Comments on the sort program

Yves Bertot and Pierre Castéran

July 15, 2003

We give some comments on a little developement in *Coq* : a certified sort program on lists of integers.

1 Importing modules

Our program will work on lists of integers. The modules `PolyList` and `ZArith` of *Coq*'s standard library contain the definition and some functions and already proved theorems on (respectively) polymorphic lists and integers.

```
Require PolyList.
Require ZArith.
```

The type of lists of integers is now `(list Z)`.

2 List equivalence

A sorting function working on lists must return a list which has the same elements (with the same order of multiplicity) as its argument.

2.1 Counting occurrences in a list

In order to specify this constraint, we first define a function which counts the number of occurences of some integer z in a list l. This function is defined recursively and uses a comparison function for integers named `Z_eq_dec` (which we found in module `ZArith`).

```
Fixpoint nb_occurences[z:Z; l:(list Z)]:nat :=
  Cases l of
    nil => 0
  | (cons z' l') =>
     Cases (Z_eq_dec z z') of
       (left _) => (S (nb_occurences z l'))
     | (right _) => (nb_occurences z l')
     end
  end.
```

We can test this function, for instance by computing the number of occurences of 3 in the list $(3, 7, 3)$. Notice we use a *Lisp*-like notation of lists.

```
Eval Compute in
  (nb_occurences `3` (cons `3` (cons `7` (cons `3` (nil ?))))).
```

```
= (2)
  : nat
```

2.2 An equivalence relation on lists

We can now define a binary relationship on `(list Z)`. Lists l and l' are related if they have the same number of occurrenes of z, for any z.

```
Definition equiv[l,l':(list Z)]:=
     (z:Z)(nb_occurences z l)=(nb_occurences z l').
```

2.3 Some proofs

The correctness proof of our sorting functions relies on some properties of the `equiv` relationship. These properties are interactively proved within the system *Coq*.

Each lemma or theorem is announced with its name and statement. One can find between the keywords `Proof` and `Qed` a sequence of commands for guiding the proof with the help of *tactics*. Instead of reading passively such *proof scripts*, you should *replay* them on your computer, and watch the answers of the system.

2.3.1 The empty list is equivalent to itself

The following proof is very simple. In few words, one computes the number of occurrences of any number z in the empty list, and the statement we want to proves reduces to the trivial equality $0 = 0$.

```
Lemma equiv_nil : (equiv (nil ?) (nil ?)).
Proof.
  Intro z;Simpl;Auto.
Qed.
```

2.3.2 The `equiv` relation is a congruence *wrt* `cons`

We have to prove that, if l and l' are equivalent, and if we insert the same z in front of l and l', the resulting lists are equivalent too.

The proof is quite simple. We count the number of occurrences of some z' in both resulting lists. We have to consider the cases where $z = z'$ and $z \neq z'$.

The statement of theorem `equiv_cons` must be read as follows :

> For each z of type `Z`, for each l and l' of type `(list Z)`, if l is equivalent to l', then `(cons` z l`)` is equivalent to `(cons` z l'`)`.

```
Lemma equiv_cons : (z:Z;l,l':(list Z)) (equiv l l')->
                                       (equiv (cons z l) (cons z l')).
Proof.
 Intros z l l' H z';Simpl;Auto.
 Case (Z_eq_dec z' z);Simpl;Auto.
Qed.
```

2.3.3 More properties of `equiv`

The relationship *equiv* is reflexive and transitive.

Proofs of both statement are quite easy. Notice the proof of reflexivity, which is done by induction on lists, and uses the lemma *equiv_ cons*.

```
Lemma equiv_refl : (l:(list Z))(equiv l l).
Proof.
 Induction l;Intro z;Simpl;Auto.
 Intros;Apply equiv_cons;Auto.
Qed.

Lemma equiv_trans : (l,l',l'':(list Z))(equiv l l')->
                                       (equiv l' l'')->
                                       (equiv l l'').
Proof.
 Intros l l' l'' H H0 z.
 Transitivity  (nb_occurrences z l').
 Apply H;Auto.
 Apply H0;Auto.
Qed.
```

Finally, we prove that our equivalence relation is compatible with the permutation of the two first items of a list.

```
Lemma equiv_perm : (a,b:Z; l,l':(list Z))
                   (equiv l l')->
                   (equiv (cons a (cons b l))
                                  (cons b (cons a l'))).
Proof.
 Intros a b l l' H z;Simpl.
 Case (Z_eq_dec z a);Case (Z_eq_dec z b);Simpl;Case (H z);Auto.
Qed.
```

2.3.4 Helping *Coq* to help yourself

The following command adds the four lemmas `equiv_nil`, `equiv_cons`, `equiv_refl` and `equiv_perm` to a base of theorems which can be used by the `Auto` tactics.

