Logic and specification in Coq

Yves Bertot

September 2019

Expressing properties of programs

- Expressing properties of the output
- Expressing a relation between the input and the output
- Expressing properties that are *always* satisfied
- Properties are similar to tests

Starting from tests

- Tests rely on two components
  - A piece of code to generate test cases
  - A piece of code to verify correct behavior on all cases
- In our approach, we use logic
  - to describe what are all possible inputs
  - to express what is the correct behavior

Example on the filter function

- What do you expect from the filter function?

```
Fixpoint my_filter {T : Type} (p : T -> bool)
    (l : list T) : list T :=
match l with
  nil => nil
| a :: l1 =>
  if p a then a :: my_filter p l1 else my_filter p l1
end.
```

Filter function specification

- The output of the filter function must only contain values satisfying the boolean property
- All values satisfying the boolean property should be taken

Filter function specification

- The output of the filter function must only contain values satisfying the boolean property
- All values satisfying the boolean property should be taken
- The multiplicity of values is preserved
- The order of values is preserved

Expressing a logical property

- Difficulty of Coq: logical values are not boolean
- In a sense, `bool` is restricted to logical statements that can be decided
  - For instance, if f and g are functions on `nat`, the fact that coincide everwhere cannot be decided
- A new type `Prop` is used for logical propositions
- For `T` a type and `a b : T`, `a = b : Prop`
- `Prop` also has connectives `and` (`/\`), `or` (`\/`), `not` (`~`), implication is written `->`
- Universal quantification is written `forall x : T, P x`
- Existential quantification `exists x : T, P x`
- A boolean value v can be mapped to a proposition by writing v `= true`

Propositions for lists and numbers

- `In x l` is defined by a recursive computation that yields a proposition based on = and \/
- Numbers have <=, <

```
Compute In (2 + 3) (3 :: 5 :: 8 :: nil).
= 3 = 5 \/ 5 = 5 \/ 8 = 5 \/ False : Prop

Compute 2 <= 3.
= 2 <= 3 : Prop

Compute 2 <=? 3.
= true : bool
```

Specifications for `filter`

```
forall (A : Type) (f : A -> bool) (x : A) (l : list A),
  In x (myfilter f l) -> In x l /\ f x = true

forall (A : Type) (f : A -> bool) (x : A) (l : list A),
  In x l /\ f x = true -> In x (myfilter f l)
```

specifications based on tests

- Write your function: `f : A -> B`
- Write extra functions to verify that the output is correct, `verif : B -> bool`
- Express a universal statement `forall x :A, verif (f x) = true`
- Being able to prove such a statement is equivalent to exhaustive testing.

Example: stating that a list is sorted

```
Fixpoint is_sorted {T : Type} (R : T -> T -> bool)
  (l : list T) : bool :=
match l with
  a :: (b :: _ as tl) =>
  if R a b then is_sorted tl else false
| _ => true
end.
```

A *partial* specification for a sort function is

```
forall l, is_sorted (sort R l) = true
```

Non computable properties

- Type theory is strong enough to describe non-decidable properties
- Example : Collatz (aka. Syracuse) sequences.

```
Inductive Collatz : Z -> Prop :=
| c1 : Collatz 1
| c2 : forall n, n mod 2 = 0 -> Collatz (n / 2) ->
       Collatz n
| c3 : forall n, n mod 2 = 1 -> Collatz (3 * n + 1) ->
       Collatz n.
```

Inductive properties

- Describe sets that are stable modulo some operations
- Take the least set satisfying these operations.
- Suitable for a many applications
  - semantics of programming language
  - Describing grammars

Example: a grammar as an inductive property

```
Require Import String.

Open Scope string_scope.

Inductive wfpar : string -> Prop :=
  wf0 : wfpar EmptyString
| wfcat : forall s1 s2, wfpar s1 -> wfpar s2 ->
          wfpar (s1 ++ s2)
| wfadd : forall s1, wfpar s1 ->
          wfpar ("(" ++ s1 ++ ")").
```

- Be careful about the sense in which one reads arrows
- When describing a process (e.g. parsing), work follows arrows in reverse

Example: transitive closure as an inductive property

```
Inductive t_closure {T : Type} (R : T -> T -> Prop) :
  T -> T -> Prop :=
  tc1 : forall x y, R x y -> t_closure R x y
| tc_s : forall x y z, R x y -> t_closure R y z ->
  t_closure R x z.
```

Example: transitive closure, take 2

```
Inductive t_closure2 {T : Type} (R : T -> T -> Prop) :
  T -> T -> Prop :=
  tc2_1 : forall x y, R x y -> t_closure2 R x y
| tc2_s : forall x y z,
  t_closure2 R x y -> t_closure2 R y z ->
  t_closure2 R x z.
```

Induction principle for an inductive property

Every relation `P` that satisfies the same stability property as `t_closure` is a consequence.

```
About t_closure_ind.
t_closure_ind :
forall (T : Type) (R P : T -> T -> Prop),
(forall x y : T, R x y -> P x y) ->
(forall x y z : T, R x y ->
            t_closure R y z -> P y z -> P x z) ->
forall y y0 : T, t_closure R y y0 -> P y y0
```

The power of inductive properties

- Logical connectives
- In Coq, logical connectives, equality, comparison between natural numbers are inductive properties
- Beware of excessive use
- Mathematical Components advocates a different style
  - Express as much as possible using boolean functions
  - Provide bridges from `bool` to `Prop`