Coq: a technical introduction

Yves Bertot

May 2015

Basic idea : Proofs as programs

1. A proof of $A \Rightarrow B$ is a processor that produces proofs of B
   - if given proofs of A as input
2. A proof of $\forall x : nat, B(x)$ is a processor that produces proofs of $B(1)$, $B(2)$

- It is how you construct these proofs
- It is how you use them

Propositions as types

- Overload notation A `->` B
  - type of functions from type A to B
  - proposition A *implies* B
  - Variant 1 is like programs in C
  - Variant 2 is exotic for programmers
- Build up from implication and universal quantification: other connectives (first solution, but not the one in Coq)

```
Definition and1 (A B : Prop) :=
  forall C : Prop, (A -> B -> C) -> C.
```

Programs: functions

- In theory, you can define a programming language with only one-argument functions
- Beware of notations
  - `fun x : T =>` *e*
    functions don't necessarily have a name!
    But Coq makes it possible to name terms
    Sometimes the type `T` is omitted (syntax : `fun x =>` *e*)
  - *f e*
    when applying functions: no parentheses
- Example:
```
(fun f => fun x => f (f x))
     (fun x => fun y => y)
```
- Theoretical background: in Church's papers (ca. 1940) keyword `fun` was λ, symbol `=>` was a period.

Deeper foundation : pure λ-calculus

- Not enough time for this but, if interested
  `http://www-sop.inria.fr/members/Yves.Bertot/misc/lambda.ml`
  Shows a 250 lines program (in ocaml) which implements lambda-calculus and performs a few computations

Programs as proofs

- Challenge: make basic proofs about the “and” connective as given in previous slide

Inventing new types

Aggregating data and providing alternatives

- Illustration using pre-existing types `A`, `B`, `C`
- `Inductive test1 := ct1 (x : A) (y : B)`
  `| ct2 (z : C).`
- If e_a `:` `A` and e_b `:` `B`, then
  `ct1` e_a e_b has type `test1`
- if e `:` `test1`, then you should consider it can have the form
  `ct1` e_1 e_2 or the form `ct2` e_3
- Programming construct:

```
match e with
  ct1 a b => E
| ct2 c => F
end
```

Example new type

```
Inductive nat := O | S (n : nat).
```

- Use the term `S (S (S O))` to represent 3
- Do not represent negative integers
- No bounds
- Pattern-matching gives two features
  - perform different actions for expressions `O` and `S` x
  - give access to sub-term x
- This type as inherent recursion
  - safe recursive programming
- numerical notation is programmed layer

Example new type: lists

```
Inductive list (A : Type) :=
 | nil
 | cons (a : A) (l : list A).

Check cons 1 (cons 2 (cons 3 nil)).
```

- beware of notation `cons 3 nil` : two arguments
- *In principle*, `nil` takes a type as first argument
  `cons` takes a type t, a value in t, a value in the type `list` t
- Implicit arguments: easier to write, not easy for beginners

Coq as a programming language

- Programming with lists of natural number, you can already represent many programs
- For some applications, Coq datatype are just as good as others example : a C compiler
- You can also write programs that perform proofs for you

Goal directed proof

- Given a proposition, how do you build a proof?
- *top-down construction of proof*
- Programs with *holes*, Each hole has an associated proposition.
- Example: attempt to prove B
- There exists a theorem `th` that proves A `->` B
- One possibility is `th ?1`
- But now you have to find a proof of A

Goal directed proof (2)

- Attempt to prove A `->` B
- One possibilty is `fun h :` A `=>` ?2
- But now you have to find a proof of B (allowed to use h)
- Goals : a context and a conclusion
- Operations of this slide and previous slide are performed by *tactics*
- Demo time

Proving programs

- Confront a program with a specification
- A specification is like a test suite
  - When testing, sometimes write a function that tests outputs
  - pick a few sample of possible inputs, run the program and test outputs
- A proof is often the same except:
  - cover all possible inputs
  - Even if input set is infinite (thanks to induction)

Coq : an international success

- Not described all powerful features
  - Separate compilation
  - Higher-order reasoning
  - Dependent types
  - Generation of executable programs
- Major examples
  - Compiler correctness proofs
  - Security proofs
  - Cryptography (probabilistic reasoning)
  - Numerical approximations
  - Pure mathematics

Coq maturity

- Two awards in 2013-2014,
- Open source, hosted on inria.gforge.fr, mirrored on github
- One (American) company in Germany has 9 developers on Coq
- Airbus might be interested in using a Coq derived product (CompCert)
- One French company used Coq for 5-10 years but practically stopped.
- Maybe a dozen researchers at Microsoft are using Coq
- Bug tracking, tar balls, available on `http://coq.inria.fr`
- Development in Ocaml mainly by researchers and PhD students
- Traditionally difficult to integrate engineers