Verifying programs and proofs
part II. describe program properties

Yves Bertot

September 2012

1 Motivating introduction

In version 8.4, the Coq system does not provide any debugging tool. On the other hand, it provides ways to express logical expressions between various pieces of data. For instance, we can express logically the relation that should exist between a function and its input, in a way that is independent from the actual algorithm implemented in the function. If we think about a division function (as we learn it in school), there is a simple relation between the divided number `x`, the divisor `y`, the computed quotient `q`, and the computed remainder `r`:

```
x = y * q + r /\ r < y
```

In English, this relation states that `x` is equal to the sum of the product of `y` and `q` and `r` and that `r` is smaller than `y`. This relation should be satisfied by the pair of numbers returned by a division algorithms, but this can only be satisfied if we avoid dividing by 0. In this course, we will concentrate on the language used to express relations between pieces of data.

In this lesson, we will learn how to write logical formulas that represent correctly what we want to express and how to prove these formulas. This may be difficult for readers that have little acquaintances with logic, but somehow, we should try to learn by practice. We know that we understand it when we start being able to write concise formulas that we can prove, thus getting lemmas, and when we can use these lemmas to prove other formulas, thus getting new theorems.

2 Writing logical formulas

2.1 Predicates

In this section, we will consider a new type, the type of propositions and ways to build atomic objects in this type. The simplest way to assert a proposition is to say that some value is equal to another. Here two examples:

```
Check 3 = 2 + 1.
3 = 2 + 1 : Prop

Check 3 = 4.
3 = 4 : Prop
```

Note that you can always write a logical proposition, as long as it is well-formed, even if this proposition is false (here `3 = 4` is a well-formed logical proposition, even if it is false).

In this course, we will define new predicates by simply inventing new functions of one or two arguments that return a value that is then compared to another value. For instance, we can write a function that returns the boolean value `true` whenever its argument is even, and then make this into a predicate by expressing that the value of the function is `true`.

```
Fixpoint even (n : nat) :=
  match n with 0 => true | 1 => false | S (S p) => even p end.

Check even 3.
even 3 : bool

Check even 3 = true.
even 3 = true : Prop
```

We see here that there is a distinction between boolean values and propositions. In practice, boolean values are meant to be the result of tests that can be checked by a program, but propositions may represent statements that no algorithm can verify in one run. Boolean values can be produced and tested in programs. Logical proposition do not belong inside programs, at least for our lessons.

2.2 Logical connectives

There are four simple connectives *implies*, *and*, *or*, *not*. These connectives are given with notations that make it possible to write logical formulas in a short and readable way.

- Implication is noted `->`.
- Conjunction (*and*) is noted `/\`.
- Disjunction (*or*) is noted `\/`.
- Negation is noted `~`.
- Equivalence is noted `<->`. In fact `A <-> B` is a defined as `A -> B /\ B -> A`.

For instance, we can write the following proposition.

```
Check ~ (even 2) = true -> even 3 = false.
even 2 <> true -> even 3 = false
   : Prop
```

2.3 Quantifications

We can also write formulas that give properties about a whole type. There are two main connectives for this. The first one, `forall`, describes *universal quantification*. In other words, this is used to express that every element of a type satisfies some property. The second one, `exists`, describes *existential quantification*. This is used to express that at least one element satisfies some property. For instance, one can write the following proposition.

```
Check forall x, even x = true -> exists y, x = 2 * y
```

Here, the example chosen as illustration is true and can be proved. In a sense, this means that we can prove that the `even` function does what we intend it to do. On the other hand, we cannot write an algorithm that verifies that the property holds, because it would need to verify the statement for every natural number, but the set of natural numbers is infinite and the verification would never end. This is why we make the distinction between propositions (that can be stated and proved) and boolean values (that can be computed by an algorithm).

