Coq in a Hurry

Yves Bertot

June 30, 2005

These notes provide a quick introduction to the Coq system and show how it can be used to define logical concepts and functions and reason about them. It is designed as a tutorial, so that readers can quickly start their own experiments, learning only a few of the capabilities of the system. A much more comprehensive study is provided in [1], which also provides an extensive collection of exercises to train on.

1 Expressions and logical formulas

The Coq system provides a language in which one handles formulas, verify that they are well-formed, and prove them. Formulas may also contain functions and limited forms of computations are provided for these functions.

The first thing you need to know is how you can check whether a formula is well-formed. The command is called `Check`. Here are a few examples, which use a few of the basic objects and types of the system.

```
Check True.
True : Prop

Check False.
False : Prop

Check 3.
3 : nat

Check (3+4).
3 + 4 : nat

Check (3=5).
3=5 : Prop

Check (3,4).
(3,4) : nat * nat
```

```
Check ((3=5)/\True).
3 = 5 /\ True

Check nat -> Prop.
nat -> Prop : Type

Check (3 <= 6).
3 <= 6 : Prop
```

The notation A :B is uniformly used to indicate that the type of the expression A is the expression B.

Among these formulas, some can be read as propositions (they have type `Prop`), others may be read as numbers (they have type `nat`), others may be read as elements of more complex data structures. You can also try to check badly formed formulas and in this case the Coq system returns an informative error statement.

Complex formulas can be constructed by combining propositions with logical connectives, or other expressions with addition, multiplication, the pairing construct, and so on. You can also construct a new function by using the keyword `fun`, which replaces the λ symbol of lambda calculus and similar theories.

```
Check (fun x:nat => x = 3).
fun x : nat => x = 3 : nat -> Prop

Check (forall x:nat, x < 3 \/ (exists y:nat, x = y + 3)).
forall x : nat, x < 3 \/ (exists y : nat, x = y + 3) : Prop

Check (let f := fun x => (x * 3,x) in f 3).
let f := fun x : nat => (x * 3, x) in f 3 : nat * nat
```

Please note that some notations are *overloaded.* For instance, the `*` sign is used both to represent conventional multiplication on numbers and the cartesian product on types. One can find the function hidden behind a notation by using the `Locate` command.

```
Locate "_ <= _".
Notation               Scope
"x <= y" := le x y     : nat_scope
                         (default interpretation)
```

The conditions for terms to be well-formed have two origins: first, the syntax must be respected (parentheses, keywords, binary operators must have two arguments); second, expressions must respect a type discipline. The `Check` not only checks that expressions are well-formed but it also gives the type of expressions. For instance we can use the `Check` command to verify progressively that some expressions are well-formed.

```
Check True.
True : Prop
```

```
Check False.
False : Prop

Check and.
and : Prop -> Prop -> Prop

Check (and True False).
True /\ False : Prop
```

In the last example, `and` is a function that expects an argument of type `Prop` and returns a function of type `Prop -> Prop`. It can therefore be applied to `True`, which has the right type. But the function we obtain expects another argument of type `Prop` and it can be applied to the argument `False`. The notation

$$a \texttt{ -> } b \texttt{ -> } c$$

actually stands for

$$a \texttt{ -> } (b \texttt{ -> } c)$$

and the notation $f\ a\ b$ actually stands for $(f\ a)\ b$.

The last example also shows that the notation `/\` is an infix notation for the function `and`.

Some constructs of the language have a notion of *bound* variable. Among the examples we have already seen, the `forall` and `exist` logical quantifiers and the `fun` function constructor and the `let .. in` local declaration construct have this characteristic. When constructs have a bound variable, this variable can be used with some type inside some part of the construct called the scope. The type is usually given explicitely, but it may also sometimes be left untold, and the Coq system will infer it.

2 Defining new constants

You can define a new constant by using the keyword `Definition`. Here is an example:

```
Definition example1 := fun x : nat => x*x+2*x+1.
```

An alternative, exactly equivalent, definition could be:

```
Definition example1 (x : nat) := x*x+2*x+1.
```

3 Proving facts

The notation A:B is actually used for several purposes in the Coq system. One of these purposes is to express that A is a proof for the logical formula B. This habit is usually referred to under the name *Curry-Howard Isomorphism.* One can find already existing proofs of facts by using the `Search` command.

```
Search True.
I : True
Search le.
le_n : forall n : nat, n <= n
le_S : forall n m : nat, le n m -> le n (S m)
```

The theorem `le_S` uses a function `S`, this function maps any natural number to its successor. Actually, the notation `3` is only a notation for `S (S (S O))`.

