Verifying programs and proofs
part III. prove program properties

Yves Bertot

September 2012

1 Motivating introduction

To prove that programs do what is intended, we need to cover all possible cases in their execution. Most of the time, the programs take their input in types with an infinite numbers of elements. It is thus impossible to check all possible inputs one by one. Instead, we reason on subsets of the types that correspond to different behaviors of the considered programs. This is usually done by following the structure of the program. Thus we reason logically on the various cases that may arise during the execution of programs.

When functions are recursive, this approach relies on a complex logical tool, called *proof by induction*. In this lecture, we want to understand the various aspects of reasoning about program behavior, including the idea of proofs by recursion. We shall restrict this study to programs that compute on numbers and lists.

There are many ways to describe what is the expected behavior of a program. In this course, we will study only a simple approach, where the expected behavior is described with the help of secondary programs that are used either to produce specific inputs or to perform tests on on the output of a program.

For instance, if we wrote a function `evenb` that computes a boolean value that is true whenever the input is an even number (a multiple of 2), we want to write a function that computes all even numbers.

```
Definition mult2 (n : nat) := 2 * n.
```

Then we just want to prove that `evenb` returns the correct value for every result of `mult2`:

```
Lemma evenb_complete :
  forall n : nat, evenb (mult2 n) = true.
```

This is not enough, because the trivial boolean predicate that always returns true would also satisfy this lemma. We may also want to make sure that the function `evenb` accepts only numbers that should be accepted. One way to express this is with the following statement:

```
Lemma evenb_sound :
  forall n : nat, evenb n -> exists y : nat, n = mult2 y.
```

Again, lemma `evenb_sound` is not enough, because the trivial boolean predicate that always returns `false` would also satisfy this lemma.

In the end, every lemma that we write is like a symbolic test that we run on the input program, but the proof shows that this test is satisfies for all possible inputs instead of a random sample. In this sense, the coverage brought by formal proofs is more complete than the coverage brought by random tests and people like to say that we provide 100% correctness, but we should keep in mind that the lemmas may describe only partially the intended behavior of the program (like the predicate `evenb_complete` and `evenb_sound` taken separately) and that the way we consider the inputs (like the function `mult2`) may also be faulty.

2 Reasoning on pattern-matching constructs

A pattern matching construct describes several possible cases of execution. When proving that a program is correct, we need to cover all possibilities. There are commands to decompose the problem and to observe separately each of the cases. Then, it may occur that some cases are inconsistent, because they exhibit assumptions like `0 = 1` or `true = false`. In some other cases, one may have equalities of the form `a::l = b::l'`, from which one should be able to deduce at least `a = b` and `l = l'`. We shall see different approaches for this.

2.1 case, destruct, and case_eq

Let us consider the following goal:

```
x : nat
======================
 match x with 0 => true | S p => negb (even p) end = false
 -> x <> 0
```

The conclusion of this goal contains a pattern matching construct. We don't know the value of `x`, but we know that `x` is a natural number, and by consequences `x` may follow one of 2 cases: `x` is `0` or `x` is `S y` for some `y`. When `x` is `0`, the whole pattern-matching construct will compute to `true`, so the goal's conclusion will become

```
 true = false -> 0 <> 0
```

In the other case, we know that the pattern-matching construct will be reduced to `negb (even y)`, so the goal's conclusion will become

```
negb(even p) -> S y <> 0
```

There are three tactics that express this decomposition in two cases, with slightly different behaviors. First, let's observe the behavior of the `destruct`

tactic. It produces as many goals as the number of cases in the argument's type. Here we use `destruct x` and `x` has type `nat`. Because the type `nat` has two constructors, there are two cases: either `x` is 0 or `x` is `S n` for some other natural number `n`.

```
destruct x.
2 subgoals
  ======================
   true = false -> 0 <> 0

Subgoal 2 is
   negb(even n) -> S n <> 0
```

So the tactic simply produced two instances of the goal where `x` is replaced with cases taken from the type. In the first goal, all occurrences of `x` are replaced by `0`, then the pattern matching construct is computed to take this new information into account.

In this example, the two goals can be proved because they mention equalities that cannot hold. This is taken care of by the tactic described in the next section.