For instance, the foolowing goal (*i.e.* prove that the lists (a, b, c, d) and (b, a, d, c) are always equivalents) is solved in one automatic interaction.

```
Lemma abcd : (a,b,c,d:Z)
  (equiv (cons a (cons b (cons c (cons d (nil ?)))))
         (cons b (cons a (cons d (cons c (nil ?)))))).
Proof.
 Auto.
Qed.
```

2.4 ordered lists

The following piece of code defines the predicate " to be sorted " on lists of integers. There are wo base cases, corresponding to lists with 0 or 1 item. The recursive case corresponds to lists of the form $z_1 z_2.l$. Such a list is sorted if $z_1 \leq z_2$ and the list $z_2.l$ is sorted. Definitions like the following one are called *inductive definitions* and are close – but much more powerful – to *Prolog*'s clauses.

```
Inductive sorted : (list Z)-> Prop :=
 sorted0 : (sorted (nil Z))
|sorted1 : (z:Z)(sorted (cons z (nil ?)))
|sorted2 : (z1,z2:Z;l:(list Z))'z1<=z2' -> (sorted (cons z2 l))->
                                           (sorted (cons z1 (cons z2 l))).
Hints Resolve sorted0 sorted1 sorted2.
```

The following interaction shows how to get automatically a proof that some list is already sorted. Note that the command "`Auto with zarith`" loads the hints from the base of theorems about arithmetics on integers.

```
Lemma sorted_1135 : (sorted
    (cons '1' (cons '1' (cons '3' (cons '5'  (nil ?)))))).
Proof.
 Auto  with zarith.
Qed.
```

3 Inserting in a sorted list

The kernel of an insertion sort is a function which inserts some item at the right place in an already sorted list.

3.1 An insertion function

Let us define a function for inserting some integer z in a list l. This function is defined by a simple recursion on l.

- If l is empty, we return the list compsed of a single occurrence of z,
- If l is of the form $a.l'$,

- if $z \leq a$, then we return $za.l$,
- if $a < z$, we return a(`insert z l'`).

```
Fixpoint insert[z:Z;l:(list Z)]:(list Z) :=
  Cases l of
          nil => (cons z (nil ?))
         |(cons a l') => Cases (Z_le_gt_dec z a) of
                           (left _)=>(cons z (cons a l'))
                          |(right _) =>(cons a (insert z l'))
                         end
  end.

Eval Compute in (insert `4` (cons `2` (cons `5` (nil ?)))).
    = (cons `2` (cons `4` (cons `5` (nil Z))))
    : (list Z)
```

3.2 Some properties of `insert`

The function `insert` has some properties, which will be used in the validation of our sorting program.

3.2.1

We first prove that (`insert` x l) has exactly the same elements as $x.l$.

```
Lemma insert_equiv : (l:(list Z);x:Z)(equiv (cons x l) (insert x l)).
Proof.
 Induction l ; Simpl ; Auto.
 Intros a l0 Hrec x.
 Case (Z_le_gt_dec x a);Simpl;Auto.
 Intro ;  Apply equiv_trans with (cons a (cons x l0)); Auto.
Qed.
```

3.2.2

The following lemma is a justification for using `insert` in a sort : if l is sorted, then (`insert` x l) returns a sorted list.

```
Lemma insert_sorted : (l:(list Z);x:Z)(sorted l)->(sorted (insert x l)).
Proof.
 Induction 1 ;  Simpl; Auto.
 Intro z; Case (Z_le_gt_dec x z);Simpl ; Auto with zarith.
 Intros z1 z2 ;Case (Z_le_gt_dec x z2);Simpl;Intros.
 Case (Z_le_gt_dec x z1);Simpl;Auto with zarith.
 Case (Z_le_gt_dec x z1);Simpl;Auto with zarith.
Qed.
```

4 A certified sorting program

4.1 From specification to realization

We are now able to define a sorting function. Instead of defining it as a plain recursive function like `insert`, we choose to ensure that the function we define has all the required properties of a sort.

The following definition of the constant `sort` specifies its type, not as the type of any function from `(list Z)` to itself, but as the type of any function taking as argument a list l, and returning a list l' *which has the same elements as* l *and is sorted too.*

Because of the constraints in its specification, it is easier to define `sort` interactively, in the same style as the preceding proofs.

```
Definition sort : (l:(list Z)){l':(list Z)|(equiv l l')/\(sorted l')}.
 Intro l;Induction l.
 Exists (nil Z);Split;Auto.
 Case Hrecl;Intros l' [H0 H1].
 Exists (insert a l').
 Split.
 Apply equiv_trans with (cons a l') ; Auto.
 Apply insert_equiv.
 Apply insert_sorted ; Auto.
Defined.
```

4.2 Extraction to *ML* code

The command `Extraction "insert" insert sort` writes an *OCAML* module containing code extracted from the development above. This code can be compiled, then executed or called from another *OCAML* program.

The theoretical principles on which *Coq* has been built ensure us that this program is conform with its specification.

5 Conclusion

This little example shows some of the main features of *Coq* : possibility of defining datatypes and programs, proving properties of programs with a reasonable degree of automation, and deriving correct programs from their specification.

The reader will learn to develop and certify more complex programs, including imperative programs, polymorphic functions, and even entire modules. *Coq*'s field of application includes mathematics, logic, and many other domains (look at the users contributons).