A small example : proof of a functional program

In Gallina, the language used to describe terms, types, proofs and programs, we can express program specifications.

> *"The function f is a correct function for sortings lists of elements of type A with respect to a given binary relation R"*

(The predicates `Permutation` and `Sorted` are defined in Coq's standard library.)

```
Definition Sort_spec (f : list A -> list A) :=
    forall l, let l' := f l in
        Permutation l' l /\ Sorted R l'.
```

Our simple sorting function is described under the form of two recursive functions on lists.

Insertion of an element a in an already sorted list l

```
Function insert (a:A) (l: list A) : list A:=
    match l with
        [] => [a]
      | b::l' => if R a b then a::l else b::insert a l'
    end.
```

Main sorting function, recursively calling `insert`*:*

```
Function sort (l: list A) : list A :=
    match l with
        nil => nil
      | a::l' => insert a (sort l')
    end.
```

A correctness proof of `sort` is a sequence of interactively proved lemmas leading to a final correctness statement. Let us look at some extract of this proof.

By induction on the list `l`*, we prove that the elements of the list* `insert x l` *contains exactly the same elements as* `x :: l` *(with the same multiplicity).*

```
Lemma  insert_perm : forall x l, Permutation (x :: l)
                                              (insert x l).
Proof.
  induction l.
```

The first case (the empty list) is trivially solved.

```
A : Type
R : A -> A -> bool
==============================
Permutation [x] (insert x [])
```

```
trivial.
```

For the second case, Coq provides us with an *induction hypothesis* `IHl` on a given list `l`. The new goal consists in proving the property for the bigger list `a::l`.

```
a : A
l : list A
IHl : Permutation (x :: l) (insert x l)
==============================
Permutation (x :: a :: l) (insert x (a :: l))
```

We solve this goal through a sequence of *tactics*.

```
simpl.
case (R x a); trivial.
+ transitivity (a:: x :: l); auto.
```

```
l : list A
 IHl : Permutation (x :: l) (insert x l)
 ==============================
 Permutation (x :: a :: l) (a :: x :: l)
```

```
l : list A
IHl : Permutation (x :: l) (insert x l)
==============================
Permutation (x :: a :: l) (a :: x :: l)
```

We can send queries to the libraries of already proven lemmas:

```
Search (Permutation (?x :: ?y ::?l) (?y :: ?x :: ?l)).
```

```
perm_swap:
  forall (A : Type) (x y : A) (l : list A),
  Permutation (y :: x :: l) (x :: y :: l)
```

```
  apply perm_swap.
Qed.
```

Finally, we prove that our function `sort` is correct.

```
Theorem sort_correct : Sort_spec sort.
  Proof.
    split.
    - apply sort_perm.
    - apply sort_sorted.
  Qed.
```

We can also *extract* our function towards a programming language like *Ocaml*, *Haskell* or *Scheme*.

```
Extraction Language Ocaml.


Recursive Extraction sort .
```