Verifying programs and proofs
part IV. Proofs about arithmetic programs

Yves Bertot

September 2012

1 Motivating introduction

With the tactics we have learned so far, we can prove many facts about arithmetic expressions, but we need to do it step by step, combining simple theorems manually. In this chapter, we will learn to use automatic proof tactics that take care of big formulas in one shot.

In a first section, we will observe different types of numbers that are available in the Coq system. First, the datatype of natural numbers is not suited for large computations; second a datatype with negative numbers makes it possible to handle subtraction in a more regular way. We can also consider a type of real numbers, but this type is not really a datatype, in the sense that many computations about these numbers cannot be described algorithmically.

2 A variety of number types

From the computer complexity point of view, the type of natural numbers is quite inefficient. The amount of memory needed to store a natural number is proportional to the value of this number. Thus, to store the number 10 in memory, we use approximately 20 words of memory, to store the number 20, we use approximately 40 words of memory and so on.

Obviously, there exist more efficient ways to store numbers. For instance, the decimal representation that we all learned in school uses an amount of memory that is proportional to the log of the number represented. Thus it takes 3 + 1 digits to write the number 1000 and 6+1 digit to write the square, 1000000. The Coq system also provides a datatype that has this *logarithmic* memory consumption characteristic: the type of integers `Z`.

The type of integers is loaded in Coq if we execute the following command (please mind the exact use of upper case letters):

```
Require Import ZArith.
```

The datatype is described in two stages. The first stage only describes positive integers in binary form.

```
Print positive.

Inductive positive : Set :=
  xI : positive -> positive
| xO : positive -> positive
| xH : positive
```

There are three cases in this datatype. The case `xH` represents the number 1, the case `xI` represents the function $x \mapsto 2 \times x + 1$, and the case `xO` represents the function $x \mapsto 2 \times x$. For instance, 1 is `xH`, 2 is `xO xH`, 3 is `xI xH`, 4 is `xO (xO xH)`, and 10 is `xO (xI (xO xH))`.

To construct integers, we have a second datatype that chooses between three cases again, but this time to express whether the number is zero, positive, or negative.

```
Print Z.
Inductive Z : Set :=
  Z0 : Z | Zpos : positive -> Z | Zneg : positive -> Z
```

Thus the number 10 is actually represented by `Zpos (xO (xI (xO xH)))`. We need to get accustomed to the fact that the same number is represented differently depending on the type in which we want to represent it.

```
Check Zpos (xO (xI (xO xH))).
10%Z : Z

Check S (S (S (S (S (S (S (S (S (S 0))))))))).
10 : nat
```

To use integer numbers by default, we need to tell the parser that we are writing in the corresponding scope:

```
Open Scope Z_scope.
```

If we want to go back to using natural numbers by default, we can revert this directive by typing `Close Scope Z_scope`, but this command is not used for what follows.

```
Check Zpos (xO (xI (xO xH))).
10 : Z

Check S (S (S (S (S (S (S (S (S (S 0))))))))).
10%nat : nat
```

So the scope mechanism instructs the printer to convert a number to its representation in one type or the other. We can change scopes locally by using the percent sign, as in `%Z`.

The scope mechanism is also responsible for understanding the arithmetic operations like `+`, `*`, or `-`, it adds a power function `^`.

The integer data representation is efficient enough that we can compute very large numbers in reasonable time, a property that was not true for natural numbers.

```
Compute 1001 * 1001 * 1001 * 1001.
     = 1004006004001 : Z
```

3 Proving equalities

If you execute the command `Require Import Arith.`, the Coq systems loads extra theorems and tactics to reason about natural numbers. If you execute the command `Require Import ZArith.`, it load extra theorems and tactics to reason about integers.