The tactic `case` performs approximately the same operations as the tactic `destruct`, except that its effect only applies in the conclusion of the goal and the data that is analyzed is not removed from the context. So if we restart the example, we have the following behavior:

```
  x : nat
  ======================
   match x with 0 => true | S p => negb (even p) end = false
   -> x <> 0
case x.
2 subgoals
  x : nat
  ======================
   true = false -> 0 <> 0

Subgoal 2 is
   forall n : nat, negb(even n) -> S n <> 0
```

However, the variable `x` that remains in the context is now completely disconnected from the values that replace it in the two goal conclusions. That this variable is disconnected sometimes poses a problem. Let us show an example where this is a problem. Let's take again the same example, but assume that we perform a stronger call to `intros` before calling `case`.

```
  x : nat
  ======================
   match x with 0 => true | S p => negb (even p) end = false
   -> x <> 0
```

```
intros H.
  x : nat
  H : match x with 0 => true | S p => negb (even p) end = false
  ======================
   x <> 0
case x.
 x : nat
  H : match x with 0 => true | S p => negb (even p) end = false
  ======================
   0 <> 0

Subgoal 2 is
   forall n : nat, S n <> 0
```

The first goal cannot be proved, because the instance of `x` in the hypothesis `H` has not been replaced by 0.

This problem is solved by another tactic named `case_eq`. This tactic adds in the goals equalities that keeps a connection between `x` and the values that replace it.

```
  x : nat
  H : match x with 0 => true | S p => negb (even p) end = false
 ======================
  x <> 0
case_eq x.
2 subgoals
  x : nat
  H : match x with 0 => true | S p => negb (even p) end = false
  ======================
   x = 0 -> 0 <> 0

Subgoal 2 is
   forall n : nat, x = S n -> S n <> 01
```

This time, the occurrences of `x` still remain unchanged in the hypothesis `H`, but we now have an equality in the goal that can be introduced and used to modify the hypothesis, with the help of the `rewrite` tactic.

2.2 the `induction` tactic

Reasoning by cases is not adapted for recursive functions, because we often need hypotheses on the values returned by recursive calls. These hypotheses often have the same form as the statement that one attempts to prove. This follows the a well-known pattern seen in proofs about natural number, known as proof by induction.

induction on natural numbers If a predicate on natural numbers `P` is such

that `P 0` holds and for every `n` one can deduce `P (S n)` from `P n`, then this predicate holds for every natural number.

induction on lists If a predicate on lists of natural numbers `P` is such that `P nil` holds and for every list `l` and every natural number `a`, if `P l` holds then we can deduce `P (a::l)`, then this predicate holds for every list of natural numbers. This can be generalized to list of any type of elements.

When performing a proof by induction on natural numbers, we also have two cases to study, the first case for the situation where the value is `0` and the second for the situation where the value has the form `S n` for some `n`. In this respect, the induction tactic is very close to the `destruct` tactic. However, when considering the second case, we have more information: we can use an *induction hypothesis* stating that `n` already satisfies the expected predicate. Let's observe such a proof by induction concerning the addition of a number with 0. We first observe the definition of addition:

```
Locate "_ + _".
Notation            Scope
"x + y" := sum x y   : type_scope

"n + m" := plus n m  : nat_scope
                      (default interpretation)
Print plus.
fix plus (n m : nat) struct n : nat :=
  match n with
  | 0 => m
  | S p => S (plus p m)
  end
     : nat -> nat -> nat
```

We see that addition is given as a recursive function where the first argument decreases at each recursive call. By definition `0 + n` computes to `n` in a single step; on the other hand, computing `n + 0` does not do anything directly, but we can prove that it computes to `n`. Here is the proof in Coq:

```
Lemma example_induction_plus : forall n, n + 0 = n.
induction n.
2 subgoals

  ============================
   0 + 0 = 0

subgoal 2 is:
 S n + 0 = S n
```

As expected, we have two cases where `n` is replaced either by `0` or by `S n`. For the first case, immediate computation yields the result.

```
reflexivity.
1 subgoal

  n : nat
  IHn : n + 0 = n
  ============================
   S n + 0 = S n
```

In this goal, the context contains the hypothesis `IHn`, which states exactly that the property we want to prove already holds for `n`. So the induction principle is being used, and the predicate `P` is instantiated with the function

```
fun x => x + 0 = x
```

2.3 the `discriminate` tactic

The datatypes of boolean values, natural numbers, and lists are all described in the Coq system as *inductive types*, where the data may each time correspond to two patterns. We can see this by calling the command `Print`.

```
Print bool.
Inductive bool : Set :=  true : bool | false : bool

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

Print list.
Inductive list (A : Type) : Type :=
    nil : list A | cons : A -> list A -> list A
```

The description of natural numbers means that numbers are either of the form `O` or of the form `S n`. Moreover, it also means that the number `O` is not of the form `S n`[1]. As a result, any goal whose conclusion has the form `O <> S n` should be easily provable. The Coq system provides a specific tactic for that, called `discriminate`. This tactic also takes care of cases where a goal has an arbitrary conclusion but one of its hypotheses is an hypothesis of the form `O = S n`.

