Programs in arithmetics

Yves Bertot

October 2018

Objectives

- Numbers are important in software
- Several possible data-structures, different advantages
  - `nat` : proofs are easy
  - `N` : same collection of numbers, but proofs harder
  - `Z` : contains also negative numbers, proofs also hard
  - `Q` : fractions
  - `R` : real numbers, computation incomplete

Definition of positive integers

- three exclusive cases for positive integer x
  1. $x = 1$
  2. $x = 2 \times x'$ where x' is also a positive integer
  3. $x = 2 \times x' + 1$ where x' is also a positive integer
- Intuition : like a sequence of bits, the last one always a 1
- Binary representation

Coq implementation of positive

```
Require Import ZArith.

Print positive.

Inductive positive : Set :=
  xI : positive -> positive
| xO : positive -> positive
| xH : positive

Compute (Pos.to_nat (xO (xI xH))).
     = 6 : nat

Check (xO (xI xH)).
 6%positive : positive
```

Definition of integers

- three possible cases for x integer, no recursion
  1. $x = +x'$ where x' a positive number
  2. $x = 0$
  3. $x = -x'$ where x' a positive number
- Integers not practical for recursive definitions
- Rely on *fuel* technique or *well-founded recursion*

Library of Z functions

- use `Open Scope Z_scope.` to have common notations on addition, multiplication, etc.
- Use `Z.ltb`, `Z.leb`, `Z.eq` as comparison functions
- Also `Z.div`, `Z.mod`, `Z.sqrt`, quite efficient

Automatic reasoning about integers

- Two big tools for reasoning
  1. Polynomial equalities
     - formula obtained solely by using `+`, `*`, `-` and constants
     - Use the tactic `ring`
  2. Linear arithmetic
     - Comparisons f_1 `<` f_2, f_1 `<=` f_2, or f_1 `=` f_2
     - formula obtained solely by using `+`, `-` and multiplications by constants
     - Use the tactic `lia` (for linear integer arithmetic)
       Must have `Require Import Psatz.`
     - Also can use the tactic `omega`
       Must have `Require Import Omega.`

Illustration

Demo time!

Proofs by induction

- Programming on positive numbers allows recursion on half number
- Three cases because three constructors in type
- Proofs by induction are similar
  - Three cases
  - Always only one recursive hypothesis (not for all smaller numbers!)

The standard library of Real numbers

- Real numbers cannot be described as an inductive type
- Real numbers form a complete archimedean field
  - this expressed as a collection of axioms and parameters
- Reasoning in Coq establishes consequences of the axioms
- Computability is not complete (no inductive type)
  - *what does it mean to compute* $\sqrt{2}$*?*
- Approximate computation is still possible, using integers, rational numbers, floating point numbers
- The tools are `coq-interval`, `flocq`, `lra`

Example real number computation

```
Require Import Reals Interval.Interval_tactic.

Open Scope R_scope.

Lemma try_sqrt2 : 14142/10000 <= sqrt 2 <= 14143 / 10000.
Proof.
interval.
No more subgoals
Qed.
```

real number computation continued

```
Lemma try_sqrt3_5 : 2 * sqrt 2 < sqrt 3 + sqrt 5.
Proof.
interval_intro (sqrt 3 + sqrt 5) as s3s5.

s3s5 : 1065183775 / 268435456 <= sqrt 3 + sqrt 5
         <= 1065183777 / 268435456
interval.
No more subgoals
Qed.
```

Possible applications

People compute trillions of decimals of π

- How do they know the computation is right?
- They use a second different algorithm

We can prove high-precision algorithms (BRT18)

- this requires proofs about integrals, arithmetic-geometric means, ...

Hard mathematical proofs are also necessary to make sure arithmetic hardware is correct

- This proofs can also be verified