Approximation Theory and Proof Assistants: Certified Computations

CR16, Nicolas Brisebarre & Damien Pous

Certified approximations

- Course organisation:
  - the maths: Nicolas Brisebarre, generally on Thursdays
  - their formalisation in Coq: Damien Pous, generally on Fridays
  - + two courses by guest stars: Mioara Joldes (Toulouse) and Florent Bréhard (Lille)
- Evaluation:
  - exercises from one week to the other, both in Coq and on paper
  - progressively, Coq exercises become a project
  - table exam during the last week (on paper)
- Website, references:
  `https://m2coqapprox.gitlabpages.inria.fr/`

Today

- Overview of the Coq proof assistant (me)
- Hands on (you&me)

Coq is a proof assistant

It can be used in order to

- prove/certify mathematical theorems
- certify existing programs/libraries
- design certified software

Coq is not:

- a fast/distributed/bitcoin-oriented programming language
- a Turing-complete programming language
- a model-checker
- an automatic prover
- an oracle
- something easy to work with

Principles: Poincaré

- mathematical proofs can be arbitrarily complex

  (and thus difficult to find)
- proofreading is easy...

  (and thus boring)

Principles: Poincaré

- mathematical proofs can be arbitrarily complex

  (and thus difficult to find)
- proofreading is easy...

  (and thus boring)

... once we agree about what a proof is

Principles - Curry-Howard correspondance

"proofs are programs"

$$(A \to B) \land (B \to C) \land A \quad \to \quad C$$

Principles - Curry-Howard correspondance

"proofs are programs"

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

Principles - Curry-Howard correspondance

"proofs are programs"

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

Principles - Curry-Howard correspondance

"proofs are programs"

$$\begin{array}{rcccc} p: & (A \rightarrow B) \wedge (B \rightarrow C) \wedge A & \rightarrow & C \\ & (f, g, x) & \mapsto & g(f(x)) \end{array}$$

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

Outline

Quick tour of syntax and basic principles

Three kinds of use

- Prove/certify mathematical theorems
- Certify existing software
- Build certified software

Summary

Syntax

Let's play

What did we learn?

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

  This is a purely functional programming language.

What did we learn?

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

  This is a purely functional programming 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 - Gallina

- Checking a proof is easy: this is just type-checking...
  ... but we need to trust the type-checker.
- Gallina is a 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.

Outline

Quick tour of syntax and basic principles

Three kinds of use

- Prove/certify mathematical theorems
- Certify existing software
- Build certified software

Summary

Prove/certify mathematical theorems

- We just proved some elementary theorems,
  more complex ones can be proved too!
- Three major examples:
  - Four-colours theorem;
  - Feit-Thompson's theorem (finite groups classification)
  - Kepler's conjecture

Certify existing software

- Given an existing program, we might want to prove:
  - the absence of runtime errors,
  - termination,
  - behavioural correctness.

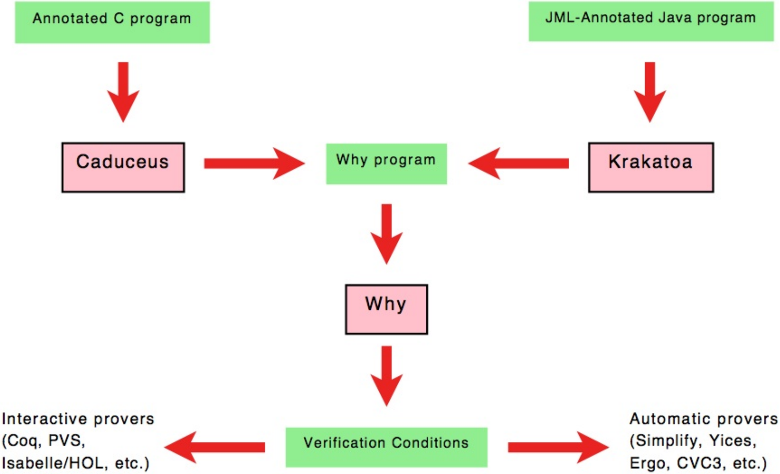
Certify existing software

- 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. . .

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: Why3 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 and Sandrine Blazy obtained their certified compiler for C:
  `http://compcert.inria.fr/`

Outline

Quick tour of syntax and basic principles

Three kinds of use

- Prove/certify mathematical theorems
- Certify existing software
- Build certified software

Summary

Summary

- Coq is a programming language:
  - purely functional;
  - interpreted (rather slow), but programs can be extracted to fast, compiled, languages;
- 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.

History / people

- 1984: Thierry Coquand and Gérard Huet implement the Calculus of Constructions
- 1991: Christine Paulin adds *Inductives*
- 2005: Georges Gonthier: 4-colours theorem
- 2008: Xavier Leroy & Sandrine Blazy: compcert
- 2012: Georges Gonthier, Assia Mahboubi, & many others: Feit-Thompson theorem
- 2013: Vladimir Voevodski: univalence axiom, HoTT book

Related software

- Isabelle/HOL
  Larry Paulson - Cambridge & Tobias Nipkow - München
- Agda
  Catarina Coquand - Chalmers
- Lean
  Leo de Moura - Microsoft, AWS

Hands-on

`https://m2coqapprox.gitlabpages.inria.fr/`