Formalizing Mathematics In A Proof Assistant

An Introduction

Assia Mahboubi

A formal language for mathematics

Formalizing mathematics requires at least:

- Defining a precise and unambiguous language representing mathematical assertions and their proofs;
- Obtaining a small and simple set of well-formedness rules which reduce proof checking to a mechanical task.

Machine checking

Machines are better than humans for routine checking:

- If proof checking boils down to a mechanical task we can use a computer to check mathematical proofs.
- If one trusts the correctness of the program which checks proof, one trusts every proof validated by this program.

(De)Motivations

- Reducing proof checking to a mechanical task is a very old dream

  e.g. Leibniz' Calculus ratiocinator, 1666
- But it is often considered as either not realistic or too boring a topic among mathematicians

  e.g. "The architecture of mathematics", N. Bourbaki, 1962
- This idea nonetheless gained a renewed interest, mostly from computer scientists, after the late 60's.

  e.g. de Bruijn's Automath project, circa 1967

(De)Motivations

Codifying this language, ordering its vocabulary and clarifying its syntax is a useful work which is indeed one of the aspects of the axiomatic method [...]. But - and we insist on this point - this is only one of its aspects, and it is certainly the less interesting.

N. Bourbaki "The architecture of mathematics" 1962.

(De)Motivations

Codifying this language, ordering its vocabulary and clarifying its syntax is a useful work which is indeed one of the aspects of the axiomatic method [...]. But - and we insist on this point - this is only one of its aspects, and it is certainly the less interesting.

The essential motivation of the axiomatic method is precisely to define what the logical formalism is alone unable to provide, which is the profound intelligibility of mathematics.

N. Bourbaki "The architecture of mathematics" 1962.

Proofs and programs

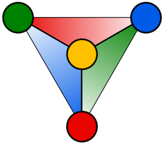
Several reasons can explain why the interest was stronger from computer science inclined people:

- These were more familiar with alternative presentations of foundations, that are more tractable for a concrete use of computers.
- Programmers themselves are exposed to the difficult task of checking the properties of programs.
- They are conceiving, writing and using programs called decision procedures.

Programs and proofs

- SAT/SMT solvers: decision of propositional (modulo theory) formulae
- Termination checkers: termination, liveness properties
- Constraint solving: operation research, scheduling,...
- ...

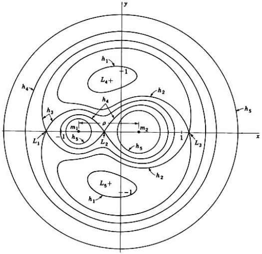
Programs and proofs



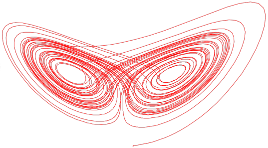
The Four Colour Theorem
K. Appel - W. Haken
G. Gonthier - B. Werner

The Kepler conjecture
Th. Hales - S. Ferguson
The Flyspeck project

Programs and proofs



Celestian mechanics



Existence of the Lorenz attractor
W. Tucker, 2002

Every odd number $n \geq 7$ is the sum of three prime numbers.

H. Helfgott, 2013

This is not only about large calculations

- Theory of programming languages

  e.g. the ▸ Poplmark challenge
- Classification of finite simple groups

  e.g. the formal proof of the ▸ Odd Order Theorem
- Homotopy theory

  see V. Voevodsky' recent talk at IAS, ▸ Pdf slides ▸ Video

In fact this is not (only) about finding bugs in proofs.

Motivations

Indeed every mathematician knows that a proof has not been "understood" if one has done nothing more than verifying step by step the correctness of the deductions of which it is composed, and has not tried to gain a clear insight into the ideas which have led to the construction of this particular chain of deductions in preference to every other one.

N. Bourbaki "The architecture of mathematics" 1962.

The proof assistant zoo

- As of today, there exists many interactive proof assistants that can be used for the purpose of formalizing mathematics.

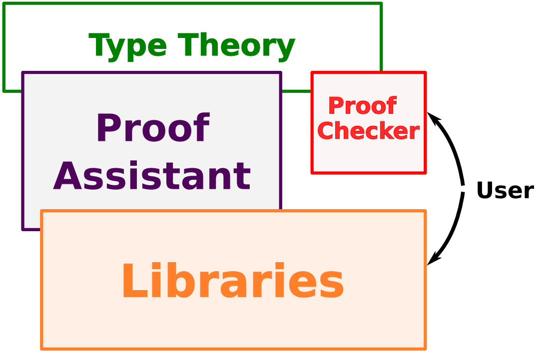
  ACL2, Mizar, PVS, HOL, Isabelle, HOL-Light, Agda, Coq,...

