Verifying programs and proofs
part V. More data-structures

Yves Bertot

September 2012

1 Motivating introduction

Until now, we have only studied proofs about numbers and lists. Data-structures can become more complex than that and more complex data-structures will make it possible to have better complexity. We will illustrate this by looking more precisely at two kinds of data structures: positive numbers (with an algorithm for addition) and binary trees (with an algorithm for list sorting).

2 Recursion on positive numbers

2.1 Restrictions on programming

Positive numbers are represented by a datatype with three constructors. The constructor named `xH` is used to represent 1, the constructor named `xO` is used to represent the function that maps any x to $2x$, and the constructor named `xI` is used to represent the function that maps any x to $2x + 1$.

```
Require Import ZArith Arith.

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

For instance, `xO (xI (xO xH))` represents the number 10. We can check this by the following command:

```
Check xO (xI (xO xH)).
10%positive : positive
```

When writing recursive algorithms on this data-structure, we can only have recursive calls on the sub-components of the `xO` and `xI` constructors. When looking at the representations, this means that recursive calls are only allowed on the half (rounded down) of the initial argument of the function.

When designing algorithms, we need to take this into account. We can illustrate this for the conversion functions between types `nat` and `positive` (respecting the number being represented) and the operation of adding two numbers.

2.2 From positive numbers to natural numbers

To compute the natural number corresponding to a positive number, we can just write the interpretation using the constants 2 and 1, addition, and multiplication. Each case naturally relies on recursive calls with the expected argument.

```
Fixpoint pos_to_nat (x : positive) : nat :=
  match x with
    xH => 1
  | xO p => 2 * pos_to_nat p
  | xI p => S (2 * pos_to_nat p)
  end.
```

2.3 The function that adds one to a positive numbers

We now have to decompose the description of functions into three cases. For the function that just adds one goes along the following lines:

1. For the base case: if the input is 1 (represented by `xH`), the output must be 2 (represented by `xO xH`),

2. For the case where the input is even with half x_p, the output must be the odd number with the same half. In other words, if the input is `xO p` then the output is `xI p`,

3. For the case where the input is odd with half x_p, the output must be even. But we have $(2 \times x_p + 1) + 1 = 2 \times (x_p + 1)$, so we need to compute the result of adding 1 to x_p and then multiply by 2 using the constructor `xO`. In other words, if the input is `xI p` then the output is `xO (add1 p)`; assuming that `add1` is the function that we are defining. All this is described in the following definition:

```
Fixpoint add1 (x : positive) : positive :=
  match x with
    xH => xO xH
  | xO p => xI p
  | xI p => xO (add1 p)
  end.
```

2.4 Adding two positive numbers

When adding two numbers, we use the same algorithm as the one we learn at school to add numbers written in base 10, except that here is the base is 2 and we don't need to know addition tables. On the other hand, addition may involve a carry at anytime. So we will use a three argument function, the third argument being a boolean value indicating whther there is a carry or not. The algorithm involves a case analysis with the two arguments and with the carry, so in the end there are $18 = 3 \times 3 \times 2$ cases.

Let's first look at the case where there is no carry.

1. If one of the arguments is 1, then we can use the `add1` function,

2. If the two numeric inputs are odd, then there is a carry in the recursive call, the computation goes along the line
$$(2x + 1) + (2y + 1) = 2(x + y + 1)$$

3. If the two numeric inputs are even, then there is no carry in the recursive call. The computation goes along the line
$$2x + 2y = 2(x + y)$$

4. If only one of the inputs is odd, then the result is odd and there is no carry in the result

$$(2x + 1) + 2y = 2(x + y) + 1$$

Now let's look at the case where there is a carry.

1. If the two numeric inputs are 1, then the result is 3,
2. Otherwise, if one of the arguments is 1, then we can use the `add1` function, but on the half of the other argument:

$$(2x) + 1 + 1 = 2(x + 1)$$

$$(2x + 1) + 1 + 1 = 2(x + 1) + 1$$

3. If the two numeric inputs are odd, then there is a carry in the recursive call, the computation goes along the line

$$(2x + 1) + (2y + 1) + 1 = 2(x + y + 1) + 1$$

4. If the two numeric inputs are even, then there is no carry in the recursive call. The computation goes along the line

$$2x + 2y + 1 = 2(x + y) + 1$$

5. If only one of the inputs is odd, then the result is even and there is a carry in the recursive call

$$(2x + 1) + 2y + 1 = 2(x + y + 1)$$

We can now condense all of this reasoning in the algorithm

