒ Some b
  | S n′, Vcons _ p′ _ v′ ⇒ vector_nth A n′ p′ v′
  end.
Implicit Arguments vector_nth [A p].

We can now prove — using the double induction combinator — a simple property relying vector_nth and bitwise_or:

Lemma nth_bitwise :
  ∀ (n:nat) (v1 v2: vector bool n) i a b,
  vector_nth i v1 = Some a →
  vector_nth i v2 = Some b →

59
\[ \text{vector_nth } i \ (\text{bitwise_or} \ v1 \ v2) = \text{Some} \ (\text{orb} \ a \ b). \]

Proof.
\[ \text{intros } n \ v1 \ v2; \text{pattern } n,v1,v2. \]
\[ \text{apply vector_double_rect.} \]
\[ \text{simpl.} \]
\[ \text{destruct } i; \text{discriminate } 1. \]
\[ \text{destruct } i; \text{simpl;auto.} \]
\[ \text{injection } 1; \text{injection } 2; \text{intros}; \text{subst } a; \text{subst } b; \text{auto.} \]
\[ \text{Qed.} \]

7 Co-inductive Types and Non-ending Constructions

The objects of an inductive type are well-founded with respect to the constructors of the type. In other words, these objects are built by applying a finite number of times the constructors of the type. Co-inductive types are obtained by relaxing this condition, and may contain non-well-founded objects \([10, 9]\). An example of a co-inductive type is the type of infinite sequences formed with elements of type \(A\), also called streams. This type can be introduced through the following definition:

\[
\text{CoInductive Stream } (A: \text{Type}) : \text{Type } := \\
| \text{Cons} : A \rightarrow \text{Stream } A \rightarrow \text{Stream } A.
\]

If we are interested in finite or infinite sequences, we consider the type of lazy lists:

\[
\text{CoInductive LList } (A: \text{Type}) : \text{Type } := \\
| \text{LNil} : \text{LList } A \\
| \text{LCons} : A \rightarrow \text{LList } A \rightarrow \text{LList } A.
\]

It is also possible to define co-inductive types for the trees with infinitely-many branches (see Chapter 13 of [3]). 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 \textit{head} and \textit{tail}.

\[
\text{Definition head } (A: \text{Type})(s : \text{Stream } A) : = \\
\text{match } s \text{ with } \text{Cons } a \ s' \Rightarrow a \text{ end}. \\
\text{Definition tail } (A : \text{Type})(s : \text{Stream } A) : = \\
\text{match } s \text{ with } \text{Cons } a \ s' \Rightarrow s' \text{ end}. \\
\]
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. For example, the following definition introduces the infinite list \([a, a, a, \ldots]\):

\[
\text{CoFixpoint repeat } (A: \text{Type})(a: A) : \text{Stream } A :=
\text{Cons } a \text{ (repeat } a) .
\]

However, not every co-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 [8]. The following definitions are examples of valid methods of construction:

\[
\text{CoFixpoint iterate } (A: \text{Type})(f: A \rightarrow A)(a : A) : \text{Stream } A :=
\text{Cons } a \text{ (iterate } f \text{ (f } a) .
\]

\[
\text{CoFixpoint map } (A B: \text{Type})(f: A \rightarrow B)(s : \text{Stream } A) : \text{Stream } B :=
\text{match } s \text{ with } \text{Cons } a \text{ tl } \Rightarrow \text{Cons } (f a) \text{ (map } f \text{ tl) end.}
\]

**Exercise 7.1** Define two different methods for constructing the stream which infinitely alternates the values **true** and **false**.

**Exercise 7.2** Using the destructors **head** and **tail**, 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.

\[
\text{Eval simpl in (fun } (A: \text{Type})(a: A) \Rightarrow \text{repeat } a).\]

\[
= \text{fun } (A : \text{Type}) (a : A) \Rightarrow \text{repeat } a
\]

\[
: \forall A : \text{Type}, A \rightarrow \text{Stream } A
\]

\[
\text{Eval simpl in (fun } (A: \text{Type})(a: A) \Rightarrow \text{head } (\text{repeat } a)).\]

\[
= \text{fun } (A : \text{Type}) (a : A) \Rightarrow a
\]

\[
: \forall A : \text{Type}, A \rightarrow A
\]
7.1 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 [9]. 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 \( \text{iterate } f \ (f \ x) \) and \( \text{map } f \ (\text{iterate } f \ x) \). 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:

\[
\text{CoInductive EqSt } (A : \text{Type}) : \text{Stream } A \rightarrow \text{Stream } A \rightarrow \text{Prop} := \\
eq \text{st} : \forall s_1 \ s_2 : \text{Stream } A, \\
\quad \text{head } s_1 = \text{head } s_2 \rightarrow \\
\quad \text{EqSt } (\text{tail } s_1) (\text{tail } s_2) \rightarrow \\
\quad \text{EqSt } s_1 \ s_2.
\]

