Proving properties of programs

Yves Bertot

October 2018

Objectives

- Usual approach to removing bugs in programs: testing
- Write testing context, construct sample inputs, run
- This course: perform test with symbolic values
- Use quantification to introduce symbolic values

Examples of programs

```
Require Import Arith List.

Fixpoint evenb (n : nat) : bool :=
  match n with
  | 0 => true  | S p => negb (evenb p)
  end.
Fixpoint max_list (l : list nat) : nat :=
  match l with
  | nil => 0
  | a::tl => max a (max_list tl)
  end.
Definition swap_first_two (l:list nat) : list nat :=
  match l with
  | a::b::tl => b::a::tl
  | _ => l
  end.
```

Reasoning on case expressions

- When a `match` appears in the goal
- Use `case`, `case_eq`, `destruct` to look separately at the various cases of execution
- In effect, we cover all alternatives of execution behavior
- Demo time!

Impossible cases

- Impossibility can be expressed in several ways:
  1. Premise or hypothesis `true = false`, `0 = 1`, or `nil = a::tl`
  2. Premise or hypothesis `e <> e` or `e1 <> e2` when `e1` actually equals `e2`
  3. Premise or hypothesis `False`
  4. A premise `~A` when `A` can actually be proved
- Impossibility 1: `discriminate`
- Impossibility 2,3,4: `case H`

Reasoning by induction : lists

- View proofs on lists like recursive programs
- The result of a recursive calls proves the statement for the sublist
- The case of the empty list must also be covered
- Like a proof covering all cases, but with the extra recursive statement
- Demo on a proof concerning the filter function

Reasoning by induction : natural numbers

- Mathematicians prove properties of natural numbers by induction
- For any predicate `P` on natural numbers
  - If `P 0` holds
  - If one can deduce `P (1 + n)` from `P n` for any `n`
- Then the properties holds for every natural number
- Only two cases, but infinity of results!
- Like proof by cases, but with an induction hypothesis
- Also the same pattern as proofs on lists

Using induction to prove properties on `evenb`

Demo time!

Non confusion of data-type constructors

- Constructors of data-types are manipulated as functions
- These functions have specific properties
  - Different constructors always yield different values
  - Each constructor is injective
- These properties are consequences of `match ... with ... end` behavior
- In proofs two tactics are provided to use these characteristics
  - `discriminate` to prove `O <> S p` and goals of the same shape
  - `injection` to prove `S p = S q -> p = q`

Guiding computation

- Sometimes we want to replace sub-expressions with others that are equal
- If the system should be able to recognize it, use `change`
- If the system can't recognize it, but you are sure you can prove it, use `replace`
- If you don't want to write the result, use `unfold` or `simpl`