2.4 Some more predefined predicates

With the `Arith` and `List` libraries loaded, the Coq system also provides a predefined predicate to compare two natural numbers, noted `x <= y`. Different variants are also provided for strict comparisons (`x < z`), and for intervals (`x <= y < z`).

For lists there is also the `In` predicate, such that `In x l` means that `x` is inside `l`.

2.5 Playing with logical formulas

To progress further you should be able to write simple logical formulas that carry the meaning of some of your knowledge about real life, programs, or mathematical facts.

In general, if you write `A -> B` and you mean this formula to be true, it is not necessary that `A` is always true: you only want to express that `B` is true in all cases where `A` is true, but `A` may sometimes be false and in those cases, `B` can also be false. On the other hand, if you write `A /\ B` and you mean this formula to be true, then `A` should always be true.

Here is a sentence: *in winter, days are always shorter than nights, and there is always a day that is rainy*. We could capture this by assuming that we have a type of dates that contains all dates in the year, a predicate `winter` that holds for dates in winter, a function `night_length` that returns the length of the night for every date, as a natural number (for instance counting the number of minutes) and a similar function `day_length` that returns the length of the day. Also, we could have a function `weather` that associates a date and a year to the weather on that day. This function would return its value in a type that contains at least the values `sunny` and maybe `rainy`, or `cloudy`. The sentence could then be described by the following logical formula:

```
(forall d: date, winter d -> day_length d <= night_length) /\
(exists y : nat, exists d : date, weather y d = rainy)
```

The distinction between conjunction and implication is important. If you want to define a predicate `P` and you know that some predicate `A` has to hold for every object that satisfies `P`, then you should probably use a conjunction in the definition of `P`, not an implication, even though there may be objects that do no satisfy `A` (and therefore they do not satisfy `P` either). Here is an example with prime numbers: We know that all prime numbers are larger than or equal to 2. So when writing a definition for the predicate `prime` we write something like that:

```
Definition prime n := 2 <= n /\ ...
```

If we replaced this conjunction by an implication, this would allow `0` to be considered a prime number, because a formula of the form `2 <= 0 -> ...` can always be proved.

3 Performing simple proofs

To perform a proof, we state the proposition we want to prove, then we decompose the proposition into simpler propositions, until they can be solved. The commands used to decompose propositions are called tactics. To learn how to use the tactics, it is handy to classify them according to the connectives and according on whether the connectives appears in something we want to prove or in something we already know.

3.1 Stating a logical formula

It is better to show that in an example.

```
Lemma ex1 : 2 = 3 -> 3 = 2.
```

The keyword is `Lemma`: we will use it every time we want to start a new proof. Then comes a unique name, which we choose, `ex1`. Then comes a colon "`:`". Then comes a logical formula (here an implication between two equalities). We finish the command with a period.

3.2 Known facts and facts to prove

At any time during a proof, the Coq system displays *goals*, which describe what we have to prove. Each goals has two parts, the first part is *context* of temporary known facts. The second part is a *conclusion*, a fact that needs to be proved. A simple case of proof is when the fact we want to prove is present among the known facts. In this case, it suffices to use the basic tactic `assumption` to solve the goal. If we do that, the Coq system displays the next unsolved goal, or says that the proof is complete.

3.3 Finishing a proof

When there are no more subgoals to solve, the system says that the proof is complete, but we still have an operation to do: we must instruct the system to record the completed proof in memory. This is done by typing in the command `Qed.`

3.4 Handling connectives

3.4.1 implication

When a goal's conclusion is an implication, we can make it simpler by applying the tactic `intros H`. This produces a new goal with an extra element in the context, which corresponds to the proposition that was initially in the left hand side of the implication.

```
  H : A
  ==================
  2 = 3 -> 3 = 2

intros H'.
  H : A
  H' : 2 = 3
  ===================
  3 = 2
```

The name used as argument to the `intros` tactic is used to name the new fact in the context.

