Verifying programs and proofs
part III. prove program properties

Yves Bertot

January 2025

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 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, we described a function `Znth` that extracts the nth element of a list for suitable values of n, and returns `None` when n is out of bounds. We want to prove that this function behaves without producing `None` when the argument is inbound, but for this, we need to also have a way to compute the size of list. The auxiliary function is `Zlength`, and this function is already defined in the `ZArith` library.

```
Lemma Znth_safe :
  forall l n, 0 <= n < Zlength l -> Znth l n <> None.
```

This is not enough, because the trivial function that always returns `Some 0` also satisfies this logical statement.

We need to add more properties to capture the intended behavior, but for the very basic functions, it is hard to go beyond expressing exactly what is in the code. Here is an attempt at capturing the intended behavior, in relation to size of lists.

```
Lemma Znth_cat l1 l2 x :
  Znth (l1 ++ x :: l2) (Zlength l1) = Some x.
```

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 satisfied 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 `Znth_safe`).

2 Reasoning about integers and bounds

In the examples used in this chapter, we will need to reason about small operations on integer values, adding 1 or subtracting 1 and comparing with bounds. This will often be solved by an automatic tool called `lia`. As long as we write only formulas in which we do not multiply variables with variables, this automatic tool `lia` will usually prove the facts that we wish to prove (as long as they are true!).

Here is an example of a statement that we will delegate to `lia`.

```
Lemma lia_example (l : list Z) x : 0 <= x < Zlength l ->
  0 <= 1 + x < 1 + Zlength l.
Proof.
intros xbounds.
lia.
Qed.
```

The acronym `lia` stands for *linear integer arithmetic*, this means that this proof procedure is adapted to working on formulas that have the following shape:

```
  x * 3 + y * 5 <= z * 7 + 1
```

and when these formulas are accumulated in the goal context. The words *linear* expresses that multiplications are only accepted when one of the factors is an integer constant.

The initial example in this section `lia_example` contains a fragment that is not a variable, it is a function call. Such a subterm is considered as yet another variable.

For comparisons between integers, there are two different ways to write them. For instance, `x <= y` can also be written as `(x <=? y) = true` and `x < y` can also be written as `(x <=? y) = false`. The `lia` proof procedure is designed to understand these different ways to express the same property.

Comparisons are also involved in functions that compute the minimum and maximum of two numbers. The functions `Z.min` and `Z.max` are also handled gracefully by `lia`.

3 Reasoning about plain and recursive functions

When we want to reason about the behavior of a function, we need to *unfold* that function. Functions are defined in two different ways in the proof system, one way for plain function and a different way for recursive functions. We have two different tactics to perform unfolding on these functions. To illustrate this, we will used the following functions.

```
Fixpoint Zlist_min (l : list Z) : option Z :=
match l with
| nil => None
| a :: tl =>
  match Zlist_min tl with
  | None => Some a
  | Some b => if a <=? b then Some a else Some b
  end
end.

Definition Zlist_min_w_k k l :=
  match Zlist_min l with
  | Some a => Z.min a k
  | None => k
  end.
```

The function `Zlist_min` will take a list of integer values and return the minimum of that list. However, there is no good choice for the minimum of a list when this list is empty, so that the returned value is actually in an option type. When the input list is empty, the function `Zlist_min` returns `None`.

The function `Zlist_min_w_k` returns the minimum of the value computed in the previous function and a chosen value. Such a function can be handy if we already know that all values present in the list are bounded by some upper bound.

The tactic designed to *unfold* plain definitions is simply called `unfold`. You have to give the name of the function you want unfold, and you may also specify which occurrence in the goal want to unfold (please refer the proof system documentation to see how to do that) . The tactic designed to unfold recursive tactics is called `simpl`. The proof system will not complain if you use `unfold` on a recursive function, but the resulting goal will be quite unpleasant to read. Actually, the name of the tactic `simpl` was probably chosen to express that the result is simpler than when using the `unfold` tactic.

Here is an example:

```
Lemma Zlist_min_w_k_under : forall k l, Zlist_min_w_k k l <= k.
Proof.
intros k l.
unfold Zlist_min_w_k.

  (1 / 1) ======================
  match Zlist_min l with
  | Some a => Z.min a k
  | None => k
  end <= k.
Abort.
```

We see that the definition of `Zlist_min_w_k` has been expanded.

