Programming in Coq

Yves Bertot

October 2018

Why use Formal verification tools

- Write programs with less bugs
- Document the programs with logical statements
- Verify the logical statements with the computer
- Model and verify existing programs or systems

Example of program with less bugs

- Compcert C compiler
- Use a formal description of the C programming language and assembly languages
- Construct a formal description of a compiler from C to assembly
- **Prove** that the compiler respects the semantics of programs
- Independently tested by the C-smith tool
- More information on `http://compcert.inria.fr/motivations.html`

Compiler correctness statement

Taken from CompCert 3.4, file `driver/Compiler.v`

```
Theorem transf_c_program_correct:
  forall p tp,
  transf_c_program p = OK tp ->
  backward_simulation (Csem.semantics p) (Asm.semantics tp).
Proof.
  intros.
  apply c_semantic_preservation.
  apply transf_c_program_match; auto.
Qed.
```

Ambitious projects

- End-to-end verification of large systems
- Formal verification brings composability
- Complex inner parts can be factored out
- You only need to understand the definition and the top statement
- The odd-order theorem in group theory (Feit-Thompson) is an example
  - The definitions and the statements fit in two pages

Let's start easy

Two ways to develop software in Coq

- Describe algorithms inside Coq, Execute outside
  - Stronger programming tools
  - Lighter runtime environment
- Do everything inside Coq
  - Simpler programming language
  - Use Coq as an interpreter
  - Instant feedback
- We will mostly show the latter

Basic data structures

- numbers 1, 42, 1024
- boolean values `true`, `false`
- pairs `(1, true)`
- lists of things `1 :: 2 :: 3 :: 4 :: nil`
- functions `fun x => x`

More about functions

- binary operations on numbers `+`, `*`, `/` `mod`, `-`
- boolean relations on numbers `<?`, `=?`, `<=?`
- boolean operations on boolean values `&&`, `||`
- boolean negation `negb`
- test on boolean value `if then else`
- projections on pairs `fst`, `snd`
- more complex programming structure for lists (to be given later)

Defining and using your own functions

- Give a name to a value : `Definition name := value.`
  - Give a name to a function :
    `Definition fname := fun x => x.`
  - Alternative : `Definition fname x := x.`
- Use a function: write the name before the argument
  write f (fname 1)
  - parentheses not always needed
- Check your own steps using the `Check` command.
- Compute your examples using the `Compute` command.
- Know what is defined using the `Print`.

Examples

```
Require Import Arith Bool List.

Definition add2 x := x + 2.

Check add2 3.
add2 3 : nat

Compute add2 3.
= 5 : nat

Definition twice (f : nat -> nat) (x : nat) := f (f x).

Compute twice (twice add2) 1.
= 9 : nat
```

Comments on the examples

- `twice` is a function with two arguments
  - the syntax is really different from `C`, `java`, `etc.`
- parentheses are needed around `f x` in the definition of twice
- **No parentheses** around the two arguments in the use of `twice`
- `twice` can also be used with only one argument

Functions about data-structures

- components of a pair : `fst`, `snd`
- Fetching elements of a list

```
match l with
| a :: l1 => f a l1
| nil => v
end
```

Programming with lists

```
Definition headplus1 (l : list nat) :=
  match l with
    a :: l1 => a + 1
  | nil => 0
  end.
```

Recursive programming with lists

- Lists can be arbitrary long
- A list has a sub-component that is itself a list
- A recursive program can call itself on that sub-component

```
Fixpoint grow_nat (l : list nat) :=
match l with
  nil => nil
| a :: l1 => 2 * a :: 2 * a + 1 :: grow_nat l1
end.

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.
```

Comments on list programming

- Lists and pairs are *polymorphic* data structures
- You don't need to know the type of elements for many operations
- You can choose for the type argument to be implicit.
- No undefined behavior: all functions must cover the case where the argument is empty

Making your own data type

- Lists are an example of data type with two cases, and one of the cases has two sub-components
- You can make your own choice.
- Example drawn from an example available in `coq-contribs`, `semantics`

```
Inductive aexpr0 : Type :=
  avar (s : string)
| anum (n :Z)
| aplus (a1 a2:aexpr0).

Inductive bexpr0 : Type :=  blt (_ _ : aexpr0).

Inductive instr0 : Type :=
  assign (_ : string) (_ : aexpr0)
| sequence (_ _: instr0)
| while (_ :bexpr0)(_ : instr0)
| skip.
```