Verifying programs and proofs
part II. describe program properties

Yves Bertot

January 2025

1 Motivating introduction

While the motivation of the proof system is to avoid bugs in functional programs, it is amazing that this system does not provide any advance tool to step through the execution of programs.

On the other hand, it provides ways to express logical relations between various pieces of data. For instance, we can express logically the relation that should exist between a function output 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 /\ 0 <= 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 between `0` and `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. So the logical statement expressing that division is implemented correctly should both express this expected outcome and this restriction on the inputs.

In this course, we will concentrate on the language used to express relations between pieces of data.

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 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 Prerequisites

To run the examples that follow with the Coq system, we need to place ourselves in the right context. This is done by telling the proof system to load predefined libraries for integers, lists, boolean values.

```
Require Import ZArith List Bool.

Open Scope Z_scope.
```

For the exercises, the author of these notes also prepared some sets of predefined functions which can be loaded in this manner. If you use the docker image `ybertot/course2025`, then you only have to type the following command inside Coq to get access to these predefined functions.

```
From YBCourse2025 Require Import predefined_functions.
```

2.2 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
```

An equality is well-formed when the left-hand-side and the right-hand-side have the same type.

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). The true and false propositions are not distinguished by the fact that they can be written, but by the fact that they can be proved.

When it comes to numbers, the Coq system also provides us with other predicates that correspond to comparisons : `3 <= 4`, 3 ¡ 4, and the symmetric relations. To verify algorithms about cryptography, it may also be useful to know when a number is prime, for instance.

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 say that a number is even if that number modulo 2 is equal to 0.

```
Definition even (x : Z) := x mod 2 = 0.

Check 3 mod 2 =? 0 : bool.
3 mod 2 =? 0 : bool

Check even 3.
even 3 : 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, and these tests can be used in other program. On the other hand, 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.

2.3 Examples with lists

Lists manipulations are a nice source of exercises for recursive programming and verifying algorithms. We will exercise a small collection of functions.

1. The function `Zlength` is a function designed for this lesson. It takes as input a list of elements in an arbitrary type and returns the length of this list in type `Z`.

   ```
   Fixpoint Zlength {A : Type}(l : list A) : Z :=
     match l with
     | nil => 0
     | _ :: tl => 1 + Zlength tl
     end.
   ```

2. The predicate `In` is already provided in the `List` library. It takes two arguments, an element x in any type and a list l of elements of that type and it holds when x appears among the elements of that list.

3. The function `Znth` is a function designed specifically for this lesson. It takes as input a list of elements in an arbitrary type and an integer n, and it returns the element at position n in that list, starting count at 0. However, there may be reasons for this to fail, because n is negative or larger than the size of the list, no value can be found in the list.

```
Fixpoint Znth {A : Type}(l : list A) (n : Z) : option A :=
  match l with
  | nil => None
  | a :: tl =>
    if n =? 0 then Some a else Znth tl (n - 1)
  end.
```

4. The function `app` is already provided in the `List` library. It takes two lists and returns the concatenation of these two lists. There is an infix notation that is also predefined for this function, so that `app l1 l2` is also written `l1 ++ l2`. This notation is right associative, so that `l1 ++ (l2 ++ l3)` means `l1 ++ l2 ++ l3`.

5. The function `rev` is already provided in the `List` library. It takes as input a list a_1::$a_2 \cdots$::a_n:: `nil` and returns the list a_n::$a_{n-1} \cdots$::a_1:: `nil`.

2.4 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 concerning the function `Znth`.

```
Check ~(Znth (1 :: 2 :: nil) 1 = None) ->
   0 <= 1 < Zlength (1 :: 2 :: nil).
~(Znth (1 :: 2 :: nil) 0 = None) ->
       0 <= 1 < Zlength (1 :: 2 :: nil) : Prop
```

It happens that this proposition is provable, as we shall see later.

2.5 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 of the type satisfies some property. For instance, one can write the following proposition.

```
Check forall {A : Type} l x, Znth l x -> 0 <= x < Zlength l.
```

It happens that this proposition can be proved, as we shall see later. Here, the example chosen as illustration is true and can be proved. In a sense, this means that we can prove that the `Znth` and the `Zlength` function are consistent with the explanations in plain language. However, we cannot write a test that guarantees this property, because such a test would have to cover all possible lists with elements in all possible types and with all possible lengths.