Among the available reasoning capabilities, there is the `ring` tactic. For natural numbers, it handles self-contained equalities between formulas containing natural number constants, addition and multiplication. Here is an example:

```
Close Scope Z_scope.
Lemma ex1 :
  forall x y : nat, (3 * x + 2) * y = 2 * (x * y + y) + y * x.
intros x y.
  x : nat
  y : nat
  ======================
   (3 * x + 2) * y = 2 * (x * y + y) + y * x
ring.
No more subgoals.
```

The equality has to be self-contained: its truth should not depend on an hypothesis. For instance, the following goal is provable, but not directly.

```
  x : nat
  y : nat
  H : 2 * (y * y) = x * x
  ========================
   3 * x * x = 2 * (y * y) + 2 * x * x
ring.
Error: Tactic failure: not a valid ring equation.
```

For integers, the `ring` tactic solves the same family of problems, but with two more allowed operation: subtraction and exponentiation by a constant integer.

```
Open Scope Z_scope.

Lemma ex2 : forall x y, 2 * (y * y) = x * x ->
  2 * (x - y) * (x - y) = (2 * y - x) * (2 * y - x).
intros x y H.
```

```
replace (2 * (x - y) * (x - y)) with
   (2 * (x * x) - 4 * (x * y) + 2 * (y * y)).
2 subgoals
  ...
  H : 2 * (y * y) = x * x
  ============================
   2 * (x * x) - 4 * (x * y) + 2 * (y * y) =
   (2 * y - x) * (2 * y - x)

subgoal 2 is:
 2 * (x * x) - 4 * (x * y) + 2 * (y * y) = 2 * (x - y) * (x - y)

replace ((2 * y - x) * (2 * y - x)) with
  (4 * (y * y)- 4 * (x * y) + x * x).
3 subgoals
  ...
  H : 2 * (y * y) = x * x
  ============================
   2 * (x * x) - 4 * (x * y) + 2 * (y * y) =
   4 * (y * y) - 4 * (x * y) + x * x

   ...
rewrite <- H.
3 subgoals
  ...
  ============================
   2 * (2 * (y * y)) - 4 * (x * y) + 2 * (y * y) =
   4 * (y * y) - 4 * (x * y) + 2 * (y * y)

  ...
```

At this point, the formulas on both sides of the equality are quite complex, but usual algebraic computation on these formulas makes it possible to see that they are equal. The `ring` tactic does that. The other two goals that were generated by the `replace` tactic can also be verified using simple algebraic computation. Again, the `ring` tactic suffices for this task. So this proof is completed with two uses of the `ring` tactic. So combining `replace` and `ring` makes it possible to progress quickly through computation of numeric formulas.

4 Proving comparisons

Comparison problems are formulas with implications where the premise and conclusion of the implications are of the form `A < B`, `A = B`, `A <= B`, where `A` and `B` are numeric formulas using addition, multiplication, and subtraction. If the formulas contain arbitrary multiplications and the numbers are either natural numbers or integers, comparison problems are known to be undecidable

in general. In other words, it is impossible to construct a tactic that will solve all true comparison problems.

But if `A` and `B` contain only additions and subtractions, then comparison problems become decidable. This is equivalent to accepting multiplication by numeric constants. There is a tactic that performs this kind of proofs for natural numbers and integers, it is called `lia`. The comparison problems accepted by `lia` are called linear comparison problems. To be able to use this tactic you need to load another package, named `Psatz`.

This tactic is included in the library `ZArith`, but not in the library `Arith`. To load it, you may have to require explicitely, as is done in the following example.

```
Require Import Psatz.

Lemma lia_ex : forall x y, x > y -> 2 * y > 3 * x -> False.
1 subgoal

  ============================
   forall x y : nat, x > y -> 2 * y > 3 * x -> False

intros; lia.
No more subgoals.
Qed.
```

5 Advanced proof by induction for numbers

For proofs by induction, it is important to make sure that the induction hypothesis is strong enough for the proof we need to perform. The most basic way to make this induction hypothesis stronger is to prove a strong statement by induction. Another approach is to rely on generic theorems that systematically provide strong induction hypotheses. The next two sections illustrate these approaches.

5.1 Strengthening induction

When proving statements by induction, it may be easier to prove stronger statements. This may seem paradoxal, because stronger statements will be harder to prove. However, in proofs by induction the fact to be proved is repeated in the induction hypothesis, which also becomes stronger.

