Program proving and synthesis with the Coq proof assistant

Pierre Castéran
University of Bordeaux, and LaBRI

Kraków, May 9 2016

Overview

What is *Coq*?

- A (not so efficient) purely functional, statically typed, not Turing complete, programming language,
- A proof assistant, allowing to build interactively large proofs of mathematical statements and/or program correctness.
- A proof checker, that verifies whether its argument is a correct proof of a given statement.
- In fact, proofs are programs, theorem statements and program specifications are types, and the proof checker is "just" a type checker.

History

The ancestors

1971	Nqthm (Boyer – Moore) Automated proofs of Lisp functions. *ACL2* is the current stage of this research.
1980	LCF (Robin Milner). Interactive proofs of program correctness; invention of *ML* for programming *proof tactics*.
1980	Intuitionnistic Theory of types (P. Martin-Löf).
1984	Nuprl (R. Constable) Software design from formal proofs, based on Martin-Löf's theory.

The Calculus of Constructions and Coq

1984 The Calculus of Constructions (T. Coquand and G. Huet)

1985 CoC: First Implementation of this calculus

1988 The Calculus of Inductive Constructions (C. Paulin-Mohring). The Coq system takes its definitive form:

- A (rather small) kernel for verifying the proofs
- A set of tools for interactive proof construction, in a LCF-like way,
- A standard library, containing definitions, theorems and tactics for many data-types.
- A repository of users contributions.

Main realizations

2002	EAL7 certification of JavaCard
2005	Formal proof of the Four-colors theorem (G. Gonthier and B. Werner)
2008	First version of the Compcert verified C compiler (X. Leroy *et al.*)
2012	Proof of the Feit-Thompson theorem (Mathematical Components Group)
2015	Verasco : a static analyzer for the Compcert subset of ISO C 1999 that establishes the absence of run-time errors in analyzed programs.

2013: ACM Sigplan Programming Languages Award *and* ACM Software System Award

Because it can be used to state mathematical theorems and software specifications alike, Coq is a key enabling technology for certified software.

Coq has played an extremely influential role in several disciplines including as formal methods, programming languages, program verification, and formal mathematics. With increasing emphasis on the design and development of secure applications, certification has become important to the academic community as well as industry.

Coq as a (strange) programming language

Let us consider a simple programming example : efficient computation of a^n.

- Definition in Gallina, the specification language of Coq:
  - Statically (strongly) typed
  - Not even Turing complete (all computations must terminate)
- Formal specification and correctness proof

- In Coq, every variable is typed.
- Polymorphism is described with *type variables*.
- Type variables are variables, so they also have a type (called a sort).

```
Section Definitions.

 Variable A : Type.
 Variable one : A.
 Variable mult : A -> A -> A.

 Infix "*" := mult.
```

Coq's logic requires that the evaluation of any expression terminates. Thus, it loses Turing completeness.
Allowed recursive definition must follow the structure of the datatype on which they operate. For instance, parity test and division by 2 on positive integers are expressed through pattern matching on their binary representation.

```
Function bin_expt (a: A) (p : positive) : A :=
match p with
| 1 => a
| (* 2 * q *)
   q~0 => bin_expt (a * a) q
| (* 2 * q + 1 *)
   q~1 => a * bin_expt (a * a) q
end.
```

Remarks

- Programming in Coq is not limited to primitive recursive functions.
- The class of functions definable in Gallina contains all functions that are provably total in higher-order logic.
- The cost of this large expressive power is that some complex function definitions must include proofs of their termination.
- Fortunately, a collection of “design patterns” is being developed and documented.

Propositions, predicates, etc.

Coq's type system, in addition to types like Z, positive, Type, etc., contains a sort Prop, that contains logical statements.
The following declarations express that all our forthcoming constructions will be guaranteed only for types that can be provided with a monoid structure.

```
Hypothesis mult_assoc :
    forall a b c: A, a * (b * c) = (a * b) * c.

Hypothesis one_left : forall a, one * a = a.
Hypothesis one_right : forall a, a * one = a.
```

Expressing correctness

Correctness of a function like bin_expt can be expressed in terms of a naïve, straightforward implementation of exponentiation.

```
Function  power (x:A)(n:nat) : A :=
match n with
| 0%nat => one
| S p => x * power x p
end.
Infix "^" := power.

Definition correct_expt (f : A -> positive -> A) :=
forall a n, f a n = a ^ n.
```

Interactive proofs

How to prove a theorem?