New theorems can be loaded from already proven packages using the `Require` command. For example, for proofs in arithmetics, it is useful to load the following packages:

```
Require Import Arith.
Require Import ArithRing.
Require Import Omega.

Search le.
between_le: forall (P : nat -> Prop) (k l : nat),
   between P k l -> k <= l
exists_le_S:
  forall (Q : nat -> Prop) (k l : nat),
      exists_between Q k l -> S k <= l

...
plus_le_reg_l: forall n m p : nat, p + n <= p + m -> n <= m
plus_le_compat_l: forall n m p : nat, n <= m -> p + n <= p + m
plus_le_compat_r: forall n m p : nat, n <= m -> n + p <= m + p
le_plus_l: forall n m : nat, n <= n + m

...
```

There is a real notion of *false* formulas, but it is only expressed by saying that the existence of a proof for a false formula would imply the existence of a proof for the formula `False`.

A theorem that proves an implication actually is an expression whose type is a function type (in other words, an arrow type). Thus, it can be applied to a term whose type appears to the left of the arrow. From the logical point of view, the argument of this theorem should be a proof of the implication's premise. This corresponds to what is usually known as *modus ponens*.

A theorem that proves a universal quantification is also an expression whose type is a function type. It can also be applied to an argument of the right type. From the logical point of view, this corresponds to producing a new proof for the statement where the universal quantification is instantiated. Here are a few examples, where theorems are instantiated and a modus ponens inference is performed:

```
Check (le_n 0).
le_n 0 : 0 <= 0
```

```
Check (le_S 0 0).
le_S 0 1 : 0 <= 0 -> 0 <= 1

Check (le_S 0 0 (le_n 0)).
le_S 0 0 (le_n O) : 0 <= 1
```

New theorems could be constructed this way by combining existing theorems and using the `Definition` keyword to associate these expressions to constants. But this approach is seldom used. The alternative approach is known as *goal directed proof*, with the following type of scenario:

1. the user enters a statement that he wants to prove, using the command `Theorem` or `Lemma`,
2. the Coq system displays the formula as a formula to be proved, possibly giving a context of local facts that can be used for this proof (the context is displayed above a horizontal line, the goal is displayed under the horizontal line),
3. the user enters a command to decompose the goal into simpler ones,
4. the Coq system displays a list of formulas that still need to be proved,
5. back to step 3.

Some of the commands sent at step 3 actually decrease the number of goals. When there are no more goals the proof is complete, it needs to be saved, this is performed when the user sends the command `Qed.` The commands that are especially designed to decompose goals into lists of simpler goals are called *tactics*.

Here is an example:

```
Theorem example2 : forall a b:Prop, a /\ b -> b /\ a.
1 subgoal

  ============================
   forall a b : Prop, a /\ b -> b /\ a

Proof.
 intros a b H.
1 subgoal

  a : Prop
  b : Prop
  H : a /\ b
  ============================
   b /\ a

split.
```

```
2 subgoals
...
  H : a /\ b
  ============================
   b

subgoal 2 is:
 a

elim H; intros H0 H1.
...
  H0 : a
  H1 : b
  ============================
   b

exact H1.
1 subgoal
...
  H : a /\ b
  ============================
   a

intuition.
Proof completed.

Qed.
intros a b H.
split.
 elim H; intros H0 H1.
   exact H1.
 intuition.
example2 is defined
```

This proof uses several steps to decompose the logical processing, but a quicker dialog would simply rely on the `intuition` tactic directly from the start. There is an important collection of tactics in the Coq system, each of which is adapted to a shape of goal. For instance, the tactic `elim H` was adapted because the hypothesis `H` was a proof of a conjunction of two propositions. The effect of the tactic was to add two implications in front of the goal, with two premises asserting one of the propositions in the conjunction. It is worthwhile remembering a collection of tactics for the basic logical connectives. We list these tactics in the following table, inspired from the table in [1] (p. 130).