It is possible to introduce proof principles for reasoning about infinite objects as combinators defined through \texttt{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 \( s_1 \) and \( s_2 \) satisfying \( R \) have equal heads, and tails also satisfying \( R \). This principle is in fact a method for constructing an infinite proof:

\[
\text{Section Parks_Principle.} \\
\text{Variable } A : \text{Type.} \\
\text{Variable } R : \text{Stream } A \rightarrow \text{Stream } A \rightarrow \text{Prop.} \\
\text{Hypothesis bisim1 : } \forall s_1 \ s_2 : \text{Stream } A, \\
\quad \text{R } s_1 \ s_2 \rightarrow \text{head } s_1 = \text{head } s_2. \\
\text{Hypothesis bisim2 : } \forall s_1 \ s_2 : \text{Stream } A, \\
\quad \text{R } s_1 \ s_2 \rightarrow \text{R } (\text{tail } s_1) (\text{tail } s_2). \\
\text{CoFixpoint park_ppl :} \\
\forall s_1 \ s_2 : \text{Stream } A, \text{R } s_1 \ s_2 \rightarrow \text{EqSt } s_1 \ s_2 :=
\]
fun s1 s2 (p : R s1 s2) ⇒
  eqst s1 s2 (bisim1 s1 s2 p)
  (park_ppl (tail s1)
    (tail s2)
    (bisim2 s1 s2 p)).

End Parks_Principle.

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

Theorem map_iterate :
  ∀ (A:Type)(f:A→A)(x:A),
  EqSt (iterate f (f x))
    (map f (iterate f x)).

Proof.
  intros A f x.
  apply park_ppl with
    (R:= fun s1 s2 ⇒
      ∃ x: A, s1 = iterate f (f x) ∧
      s2 = map f (iterate f x)).

  intros s1 s2 (x0,(eqs1,eqs2));
  rewrite eqs1; rewrite eqs2; reflexivity.
  intros s1 s2 (x0,(eqs1,eqs2)).
  exists (f x0);split;
    [rewrite eqs1|rewrite eqs2]; reflexivity.
  exists x;split; reflexivity.
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 indicated in Section 5.3, 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:

Ltac infiniteproof f :=
  cofix f;
  constructor;
  [clear f| simpl; try (apply f; clear f)].
In the example above, this tactic produces a much simpler proof that the former one:

\[ \text{Theorem } \text{map\_iterate'}: \forall (A : \text{Type}) (f : A \to A) (x : A), \]
\[ \text{EqSt} (\text{iterate } f (f x)) = (\text{map } f (\text{iterate } f x)). \]

Proof.
  \text{infiniteproof map\_iterate'}.  
  \text{reflexivity}.  
  \text{Qed}.

**Exercise 7.3** Define a co-inductive type of name `Nat` that contains non-standard natural numbers—this is, verifying

\[ \exists m \in \text{Nat}, \forall n \in \text{Nat}, n < m. \]

**Exercise 7.4** Prove that the extensional equality of streams is an equivalence relation using Park’s co-induction principle.

**Exercise 7.5** 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 5.2.

### 7.2 About injection, discriminate, and inversion

Since co-inductive types are closed w.r.t. their constructors, the techniques shown in Section 4 work also with these types.

Let us consider the type of lazy lists, introduced on page 60. The following lemmas are straightforward applications of discriminate and injection:

**Lemma** `Lnil_not_Lcons` : \( \forall (A : \text{Type}) (a : A) (l : \text{LList } A), \)
\[ \text{LNil} \neq (\text{LCons } a \ l). \]

Proof.
  \text{intros;discriminate}.  
  \text{Qed}.

**Lemma** `injection_demo` : \( \forall (A : \text{Type}) (a \ b : A) (l \ l' : \text{LList } A), \)
\[ \text{LCons } a \ (\text{LCons } b \ l) = \text{LCons } b \ (\text{LCons } a \ l') \to a = b \land l = l'. \]
Proof.
  intros A a b l l’ e; injection e; auto.
Qed.

In order to show inversion at work, let us define two predicates on lazy lists:

InductiveFinite (A:Type) : LList A → Prop :=
  | Lnil_fin : Finite LNil (A:=A))
  | Lcons_fin : ∀ a l, Finite l → Finite (LCons a l).

CoInductive Infinite (A:Type) : LList A → Prop :=
  | LCons_inf : ∀ a l, Infinite l → Infinite (LCons a l).

First, two easy theorems:

Lemma LNil_not_Infinite :
  ∀ (A:Type), ~ Infinite (LNil (A:=A)).
Proof.
  intros A H; inversion H.
Qed.

Lemma Finite_not_Infinite :
  ∀ (A:Type)(l:LList A),
  Finite l → ~ Infinite l.
Proof.
  intros A l H; elim H.
  apply LNil_not_Infinite.
  intros a l0 F0 I0’ I1.
  case I0’; inversion_clear I1.
  trivial.
Qed.

On the other hand, the next proof uses the cofix tactic. Notice the deconstruction of l, which allows us to apply the constructor LCons_inf, thus satisfying the guard condition:

Lemma Not_Finite_Infinite :
  ∀ (A:Type)(l:LList A),
  ~ Finite l → Infinite l.
Proof.
  cofix H.
  destruct l.
  intro;
absurd (Finite (LNil (A:=A)));  [auto|constructor].

1 subgoal

H : forall (A : Type) (l : LList A), ~ Finite l -> Infinite l
A : Type
a : A
l : LList A
H0 : ~ Finite (LCons a l)

================
Infinite l

At this point, one must not apply H! . It would be possible to solve the current goal by an inversion of “Finite (LCons a l)”, but, since the guard condition would be violated, the user would get an error message after typing Qed. In order to satisfy the guard condition, we apply the constructor of Infinite, then apply H.

constructor.
apply H.
red; intro H1; case H0.
constructor.
trivial.
Qed.

The reader is invited to replay this proof and understand each of its steps.

References


66