1. You give the statement, as a well-typed expression.
2. Proving this statement is transformed into a goal
3. You use tactics for breaking goals into subgoals
4. When no goal remains to be solved, Coq computes a term and checks whether its type is the announced theorem statement.
5. The theorem (including the proof) is registered for further applications.

Let us look (partially) at a small demo (file expt.v)

What are the tactics?

- Tactics asssociated with the basic rules of logic (connectives, quantifiers, etc)
- Tactics associated with data-types : proof by cases, induction, etc.
- Tactics defined by the user :
  - Smart application of theorems
  - Using the dedicated language Ltac (tactic composition, recursion, inspection of the subgoals)
  - Tactics by reflection (mix of computation and deduction)
  - Automation of [semi-]decidable problems : Presburger arithmetics, ring structures, etc.

A case study

- For illustrating some certification techniques in functional programming, we propose to consider a small example, still about efficient computation of x^n in any monoid.
- It is well-known (since Brauer, Knuth, etc.) that the famous binary exponentiation algorithm is not optimal.
- With S. Brlek and R. Strandh, we studied the following method :
  1. For any n, generate a specific pogram (in Scheme)
  2. Execute or compile this program
- Let us look briefly present at a certification of this method.

For instance, our generator computed the following algorithm for computing x^{87} (more efficient than the binary algorithm).

$$\begin{tikzcd}
x \arrow[r] \arrow[rr, bend left] \arrow[rrrr, bend left] & x^2 \arrow[r] & x^3 \arrow[r] \arrow[rrr, bend left] & x^6 \arrow[r] & x^7 \arrow[r] \arrow[rrrrr, bend left] & x^{10} \arrow[r] & x^{20} \arrow[r] & x^{40} \arrow[r] & x^{80} \arrow[r] & x^{87}
\end{tikzcd}$$

Here is an automatically generated description in Coq.

```
Definition f (A : Type) (x : A) :=
       x0 <--- x times x; x1 <--- x0 times x;
       x2 <--- x1 times x1; x3 <--- x2 times x;
       x4 <--- x3 times x1; x5 <--- x4 times x4;
       x6 <--- x5 times x5; x7 <--- x6 times x6;
       x8 <--- x7 times x3; Return x8
```

Which correctness statements ?

Two kinds of correctness statements deserve to be proved:

Generate, then certify

For a given n and a given program, prove that the program correctly computes x^n.

Certify a generator

Write a generator and prove one and for all that this generator always returns a correct exponentiation program.

Proof by Reflection

We want to prove efficiently that the following function computes x^{87} in any monoid.

```
Definition f (A : Type) (x : A) :=
       x0 <--- x times x; x1 <--- x0 times x;
       x2 <--- x1 times x1; x3 <--- x2 times x;
       x4 <--- x3 times x1; x5 <--- x4 times x4;
       x6 <--- x5 times x5; x7 <--- x6 times x6;
       x8 <--- x7 times x3; Return x8.
```

1

Prove that the application of f on any pair of types A and B leads to similar computations. We say that f is parametric.

2

By induction on the function body, we prove that any parametric function defined in our toy language respects monoid isomorphisms.

3

Take B as the type of natural numbers, replace multiplication by addition. We prove that f is a correct implementation of $\lambda x.\, x^n$ iff f applied to 1 returns n.

Proof by computational reflection

Thus, a simple computation of f(1) is enough for proving f's correctness.

```
(** generic *)
Ltac parametric_tac  :=
 match goal with [ |- parametric ?c] =>
   red ; intros;   repeat (right;[assumption | assumption
   left; assumption
 end.

Ltac param_correct :=
match goal with [|- correct ?c ?p ] =>
 apply param_correctness; parametric_tac
end.

(** Example *)
Lemma f_ok : correct f 87.
Proof. param_correct. Qed.
(* Finished transaction in 0.005 secs (successful) *)
```

Building a certified generator

1. Write, in Coq, or any suitable language, a function g of type nat → forall A:Type, A → A.
2. Prove in Coq that for all n, $(g(n))(x)$ computes x^n.

For instance, a generator based on continued fraction expansion of the exponent has been written:

1. A first version in Scheme (written by hand)
2. A certified version in Coq,
3. This last version can be extracted towards Scheme (to do)

What about non purely functional languages?