For the `simpl` tactic, what happens is a bit more difficult to explain. This tactic will only modified a sub-expression containing the call of a recursive function if there is some computation to perform. So if the goal contains an expression of the form `Zlist_min l`, nothing happens on this expression, because we don't know whether the list is empty or not. On the other hand, if the goal ocontains an expression of the form `Zlist_min (a :: tl)`, then `simpl` will replace this expression with another where execution has happened, reducing some of the pattern-matching constructs, etc. Here are a few examples.

```
Lemma Zlist_min_simpl_sandbox a tl : Zlist_min (a :: tl) <> None.
Proof.
simpl.

  (1 / 1) ======================
  match Zlist_min tl with
  | Some b => if a <=? b then Some a else Some b
  | None => Some a
  end <> None.
Abort.
```

We see that the first pattern matching construct in `Zlist_min` has already been processed in the expression that replaces `Zlist_min (a :: tl)`.

4 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 execution cases.

4.1 The `destruct` tactic

The function `Zlist_min` actually contains three pattern-matching constructs. The first pattern-matching construct analyzes the input list, it exhibits a first branching opportunity in the code of the function. The second pattern-matching function analyzes the result of the recursive call on the remainder list, the third pattern matching construct shows how the returned value is chosen, by comparing between the minimum of the remainder list and the first element of the list.

To illustate uses of this tactic, we will restart the proof that we used as an example for unfolding.

```
Lemma Zlist_min_w_k_under : forall k l, Zlist_min_w_k k l <= k.
Proof.
intros k l.
unfold Zlist_min_w_k.

  (1 / 1) =====================
  match Zlist_min l with
  | Some a => Z.min a k
  | None => k
  end <= k.
```

We know wish execution to make more progress, but for this we need to consider two possible cases, corresponding to the patterns in the program text. Essentially we want to reason about the case where `Zlist_min l` has value `Some x` and separately about the case where `Zlist_min l` has value `None`. This is done by typing the following command.

```
destruct (Zlist_min l) with [ x | ].
```

We need to respect the order of the patterns present in the goal and provide variable names for those patterns that contain variables. As a result, the proof system generates two goals. In each of the goals, the pattern matching expression on the *destructed*

We wish to show that when this function returns `None`, necessarily the list must be empty.

```
Lemma Zlist_min_None (l : list Z) :
  Zlist_min l = None -> l = nil.
Proof.
destruct l as [ | a tl].
Goal 1
  (1 / 2) =================
  Zlist_min nil = None -> nil = nil
```

The variable `l` is replaced by the values that it can take. Either it is the empty list, or it is a list of the form `a :: tl`. The fragment of the command `as [ | a tl]` that we wrote in the command helps choose names for subcomponents in the second case. The `nil` case is always given first.

So we have a first goal in which `l` is replaced by `nil`. We can ask the system compute the value of `Zlist_min nil` by typing the command `simpl`.

```
simpl.
Goal 1
   (1 / 2) ================
   None = None -> nil = nil
```

We now have an implication in our goal. Implications are usually useful because they give us added information, but in this case the added information is useless, because we already new that `None` is equal to itself. It is okay, because we need to prove is also very easy, and we can use a tactic adapted for that.

```
easy.
```

This tactic completes the first branch of the proof, and we can now address the second branch.

```
Goal 1
  (1 / 1) ==================
  Zlist_min (a :: tl) = None -> a :: tl = nil
simpl.
Goal 1
  (1 / 1) =================
match Zlist_min tl with
| Some b => if <=? b then some a else Some b
| None => Some a
end = None -> a :: tl = nil
```

In this case, we still require for the execution to progress using `simpl`. We obtain a new pattern-matching construct and still ask for the execution to progress, this time by requesting that the value of `Zlist_min` should be analyzed.

```
destruct (Zlist_min tl) as [b | ].
Goal 1
  (1 / 2) =================
  (if a <=? b then Some a else Some b) = None -> a :: tl = nil
```