	$\Rightarrow$	$\forall$	$\wedge$	$\vee$	$\exists$
Hypothesis	`apply`	`apply`	`elim`	`elim`	`elim`
goal	`intros`	`intros`	`split`	`left` or `right`	`exists` v
	$\neg$	$=$			
Hypothesis	`elim`	`rewrite`			
goal	`intro`	`reflexivity`			

When using the tactic `elim`, this usually creates new facts that are placed in the result goal as premises of newly created implications. These premises must then be introduced in the context using the `intros` tactic. A quicker tactic does the two operations at once, this tactic is called `destruct`.

Some automatic tactics are also provided for a variety of purposes, `intuition` is often useful to prove facts that are tautologies in first-order intuitionistic logic (try it whenever the proof only involves manipulations of forall quantification, conjunction, disjunction, and negation); `auto` is an extensible tactic that tries to apply a collection of theorems that were provided beforehand by the user, `eauto` is like `auto`, it is more powerful but also more time-consuming, `ring` and `ring_nat` mostly do proofs of equality for expressions containing addition and multiplication (and `S` for `ring_nat`), `omega` proves formulas in Presburger arithmetic.

One of the difficult points for newcomers is that the Coq system also provides a type `bool` with two elements called `true` and `false`. Actually this type has nothing to do with truth or provability, it is just a two-element type that may be used to model the boolean types that one usually finds in programming languages and the value attached to its two elements is purely conventional. In a sense, it is completely correct (but philosophically debatable) to define a function named `is_zero` that takes an argument of type `nat` and returns `false` if, and only if, its argument is 0.

4 Inductive types

Inductive types could also be called algebraic types or initial algebras. They are defined by providing the type name, its type, and a collection of constructors. Inductive types can be parameterized and dependent. We will mostly use parameterization to represent polymorphism and dependence to represent logical properties.

4.1 Defining inductive types

Here is an example of an inductive type definition:

```
Inductive bin : Set :=
  L : bin
| N : bin -> bin -> bin.
```

This defines a new type `bin`, whose type is `Set`, and provides two ways to construct elements of this type: `L` (a constant) and `N` (a function taking two arguments). The Coq system automatically associates a theorem to this inductive type. This theorem makes it possible to reason by induction on elements of this type:

```
Check bin_ind.
bin_ind
     : forall P : bin -> Prop,
       P L ->
       (forall b : bin,
         P b -> forall b0 : bin, P b0 -> P (N b b0)) ->
       forall b : bin, P b
```

The induction theorem associated to an inductive type is always named *name*`_ind`.

4.2 Pattern matching

Elements of inductive types can be processed using functions that perform some pattern-matching. For instance, we can write a function that returns the boolean value `false` when its argument is `N L L` and returns `true` otherwise.

```
Definition example3 (t : bin): bool :=
  match t with N L L => false | _ => true end.
```

4.3 Recursive function definition

There are an infinity of different trees in the type `bin`. To write interesting functions with arguments from this type, we need more than just pattern matching. The Coq system provides recursive programming. The shape of recursive function definitions is as follows:

```
Fixpoint flatten_aux (t1 t2:bin) {struct t1} : bin :=
  match t1 with
     L => N L t2
  | N t'1 t'2 => flatten_aux t'1 (flatten_aux t'2 t2)
  end.

Fixpoint flatten (t:bin) : bin :=
  match t with
    L => L
  | N t1 t2 => flatten_aux t1 (flatten t2)
  end.

Fixpoint size (t:bin) : nat :=
  match t with
    L => 1
  | N t1 t2 => 1 + size t1 + size t2
  end.
```

There are constraints in the definition of recursive definitions. First, if the function has more than one argument, we must declare one of these arguments as the *principal* or *structural* argument (we did this declaration for the function `flatten_aux`). If there is only one argument, then this argument is the principal argument by default. Second, every recursive call must be performed so that the principal argument of the recursive call is a subterm, obtained by pattern-matching, of the initial principal argument. This condition is satisfied in all the examples above.

