Coq, an overview

Damien Pous

EJC, Mai 2013

What is Coq?

- A “proof assistant”
- A “formal proof management system” (from Coq webpage)
- A programming language
- A specification language
- An interactive prover
- A project initiated by Thierry Coquand in 1984, and still under active development…

What is Coq useful for?

- Formally “certify” existing programs/libraries
- Build “certified” software
- Prove or certify mathematical theorems

What Coq is not?

- A fast/distributed/component-based programming language
- A Turing-complete programming language
- A model-checker
- A proof checker
- An automatic prover
- An oracle
- Something easy to work with

What did we learn?

- There is a single language (gallina), for:
  - programs/functions,
  - specifications,
  - proofs.

  This is a purely functionnal language.

- There is another language (tactics: Ltac):
  - for building/searching proofs,
  - that can be used interactively.

  There are primitive tactics (`intros`, `apply`, `induction`), and rather complex ones (`tauto`, `ring`).

Principles - Curry-Howard correspondance

"proofs are programs"

$$
\begin{array}{lccccccc}
p: & (A \to B) & \to & (B \to C) & \to & A & \to & C \\
 & f & \mapsto & g & \mapsto & x & \mapsto & g(f(x))
\end{array}
$$

property P	type T	(interface)
proof p	term t	(implementation)
proof-checking	type-checking	
$p \vdash P$	$\vdash t : T$	

Principles - Gallina

- Checking a proof is easy: this is just type-checking...
  ... but we need to trust the type-checker.
- Gallina is a quite small language,
  - for which type-checking is (easily) decidable;
  - and still remains really expressive.
- It relies on a strong theoretical background:
  - the "Calculus of Inductive Constructions",
  - which comes from the λ-calculus.

Principles - Ltac

- Sequences of tactics do not constitute proofs:
  tactics produce gallina terms that can be checked by Coq.
- We don't need to trust tactics: any way to obtain a proof is valid since the proof will be checked.
- Proofs can actually be searched by other means than Ltac.

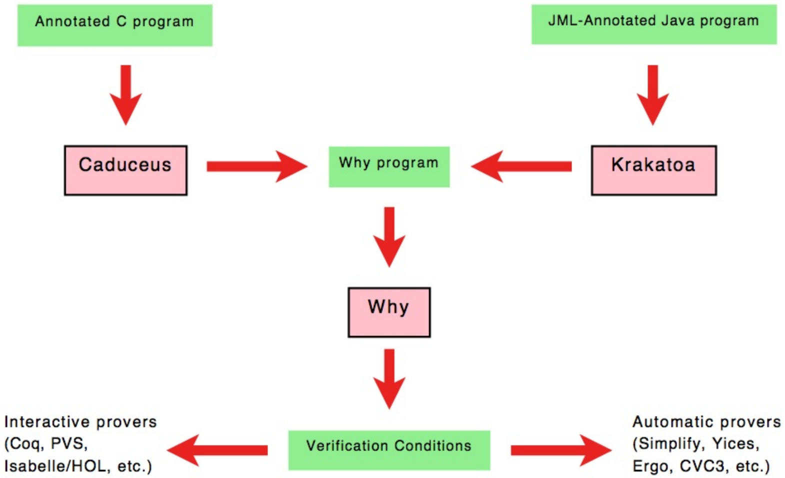
Prove/certify mathematical theorems

- We just proved some elementary theorems,
  more complex ones can be proved too!
- Two major examples:
  - Georges Gonthier et al.'s proof of four-colours theorem;
  - Poplmark challenge.
  - Feit-Thompson's theorem

Certify existing software

- Given an existing program, we might want to prove:
  - the absence of runtime errors,
  - termination,
  - behavioural correctness.
- Problem: sometimes, programs are not written in Coq...
- A solution: Why and Krakatoa/Caduceus tools.