```
Fixpoint pos_add (x y : positive) (c : bool) : positive :=
 match x, y, c with
   xI x', xI y', false => xO (pos_add x' y' true)
 | xO x', xI y', false => xI (pos_add x' y' false)
 | xI x', xO y', false => xI (pos_add x' y' false)
 | xO x', xO y', false => xO (pos_add x' y' false)
 | xH, y, false => add1 y
 | x, xH, false => add1 x
 | xI x', xI y', true => xI (pos_add x' y' true)
 | xO x', xI y', true => xO (pos_add x' y' true)
 | xI x', xO y', true => xO (pos_add x' y' true)
 | xO x', xO y', true => xI (pos_add x' y' false)
 | xH, xH, true => xI xH
 | xH, xI y, true => xI (add1 y)
 | xH, xO y, true => xO (add1 y)
 | xI x, xH, true => xI (add1 x)
 | xO x, xH, true => xO (add1 x)
end.
```

2.5 Proving the correctness

We will first prove the correctness of the `add1` function. We use the map from positive numbers to natural numbers to express correctness. As usual, we only show that a function is correct by showing that it is consistent with other functions.

```
Lemma add1_correct : forall x, pos_to_nat (add1 x) = S (pos_to_nat x).
Proof.
```

This proof is done by induction on the positive number. Because the inductive type has three cases, the proof also has three cases.

```
induction x.
3 subgoals

  x : positive
  IHx : pos_to_nat (add1 x) = S (pos_to_nat x)
  ============================
   pos_to_nat (add1 x~1) = S (pos_to_nat x~1)

subgoal 2 is:
 pos_to_nat (add1 x~0) = S (pos_to_nat x~0)
subgoal 3 is:
 pos_to_nat (add1 1) = S (pos_to_nat 1)
```

The notations `x~1`, `x~0`, and 1 respectively stand for `xI x`, `xO x`, and `xH`.

For the first case, we can force the computation of the recursive function as follows, this makes the term in the induction hypothesis appear:

```
simpl.
  ...
  ============================
   pos_to_nat (add1 x) + (pos_to_nat (add1 x) + 0) =
   S (S (pos_to_nat x + (pos_to_nat x + 0)))
rewrite IHx; ring.
```

After rewriting with the induction hypothesis, we obtain an equality that is easily solved by the `ring` tactic.

Then the second case appears. In this case forcing the computation returns a simple equality.

```
  ...
 ============================
   pos_to_nat (add1 x~0) = S (pos_to_nat x~0)
simpl.
  ...
 ============================
   S (pos_to_nat x + (pos_to_nat x + 0)) =
   S (pos_to_nat x + (pos_to_nat x + 0))
reflexivity.
```

The third case is also easily solved using reflexivity.

```
reflexivity.
Qed.
```

When proving the correctness of addition, it is clever to use the regularity of the function to treat many cases at a time. we perform a proof by induction on the first argument, by cases on the second and third arguments. This develops all 18 cases in one shot. Then, a few of this cases are directly solved by computing on the equality between numbers using the `ring` tactic.

```
Lemma pos_add_correct :
  forall x y c, pos_to_nat (pos_add x y c) =
    pos_to_nat x + pos_to_nat y + if c then 1 else 0.
Proof.
induction x as [x' | x' | ]; intros [y' | y' | ] [ | ]; simpl; try ring.
```

In this combined tactic the notation `intros [ y' | y' | ]` is a shorthand for `intros y; destruct y as [ y' | y' | ]`.

This leaves only 14 goals, so 4 goals have been solved automatically by `ring`. We see that some of these goals mention `add1`, so it should be useful to also try rewriting with the correctness statement for that function. Let's restart the proof with the following combined tactic.