4.4 Proof by cases

Now, that we have defined functions on our inductive type, we can prove properties of these functions. Here is a first example, where we perform a few case analyses on the elements of the type `bin`.

```
Theorem example3_size :
   forall t, example3 t = false -> size t = 3.
Proof.
intros t; destruct t.
2 subgoals

  ============================
   example3 L = false -> size L = 3

subgoal 2 is:
 example3 (N t1 t2) = false -> size (N t1 t2) = 3
```

The tactic `destruct t` actually observes the various possible cases for `t` according to the inductive type definition. The term `t` can only be either obtained by `L`, or obtained by `N` applied to two other trees `t1` and `t2`. This is the reason why there are two subgoals.

We know the value that `example3` and `size` should take for the tree `L`. We can direct the Coq system to compute it:

```
simpl.
2 subgoals

  ============================
   true = false -> 1 = 3
```

After computation, we dicover that assuming that the tree is `L` and that the value of `example3` for this tree is `false` leads to an inconsistency. We can use the following tactics to exploit this kind of inconsistency:

```
intros H.
  H : true = false
  ============================
   1 = 3
discriminate H.
1 subgoal
...
  ============================
   example3 (N t1 t2) = false -> size (N t1 t2) = 3
```

The answer shows that the first goal was solved. The tactic `discriminate H` can be used whenever the hypothesis `H` is an assumption that asserts that two different constructors of an inductive type return equal values. Such an assumption is inconsistent and the tactic exploits directly this inconsistency to express that the case described in this goal can never happen. This tactic expresses a basic property of inductive types in the sort `Set` or `Type`: constructors have distinct ranges. Another important property of constructors of inductive types in the sort `Set` or `Type` is that they are injective. The tactic to exploit this fact called `injection` (for more details about `discriminate` and `injection` please refer to [1] or the Coq reference manual [2]).

For the second goal we still must do a case analysis on the values of `t1` and `t2`, we do not detail the proof but it can be completed with the following sequence of tactics.

```
destruct t1.
destruct t2.
3 subgoals

  ============================
   example3 (N L L) = false -> size (N L L) = 3

subgoal 2 is:
 example3 (N L (N t2_1 t2_2)) = false ->
 size (N L (N t2_1 t2_2)) = 3
subgoal 3 is:
 example3 (N (N t1_1 t1_2) t2) = false ->
  size (N (N t1_1 t1_2) t2) = 3
```

For the first goal, we know that both functions will compute as we stated by the equality's right hand side. We can solve this goal easily, for instance with `auto`. The last two goals are solved in the same manner as the very first one, because `example3` cannot possibly have the value `false` for the arguments that are given in these goals.

```
auto.
intros H; discriminate H.
intros H; discriminate H.
Qed.
```

To perform this proof, we have simply observed 5 cases. For general recursive functions, just observing a finite number of cases is not sufficient. We need to perform proofs by induction.

4.5 Proof by induction

The most general kind of proof that one can perform on inductive types is proof by induction.

When we prove a property of the elements of an inductive type using a proof by induction, we actually consider a case for each constructor, as we did for proofs by cases. However there is a twist: when we consider a constructor that has arguments of the inductive type, we can assume that the property we want to establish holds for each of these arguments.

When we do goal directed proof, the induction principle is invoked by the `elim` tactic. To illustrate this tactic, we will prove a simple fact about the `flatten_aux` and `size` functions.

```
Theorem forall_aux_size :
 forall t1 t2, size (flatten_aux t1 t2) = size t1 + size t2 + 1.
Proof.
 intros t1; elim t1.
  ============================
   forall t2 : bin, size (flatten_aux L t2) = size L + size t2 + 1

subgoal 2 is:
 forall b : bin,
 (forall t2 : bin, size (flatten_aux b t2) =
                   size b + size t2 + 1) ->
 forall b0 : bin,
 (forall t2 : bin, size (flatten_aux b0 t2) =
                   size b0 + size t2 + 1) ->
 forall t2 : bin,
 size (flatten_aux (N b b0) t2) =
 size (N b b0) + size t2 + 1
```