To use an hypothesis that contains an implication, we use the tactic `apply`. Here is an example.

```
  H : A -> 2 = f 3
  ==================
  2 = f 3

apply H.
  H : A -> 2 = f 3
 ==================
  A
```

This works only if the left hand side of the hypothesis `H` corresponds exactly with the conclusion. The conclusion is replaced by left-hand side of the arrow. If there are several arrows in the hypothesis, then several goals are produced. For instance, if the hypothesis had been `A -> B -> 2 = 3`, then we would have had two new goals, one with `A` and the other with `B`.

3.4.2 Conjunction

When a goal's conclusion is a conjunction, we can make it simpler by applying the tactic `split`. This produces two new goals whose statements are the parts of the conjunction.

```
  H1 : A -> B
  ======================
  C /\ D
split.
  H1 : A -> B
  ======================
   C

Subgoal 2 is:
   D
```

When we want to use an hypothesis that is a conjunction, we often need to decompose this conjunction to extract its parts as new hypotheses. This is done with the destruct command.

```
  H : A /\ B
  ======================
   A
destruct H as [H1 H2].
  H1 : A
  H2 : B
  ======================
   A
```

3.4.3 Disjunction

When a goal's conclusion is a disjunction, we can make it simpler by applying one of the tactics `left` and `right`. This produces a new goal with only the chosen part of the goal.

```
  H1 : A -> B
  ========================
  B \/ 2 = 3
left.
  H1 : A -> B
  ========================
   B
```

Of course, we have to be careful and choose the tactic that will really lead us to a new goal that is provable (in this example, choosing `right` looks silly).

When we want to use an hypothesis that is a disjunction, we often need to decompose this conjunction to extract its parts as new hypotheses. But this produces two goals, because if we have to cover the two case:

```
  H : A \/ B
  ======================
   B \/ A
destruct H as [H1 | H2].
  H1 : A
 ======================
   B \/ A

Subgoal 2 is:
   B \/ A
```

The second goal contains an hypothesis named `H2` (as stated in the `destruct` tactic) with `B` as the statement.

3.4.4 Negation

When a goal's conclusion is a negation, we can make it simpler by applying the tactic `intros H`.

```
  H1 : A -> B
  ========================
   ~ A
intros H.
  H1 : A -> B
  H : A
  ========================
   False
```

When proving "not A", the idea is to show that assuming `A` would lead to a contradiction.

When we want to use an hypothesis that is a negation, we apply a tactic called `case H`. This replaces the current conclusion by the negated formula.

```
  H : ~ A
  ======================
   C
case H.
  H : ~ A
 ======================
   A
```

We should do this exactly when we know that we will be able to prove `A` more easily that proving `C`.

3.5 Quantifiers

3.5.1 Universal quantification

When trying to prove a universally quantified formula, we often use the tactic `intros x`, where `x` is a name chosen to fix the value on which we want to reason. The idea is that if we want to prove a formula for all members of a type, we should simply prove that this formula holds for a single arbitrary one, which we chose to name `x`. Here is an example:

```
  ==================
  forall A:Prop, A -> A
intros A.
  A : Prop
  ===================
   A -> A
```

This proof can be finished by typing `intros H.` and then `assumption.`

We shall see in another lesson that some universally quantified formulas can be proved by more advanced means, like `induction`.

If one wants to use an hypothesis that starts with a universal quantification, one should most of the time use the tactic `apply`. Here is an example:

```
  H : forall x: nat, P x -> Q x
  ================
   Q 3
apply H.
  H : forall x: nat, P x -> Q x
  ================
   P 3
```

Note that the Coq system found an instance of the universally quantified formula that corresponds to `Q 3`, then it applied the same behavior as for implication.

3.5.2 Existential quantification

When trying to prove an existentially quantified formula, we have to provide a candidate value that satisfies the required predicate. The tactic is called `exists` (with the same spelling as the logical connective). Here is an example.

```
  ======================
   exists x, 2 * x = 6
exists 3.
  ======================
   2 * 3 = 6
```

As a result, we simply have to prove that the provided value (here 3) satisfies the required predicate.