- They differ by their choice of logical foundations, the scope of their libraries, the size and/or interests of their community of users.

Crucial ingredients

- Appropriate logical foundations and a proof checker;
- Correct representations of mathematical concepts in the formal language;
- Helper tools for bridging the gap between the machine checker and the human writer.
- Well-designed, comprehensive and searchable libraries

The Coq proof assistant



Type Theory

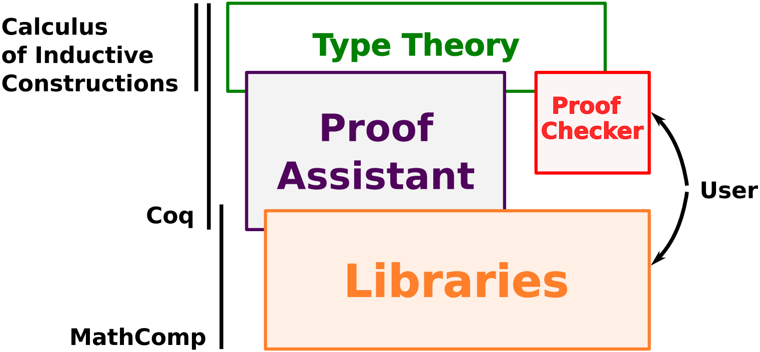
Proof Assistant

Proof Checker

User

Libraries

The Coq proof assistant



Calculus of Inductive Constructions

Type Theory

Proof Assistant

Proof Checker

User

Coq

Libraries

MathComp

The Coq proof assistant

- Calculus of (Inductive) Constructions

  Th. Coquand (1985), Th. Coquand, Ch. Paulin (1989)
- Implemented in Ocaml

  First prototype by Th. Coquand, G. Huet (1984)
- Includes:
  - a proof checker
  - a dedicated interface
  - commands to build proofs (tactics)
  - some libraries of formalized mathematics.

Material for this week

- Coq v8.4pl3: ▸ Webpage and Downloads
- (Optional) Proof General interface: ▸ Download
- Ssreflect language of tactics: ▸ Download ▸ Reference Manual
- Slides, exercises:

  `http://specfun.inria.fr/mahboubi/cirm14.html`

Coq kernel

The task of the Coq kernel is to check typing judgments:

$$x_1 : T_1, \ldots, x_n : T_n \vdash t : T$$

- $x_1, \ldots, x_n$ are variables;
- $T_1, \ldots, T_n, t, T$ are terms;
- $x_1 : T_1, \ldots, x_n : T_n$ is a context.

The judgment is read:

> "In the context $x_1 : T_1, \ldots, x_n : T_n$, the term t has type T."

Terms and Types

Terms include the usual terms of λ-calculus:

- Variables: x, A,...
- Functions: (fun $x \mapsto t$)
- Applications: $(t_1\ t_2)$
- Constants: c

The rules defining what a valid judgment explain how we can assign a type to a term.

Typing rules

A valid typing judgment

$$x_1 : T_1, \ldots, x_n : T_n \vdash t : T$$

can be derived from a typing derivation, which is a tree made with rules like:

$$\frac{\Gamma \vdash (\text{fun } x \mapsto t) : A \to B \quad \Gamma \vdash u : A}{\Gamma \vdash (\text{fun } x \mapsto t)\ u : B}$$

Our first types

We have a collection of constants $(T_i)_{i\in\mathbb{N}}$ called universes. The associated typing rules are:

$$\vdash T_i : T_j, \quad i < j$$

However in all what follows we will leave these index implicit and use the same constant Type for any T_i.

Our first non-empty contexts

A valid typing judgment

$$x_1 : T_1, \dots, x_n : T_n \vdash t : T$$

features a well-formed context $\Gamma := (x_1 : T_1, \dots, x_n : T_n)$.

Well-formed context are constructed as:

- $\emptyset$ is a well-formed context.
- $\Gamma, x : A$ is well formed if $\Gamma \vdash A : \mathit{Type}$ and x is fresh.