There are two subgoals, the first goal requires that we prove the property when the first argument of `flatten_aux` is `L`, the second one requires that we prove the property when the argument is `N b b0`, under the assumption that it is already true for `b` and `b0`. The proof progresses easily, using the definitions of the two functions, which are expanded when the Coq system executes the `simpl` tactic. We then obtain expressions that can be solved using the `ring_nat` tactic.

```
intros t2.
simpl.
...
  ============================
   S (S (size t2)) = S (size t2 + 1)
```

```
ring_nat.
intros b IHb b0 IHb0 t2.
...
  IHb : forall t2 : bin, size (flatten_aux b t2) =
                               size b + size t2 + 1
  b0 : bin
  IHb0 : forall t2 : bin, size (flatten_aux b0 t2) =
                        size b0 + size t2 + 1
  t2 : bin
  ============================
   size (flatten_aux (N b b0) t2) = size (N b b0) + size t2 + 1

simpl.
...
  ============================
   size (flatten_aux b (flatten_aux b0 t2)) =
   S (size b + size b0 + size t2 + 1)

rewrite IHb.
...
  ============================
   size b + size (flatten_aux b0 t2) + 1 =
   S (size b + size b0 + size t2 + 1)

rewrite IHb0.
...
  ============================
   size b + (size b0 + size t2 + 1) + 1 =
   S (size b + size b0 + size t2 + 1)

ring_nat.
Proof completed.
Qed.
```

4.6 Numbers in the Coq system

In the Coq system, most usual datatypes are represented as inductive types and packages provide a variety of properties, functions, and theorems around these datatypes. The package named `Arith` contains a host of theorems about natural numbers (numbers from 0 to infinity), which are described as an inductive type with `O` (representing 0) and `S` as constructors. It also provides addition, multiplication, subtraction (with the special behavior that `x - y` is 0 when `x` is smaller than `y`. The package `ArithRing` contains the tactic `ring_nat` and the associated theorems.

The package named `ZArith` provides two inductive datatypes to represent integers. The first inductive type, named `positive`, follows a binary representation to model the positive integers (from 1 to infinity) and the type `Z` is described as a type with three constructors, one for positive numbers, one for negative numbers, and one for 0. The package also provides orders and basic operations: addition, subtraction, multiplication, division, square root. The package `ZArithRing` provides the configuration for the `ring` tactic to work on polynomial equalities with numbers of type `Z`. The tactic `omega` works equally well to solve problems in Presburger arithmetic for both natural numbers of type `nat` and integers of type `Z`.

There are a few packages that provide descriptions of rational numbers, but the support for these packages is not as standard as for the other one. For instance, there is no easy notation to enter a simple fraction. In a paper from 2001, I showed that the rational numbers could be given a very simple inductive structure, but encoding the various basic operations on this structured was a little complex.

The support for computation on real numbers in the Coq system is also quite good, but real numbers are not (and cannot) be represented using an inductive type. The package to load is `Reals` and it provides descriptions for quite a few functions, up to trigonomic functions for instance.

4.7 Data-structures

The two-element boolean type is an inductive type in the Coq system, `true` and `false` are its constructors. The induction principle naturally expresses that this type only has two elements, because it suffices that a property is satisfied by `true` and `false` to ensure that it is satisfied by all elements of `bool`. On the other hand, it is easy to prove that `true` and `false` are distinct.

Most ways to structure data together are also provided using inductive data structures. The pairing construct actually is an inductive type, and `elim` can be used to reason about a pair in the same manner as it can be used to reason on a natural number.

A commonly used datatype is the type of lists. This type is polymorphic, in the sense that the same inductive type can be used for lists of natural numbers, lists of boolean values, or lists of other lists. This type is not provided by default in the Coq system, it is necessary to load the package `List` using the `Require` command to have access to it. The usual `cons` function is given in Coq as a three argument function, where the first argument is the type of elements: it is the type of the second argument and the third argument should be a list of elements of this type, too. The empty list is represented by a function `nil`. This function also takes an argument, which should be a type. However, the Coq system also provides a notion of implicit arguments, so that the type arguments are almost never written and the Coq system infers them from the context or the other arguments. For instance, here is how we construct a list of natural numbers.

```
Require Import List.

Check (cons 3 (cons 2 (cons 1 nil))).
3 :: 2 :: 1 :: nil : list nat
```

This example also shows that the notation `::` is used to represent the cons function in an infix fashion. This tradition will be very comfortable to programmers accustomed to languages in the ML family, but Haskell addicts should beware that the conventions, between type information and cons, are inverse to the conventions in Haskell.

