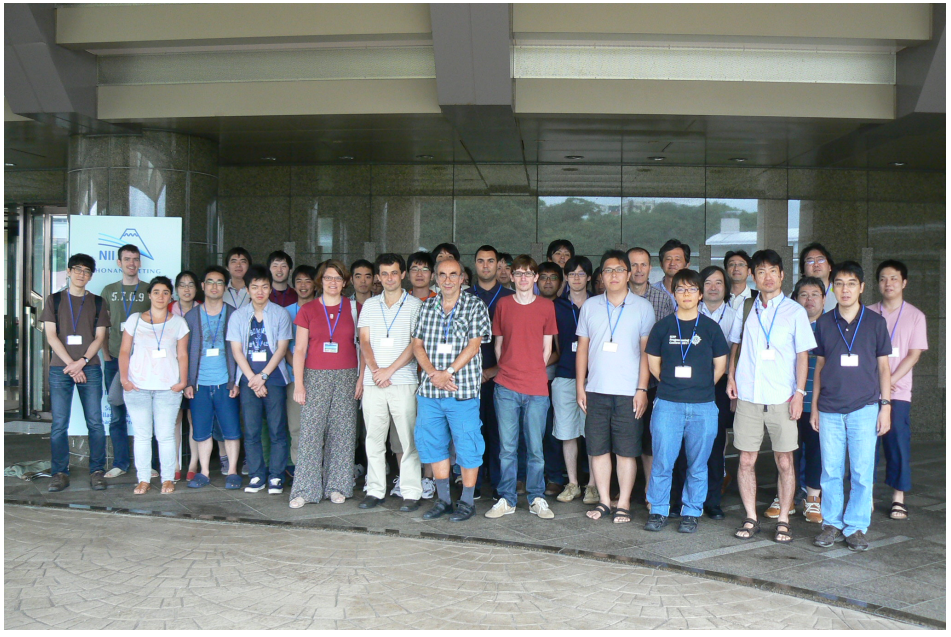
The Coq proof assistant: principles, examples and main applications

Pierre Castéran, University of Bordeaux

NII, Shonan Meetings 100th Commemorative Symposium,
Tokyo, June 22, 2018



The first Shonan School on Coq, co-organized with Jacques Garrigue (University of Nagoya, Japan) and David Nowak (CNRS and Lille 1 University, France) August 25-29, 2014.

Plan of the talk

1. The need for computer-aided theorem proving
2. Proof Assistants
3. Should we trust a program proved with Coq?
4. The daily use of Coq
5. Who is working on Coq?
6. Learning Coq
7. Conclusion

The need for computer-aided theorem proving

Computer Science

- Critical software is everywhere: transportation, health, cryptography, etc.
- Programming languages require *safe* compilers and static analysers.
- Tests and model checking cannot solve all the correctness issues (mainly because of potentially infinite number of parameters and data)
- Due to classical indecidability results, we know that no fully automatic method can guarantee the correctness of any program. Thus, many proofs of correctness require man/machine interaction.

Examples of applications

- Correctness of functional programs
  - Proof that a given function meets its specification.
  - Program extraction from a (constructive) proof of realizability.
- Study of programming languages semantics:
  - Compiler certification (*CompCert*).
  - Static analysis of *C* programs (*Verasco*)
  - Proof of correctness of *C* and/or *Java* annotated programs, with *Why3*, *Frama-C*, *Krakatoa*.
  - Validation of program transformation and composition schemes.
  - Impossibility proofs (many examples in distributed algorithmics).

Mathematics

- Some recent theorems, like the *Four colors theorem*, the *Kepler conjecture*, the *Feit-Thompson theorem*, require huge proofs, containing very large case analyses and long computations.
- For instance, the proof of the four color theorem requires the study of *633* "configurations" (potential counter-examples). Writing and reading such demonstrations is impossible to a human being.

- Some proofs contain very abstract and counter-intuitive parts. Their representation inside a computer memory allows us to explore every part of the proof, and get a better understanding than a paper proof.
- For instance, a development on the *Hydra game* uses combinatorial properties of ordinal numbers and complex induction schemes.

Proof Assistants

A *Proof assistant* is a software that combines two functionalities:

- Verifying that a given proof is complete and respects the rules of logic.
- Helping the user to build long and complex proofs.

Examples

Let us cite *LCF*, *Coq*, *Isabelle*, *HOL*, *PVS*, *ACL2*, etc.
They differ from each other according to the logic they implement, e.g. classical or intuitionistic, the extent of their standard library, their ability to derive executable programs from proofs, how they guarantee the user against logical errors, etc.

From now on, we focus on specific properties of the Coq proof assistant.

Tactics and proof scripts

Proofs may be very long, thus the user does not have to type them in full detail, but writes *proof scripts*, the execution of which builds tentative *proofs terms* that are subsequently checked for correctness.