Here is an example to illustrate this idea. The first few lines of the following proof are used to show that plain induction fails.

```
Require Import Arith Psatz.

Fixpoint mod2 (n : nat) :=
  match n with S (S p) => mod2 p | a => a end.
```

```
Lemma mod2_lt_2 : forall n, mod2 n < 2.
Proof.
induction n; simpl; [lia | ]
  n : nat
  IHn : mod2 n < 2
  ============================
   match n with
   | 0 => S n
   | S p => mod2 p
   end < 2
destruct n;[lia | ].

  n : nat
  IHn : mod2 (S n) < 2
  ============================
   mod2 n < 2
```

Invoking `destruct` is natural because the goal's conclusion contains a pattern-matching construct. But the goal that we obtain is not easy to prove: there is a mismatch between the statement that concerns `n` and the hypothesis that concerns `S n`. We should abort this proof and start afresh, with a stronger statement.

The statement that we will choose to prove is not only that `mod2 n` is less than 2, but also that `mod2 (S n)` is less than 2. We use the tactic `assert` to introduce the strong statement.

```
Lemma mod2_lt : forall n, mod2 n < 2.
intros n; assert (H : mod2 n < 2 /\ mod2 (S n) < 2);
 [ | destruct H; assumption].
  n : nat
  ============================
   mod2 n < 2 /\ mod2 (S n) < 2
induction n;[simpl; lia | ].
  n : nat
  IHn : mod2 n < 2 /\ mod2 (S n) < 2
  ============================
   mod2 (S n) < 2 /\ mod2 (S (S n)) < 2
```

The new goal is more complex: we now have to prove a conjunction instead of an equality. But the induction hypothesis is also a conjunction, and the right-hand side of the hypothesis coincides with the left hand-side of the goal's conclusion. Thus, we can destruct the hypothesis and solve the first part of the proof quickly.

```
destruct IHn as [H1 H2]; split;[assumption | ].
  n : nat
  H1 : mod2 n < 2
```

```
  H2 : mod2 (S n) < 2
  ============================
   mod2 (S (S n)) < 2
```

By computation, `mod2 (S (S n))` is the same as `mod2 n`. Thus, this goal is easily solved using the `assumption` tactic.

5.2 course-of-value induction

The example given in the previous section shows that we can have induction hypothesis not only on the predecessor of a number, but on the predecessor and the predecessor of the predecessor. Sometimes, we may want to have an induction hypothesis on all numbers smaller than a number.

There exists a theorem that covers this kind of need, but it applies in a more general setting. The name of the theorem is `well_founded_ind` and it applies to a variety of binary relations, as long as they satisfy the property to be "well founded". The relation `lt` satisfies this property. Thus we can combine two theorems together to obtain an induction principle that is stronger than the basic one we have already used.

```
Check well_founded_ind lt_wf.
well_founded_ind lt_wf
 : forall P : nat -> Prop,
   (forall x : nat, (forall y : nat, y < x -> P y) -> P x) ->
   forall a : nat, P a
```

Let's read carefully the statement of this theorem. It is universally quantified over a predicate `P`. It has only one case to cover (instead of two for a regular induction principle) but this case contains induction hypotheses for all numbers less than the number for which the property needs to be proved. If we manage to prove this single case, the predicate holds universally.

For the number 0, there is no other number that is less than 0, so there is no number for which induction hypotheses hold. So this is similar to the base case of the conventional induction principle.

This is illustrated in an example reasoning on an implementation of a division function on natural numbers. This function returns a pair containing the quotient and the remainder of the division. Note that this function is tuned to handle divisions by 0 gracefully. When dividing by 0, the quotient is set to 0 and the remainder is the number being divided.

```
Fixpoint div (n m : nat) :=
 match m, n with
   S m', S n' =>
   if leb m n then
      let (q, r) := div (n' - m') m in (S q, r)
    else
     (0, n)
```

```
 | _, _ => (0, n)
 end.
```

When this function computes it has recursive calls, where the second argument never decreases (the initial value is `m` the value of the second argument in the recursive call is also `m`). The first argument decreases by a variable amount. For instance, if `m` is 1, then `m'` is 0, and if we compute `div 7 1`, then the recursive calls are `div 6 1`, `div 5 1`, etc. On the other hand, if `m` is `3`, then `m'` is `2`, and if we compute `div 7 3`, then the recursive calls are `div 4 3` and `div 1 3`.