The use of Coq is not restricted to purely functional terminating programs. For instance, it is used in the following contexts :

- Certified compilers : Compcert,
- Verifying assertions in imperative programs : Why3, Krakatoa, Frama-C,
- Static analysis of C programs : Verasco,
- Research on distributed algorithms and reactive systems.

How does it work?

- The considered language (for instance C) is enriched with a language of assertions, (ACSL for C),
  - pre- and post-conditions,
  - loop invariants,
  - safety properties (pointer dereferencing, etc.)
- A dedicated library (definitions, theorems and tactics) describes the operational semantics of the considered language (as a binary relation between machine configurations).

Two levels of verification

Certification of the tool:

Definition and certification in Coq of a verification condition generator that reduces the validity of every assertions to the proof of purely logical statements: the validity of all generated verification conditions entails the correctness of every assertion in any execution that respects the pre-conditions.

Certification of a program annotated with logical formulas

Attempt to *prove* each generated verification condition.

- whenever possible, with an automated theorem prover : Alt-Ergo, Simplify, Z3, etc.
- or with proof assistants like Coq, PVS, Isabelle/HOL, etc.

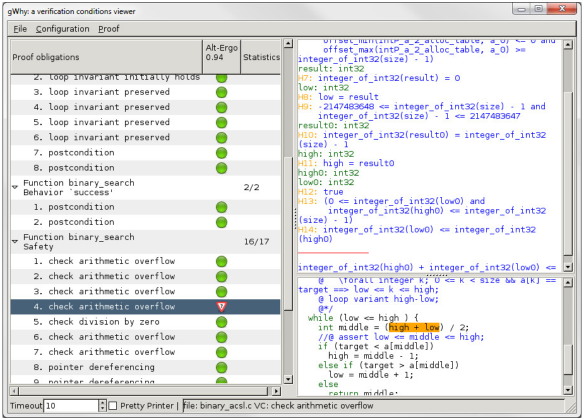
Example : Certification of a small C function

From frama-c.com/jessie.html

```
//@ requires n >= 1 && valid_range(t,0,n-1);
int binary_search(long t[], int n, long v) {
  int low = 0, high = n - 1;
  //@loop invariant 0 <= low && high <= n-1;
  while (low <= high) {
    int middle = (low + high) / 2;
    if  (t[middle] < v)
       low = middle + 1;
    else if (t[middle] > v)
     high = middle -1;
    else return middle;
 }
 return -1;
```

```
//@ requires n >= 1 && valid_range(t,0,n-1);
int binary_search(long t[], int n, long v) {
  int low = 0, high = n - 1;
  //@loop invariant 0 <= low && high <= n-1;
  while (low <= high) {
    int middle = (low + high) / 2;
    ...
  }
```

Arithmetic Overflow ☹.



Proof obligations	Alt-Ergo 0.94	Statistics
2. loop invariant initially holds	🟢	
3. loop invariant preserved	🟢	
4. loop invariant preserved	🟢	
5. loop invariant preserved	🟢	
6. loop invariant preserved	🟢	
7. postcondition	🟢	
8. postcondition	🟢	
Function binary_search Behavior `success'		2/2
1. postcondition	🟢	
2. postcondition	🟢	
Function binary_search Safety		16/17
1. check arithmetic overflow	🟢	
2. check arithmetic overflow	🟢	
3. check arithmetic overflow	🟢	
4. check arithmetic overflow	?	
5. check division by zero	🟢	
6. check arithmetic overflow	🟢	
7. check arithmetic overflow	🟢	
8. pointer dereferencing	🟢	
9. pointer dereferencing	🟢	

```
   offset_min(intP_a_2_alloc_table, a_0) <= 0 and
   offset_max(intP_a_2_alloc_table, a_0) >=
integer_of_int32(size) - 1)
result: int32
H7: integer_of_int32(result) = 0
low: int32
H8: low = result
H9: -2147483648 <= integer_of_int32(size) - 1 and
    integer_of_int32(size) - 1 <= 2147483647
result0: int32
H10: integer_of_int32(result0) = integer_of_int32
(size) - 1
high: int32
H11: high = result0
high0: int32
low0: int32
H12: true
H13: (0 <= integer_of_int32(low0) and
      integer_of_int32(high0) <= integer_of_int32
(size) - 1)
H14: integer_of_int32(low0) <= integer_of_int32
(high0)

integer_of_int32(high0) + integer_of_int32(low0) <=
```

```
    @   \forall integer k; 0 <= k < size && a[k] ==
target ==> low <= k <= high;
    @ loop variant high-low;
    @*/
  while (low <= high ) {
    int middle = (high + low) / 2;
    //@ assert low <= middle <= high;
    if (target < a[middle])
      high = middle - 1;
    else if (target > a[middle])
      low = middle + 1;
    else
      return middle;
```

file: binary_acsl.c VC: check arithmetic overflow

```
integer_of_int_32(low) + integer_of_int_32(high) <=
2^31 - 1
```