  (see Jean-Christophe Filliâtre' gallery of certified programs: `http://why.lri.fr/examples/`)


Annotated C program
JML-Annotated Java program
Caduceus
Why program
Krakatoa
Why
Interactive provers
(Coq, PVS,
Isabelle/HOL, etc.)
Verification Conditions
Automatic provers
(Simplify, Yices,
Ergo, CVC3, etc.)

Build certified software

- If we have to write a new program, why not writing it and certifying it within Coq?
- Not so realistic, Coq is definitely too slow:
  - it's interpreted;
  - integers, floats... are not 'native'.
- However, Coq programs can be extracted to other languages: OCaml, Haskell and Scheme.
- This is how Xavier Leroy obtained its certified compiler for C: `http://compcert.inria.fr/`

More about the programming language

- Section mechanism:
  - allows one to work under hypothesis,
  - easy way to define polymorphic objects.
- Module system, functors:
  - makes it possible to structure code and proofs,
  - facilitate code reuse.
- Rather large standard library (functions/theorems):
  - $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$;
  - logic, relations;
  - lists, finite sets...

More about the programming language - bis

Dependent types: types may contain (depend on) terms.

- In OCaml: `list` is a polymorphic type constructor (`list int`, `list float`, `list (list int)`), that is, a function from types to types: `list: Type→Type`.
- In Coq we can also define functions from terms to types; for example the function `vect: nat→Type`, that associate to each natural number, the set of vectors of that length.

- For type-checking to remain decidable, we need strong normalization: otherwise, how to decide that
  `vect (f 0) = vect (f 1)`
  for an arbitrary function `f: nat→nat` ?

More about the programming language - ter

Inductive constructions: how datatypes and predicates are defined.

- ```
  Inductive nat: Set :=
  | O: nat
  | S: nat -> nat.
  ```
- ```
  Inductive vect (X: Set): nat -> Set :=
  | vnil: vect X O
  | vcons: forall n, X -> vect X n -> vect X (S n).
  ```
- ```
  Inductive le: nat -> nat -> Prop :=
  | le_n: forall n, le n n
  | le_S: forall m n, le m n -> le m (S n).
  ```
  ```

More about tactics

We have seen some tactics, either quite simple (`apply`), or quite impressive (`ring`).

- There actually is a tactic language, that makes it possible to build complex tactics from simpler ones;
  - let's play with it...
- But we can also use Coq itself to resolve some problems!
  - this is called reflection;
  - let's give an example...
- Last, we can use any external tool in order to find a proof:
  - Coq can communicate to the outside world through XML documents;
  - you can use your favorite language to hack a particular tactic.

Sum-up

- Coq is a programming language:
  - purely functionnal;
  - interpreted (rather slow), but programs can be extracted to fast, compiled, languages;
  - one real constraint: all programs terminate.
- Coq is an expressive specification language:
  - any mathematical property can be stated.
- Coq certifies proofs by a simple type-checking algorithm.
- Coq is a proof assistant:
  - the interactive mode allows us to prove a theorem progressively, by using tactics;
  - tactics can be more or less elaborated, and can be defined by the user.

Related software

- Why, Krakatoa/Caduceus,
  - for the analysis of Java and C programs.

- Isabelle/HOL
  - Larry Paulson - U. of Cambridge
    & Tobias Nipkow - U. of München
- Twelf
  - Karl Crary & Robert Harper - Carnegie Mellon U., USA

History / people

- 1984: Thierry Coquand and Gérard Huet implement the Calculus of Constructions (INRIA-Rocquencourt)
- 1991: Christine Paulin extended it to the Calculus of Inductive Constructions
- 2005: Georges Gonthier et al. use Coq to prove the 4-colours theorem
- 2008: Xavier Leroy et al. build a certified compiler for C
- 2012: Feit-Thompson: classification of finite groups of odd order
- Other developpers: Chet Murthy, Jean-Christophe Filliâtre, Bruno Barras, Hugo Herbelin, Assia Mahboubi and more than thirty people...
  TypiCal (formerly LogiCal), ProVal and Marelle projects