We observe here that the pattern-matching construct has been replaced by the branch corresponding to the case where `Zlist_min tl` has value `Some b`. If we continue with the analysis corresponding to the next pattern-matching construct (the `if-then-else`), we see that we now have three goals.

```
destruct (a <=? b).
Goal 1
  (1 / 3) =================
  Some a = None -> a :: tl = nil
```

In this goal, we have an assumption that `Some a = None`, but this assumption contains an inconsistency. A value of the form `Some _` is not of the value of the form `None`, because these are the two distinct forms of the `option` datatype. This equality is inconsistent and the `easy` tactic is designed to take advantage of this kind of inconsistent hypothesis. It is a good think that there is an inconsistent hypothesis here, because we would never be able to prove the conclusion which expresses that an empty list is non-empty.

In fact, all three goals we to prove now have this kind of inconsistent hypothesis, and they can all be solved by the easy tactic. We can express this in one line.

```
all: easy.
Qed.
```

4.2 The `destruct` tactic with the `eqn:` variant

Sometimes, we need `destruct` to add some information in the context to describe in which case we are. For instance, we may need to show that the minimum of a list is smaller than the first element. If we do not know the outcome of the comparison, we will not be able to conclude.

```
Lemma Zlist_min2 a b: Zlist_min (a :: b :: nil) = Z.min a b.
Proof.
simpl.
Goal 1
  (1 / 1) =================
  if a <=? b then Some a else Some b
```

From this goal, if we just peform the plain `destruct` operation we obtain two goals that will not be able to prove. Let us just observe the first goal that would be created.

```
destruct (a <=? b).
Goal 1
  a, b : Z
  (1 / 2) ================
  Some a = Some (Z.min a b)
```

In this first goal, the `if-then-else` constructed has been replaced by the first branch, but the context of the goal only contains the information that `a` and `b` are integers. There is some information missing, the expression was initially a pattern-matching construct, which has been replace by one of its branches, and the test that made this branch to be chosen does not appear as extra information. We need to undo this step.

The solution to this problem is to request that the `destruct` tactic adds information to the context which indicates in which case the tested expression is. This is done by adding the `eqn:` *name* modifier to the destruct command. With this modifier, the `destruct` tactic adds an extra hypothesis called *name* in the goal's context that indicates in which case we are. To illustrated, we can restart the proof above and proceed.

```
Lemma Zlist_min2 a b: Zlist_min (a :: b :: nil) = Z.min a b.
Proof.
simpl.
Goal 1
  (1 / 1) =================
  if a <=? b then Some a else Some b
```

We now use the `destruct` tactic with the `eqn:` modifier.

```
destruct (a <=? b) eqn:comparison_case.
Goal 1
  a, b : Z
  comparison_case : a <=? b = true
  (1 / 2) ==============
  Some a = Some (Z.min a b)
```

We now have the same thing to prove, but there is this extra hypothesis in the context, `comparison_case`, which state that the boolean test returns `true`. This statement expresses that `a` is smaller than or equal to `b`. The tactic `lia` is designed to exploit this kind of information. We finish the proof of this branch in the following manner.

```
assert (a_min : a = Z.min a b).
Goal 1
  a, b : Z
  comparison_case : a <=? b = true
  (1 / 3) ==============
  a = Z.min a b
lia.
Goal 1
  a, b : Z
  comparison_case : a <=? b = true
  a_min : a = Z.min a b
  (1 / 2) ==============
  Some a = Some (Z.min a b)
rewrite <- a_min.
easy.
```

The other case can be treated similarly, we can finish this whole proof using the following script.

```
assert (b_min : b = Z.min a b).
Goal 1
  a, b : Z
  comparison_case : a <=? b = false
  (1 / 3) ==============
  b = Z.min a b
lia.
Goal 1
  a, b : Z
  comparison_case : a <=? b = true
  b_min : b = Z.min a b
  (1 / 2) ==============
  Some b = Some (Z.min a b)
rewrite <- a_min.
easy.
Qed.
```

It takes some experience to understand when `destruct` is enough and when `destruct` with the `eqn:` modifier is needed. For a beginner, it may be useful to use the latter more often. Most of the time, algorithms perform tests for important reasons, so it is necessary to record that a test has been done in the context when reasoning about a test present in an algorithm.