Valid judgments on variables follow from well-formed context:

$$\Gamma \vdash x : A \quad \text{if } (x : A) \in \Gamma$$

First steps with the system

Let experiment a small demo illustrating:

- The interaction with the system through tactics;
- The structure of goals;
- The guidance of the system;
- The a posteriori, independent check.

Propositions as Types

Formalizing mathematics in Coq consists in building correct derivations establishing statements of the form:

$$x_1 : T_1, \ldots, x_n : T_n \vdash t : T$$

Certain such judgments can be interpreted as proofs of statements:

$$\frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B}$$

Propositions as Types

Formalizing mathematics in Coq consists in building correct derivations establishing statements of the form:

$$x_1 : T_1, \ldots, x_n : T_n \vdash t : T$$

Certain such judgments can be interpreted as proofs of statements:

$$\frac{\Gamma \vdash t : A \to B \quad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B}$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash (\mathsf{fun}\ x\ \mapsto\ t) : A \to B}$$

Equality

We have a family of equality predicates, which expresses a comparison between two inhabitants of the same type:

$$A : \mathrm{Type}, x : A, y : A \vdash x =_A y : \mathrm{Type}$$

Equality

We have a family of equality predicates, which expresses a comparison between two inhabitants of the same type:

$$A : \mathrm{Type}, x : A, y : A \vdash x =_A y : \mathrm{Type}$$

Equality is reflexive:

$$\frac{}{A : \mathrm{Type}, x : A \vdash \mathrm{eqrefl\ x} : x =_A x}$$

Equality

We have a family of equality predicates, which expresses a comparison between two inhabitants of the same type:

$$A : \mathrm{Type}, x : A, y : A \vdash x =_A y : \mathrm{Type}$$

Equality is reflexive:

$$\frac{}{A : \mathrm{Type}, x : A \vdash \mathrm{eqrefl\ x} : x =_A x}$$

Equality is substitutive, in a sense we will make precise later.

In all what follows, we write $=$ for any instance of $=_A$.

Conversion Rule and Computation

The type system is parametrized by an equivalence relation $\equiv$

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash t : B} \quad \text{if } A \equiv B$$

This relation can be understood as:

"A and B are equal modulo computation".

Conversion Rule and Computation

For instance β-reduction, which models the evaluation of functions:

$$(\text{fun } x \ \mapsto \ t) \ u \rightarrow_\beta t[x \leftarrow u]$$

is included in the conversion relation:

$$(\text{fun } x \ \mapsto \ t) \ u \quad \equiv \quad t[x \leftarrow u]$$

Hence these two types are convertible:

$$(\text{fun } x \ \mapsto \ f(x)) \ u = f(u) \quad \equiv \quad f(u) = f(u)$$

Dependent Types

Types can depend on terms: in this case they are called dependent types.

This was the case for the type of equality statements:
the type $x = x$ depends on a type A and on a term $x : A$.

More generally, $\forall x : A, B$ is:

- the type of functions f
- that take as argument a term $a : A$
- and output a term $f\ a : B\ [x \leftarrow a]$ whose type depend on a.

Type $\forall x : A, B$ can also be denoted $\Pi x : A, B$.

Dependent Types

Examples:

- The type of our constructor of equality:

$$\mathrm{eqrefl} \;:\; \forall A : \mathrm{Type}, \forall x : A, x = x$$

Dependent Types

Examples:

- The type of our constructor of equality:

$$\mathrm{eqrefl} \ : \ \forall A : \mathrm{Type}, \forall x : A, x = x$$

- The substitutivity is expressed by a term of type:

$$\forall A : \mathit{Type}, \forall P : A \to \mathit{Type}, \forall x : A, Px \to \forall y : A, x = y \to Py$$

Dependent Types

Typing rules:

$$\frac{\Gamma \vdash A : \mathrm{Type} \quad \Gamma, x : A \vdash B : \mathrm{Type}}{\Gamma \vdash \forall x : A, B : \mathrm{Type}}$$

Dependent Types

Typing rules:

$$\frac{\Gamma \vdash A : \mathrm{Type} \quad \Gamma, x : A \vdash B : \mathrm{Type}}{\Gamma \vdash \forall x : A, B : \mathrm{Type}}$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash (\mathrm{fun}\ x \ \mapsto\ t) : \forall x : A, B}$$