- A script is composed of *tactics* like *perform an induction, a proof by cases*, *apply a given theorem*, etc. The user can also program his/her own tactics (in Coq or OCaml).
- Search tools help the user to find which already proven theorem can be applied in a given situation.

Example

See revrev.v

Should we trust a theorem or program proved with Coq?

ACM Software System Award (2013)

Because it can be used to state mathematical theorems and software specifications alike, Coq is a key enabling technology for certified software. The system is open source, is supported by a substantial and useful library as well as full documentation, and has attracted a large and active user community.

Before trusting any "proved with Coq" software or theorem, we must take Coq's structure into consideration.

The *Calculus of Constructions* was defined in Thierry Coquand's PhD thesis (1985), and was enriched by Christine Paulin-Mohring in 1990, becoming the *Calculus of Inductive Constructions (CIC).*

The CIC is a *typed lambda-calculus* powerful enough to represent the usual types of programming languages, program specifications, as well as mathematical statements and proofs.
On top of the so-called *Barendregt cube*, the CIC allows to write *polymorphic*, *higher-order* and *dependently typed*[a] functions.

[a] For instance, a function that is guaranteed to return a prime number, a sorted list, etc.

Some types of the CIC

- The type of lists of elements of type `A`:

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

- The type of prime numbers:

```
{n : nat | 1 < n /\
  ∀ p q: nat, p * q = n -> p = n \/ q = n}.
```

- The type of the polymorphic function `List.app` (append)

```
∀ A: Type, list A -> list A -> list A.
```

- A logical rule:

```
∀ (A: Type)(P:A -> Prop), ~(∃ x:A, P x) ->
                          ∀ x:A, ~(P x).
```

Example : a polymorphic function

```
Function app {A:Type}(l l': list A): list A :=
match l with
| nil => l'
| hd :: tl => hd :: (app tl l')
end.
```

A *proof term* (automatically built by Coq) of a derived logical rule:

```
Check
fun (A : Type) (P : A -> Prop)
(H : not (∃ x : A, P x))
(x : A) (H0 : P x) =>
 H (ex_intro (fun x0 : A => P x0) x H0).
```

```
∀ (A : Type) (P : A -> Prop),
    ~(∃ x : A, P x) -> ∀ x : A, ~(P x)
```

Main components of Coq

- The *critical kernel*, a type-checker for the Calculus of Inductive Constructions. A theorem is accepted only if its statement is the type of its proof term.
- A programmable tactic engine, which helps the user to build semi-automatically proofs of statements (mathematical theorems and program proofs) *(not critical)*
- A standard library containing definitions, theorems and tactics about mathematical structures and data types. *(built during Coq's compilation, hence to be trusted)*

Consequence

To trust a development in Coq, one just has to check the adequacy of the definitions (specifications and programs), not looking at the details of the proofs. The Coq development team, as well as expert users, constantly watch the consistency of the CIC, as well as its translation into the kernel wtriten in OCaml.

Directly or indirectly, the user of the Coq proof assistant *accepts* the theoretical studies initiated by Thierry Coquand et al., *and* the *open source* implementation of this theory (in the functional programming language OCaml).

Some typing rules may look quite abstract at first reading. Let us quote two examples:

$$\frac{E[\Gamma] \vdash t : \forall x : U,\ T \quad E[\Gamma] \vdash u : U}{E[\Gamma] \vdash (t\ u) : T\{x/u\}}\ App$$

$$\frac{E[\Gamma] \vdash U : s \quad E[\Gamma] \vdash t : T \quad E[\Gamma] \vdash T \leq_{\beta\delta\iota\zeta\eta} U}{E[\Gamma] \vdash t : U}\ Conv$$

Fortunately, these rules are not to be learned by heart. Many examples are shown in books, tutorials, and the standard library.

Let us consider a huge formal proof in Coq of a given theorem. To accept this proof, we do not have to read it in detail. It suffices to trust the software's *critical kernel* in charge of verifying all the details of the proof.

The implementation of *tactics* may be bugged, in which case the theorems would be more difficult to prove. But please keep in mind that the major risk would be to accept as true a false statement.

Remark

Other proof assistants : Isabelle, HOL, Agda, etc. offer the same level of safety, possibly with other means.

To make a development in Coq trustable, one should respect the following advice.

- Avoid the use of *axioms*, which can be the source of inconsistencies. Coq's documentation lists which [sets of] well-studied axioms can be safely added to Coq's logic.
- Make your *definitions* understandable by other people. For instance a specialist in graph theory should accept that the work by Georges Gonthier and Benjamin Werner truly proves the 4 color theorem.

The daily use of Coq

Correctness proofs of functional programs

- Define in *Gallina* (the specification language of Coq) some functions (terminating and side-effect free).
- Prove some *companion lemmas* (important properties of the functions).
- Infer from those lemmas that the functions meet a given *specification*.