Functional specification

```
// @behavior success:
// @assumes
//  forall integer k1, k2; 0 <= k1 <= k2 <= n-1
//    ==> t[k1] <= t[k2];
// @assumes
//  exists integer k; 0 <= k <= n-1  && t[k] == v;
// @ensures 0 <= \result <= n-1 && t[\result] == v

// @behavior failure:
// @assumes  // v does not appear anywhere in the array t
//  forall integer k; 0 <= k <= n - 1 ==> t[k] != v;
// @ensures \result == -1;
```

The CompCert certified compiler

CompCert

The CompCert project investigates the formal verification of realistic compilers usable for critical embedded software. Such verified compilers come with a mathematical, machine-checked proof that the generated executable code behaves exactly as prescribed by the semantics of the source program. By ruling out the possibility of compiler-introduced bugs, verified compilers strengthen the guarantees that can be obtained by applying formal methods to source programs.

The main result of the project is the CompCert C verified compiler, a high-assurance compiler for almost all of the ISO C90 / ANSI C language, generating efficient code for the PowerPC, ARM and x86 processors.

Should we trust a Coq-certified software ?

The tool

- According to Thomas Hales, any computer program contains a bug in every 500 lines of source code.
- For instance, the size of Coq's sources tarfile is 4 MB of code written in `C`, `Ocaml`, `Coq`.

- What must be really bug-free is Coq's critical kernel, which is called for verifying whether a proof is correct. The risk is that a buggy software gets a certificate.
- The part of the software that helps the user to build a proof is not critical. It would just make harder to obtain a proof.

Should we trust a Coq-certified software (2) ?

A development written by someone else

A development is composed of:

- Definitions : data types, functions, predicates
  - Give human-readable, almost naïve definitions.
  - Give also smarter definitions, that make proving or computing easier.
  - Prove that all these definitions are logically equivalent.
- Axioms: Unsafe! may be (mutually) inconsistent
  - Prefer Hypotheses (with local scope).
  - At worst, non satisfiable hypotheses will generate useless, but harmless results.
- Theorems and certified programs (with kernel's safety)

Interactive vs. Automatic Theorem/Program Proving

Proving a program's correctness requires the proof of many statements, of various difficulty.

- Use several trusted provers (automatic and/or interactive).
- Begin with the most automatic solvers.
- Keep interactive proving for the unsolved lemmas.

This methodology is compatible with tools like Why3 /Frama-C, but also in the B-familiy : Rodin, Atelier-B, etc.

On can also program complex tactics, for automating whole classes of [semi]-decidable problems.

About the Diversity of Proof Assistants

- According to wikipedia (article on Proof Assistants), there is about 12 proof assistants, with various characteristics.
- This diversity is excellent for the evolution of interactive theorem proving. All these systems are mutually influenced.

Some specificities of Coq

- Embedding of logic in functional programming: writing a proof ort a tactic is writing a functional program.
- Dependent types: *e.g.* prime number, sorted list, etc.
- Possibility of deriving dedicated induction principles and tactics.
- Learning Coq requires some experience in functional programming. Dependent types require some training.

Perspectives (non-exhaustive list)

- Improve extraction towards Scheme, consider also typed racket.
- Consider Common Lisp or Scheme programs with assignment, in a Why3 plug-in.

Some links

- Documentation and download: coq.inria.fr
- Static Analysis and Certified Compiler:
  - frama-c.com/
  - compcert.inria.fr/
  - compcert.inria.fr/verasco/
- Books:
  - Interactive Theorem Proving and Program Development www.labri.fr/∼casteran/CoqArt
  - Software Foundations: cis.upenn.edu/∼bcpierce/sf
  - Certified Programming with Dependent Types: adam.chlipala.net/cpdt/
- Mailing list : `coq-club@inria.fr`