Dependent Types

Typing rules:

$$\frac{\Gamma \vdash A : \mathrm{Type} \quad \Gamma, x : A \vdash B : \mathrm{Type}}{\Gamma \vdash \forall x : A, B : \mathrm{Type}}$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash (\mathrm{fun}\ x\ \mapsto\ t) : \forall x : A, B}$$

$$\frac{\Gamma \vdash t : \forall x : A, B \quad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B[x \leftarrow u]}$$

Dependent Types

Typing rules:

$$\frac{\Gamma \vdash A : \mathrm{Type} \quad \Gamma, x : A \vdash B : \mathrm{Type}}{\Gamma \vdash \forall x : A, B : \mathrm{Type}}$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash (\mathrm{fun}\ x \ \mapsto\ t) : \forall x : A, B}$$

$$\frac{\Gamma \vdash t : \forall x : A, B \quad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B[x \leftarrow u]}$$

If x does not appear in B, $\forall x : A, B$ is denoted $A \rightarrow B$

Description of mathematical objects

The type system of the Coq proof assistant allows the user to define new types (and their inhabitants) with inductive definitions.

This is a very powerful tool to describe mathematical objects at a high level.

Disclaimer: in this course we will provide only an informal account of Coq's inductive types via examples of increasing sophistication.

More material by Ch. Paulin: ▸ slides ▸ Habilitation memoir (in French)

Enumerated Types

Enumerated types are defined by the exhaustive description of their named inhabitants, which are all distinct:

```
Inductive color : Type :=
 |blue : color
 |green : color
 |magenta : color
 |yellow : color.
```

The terms `blue, green, magenta, yellow` are called the constructors of the inductive type `color`.

Enumerated Types

An arbitrary judgment $\Gamma \vdash T$: Type imposes a priori no special property on the nature, number or properties of the inhabitants of `T`. An inductively defined type does:

- We can program by case analysis on inhabitants of an inductive type;
- We can reason by case analysis on inhabitants of an inductive type;
- We can use the fact that two distinct labels refer to distinct inhabitants.

Enumerated Types

In practice:

- Program by (exhaustive) case analysis:

```
match x with
| blue => ... | green => ... | _ => ... end.
```

- Reason by (exhaustive) case analysis:

  using the tactic `case`

- Derive absurdity from a hypothesis of the form `h : blue=magenta`:

  using the tactic `discriminate`

Conversion Rule

Remember the type system is parametrized by an equivalence relation $\equiv$

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash t : B} \quad \text{if } A \equiv B$$

The conversion relation $\equiv$ also includes the reduction of case analysis:

$$(\text{match } x \text{ with } |c_1 \Rightarrow t_1| \ldots |c_n \Rightarrow t_n \text{ end})\ c_i \quad \equiv \quad t_i$$

Inductive Types

Inductive types can describe more than enumerations:

```
Inductive nat : Type := O : nat | S : nat -> nat.
```

- This type has two constructors: `O` and `S`.
- `O` is a constant of type `nat`.
- `S` is a constant of type `nat -> nat`..
- The inhabitants of `nat` are closed under function `S`.

Inductive Types

Inductive types can describe more than enumerations:

```
Inductive nat : Type := O : nat | S : nat -> nat.
```

- This type has two constructors: `O` and `S`.
- `O` is a constant of type `nat`.
- `S` is a constant of type `nat -> nat`..
- The inhabitants of `nat` are closed under function `S`.

Otherwise said, `nat` types the smallest collection of terms including `O` and closed under `S`.

Inductive Types

Just like in the case of enumerated types:

- We can program by case analysis on inhabitants of an inductive type;

  `match x with ... end`.

- We can reason by case analysis on inhabitants of an inductive type;

  using the `case` tactic.

- We can use the fact that two distinct head constructors imply two distinct inhabitants.

  using the `discriminate` tactic.

Inductive Types

Moreover:

- We can use the fact that constructors are injective functions.

  using the tactic `injection h` (with `h` an equality).
- We can program by (well-founded) recursion.

  using the `Fixpoint` (or `fix`) syntax.
- We can reason by (well-founded) induction.

  using the `elim` tactic.

Conversion rule

Remember the type system is parametrized by an equivalence relation $\equiv$

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash t : B} \quad \text{if } A \equiv B$$

The conversion relation $\equiv$ also includes the reduction of recursive definitions:

$$(\text{match } x \text{ with } |c_1 \Rightarrow t_1| \dots |c_n \Rightarrow t_n \text{ end})\ c_i \quad \equiv \quad t_i$$