Continuing the example given in the previous section, we had two goals that we repeat again here:

```
2 subgoals
  ======================
   true = false -> 0 <> 0

Subgoal 2 is
   negb(even n) -> S n <> 0
```

[1] This is also simply a consequence from the fact that we can define expressions by pattern-matching on natural numbers!

For the first goal, we use the `intros` tactic and we get a new hypothesis:

```
intros Htf.
  ...
  Htf : true = false
  ======================
   0 <> 0
```

In this goal, the conclusion would be unprovable in an empty context, but the hypothesis `Htf` assumes an equality between two values that are different by definition. This goal can be solved using the `discriminate` tactic or more precisely the `discriminate Htf`.

The second goal has the form

```
  negb(even n) = true -> S n <> 0
```

Here the ultimate conclusion is the negation of an equality between two cases that are forced to be different. So it falls in the same area of reasoning. Here, the `discriminate` tactics also solves the problem.

2.4 the `injection` tactic

We often have to express that the constructors of types like `nat` and `list` are *injective*. In other words, if they give equal outputs for two sets of inputs, then the inputs must be pairwise equal. for lists this is easily expressed with the following example:

```
Lemma example_injection_list :
  forall (a b : nat) (l1 l2 : list nat),  a::l1 = b::l2 ->
    a = b /\ l1 = l2.
intros a b l1 l2 Hq.
  ...
  Hq : a::l1 = b::l2
  ============================
   a = b /\ l1 = l2
```

In this goal, the hypothesis `Hq` describes an equality between two composed lists. The conclusion expresses that the list components correspond to each other. To go from `Hq` to the conclusion, we call the tactic `injection` with the name of the hypothesis.

```
injection Hq.
  ...
  ========================
   l1 = l2 -> a = b -> a = b /\ l1 = l2
intros ql qa; rewrite ql qa; split; reflexivity.
Qed.
```

In the generated goal, two new implications are created, with the equalities between components appearing as left-hand sides of these implications. The last two lines of the example show how to use these hypotheses.

Similarly, if we know two equal numbers that respect the `S` pattern, we can deduce that the sub-components are equal.

```
Lemma example_injection_nat :
  forall (a b : nat),  Sa = S b -> a = b.
intros a b Hq.
  ...
  Hq : S a = S b
  ============================
   a = b
injection Hq.
  ...
  ============================
   a = b -> a = b
intros q1; exact q1.
Qed.
```

3 Manipulating function computation

In goals and logical statements, the Coq system manipulates functions without executing them. We sometimes need to force at least a few steps of computation.

3.1 The `unfold` tactic

The first approach is to simply require that the system expands the definition. The word used in Coq tactics is `unfold`.

```
Definition add3 (n : nat) := n + 3.

Lemma example_add3 : forall n, add3 n = 3 + n.
intros n.
  ...
  ======================
   add3 n = 3 + n.
```

At this point, we would like to replace `add3 n` with the expression it computes. We use the `unfold` tactic.

```
unfold add3.
  ======================
    n + 3 = 3 + n
```

3.2 The `simpl` tactic

When dealing with a recursive function, the `unfold` tactic often makes goals unreadable, because it expands the value of the recursive function into something that repeats the text of the recursive function several times. To avoid this, there is a tactic that specifically tuned to handle recursive function. This tactic is called `simpl`.