Examples

- List reversing is involutive: revrev.v
- A correctness proof of (purely functional) insertion sort: sort.v

Correct program synthesis

- We start from some *specification*, often expressed as an input/output relationship.
- For instance, "the list l' is a sorted permutation of l "
- Give a (constructive) proof that for any list l, there exists some list l' such that the relation holds.
- By *extraction*, transform this proof into a functional program that may be compiled and executed.

Remark

This technique is used to obtain the *CompCert* certified *C* compiler [Com].

Proving imperative programs

Software of the *Why3* family: *Frama-C*, *Krakatoa* take as inputs an *annotated* program written in *C*, *Java*, ...The annotations specify pre- and post-conditions, loop invariants and variants.

```
/*@ requires n >= 0 && \valid_range(t,0,n-1);
  @ assumes Sorted(0,n-1,t); */
int binary_search(int *t, int n, int v) {
  int l=0, u = n-1;
  /*@ loop invariant
    @ 0 <= l && u <= n-1 &&
    @ forall integer k; 0 <= k < n && t[k]=v ==>
    @                   l <= k <= u; */
  while (l<= u) {
    int m = l + (u - l) / 2;
            ...
    }
  }
```

- Some implicit annotations control safety properties (invalid memory access for instance).
- *Why3* associates to every annotation a purely logical theorem statement called *verification condition* that ensures its validity in every execution of the program.
- Each verification condition is solved, either automatically or interactively, depending on its difficulty.

Please note that Coq is used at two levels:

- To solve the most complex verification conditions (the automatic provers fail to solve).
- At the meta-level, for proving the correctness of the *generator* of verification conditions from the annotated program. *If every condition is valid, then all the annotations will be satisfied in every execution of the program.*

Who is working on Coq?

- The Coq development team, which maintains and document the software.
- Researchers in type theory and fundations of mathematics.
- Contributors to proof pearls, libraries, and plugins.

Who is using Coq?

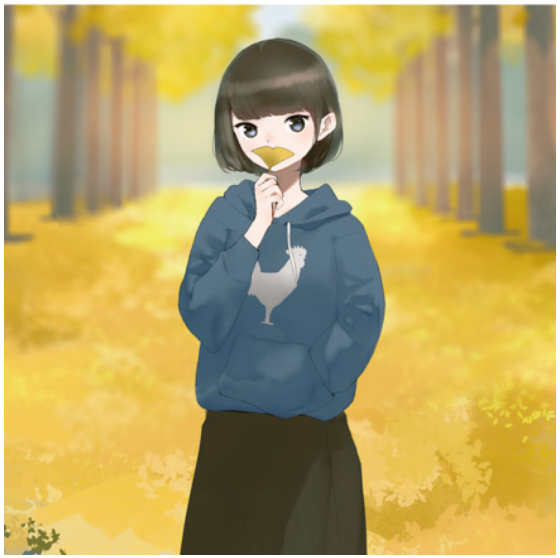
- People who need a formal proof that cannot be built by a automatic theorem prover.
- Researchers who want to guarantee the correctness of a new algorithm or a complex theorem.
- Referees who have to check whether some submitted proof is correct.
- etc.

Learning Coq

- Coq is a quite complete, hence big, software. Its reference manual is more than 500 pages long [Coq].
- Happily, in a few days, one is able to write simple proofs of small recursive functions.
- In a second step, the user has a project (in computer science or math) and s/he learns the rest of Coq by necessity.

Documentation

- Many tutorials on Coq's page
- Several books, alraedy available [BC04, Chl11, P$^+$], or in preparation (including books on *SSreflect*).
- Many posts on the *coq-club* mailing list and *stack overflow*.
- Coq summer schools and workshops are organized.



(by Tsukuba Coq Users' Group)

As a conclusion ...

> *The process of using the proof assistant becomes pretty natural. In fact, it's a bit like playing a video game. You interact with the computer. You tell the computer, try this, and it tries it, and it gives you back the result of its actions. Sometimes it's unexpected what comes out of it. It's fun."*

Vladimir Voevodsky, interview for *Scientific American*.

[BC04] Yves Bertot and Pierre Castéran.
Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.
Springer, 2004.
http://www.labri.fr/perso/casteran/CoqArt/index.html.

[Chl11] Adam Chlipala.
Certified Programming with Dependent Types.
MIT Press, 2011.
http://adam.chlipala.net/cpdt/.

[Com] CompCert Development Team.
The CompCert compiler.
http://compcert.inria.fr.

[Coq] Coq Development Team.
The coq proof assistant.
coq.inria.fr.

[P+] Benjamin Pierce et al.
Software foundations.
https://softwarefoundations.cis.upenn.edu/.