4.3 the `induction` tactic

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

In mathematics, a natural number is an integer that is larger than or equal to 0. In early math classes, students learn to prove properties about all natural numbers by stating that when a property is satisfied for 0 and when this property can be shown to hold for any number of the form $n + 1$ if it already holds for n. In type theory based proof assistants, induction is generalized to other kind of types and sets.

In this section, we will concentrate on proof by induction concerning lists. It relies on the following principle, which is germane to the one concerning proofs by induction on natural numbers.

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

In practice, when performing a proof by induction on lists, we also two cases to study, the first case for the situation where the considered list is `nil` and the second for the situation where the value has the form `a :: l` for some `a` and `l`. 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 `l` already satisfies the expected predicate.

We can now study such a proof by induction concerning the concatenation of two lists. To make things clearer, we can recall the definition of list concatenation.

```
Locate "_ ++ _".

"x ++ y " := (app x y)  : list_scope
                      (default interpretation)
Print app.

fun A : Type =>
fix app (l m : list A) struct l : list A := match l with
| nil => m
| a :: l1 => a :: app l1 m
end
     : forall [A : Type], list A -> list A -> list A
```

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

Here is the proof in Coq:

```
Lemma example_induction_app : forall A : Type (l : list A), l ++ nil = l.
intros A l.
induction l as [ | a tl Ih].
2 subgoals

  ============================
   nil ++ nil = nil

subgoal 2 is:
 (a :: tl) ++ nil = a :: tl
```

As expected, we have two cases where `l` is replaced either by `nil` or by `a :: tl`. Note that we gave the names for the variables containing the subcomponents `a` and `tl` and extra name for the induction hypothesis, `Ih`.

For the first case, immediate computation yields the result.

```
easy.
1 subgoal
```

```
  A : Type
  a : A
  l : list A
  Ih : tl ++ nil = tl
  ============================
   (a :: tl) ++ nil = a :: tl
```

at this point we have to look carefully at what is written in the goal's conclusion. The expression `(a :: tl) ++ nil` represents an application of `app` where the first argument is an expression of the cons form. In the code of the `app` algorithm, this should compute to a list of the cons form, where the first component is the element that was in the head, and the second componet is an application of `app` where the first argument is the sublist of the initial first argument. We can request for this computation to be performed by calling the `simpl` tactic.

```
simpl.
1 subgoal

  A : Type
  a : A
  l : list A
  Ih : tl ++ nil = tl
  ============================
   a :: tl ++ nil = a :: tl
```

Now that the structure of the equality left-hand side has changed, this side now contains an instance of the induction hypothesis. We can perform a rewrite using this hypothesis.

```
rewrite Ih.
1 subgoal

  A : Type
  a : A
  l : list A
  Ih : tl ++ nil = tl
  ============================
   a :: tl = a :: tl
```

This proof can easily be finished as follows.

```
easy.
Qed.
```

4.4 the `discriminate` tactic

The datatypes of boolean values, integers, 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 list.
Inductive list (A : Type) : Type :=
    nil : list A | cons : A -> list A -> list A
```

This description of lists means that lists are either of the form `nil cons A a tl` (written usually as `a :: tl`. Moreover, it also means that the form `nil` is not the form `a :: tl`[1]. As a result, any goal whose conclusion has the form `nil <> a :: tl` 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 `nil = a :: tl`. The tactic `discriminate` is part of the behavior of the `easy` tactic.

When an equality between different integer values appears as hypothesis in a goal, it is an instance of this pattern, and `easy` or `discriminate` can solve them.

4.5 the `injection` tactic

We often have to express that the `cons` constructor of `list` is*injective*. In other words, if it gives equal outputs for two sets of inputs, then the inputs must be pairwise equal. Here is an 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.

5 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.

[1]This is also simply a consequence from the fact that we can define expressions by pattern-matching on natural numbers! In some documents, this is also referred to as a *non-confusion* property.

5.1 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 : Z) := 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
```

5.2 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 full text of the recursive function, possibly several times. To avoid this, there are tactics specifically tuned to handle recursive functions. These tactics are called `simpl` and `cbn`.