The example we have already seen about reasoning on the addition function provides an illustration for this. Let start again with this proof.

```
Lemma example_induction_plus : forall n, n + 0 = n.
induction n.
reflexivity.
1 subgoal

  n : nat
  IHn : n + 0 = n
  ============================
   S n + 0 = S n
```

Here, we can request that Coq performs a little computation with `S n + 0`. We simply need to call the `simpl` tactic:

```
simpl.
  ...
  IHn : n + 0 = n
  ============================
   S (n + 0) = S n
```

The left hand side of the equality in this goal's conclusion is an occurrence of the left hand side of the hypothesis `H`. We can rewrite and conclude the proof.

```
rewrite IHn; reflexivity.
Qed.
```

3.3 Manual computation: the `change` tactic

Sometimes the `simpl` tactic performs too much computation. In this case, it is a good idea to state explicitely the result that we want to see after computation, as long as this result really corresponds to a computation. Here is an example.

```
Lemma example_change_plus :
  forall n m p, (1 + n) * m = p -> (1 + (1 + n)) * m = m + p.
intros n m p H.
1 subgoal

 H : (1 + n) * m = p
  ============================
   (1 + (1 + n)) * m = m + p
```

```
change ((1 + (1 + n)) * m) with (m + (1 + n) * m).
  ...
  ============================
   m + (1 + n) * m = m + p
rewrite H; reflexivity.
Qed.
```

3.4 Manual computation with the `replace` tactic

The tactic `change` performs replacements only if the two expressions are the same modulo computation. Sometimes, we want to relax the condition and perform replacement as soon as we can prove the equality between two expressions. For this we use the `replace` tactic. This tactic produces a second goal, because the equality between the two expressions needs to be proved.

3.5 Generating one-step recusion lemma

Using the `change` tactic gives the user complete control on what is executed, but it is very cumbersome to use, as it requires that one writes a lot of text. One efficient approach is to provide an unfolding lemma for the function that one produces. This can easily be done by copying the body of the function in an equality and encapsulating it the universal quantifications that fit. The proof can usually be done by a simple case analysis on the first argument of the function to be analysed. Here is an example for a function of multiplication by 3:

```
Fixpoint mult3 (n : nat) : nat :=
match n with 0 => 0 | S p => 3 + mult3 p) end.

Lemma mult3_step (n : nat) :
  mult3 n =
  match n with 0 => 0 | S p => 3 + mult3 p) end.
Proof.
intros n; case n; reflexivity.
Qed.
```

4 Exercises

1. Define a function `lo` that takes a natural number `n` as input and returns the list containing the first `n` odd natural numbers. For instance `lo 3 = 5::3::1`.

2. Prove that `length (lo n) = n`.

3. Define a function `sl` that takes a list of natural numbers as input and returns the sum of all the elements in the list.

4. Prove that `sl (lo n) = n * n`.

5. We define a function `add` with the following code:

   ```
   Fixpoint add x y := match x with 0 => y | S p => add p (S y) end.
   ```

   Prove the following lemmas:

   (a) `forall x y, add x (S y) = S (add x y)`
   (b) `forall x, add x 0 = x`
   (c) `forall x y, add (S x) y = S (add x y)`
   (d) `forall x y z, add x (add y z) = add (add x y) z`
   (e) `forall x y, add x y = x + y`

6. In the exercises part of the first chapter of these course notes, you are required to define a function that describes when a list of numbers is a licit representation of a natural number (it verifies that all digits are less than 10) and a function that computes the successor of a number when represented as a list of digits. Add the function `to_nat` that maps any list of digits to the natural number it represents and show that the successor function is correct in this context.

5 More information

You can use the book [1] (available in French on internet, otherwise you should find English versions at the library) and the reference manual [4]. There is also a tutorial in French [7]. There are also tutorials on the web [5, 3].

References

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

[2] B. Pierce et al. *Software Foundations* `http://www.cis.upenn.edu/~bcpierce/sf/`

[3] Y. Bertot *Coq in a Hurry* Archive ouverte "cours en ligne", 2008. `http://cel.archives-ouvertes.fr/inria-00001173`

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

[5] G. Huet, G. Kahn, C. Paulin-Mohring, *The Coq proof Assistant, A Tutorial*, Ecole Polytechnique, INRIA, Université de Paris-Sud, 2004. `http://coq.inria.fr/V8.1/tutorial.html`

[6] E. Giménez, P. Castéran, *A Tutorial on Recursive Types in Coq*, INRIA, Université de Bordeaux, 2006. `http://www.labri.fr/Perso/~casteran/RecTutorial.pdf.gz`

[7] A. Miquel, *Petit guide de survie en Coq*, Université de Paris VII. `http://www.pps.jussieu.fr/~miquel/enseignement/mpri/guide.html`