When trying to use an hypothesis that starts with an existential quantification, we actually want to decompose the information in this quantification, so that we obtain a new context that really contains a value satisfying the interesting property. Here is an example:

```
  H : exists x : nat, even x = true
   ======================
    C
destruct H as [w Pw].
  w : nat
  even w = true
  ========================
   C
```

In the goal before the `destruct` tactic, there is no natural number that satisfies the property. In the goal after the tactic, there is a natural number called `w` and we know that `w` satisfies the property, so we can use it for various purposes.

3.6 Predicates

3.6.1 Equality

When we want to prove an equality, the simplest approach is when the two members of the equality are equal (even modulo computation). Here is an example.

```
  ======================
  2 = 3 - 1
reflexivity.
  No more subgoals.
```

This tactic can also be used if the computation uses functions that we have defined ourselves, like `even`.

If we want to use an hypothesis that contains equalities, we can use the `rewrite` tactic.

```
  H : 3 = 2
  ====================
   2 = 3
rewrite H.
  H : 3 = 2
  ====================
   2 = 2
```

The tactic `rewrite` can also be used if the equality is wrapped inside a universal quantification. In that case, it finds the first relevant instantiation of the universally quantified variable before performing the replacement.

```
  H : forall x : nat, f (f x) = g (x + x)
  ====================
   g (2 + 2) = E
rewrite <- H.
  H : forall x : nat, f (f x) = g (x + x)
  ====================
   f (f 2) = E
```

Note also, that the `<-` modifier makes it possible to use the equality in a different direction.

4 Un exemple de preuve

```
Lemma distr_or_comm_l :
  forall a b c, a \/ (b /\ c) -> (a \/ b) /\ (a \/ c).
intros a b c h.
destruct h as [ha | hconj].
 split.
  left.
  exact ha.
 left.
 exact ha.
destruct hconj as [hb hc].
split.
 right.
 exact hb.
right.
exact hc.
Qed.
```

5 Exercises

1. Write a predicate `multiple` of type `nat -> nat -> Prop`, so that `multiple a b` expresses that `a` is a multiple of `b` (in other words, there exists a number `k` such that `a = k * b`).
2. Write a formula using natural numbers that expresses that when `n` is even (a multiple of 2) then `n * n` is even.
3. Write a formula using natural numbers that expresses that when a number `n` is a multiple of some `k`, then `n * n` is a multiple of `k` (you don't have to prove it yet).
4. Write a predicate `odd` of type `nat -> Prop` that characterize odd numbers like 3, 5, 37.
5. Assuming there exists a type `T` used to represent numbers and a function `nat_of_T` of type `T -> nat`, which maps any element of `T` to the element of `nat` that it represents, and assuming that `tadd` is a function of type `T -> T -> T`, how do you express that `tadd` represents addition? Beware that several elements of `T` may represent the same element of `nat`.
6. Write the script that proves the following formula

   ```
   forall P Q : nat -> Prop,
   forall x y:nat, (forall z, P z -> Q z) -> x = y -> P x ->
     P x /\ Q y
   ```

7. Write the script that proves the following formula

   ```
   forall A B C : Prop, (A /\ B) \/ C -> A \/ C
   ```

8. Write the script that proves the following formula

   ```
   forall P : nat -> Prop, (forall x, P x) ->
    exists y:nat, P y /\ y = 0
   ```

9. Write the script that proves that when `n` is a multiple of `k`, then `n * n` is also a multiple of `k`.
10. Write the script that proves that when `n` is odd, then `n * n` is also odd.

6 More information

Vous pouvez utiliser le livre [1] (disponible en français sur internet) et le manuel de référence [4]. Il existe aussi un tutoriel en français par un autre professeur [7]. Il y aussi des tutoriels en anglais [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`