Proofs about programs

Assia Mahboubi (inspired by Yves Bertot's previous slides)

Proofs of programs in Coq

Monday: Benjamin explained how to write programs in Coq (booleans, numbers, lists).

Monday and Tuesday: Pierre and Yves explained how to write statements and how to write a formal proof.

Now : We will see how to write specifications of programs written in Coq and how to prove them.

Do not be too decieved...

We know your favorite programming language is not (yet?) Coq.
But:

- Coq programs can be seen as models of realistic programs and its formal proof as a dynamic documentation.
- Coq programs can be translated (extracted) to realistic programming languages.
- Coq programs can be executed efficiently, and this is a proof automation technique.

And there exists tools too to prove even imperative programs but it is much more intricate a problem.

Proofs about computation

- Reason about functional correctness
- State properties about computation results
  - Show consistency between several computations
- Use the same tactics as for usual logical connectives
- Add tactics to control computations and observation of data
- Follow the structure of functions
  - Proving is akin to symbolic debugging
  - A proof is a guarantee that all cases have been covered
  - It is much stronger than a test!

First examples

Controlling execution

- Replace formulas containing functions with other formulas
- Manually with direct Coq control:
  - `change` f_1 `with` f_2
  - Really checks that f_1 and f_2 are the same modulo computation
- Manually with indirect control
  - `replace` f_1 `with` f_2
  - Produces a side goal with the equality $f_1 = f_2$
- Simply expand definitions
  - `unfold f`, `unfold f at 2`

Functions and specifications

- Each function should comes with theorems about it.
- In this course, sometimes called companion theorems.
- They should be usable directly through apply when the goal's conclusion fits.
- Otherwise, they can be brought in the context using `assert` `assert (H := th a b c H').`

Reasoning about pattern-matching constructs

- Pattern-matching typically describes alternative behaviors.
- Reasoning on these functions goes by covering all cases.
- Companion theorems describe the non trivial identities.

Examples with natural numbers

Tools

- `case` is the basic constructs
  - generates one goal per constructor
  - the expression is replaced by constructor-values, in the conclusion
  - the argument to `S` becomes a universally quantified variable
- `destruct` is more advanced and covers the context
  - like `case`, but nesting is authorized
  - the context is also modified
- `case_eq` remembers in which case we are
  - the context is not modified (as in `case`)
  - remembering can be crucial

How to find Companion theorems

- `SearchAbout` is your friend.
- In general `Search` commands are your friends.
  - `Search`: use a predicate name
    `Search le.`
  - `SearchRewrite`: use patterns of expressions
    `searchRewrite (_ + 0).`
  - `SearchPattern`: use a pattern of a theorem's conclusion (type `Prop`, usually)
    `SearchPattern (_ * _ <= _ * _).`

Recursive functions and induction

- When a function is recursive, calls are usually made on direct subterms.
- Companion theorems do not already exist, but have to be stated and proved by the user.
- Induction hypotheses make up for the missing theorems.
- The structure of the proof is imposed by the data-type.

Examples on recursive functions

A trick to control recursion

- Add one-step unfolding theorems to recursive functions
- Associate any definition
  `Fixpoint f x1 ...xn := body`
  with a (trivial) theorem `f_step`
  `forall x1 ...xn, f x1 ...xn = body`
- Use `rewrite f_step` instead of `change`, `replace`, or `simpl`
- This provides a better control than `simpl`.
- More concise than `replace` or `change`.
- Note that `unfold` is not well-suited for recursive functions.

Proofs on functions on lists

- Tactics `case`, `destruct`, `case_eq` also work
- Induction on lists works like induction on natural numbers
- `nil` plays the same role as `0`: base case of proofs by induction
- `a::tl` plays the same role as `S`
  - Induction hypothesis on `tl`
  - Fits with recursive calls on `tl`

Example proof on lists

```
Require Import List.

Print rev.
fun A : Type => fix rev (l : list A) : list A :=
  match l with
  | nil => nil
  | x :: l' => rev l' ++ x :: nil
  end : forall A : Type, list A -> list A

Fixpoint rev_app (A : Type)(l1 l2 : list A) : list A :=
  match l1 with
    nil => l2
  | a::tl => rev_app A tl (a::l2)
  end.

Implicit Arguments rev_app.
```

Example proof on lists (continued)

```
Lemma rev_appP : forall A (l1 : list A),
   rev_app l1 nil = rev l1.
intros A l1.
  A : Type
  l1 : list A
  ============================
  rev_app l1 nil := rev l1
assert (tmp: forall l2, rev_app l1 l2 = rev l1 ++ l2);
  [ | rewrite tmp, <- app_nil_end; reflexivity].
```

Example proof on lists (continued)

```
forall l2 : list A, rev_app l1 l2 = rev l1 ++ l2
induction l1; intros l2.
2 subgoals

  A : Type
  l2 : list A
  ============================
   rev_app nil l2 = rev nil ++ l2

subgoal 2 is:
 rev_app (a :: l1) l2 = rev (a :: l1) ++ l2
simpl; reflexivity.
```

Example proof on lists (continued)

```
  IHl1 : forall l2 : list A, rev_app l1 l2 = rev l1 ++ l2
  l2 : list A
  ============================
   rev_app (a :: l1) l2 = rev (a :: l1) ++ l2
simpl.
   rev_app l1 (a :: l2) = (rev l1 ++ a :: nil) ++ l2
SearchRewrite ((_ ++ _) ++ _).
app_ass:
    forall A (l m n:list A), (l ++ m) ++ n = l ++ m ++ n
rewrite app_ass; apply IHl1.
Proof completed.
Qed.
```