The `List` package also provides a list concatenation function named `app`, with `++` as infix notation, and a few theorems about this function.

5 Inductive properties

Inductive types can be dependent and when they are, they can be used to express logical properties. When defining inductive types like `nat`, `Z` or `bool` we declare that this constant is a type. When defining a dependent type, we actually introduce a new constant which is declared to be a function from some input type to a type of types. Here is an example:

```
Inductive even : nat -> Prop :=
  even0 : even 0
| evenS : forall x:nat, even x -> even (S (S x)).
```

Thus, `even` itself is not a type, it is `even x`, whenever `x` is an integer that is a type. In other words, we actually defined a family of types. In this family, not all members contain elements. For instance, we know that the type `even 0` contains an element, this element is `even0`, and we know that `even 2` contains an element: `evenS 0 even0`. What about `even 1`? If `even 1` contains an element, this element cannot be obtained using `even0` because $0 \neq 1$, it cannot be obtained using `evenS` because $1 \neq 2 + x$ for every x such that $0 \leq x$. Pushing our study of `even` further we could see that this type, seen as a property, is provable if and only if its argument is even, in the common mathematical sense.

Like other inductive types, inductive properties are equipped with an induction principle, which we can use to perform proofs. The inductive principle for `even` has the following shape.

```
Check even_ind.
even_ind : forall P : nat -> Prop,
       P 0 ->
       (forall x : nat, even x -> P x -> P (S (S x))) ->
       forall n : nat, even n -> P n
```

This principle intuitively expresses that `even` is the smallest property satisfying the two constructors: it implies every other property that also satisfies them. A proof using this induction principle will work on a goal where we know that `even y` holds for some `y` and actually decomposes into a proof of two cases, one corresponding to the case where `even y` was obtained using `even0` and one corresponding to the case where `even y` was obtained using `evenS`, applied to some `x` such that `y=S (S x)` and some proof of `even y`. In the second case, we again have the opportunity to use an induction hypothesis about this `y`.

When a variable `x` satisfies an inductive property, it is often more efficient to prove properties about this variable using an induction on the inductive property than an induction on the variable itself. The following proof is an example:

```
Theorem even_mult : forall x, even x -> exists y, x = 2*y.
Proof.
intros x H; elim H.
2 subgoals

  x : nat
  H : even x
  ============================
   exists y : nat, 0 = 2 * y

subgoal 2 is:
 forall x0 : nat,
 even x0 -> (exists y : nat, x0 = 2 * y) ->
 exists y : nat, S (S x0) = 2 * y

exists 0; ring_nat.
intros x0 Hevenx0 IHx.
...
  IHx : exists y : nat, x0 = 2 * y
  ============================
   exists y : nat, S (S x0) = 2 * y
```

In the last goal, `IHx` is the induction hypothesis. It says that if `x0` is the predecessor's predecessor of `x` then we already know that there exists a value `y` that is its half. We can use this value to provide the half of `S (S x0)`. Here are the tactics that complete the proof.

```
destruct IHx as [y Heq]; rewrite Heq.
exists (S y); ring_nat.
Qed.
```

In this example, we used a variant of the `destruct` tactic that makes it possible to choose the name of the elements that `destruct` creates and introduces in the context.

If we wanted to prove the same property using a direct induction on the natural number that is even, we would have a problem because the predecessor of an even number, for which direct induction provides an induction hypothesis, is not even. The proof is not impossible but slightly more complex:

```
Theorem even_mult' : forall x, even x -> exists y, x = 2* y.
Proof.
intros x.
assert (lemma: (even x -> exists y, x=2*y)/\
        (even (S x) -> exists y, S x=2*y)).
elim x.
split.
exists 0; ring_nat.
```

```
intros Heven1; inversion Heven1.
intros x0 IHx0; destruct IHx0 as [IHx0 IHSx0].
split.
exact IHSx0.
intros HevenSSx0.
assert (Hevenx0 : even x0).
inversion HevenSSx0; assumption.
destruct (IHx0 Hevenx0) as [y Heq].
rewrite Heq; exists (S y); ring_nat.
intuition.
Qed.
```

This script mostly uses tactics that we have already introduced, except the `inversion` tactic. Given an assumption `H` that relies on a dependent inductive type, most frequently an inductive proposition, the tactic `inversion` analyses all the constructors of the inductive, discards the ones that could not have been applied, and when some constructors could have applied it creates a new goal where the premises of this constructor are added in the context. For instance, this tactic is perfectly suited to prove that 1 is not even:

```
Theorem not_even_1 : ~even 1.
Proof.
intros even1.
...
  even1 : even 1
  ============================
   False

inversion even1.
Qed.
```

This example also shows that the negation of a fact actually is represented by a function that says "this fact implies `False`".