```
induction x as [x' | x' | ]; intros [y' | y' | ] [ | ];
  simpl; try rewrite add1_correct; try ring.
```

This leaves only 8 goals, so we managed to solve 6 extra goals systematically. The next idea is to use the induction hypothesis, when it exists. Let's retart the proof again with the following combined tactic.

```
induction x as [x' | x' | ]; intros [y' | y' | ] [ | ];
  simpl; try rewrite add1_correct; try rewrite IHx'; try ring.
```

This solves the goal.

```
Qed.
```

Note that the proof would not work as well if we add performed `intros x y c; induction x` as a first step. This yields weaker induction hypotheses and it turns out that they are too weak for the problem we have to solve.

3 Tree data-structures

Lists are nice data-structures for our proof work. They are easy to understand and their shape is reminiscent of arrays, which are used pervasively in conventional programming languages. So if our task was solely to reason about conventional programs, these lists would be useful. However, sometimes we need to program efficient algorithms to help our proof work and in this case lists are unsatisfactory.

The average cost of fetching a piece of data in a list is proportional to the length of that list. When using a binary tree instead of a list, or a data-structure with branching nodes, we can have more efficiency, because fetching a data in a tree (if we know where to look) can have a cost which is logarithmic in the number of elements stored in the tree.

Moreover, research in algorithmics shows that there are benefits in using *divide-and-conquer* approaches to solve problems, often making that some problems that had a high complexity end up with a much lower complexity, for instance remplacing a cost in n^2 by a cost in $n \times \ln n$. We will illustrate this with two implementations of sorting algorithms.

3.1 Insertion sort

Here is an implementation of sorting lists containing integers.

```
Fixpoint insert (a : Z) (l : list Z) :=
  match l with
    nil => a::nil
  | b::tl => if Zle_bool a b then a::b::tl else b::insert a tl
  end.

Fixpoint sort (l : list Z) :=
  match l with nil => nil | a::tl => insert a (sort tl) end.
```

To prove the correctness of this function we could try to prove two facts: sorting does not lose any data, and the result list is really sorted. Here is how we can express these two facts.

```
Fixpoint count (x : Z) (l : list Z) :=
  match l with
    nil => 0
  | y::tl =>
    if Zeq_bool x y then 1 + count x tl else count x tl
  end.

Fixpoint sorted (l : list Z) :=
  match l with
    a :: (b :: tl as l') =>
    if Zle_bool a b then sorted l' else false
  | _ => true
  end.

Lemma sort_perm : forall x l, count x l = count x (sort l).
Admitted.

Lemma sort_sorted : forall l, sorted (sort l) = true.
Admitted.
```

You are invited to perform these proofs as an exercise.

This insertion sort algorithm is not too bad on lists that are almost sorted, but it has a bad complexity on average. Let's look at the worst case, which happens when the input is sorted in the reverse order.

For instance, let review the computations performed when sorting the list `4::3::2::1::nil`. The algorithm first sorts `3::2::1::nil`, and for that it sorts `2::1::nil`, which requires one comparison and produces `1::2::nil`, then it inserts `3` in this list, so that `3` is compared with `1` and then with `2`. This makes two comparisons and produces the list `1::2::3::nil`. Then it inserts 4 into this list and this requires 3 comparisons to place the number 4 at the end of the result. Extrapolating this to a list of length `n` we see that the number of comparisons needed would be

$$\sum_{i=1}^{n}(i-1) = \frac{n(n-1)}{2}$$

In practice, on my computer sorting a one-thousand-element-list takes 0.4 seconds, and sorting a two-thousand-element-list takes 2 seconds.

3.2 Merge sort

When sorting a large list of numbers, it is more convenient to decompose the list in two lists of approximately the same length, sort the two sub-lists and then merge these two sorted lists. During the merging phase, we can make sure that the number of required comparisons remains small. Such an algorithm can be programmed in Coq using the following code.

```
Fixpoint merge_aux (a : Z) (l' : list Z)
  (f : list Z -> list Z) (l : list Z) :=
  match l with
  | b :: l' =>
    if Zle_bool a b then f l else b :: merge_aux a f l'
  | nil => l'
  end.