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

```
Lemma example_induction_app : forall A : Type (l : list A), l ++ nil = l.
intros A l.
induction l as [ | a tl Ih].
2 subgoals

  ============================
   nil ++ nil = nil

subgoal 2 is:
 (a :: tl) ++ nil = a :: tl
induction n.
easy.
1 subgoal

  A : Type
  a : A
  l : list A
  Ih : tl ++ nil = tl
  ============================
   (a :: tl) ++ nil = a :: tl
```

Here, we can request that Coq performs a little computation with `(a :: tl) ++ nil`. We simply need to call the `simpl` tactic:

```
simpl.
  ...
    A : Type
  a : A
  l : list A
  Ih : tl ++ nil = tl
  ============================
   a :: tl ++ nil = a :: tl
```

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 Ih; reflexivity.
Qed.
```

Often, cbn is practical to use, because it makes it possible to control what functions will be unfolded. Interested readers should look at the Coq documentation concerning this tactic.

5.3 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_app :
  forall n tl, nil ++ 1 + n :: tl = n + 1 :: tl.
intros n tl.
1 subgoal
  ...
  ============================
   nil ++ 1 + n :: tl = n + 1 :: tl
change (nil ++ 1 + n :: tl) with ((1 + n) :: tl).
  ...
  ============================
  1 + n :: tl = n + 1 :: tl
```

5.4 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, perform replacement, and keep the obligation to prove the equality between the two expressions for later. For this we use the replace tactic. This tactic produces a second goal, because the equality between the two expressions still needs to be proved.

We can finish the proof started in the previous section using this replace tactic.

```
  ...
  ============================
  1 + n :: tl = n + 1 :: tl
replace (1 + n) with (n + 1).
2 subgoals
  ...
  ============================
   1 + n :: tl = n + 1 :: tl

subgoal 2 is:
  n + 1 = 1 + n
easy.
ring.
```

Here the first goal is solved by `easy` and the second is solved by `ring`, because the this second goal only relies on commutativity of addition, a ring property.

6 A complete proof

We first define a function that has input a number n and a list l, and returns a list containing elements of l multiplied by successors of n. The first element of the result is the first element of l multiplied by $n+1$, the second element of the result is the second element of l multiplied by $n+2$, and so on. This can be described easily as a recursive function on lists:

```
Require Import ZArith List.

Open Scope Z_scope.

Fixpoint mulsl (n : Z) (l : list Z) :=
  match l with
  | a :: l1 => a * (n + 1) :: mulsl (n + 1) l1
  | nil => nil
  end.
```

We know wish to show that if the fist argument is positive, this function is actually injective in its second argument. Here is the statement

```
Lemma mulsl_injective n l1 l2 : 0 < n ->
  mulsl n l1 = mulsl n l2 -> l1 = l2.
Proof.
```

We wish to perform this proof by induction on one of the lists `l1` and `l2`. When observing the function `mulsl` carefully, we see that the list that must be shown to be equal to `l1` as it varies cannot be the same. So `l2` should not be fixed in the context of the goal, but rather a universally quantified variable. Similarly, we see that in recursive calls the variable `n` changes, so again `n` should not be fixed in the context. To make this change of shape of the goal, we use the `revert` tactic.

```
revert n l2.
induction l1 as [ | a l1 IH].
```

The proof by induction makes use of a predicate on lists of integers that has the following shape.

```
fun l => forall (n : Z)(l : list nat),
          0 < n -> mulsl n l = mulsl l2 -> l = l2
```

We now have two goals, where the first one corresponds to the case where this predicate is applied to the empty list `nil`, and the second one is applied to a non-empty list `a :: l1`. Thus, the first goal has the following text

```
  ============================
  forall (n : nat) (l2 : list nat), 0 < n ->
     mulsl n nil = mulsl n l2 -> nil = l2

  intros n [ | b l2].
    intros; easy.
  simpl.
  discriminate.
```

The line `intros n [ | b l2]` is equivalent to `intros n tmp; destruct tmp as [ | b l2]`. Because of this line, there are two goals. The first one requires that we prove `nil = nil`, done by `easy`. The second goal contains an implication of the form `mulsl n nil = mulsl n (b :: l2) -> ...` After applying the computation rules of `mulsl` this boils down to an equality between different constructors of a datatype (the `list` datatype). Such a premise is self-contradictory and `discriminate` recognizes it. We wrote `simpl` to make the computation happen so that a human observer of the proof can see the result, but this step is not mandatory and using `discriminate` without the `simpl` step would already work.