We can now prove a simple equality about this division function.

```
Lemma div_eq :
   forall n m, n = m * fst (div n m)  + snd (div n m).
intros n; induction n  as [n IHn] using (well_founded_ind lt_wf).
  n : nat
  IHn : forall y : nat,
        y < n -> forall m : nat,
                    y = m * fst (div y m) + snd (div y m)
  ============================
   forall m : nat, n = m * fst (div n m) + snd (div n m)
```

As we see here, the induction hypotheses states that every `y` smaller than `n` satisfies the property that we want to prove for `n`.

The rest of this proof is left as an exercise. Another exercise consists in proving that the remainder of the division is smaller than the divisor, when this divisor is not 0.

6 Advanced proof by induction on lists

For lists, the technique of strengthening induction applies as before. To illustrate this, let's look at a proof concerning reversed lists.

There are two ways to reverse lists, but only one is efficient.

```
Require Import List.

Fixpoint slow_rev_nat (l : list nat) : list nat :=
  match l with
    nil => nil
  | a::l' => slow_rev_nat l' ++ (a::nil)
  end.

Fixpoint pre_rev_nat (l l' : list nat) : list nat :=
  match l with
    nil => l'
  | a :: l' => pre_rev_nat l (a::l')
  end.
```

```
Definition rev_nat (l : list nat) : list nat :=
   pre_rev_nat l nil.
```

In the definition of `slow_rev_nat`, we use a function

`app`, with an infix notation `++` to concatenate two lists. It is easily to be convinced that `slow_rev_nat` reverses a list: when working on a list with a first element `a`, this element ends up in the end and the rest of the list is also reversed. So we can use it as a reference implementation, but it is slow, because concatenating takes a time proportional to the length of the first list being treated. In the end, the complexity of this function is quadratic.

The other function is more efficient: reversal takes place in linear time. We can see that by testing the functions on list of increasing length. So it is useful to have both functions available. The first one can be used in specifications, the second one in implementations.

If we want to express the correctness of the efficient function, we can just write the following statement.

```
Lemma rev_nat_correct : forall l, rev_nat l = slow_rev_nat l.
```

If we start our proof directly by an induction we discover that the statement loses its beautiful shape and the proof gets stuck in the recursive case.

```
induction l as [ | a l' IHl'].

  ======================
   rev_nat nil = slow_rev_nat nil
reflexivity.
simpl.

  IHl' : rev_nat l' = slow_rev_nat l'
  ======================
   rev_nat (a::l') = slow_rev_nat l'::(a::nil)
```

At this point, there is no simple way to show a correspondance between `rev_nat a::l'` and `rev_nat l'`; mainly because the latter is not a subterm of the former. This proof gets stuck.

To repair this proof, we can rely on a stronger statement, manipulating the function `pre_rev_nat` and quantifying over both its arguments.

```
Lemma rev_nat_correct : forall l, rev_nat l = slow_rev_nat l.
assert (forall l l', pre_rev_nat l l' = slow_rev_nat l ++ l').
 induction l.
  intros; reflexivity.
 intros l'; simpl.
  rewrite <- app_assoc.
 simpl; apply IHl.
intros l; unfold rev_nat; rewrite H, <- app_nil_end; reflexivity.
Qed.
```

7 Exercises

1. Here is a definition of division by 2 in natural numbers:

   ```
   Fixpoint div2 (n : nat) :=
    match n with S (S p) => S (div2 p) | _ => 0 end.
   ```

   Using the definition of `mod2` from these course notes, prove the following lemma:

   ```
   Lemma mod2_div2_eq n : n = mod2 n + 2 * div2 n.
   ```

2. Prove that `forall n, div2 (2 * n) = n`.

3. Prove that a number cannot be at the same time odd and even.

4. Write a function that sorts a list of integers, prove that it is correct (write a function that checks when a list is sorted and show that the output of the sorting function is sorted).

5. Consider the function

   ```
   Require Export ZArith List.
   Open Scope Z_scope.

   Definition myrand (x : Z) := Z.modulo (x * 3853) 4919.
   ```

   Write a recursive function to produce lists of integers of a given length and of the form:

   ```
   5::myrand 5:: myrand(myrand 5) :: myrand(myrand(myrand 5))::...
   ```

   Prove that the list has the required length. Produce a list of length 1000, and then sort it. What is the biggest list you can sort in less than 30 seconds?

6. Write a recursive function that removes duplicate elements in a sorted list, state its correctness, and prove that it is correct.