Fixpoint merge (l1 : list Z) : list Z -> list Z:=
  match l1 with
  | a :: l1' =>
```

```
    fix m2 l2 : list Z :=
    match l2 with
    | b :: l2' =>
      if Zle_bool a b then a::merge l1' l2 else b::m2 l2'
    | nil => l1
    end
  | nil => fun l2 => l2
  end.
```

This piece of code uses an advanced trick as it relies on a higher-order function (a function that takes another function as argument). The function `merge_aux` inserts in the output all the elements of the second list that are smaller than `a`. When all these elements have been exhausted, it calls the other function, which is responsible for merging the rest of the first list with any list.

We can then devise an algorithm that takes as input a binary tree and produces a list that contains all the values in this tree. For each binary node in the tree, we combine the lists obtains for the sub-trees using the `merge` function. This guarantees that the final result is sorted.

We first have to define a datatype for binary trees.

```
Inductive bin :=
  L (x : Z) | N (t1 t2 : bin).
```

So binary trees are either elementary (and in this case, they contain an integer value), or they have two parts that are themselves trees. Note that a binary tree always contains at least one integer element (on the other hand, lists were allowed to contain no element).

```
Fixpoint bintolist (t : bin) : list Z :=
  match t with
    L x => x::nil
  | N t1 t2 => merge (bintolist t1) (bintolist t2)
  end.
```

To sort a list, it is thus enough to build a binary tree that contains all the values in this list, and then to map this tree to a sorted list. To take advantage of the efficiency of the divide-and-conquer strategy, we must make sure that the intemediate tree is balanced, in the sense that all branches have approximately the same length. a clever way to achieve this is given by the following two functions.

```
Fixpoint inst (x : Z) (t : bin) :=
  match t with
    L y => N (L x) (L y)
  | N t1 t2 => N (inst x t2)  t1
  end.

Fixpoint insl l t :=
  match l with
    nil => t
  | a::tl => inst a (insl tl t)
  end.

Definition msort l :=
  match l with
    nil => nil
  | a::tl => bintolist (insl tl (L a))
  end.
```

In practice, it takes 0.02 seconds to sort a list of one thousand elements using this function and 0.04 seconds for a ls of two thousand elements.

Proofs of correctness for this code are more complex. For instance, here is a complete proof that the `merge` function produces a sorted list. We prove the property by nesting two proofs by induction, which is not surprising since there are two recursive functions in the algorithm. It is remarkable that we again have to prove a strong statement: we prove not only that the merge function produces a sorted list, but also that the first element of the result is either the first element of the first argument or the first element of the second argument.

I provide this proof to show that it is feasible, but you can assume the results in what follows. This is a proof about lists.

```
Definition head_constraint (l1 l2 l : list Z) :=
  (exists l1', exists l2', exists l', exists a, exists b, exists c,
    l1 = a::l1' /\ l2 = b::l2' /\ l = c::l' /\
    (c = a \/ c = b)) \/
  (l1 = nil /\ exists l2', exists l', exists b, l2 = b::l2' /\ l = b::l') \/
  (l2 = nil /\ exists l1', exists l', exists a, l1 = a::l1' /\ l = a::l') \/
  (l1 = nil /\ l2 = nil /\ l = nil).