2.6 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 a non-empty list of integers, there is always a number that is smaller than all the others.*

The first part of the sentence expresses that we want to discuss about *any* list of integers that is non-empty. We shall naturally have a universal quantification over a type, but the types we know do not make it possible to express that we want to look at non-empty lists. Therefore, we will proceed in two states: first we take all possible lists of integers, and then we qualify the list we discuss using an implication. We start writing our formula in this way.

```
forall l : list Z, ~ (l = nil) -> ...
```

We can now think about the formula we want to put in the dotted part of the formula. There sentence we want to translate says *there is always a number*. This part of the sentence is going to be translated using the following.

```
exists x, ...
```

Now, we need to express that this number satisfies two properties. First, we want to express that the chosen number is in the list, and second we want to express that the chosen number is smaller than all the others.

To express that the number is in the list, we use the predefined predicate `In`, so we write `In x l`.

To express that the number is smaller than any other element of the list, we want to discuss about every element in the list, so we want again to write a universal quantification, restricted to all elements of the list.

```
forall y, In y l -> ...
```

Then we want to say that `x` is smaller than `y`. Here we have to be careful if we want to state a formula that is actually provable. The natural language sentence says *that is smaller than all the others*. Resolving pronouns in this sentence, *that* would be represented by `x` and all the others would be represented by `y`. But to express that `x` is smaller than `y`, we should avoid writing `x < y`, because `x` actually be equal to `y`, because `y` can be taken among all the elements of the list. We should rather use `x <= y`.

So to finish the translation of this natural language sentence into a logical formula, we obtain the following text:

```
forall l : list Z, ~(l = nil) ->
  exists x, In x l /\ forall y, In y l -> x <= y
```

There are quite a few common mistakes that one makes when writing such a formula. One of the first mistakes is to get the parentheses in the wrong place. For better readability, we tend to lower the number of parentheses we use, but it may worth the effort to write too many parentheses at the first try and check with the computer that the formula can be written with parentheses.

The second common mistake that people make often is to use an `/\` operator in place of an `->` operator. The rule of thumb is as follows: as an immediate subterm of an existential quantification, it is more frequent to have an `/\` operator, and at an immediate subterm of a universal quantification, it is more frequent to have an `->` operator.

If we consider the following formula:

```
forall l : list Z, ~(l = nil) ->
  exists x, In x l -> forall y, In y l -> x <= y
```

It is easy to prove that formula by taking as witness any number that *is not* in the list. the implication `In x l -> forall y, In y l -> x <= y` is easy to prove, because the left hand side of the implication is false. So this mistaken formula does not mean that we found an element that is smaller than all elements of the list.

In the end, we know that a logical formula is right when we are certain we can prove it and when we can use it in other proofs.

To prove the formula we used in this section, we need to exhibit a function that computes the minimum element of a list, and then to show that this element satisfies the required property. With what you already know about programming with list and testing numbers for comparison, you can alsready define such a function.

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 to whether the connectives appears in something we want to prove or in something we already know.

3.1 Stating a logical formula to prove

It is better to show that in an example.

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

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. The next line `Proof.` is mostly useless, but we will keep the habit of writing it as it can be used as a marker to make the proof script more readable.

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 a *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.

There are a few other easy cases that are recognized by the proof system, the command to solve these easy cases is called `easy`. For beginners, it is sometimes puzzling, because it often fails to prove facts that seem obvious to us, while it may solve questions that require a little thinking on our side (especially when the fact requires some computation).

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 the 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 cases:

```
  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 choose to name `x`. Because `x` is taken arbitrarily, everything we prove about it is universal.

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: Z, P x -> Q x
  ================
   Q 3
apply H.
  H : forall x: Z, 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 property of interest. Here is an example:

```
  H : exists x : Z, Znth l x = Some v.
   ======================
    C
destruct H as [w Pw].
  w : Z
  Pw : Znth l w = Some v
  ========================
   C
```

In the goal before the `destruct` tactic, there is no integer that satisfies the property. In the goal after the tactic, there is an integer 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 `Znth`.

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 : Z, f (f x) = g (x + x)
  ====================
   g (2 + 2) = E
rewrite <- H.
  H : forall x : Z, 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 `Z -> Z -> 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 integers that expresses that when `n` is a multiple of 2 then `n * n` is also a multiple of 2.
3. Write a formula using integers 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. define a predicate `odd` of type `Z -> Prop` that characterize odd numbers like 3, 5, 37.
5. Assuming there exists a type `T` used to represent integers and a function `T_to_Z` of type `T -> Z`, which maps any element of `T` to the element of `Z` 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 `Z`.
6. Write the script that proves the following formula

   ```
   forall P Q : Z -> Prop,
   forall x y : Z, (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 : Z -> Prop, (forall x, P x) ->
    exists y : Z, 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`. You will need a theorem to reason about associativity of multiplication between integers. Use `Search (_ * _ * _).` to find such a theorem.
10. Write the script that proves that when `n` is odd, then `n * n` is also odd. Again, use `Search` to find relevant theorem.

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`