Inductive properties can be used to express very complex notions. For instance, the semantics of a programming language can be defined as an inductive definition, using dozens of constructors, each one describing a kind of computation elementary step. Proofs by induction with respect to this inductive definition correspond to what Wynskel calls *rule induction* in his introductory book on programming language semantics [3].

6 Exercices

Most of the exercices proposed here are taken from [1].

Exercise 5.6 Prove the following theorems:

```
forall A B C:Prop, A/\(B/\C)→(A/\B)/\C
forall A B C D: Prop,(A→B)/\(C→D)/\A/\C → B/\D
forall A: Prop, ~(A/\~A)
forall A B C: Prop, A\/(B\/C)→(A\/B)\/C
forall A: Prop, ~~(A\/~A)
forall A B: Prop, (A\/B)/\~A → B
```

Two benefits can be taken from this exercise. In a first step you should try using only the basic tactics given in the table page 7. In a second step, you can verify which of these statements are directly solved by the tactic `intuition`.

Universal quantification Prove

```
forall A:Set,forall P Q:A→Prop,
    (forall x, P x)\/(forall y, Q y)→forall x, P x\/Q y.
~(forall A:Set,forall P Q:A→Prop,
   (forall x, P x\/Q x)→(forall x, P x)\/(forall y, Q y).
```

Hint: for the second exercise, think about a counter-example with the type `bool` and its two elements `true` and `false`, which can be proved different, for instance.

Exercise 6.32 The sum of the first n natural numbers is defined with the following function:

```
Fixpoint sum_n (n:nat) : nat :=
  match n with  0 => 0 | S p => S p + sum_n p end.
```

Prove the following statement:

```
forall n:nat, 2 * sum_n n = n*n + n
```

Square root of 2 If $p^2 = 2q^2$, then $(2q-p)^2 = 2(p-q)^2$, now if p is the least positive integer such that there exists a positive integer q such that $p/q = \sqrt{2}$, then $p > 2q - p > 0$, and $(2q-p)/(p-q) = \sqrt{2}$. This is a contradiction and a proof that $\sqrt{2}$ is not rational. Use Coq to verify a formal proof along these lines.

7 Solutions

The solutions to the numbered exercises are available from the internet (on the site associated to the reference [1]). The short proof that $\sqrt{2}$ is not rational is also available on the internet from the author's personal web-page.

Here are the solutions to the exercises on universal quantification.

```
Theorem ex1 :
  forall A:Set, forall P Q:A->Prop,
  (forall x, P x) \/ (forall y, Q y) -> forall x, P x \/ Q x.
Proof.
```

```
 intros A P Q H.
 elim H.
 intros H1; left; apply H1.
 intros H2; right; apply H2.
Qed.

Theorem ex2 :
  ~(forall A:Set, forall P Q:A->Prop,
    (forall x, P x \/ Q x) -> (forall x, P x) \/ (forall y, Q y)).
Proof.
  intros H; elim (H bool (fun x:bool => x = true)
                  (fun x:bool => x = false)).
  intros H1; assert (H2:false = true).
    apply H1.
  discriminate H2.
  intros H1; assert (H2:true = false).
    apply H1.
  discriminate H2.
  intros x; case x.
  left; reflexivity.
  right; reflexivity.
Qed.
```

References

[1] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development, Coq'Art:the Calculus of Inductive Constructions.* Springer-Verlag, 2004.

[2] The Coq development team. *The Coq proof Assistant Reference Manual*, Ecole Polytechnique, INRIA, Universit de Paris-Sud, 2004. `http://coq.inria.fr/doc/main.html`

[3] G. Winskel. *The Formal Semantics of Programming Languages, an introduction.* Foundations of Computing. The MIT Press, 1993.