Lemma merge_sorted :
  forall l1 l2, sorted l1 = true -> sorted l2 = true ->
    sorted (merge l1 l2) = true /\
    head_constraint l1 l2 (merge l1 l2).
induction l1; intros l2.
 intros sn sl2;split;[exact sl2 | ].
 destruct l2 as [ | b l2'].
  right; right; right; repeat split; reflexivity.
 right; left;split;[reflexivity | exists l2'; exists l2'; exists b].
 split; reflexivity.
induction l2 as [ | b l2 IHl2].
 intros sal1 _; split.
  assumption.
 right; right; left; split;[reflexivity | ].
 exists l1; exists l1; exists a; split; reflexivity.
intros sal1 sbl2.
change (merge (a::l1) (b::l2)) with
  (if Zle_bool a b then a::merge l1 (b::l2) else b::merge (a::l1) l2).
case_eq (a <=? b)%Z.
 intros cab; split.
  destruct l1 as [ | a' l1].
   simpl merge.
   change ((if (a <=? b)%Z then sorted (b::l2) else false) = true).
   rewrite cab; assumption.
  assert (int: sorted (a'::l1) = true).
   simpl in sal1; destruct (a <=? a')%Z;[exact sal1 | discriminate].
  apply (IHl1 (b::l2)) in int.
   destruct int as [int1 int2].
   case_eq (merge (a'::l1) (b::l2)).
    intros q; rewrite q in int2.
    destruct int2 as [t2 | [t2 | [t2 | t2]]].
       destruct t2 as [abs1 [abs2 [abs3 [abs4 [abs5 [abs6 [_ [_ [abs9 _]]]]]]]]].
       discriminate.
      destruct t2 as [abs _]; discriminate.
     destruct t2 as [abs _]; discriminate.
    destruct t2 as [abs _]; discriminate.
   intros r l' qm.
   change ((if (a <=? r)%Z then sorted (r :: l') else false) = true).
   destruct int2 as [t2 | [t2 | [t2 | t2]]];
      try (destruct t2 as [abs _]; discriminate).
   destruct t2 as [l1' [l2' [l'' [u [v [w [q1 [q2 [qm' [H | H]]]]]]]]]].
```

```
    rewrite <- qm, int1.
    rewrite qm in qm'; injection qm';
     injection q1; injection q2; intros; subst.
    simpl in sal1; destruct (a <=? u)%Z; reflexivity || discriminate.
   rewrite qm in qm'; injection qm'; injection q1; injection q2.
   intros; subst; rewrite <- qm, int1, cab; reflexivity.
  assumption.
 left.
 exists l1; exists l2; exists (merge l1 (b :: l2)); exists a; exists b.
 exists a.
 repeat split; left; reflexivity.
intros cab.
assert (cab' : (b <=? a = true)%Z).
 apply Zle_imp_le_bool; rewrite Z.leb_nle in cab; omega.
assert (sl2 : sorted l2 = true).
 destruct l2 as [ | b' l2].
  reflexivity.
 simpl in sbl2; destruct (b <=? b')%Z; assumption || discriminate.
destruct (IHl2 sal1 sl2) as [int1 int2].
destruct l2 as [ | b' l2].
 split;[ | left; exists l1; exists nil; exists (merge (a::l1) nil); exists a;
          exists b; exists b; repeat split; right; reflexivity].
 destruct int2 as [t2 | [t2 | [ t2 | t2]]];
  try (destruct t2 as [abs _]; discriminate).
  destruct t2 as [ab1 [ab2 [ab3 [ab4 [ab5 [ab6 [_ [ab _]]]]]]]]; discriminate.
 destruct t2 as [_ [l1' [l'' [u [q qm]]]]].
 rewrite qm in int1 |- *.
 change ((if (b <=? u)%Z then sorted (u::l'') else false) = true).
 injection q; intros; subst.
 rewrite cab', int1; reflexivity.
destruct int2 as [t2 | [t2 | [t2 | t2]]];
  try (destruct t2 as [abs _]; discriminate).
destruct t2 as [l1' [l2' [l' [u [v [w [q [q' [qm [H | H]]]]]]]]]].
 injection q; injection q'; intros; subst.
 rewrite qm in *.
 split;[ | left; exists l1'; exists (v::l2'); exists (u::l');
              exists u; exists b; exists b; repeat split; right; reflexivity].
 change ((if (b <=? u)%Z then sorted (u::l') else false) = true).
 rewrite cab'; assumption.
injection q; injection q'; intros; subst.
rewrite qm in *.
split;[ | left; exists l1'; exists (v::l2'); exists (v::l');
              exists u; exists b; exists b; repeat split; right; reflexivity].
change ((if (b <=?v)%Z then sorted (v :: l') else false) = true).
simpl in sbl2; destruct (b <=? v)%Z; assumption || discriminate.
Qed.
```

Using the result about merge, we can now prove the correctness of the `bintolist` procedure. This is a proof by induction on binary trees, so there are two cases and the recursive case relies on two induction hypotheses.