The second goal generated by the induction step concerns `a :: l1` (the `cons` form), but since it is an induction proof, we can already use the fact we wish to prove for the sublist `l1`. Thus the goal has the following form:

```
  a : Z
  l1 : list Z
  IH : forall (n : nat) (l2 : list nat),
       n < 0
       mulsl n l1 = mulsl n l2 -> l1 = l2
  ============================
  forall (n : Z) (l2 : list Z),
  n < 0
  mulsl n (a :: l1) = mulsl n l2 -> a :: l1 = l2
```

So in this goal, the fact that we already know about `l1` is embodied in the hypothesis `IH` (as we prescribed in the `induction` call).

We can now fix a value `n` for the universal quantification, and reason on the possible cases for the list. In this case we know that `l1` is of the `cons` form, after computation `mulsl` will return a result of the `cons` form. If the second list if of the `nil` form, the computation of `mulsl` returns `nil` and we the equality between `mulsl` results is again an inconsistent one, and the goal is easily solved using `discriminate`.

```
intros n ngt0 [ | b l2]; simpl.
  discriminate.
```

It remains to study the case where both lists are in `cons` form. The goal then has the following shape:

```
  a : Z
  l1 : list Z
  IH : forall (n : Z) (l2 : list Z),
       mulsl n l1 = mulsl n l2 -> l1 = l2
  n : Z
  ngt0 : 0 < n
  b : Z
  l2 : list Z
  ============================
  a * (n + 1) :: mulsl (n + 1) l1 =
  b * (n + 1) :: mulsl (n + 1) l2 ->
  a :: l1 = b :: l2
```

Because of the simpl step after introducing b and l2, both computations of mulsl have produced results that are both in cons form. We know need to use the injection tactic to make this equality between two lists transform into two equalities, the first concerning the first elements of these lists, and the second concerning the results of these lists.

```
intros Heq; injection Heq as heads tails.
```

The new hypothesis tails, in combination with the induction hypothesis, will make it possible to prove that l1 and l2 are equal. We use this fact directly, knowing that the proof will be kept for later.

```
replace l2 with l1.
```

Similarly, the fact a = b should be deductible from the hypothesis heads. To make this step, we look in the database for a theorem that mentions equalities between results of multiplications. The command for this database inquiry is as follows:

```
Search (_ * ?x = _ * ?x).
```

By using two instances of the named place-holder ?x, we are able to pinpoint exactly a theorem where the same expression appears at the corresponding place on both side of the equality. The result of the Search command is as follows:

```
heads: a * (n + 1) = b * (n + 1)
Z.mul_reg_r: forall n m p : Z, p <> 0 -> n * p = m * p -> n = m
Z.mul_cancel_r: forall n m p : Z, p <> 0 -> n * p = m * p <-> n = m
```

The hypothesis tails is listed, but also a theorem about an equivalence between equalities, under the condition that the common factor p is non-zero. We first modify the hypothesis heads using this theorem, and we then use the modified hypothesis to modify the goal. What we obtain is a trivial equality, and the easy tactic can get rid of this goal.

```
  rewrite Z.mul_cancel_r in heads.
    rewrite heads; easy.
```

At this point, we still have to prove that the common factor is non-zero, and this can be done by asking a tactic that is adapted to reason about ordered formula in integers.

```
  lia.
```

We still have to prove that the replacement of l2 by l1 is justified. For this we use the induction hypothesis. However, in this induction hypothesis, the universally quantified variable does not appear in the hypothesis' conclusion, so we need to guide the proof system by showing how the instantiation will happen. Here is one possibility.

```
apply IH with (n := n + 1).
exact tails.
Qed.
```

7 Exercises

Not ready yet.

8 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. Version française `http://www-sop.inria.fr/members/Yves.Bertot/coqartF.pdf`

[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`