```
Lemma bintolist_sorted : forall t, sorted (bintolist t) = true.
induction t.
2 subgoals, subgoal 1 (ID 1478)

  x : Z
  ============================
   sorted (bintolist (L x)) = true
```

```
subgoal 2 (ID 1483) is:
 sorted (bintolist (N t1 t2)) = true
```

The base case is quite easy. When the input tree is a leaf, the output of `bintolist` is a one-element list, and the `sorted` function always returns `true` for this kind of lists. The tactic `reflexivity` solves this cases easily.

```
 reflexivity.
simpl.
  t1 : bin
  t2 : bin
  IHt1 : sorted (bintolist t1) = true
  IHt2 : sorted (bintolist t2) = true
  ============================
   sorted (merge (bintolist t1) (bintolist t2)) = true
```

The statement we want to prove is the first part of the conjunction proved by `merge_sorted`. We use `assert` to specialize the theorem accordingly. Note that we use the two induction hypotheses to satisfy the requirements of `merge_sorted`.

```
assert (tmp := merge_sorted (bintolist t1) (bintolist t2) IHt1 IHt2).
destruct tmp as [tmp' _]; assumption.
Qed.
```

We can now conclude that the function `msort` correctly produces a list that is sorted. Note that the fact `insl` produces a balanced tree does not play a role in the proof. Even if `insl` produced an unbalanced tree, the result would be correct, but in that case the computation would be slower. The kind of bug that leads to a slower result is not captured by our approach.

```
Lemma msort_sorted : forall l, sorted (msort l) = true.
intros [ | a l].
 reflexivity.
unfold msort; apply bintolist_sorted.
Qed.
```

For a more complete correctness proof, we should also show that all the elements of the input list appear in the output list. This is left as an exercise.

4 Exercises

1. Write a function `count_in_tree` that counts the number of occurrences of a given integer in a binary tree.

2. Use the function `count_in_tree` from the previous exercise to state and prove that the `inst` function increases the number of occurrences for its integer argument in the tree.

3. As a follow-up to the previous exercise, prove that `insl` preserves the number of occurrences from a list to a tree,

4. As a follow-up to the previous exercise, prove that `merge` adds the number of occurrences for two lists, and final conclude that `mergesort` preserves the number of occurrences of the list it sorts.

5. The sequence of *Fibonnacci numbers* is defined recursively by:

$$f_0 = 0 \qquad f_1 = 1 \qquad f_{n+2} = f_{n+1} + f_n$$

This definition can be implemented directly in a Coq function. What is the highest element that you can compute in less than 10 seconds? (hint: define a function that is recursive on natural numbers, but the fibancci numbers themselves should be in type `Z`). An alternative approach to computing Fibonacci numbers relies on matrices.

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \times \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} f_n \\ f_{n+1} \end{bmatrix}$$

Moreover the power of a matrix can be computed efficiently by using a divide-and-conquer approach that is associated by the positive type of natural numbers (note that in each equation, A^p should only be computed once):

$$A^{2p} = A^p \times A^p \qquad A^{2p+1} = A^p \times A^p \times A$$

Use this as the main structure for an algorithm, and then prove it correct. Test that you can compute much larger Fibonacci numbers using this approach.

6. Expressions in a program using variables, integer constants, additions, multiplications, and subtractions can be represented by the following datatype:

```
Require Import String.

Inductive exp :=
  Var (name : string)
| Const (x : Z)
| Adde (e1 e2 : exp)
| Mule (e1 e2 : exp)
| sube (e1 e2 : exp).
```

   To evaluate such an expression, one needs to know what is the environment, in other terms, the value associated to each variable. We will represent such an environment by a list of pairs associating string (names) to integers (values). The first question of this series is to write a program that evaluates expressions in a given environment (one shall assume that all variable names occuring in the expression occur in the environment; it will be fair to return an arbitrary value when a variable does not occur in the expression).

7. Instead of using an environment with precise values, we want to compute with values known only as intervals: write a program that computes the value of an expression when the variables are only known by intervals (the result of the evaluation should be an interval). Write a statement that proves the consistence of this program with the program from the previous question, and prove it correct.