Mechanized semantics, eight lecture

Coq in Coq:
Mechanizing the logic of a proof assistant

Xavier Leroy

2020-02-13

Collège de France, chair of software sciences

The approach followed in the “Mechanized semantics” course

“This course is an introduction to the formal semantics of programming languages and to their uses for building and validating programming tools and verification tools:

- type systems;
- program logics;
- static analyzers;
- compilers.

All definitions, properties and proofs are mechanized using the Coq proof assistant.

The approach followed in the "Mechanized semantics" course

"This course is an introduction to the formal semantics of programming languages and to their uses for building and validating programming tools and verification tools:

- type systems;
- program logics;
- static analyzers;
- compilers.

All definitions, properties and proofs are mechanized using the Coq proof assistant.

Coq: a tool like any other

Throughout the course, we used Coq as a programming tool and a verification tool.

Can we trust this tool?

Which formalisms could help validate this tool?

How a mechanized proof could be wrong?

(R. Pollack, *How to Believe a Machine-Checked Proof*, 1997)

Inadequacy: what is proved is not what you think.

How a mechanized proof could be wrong?

(R. Pollack, *How to Believe a Machine-Checked Proof*, 1997)

Inadequacy: what is proved is not what you think.

(Chris Casinghino, 2009-04-01)

```
Require Import Arith.
(* BEGIN PROOF OF FERMAT"S LAST THEOREM *)
Theorem fermat : forall n x y z,
  n > 2 ->
  x > 0 -> y > 0 -> z > 0 ->
  x ^ n + y ^ n <> z ^ n.
Proof.
  intros n x y z. trivial.
Qed.
(* END PROOF OF FERMAT"S LAST THEOREM *)
```

How a mechanized proof could be wrong?

(R. Pollack, *How to Believe a Machine-Checked Proof*, 1997)

Inadequacy: what is proved is not what you think.

`Admitted` proofs; axioms that are false or inconsistent.

Example: some classical logic axioms are inconsistent with the `-impredicative-set` option of Coq.

How a mechanized proof could be wrong?

(R. Pollack, *How to Believe a Machine-Checked Proof*, 1997)

Inadequacy: what is proved is not what you think.

`Admitted` proofs; axioms that are false or inconsistent.

A bug in a critical part of Coq's implementation

The implementation follows the de Bruijn architecture:

- a kernel that re-checks proof terms (critical);
- tactics that build these proof terms (not critical).

How a mechanized proof could be wrong?

(R. Pollack, *How to Believe a Machine-Checked Proof*, 1997)

Inadequacy: what is proved is not what you think.

`Admitted` proofs; axioms that are false or inconsistent.

A bug in a critical part of Coq's implementation

An inconsistency in the logic implemented by Coq.

Logical consistency

Logical consistency

A logic is consistent if it cannot deduce a paradox or an obvious absurdity, such as

$P \wedge \neg P$	for some paradox P (classical logic)
$\bot$	(written `False` in Coq) (intuitionistic logic)
$0 = 1$	(Peano arithmetic)
$\forall P.\ P$	(higher-order logic)

Equivalently: a logic is consistent if there exists at least one proposition that cannot be deduced.

(The *ex falso quod libet* principle: from absurdity, all propositions follow.)

Example: an intuitionnistic logic

$$\Gamma_1, P, \Gamma_2 \vdash P \quad \text{(Ax)}$$

$$\frac{\Gamma, P \vdash Q}{\Gamma \vdash P \Rightarrow Q}\ (\Rightarrow\text{I}) \qquad \frac{\Gamma \vdash P \Rightarrow Q \qquad \Gamma \vdash P}{\Gamma \vdash Q}\ (\Rightarrow\text{E, } \textit{modus ponens})$$

$$\frac{\Gamma \vdash P \qquad \Gamma \vdash Q}{\Gamma \vdash P \wedge Q}\ (\wedge\text{I}) \qquad \frac{\Gamma \vdash P \wedge Q}{\Gamma \vdash P}\ (\wedge\text{E}_1) \qquad \frac{\Gamma \vdash P \wedge Q}{\Gamma \vdash Q}\ (\wedge\text{E}_2)$$

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash P}\ (\bot\text{E, } \textit{quod libet})$$

Consistency = there exists one P such that we cannot derive $\vdash P$.

Gödel's second incompleteness theorem

Theorem (Gödel, 1931)

Let L be a consistent logic containing Peano arithmetic. The proposition "L is consistent" can be expressed in L but cannot be proved in L.

Corollary: a proof of consistency for a logic must be conducted in a "more powerful" logic.

PAT: Propositions As Types, Proofs As Terms

The Curry-Howard correspondence connects several logics (including that of Coq) with typed functional languages:

Langage typé	Logique
type	proposition
term	proof, “construction”
reduction	cut elimination

(See my 2018-2019 course.)

Propositions = types

Typed language	Logic
functions $\sigma \to \tau$	$P \Rightarrow Q$ implication
products $\sigma \times \tau$	$P \wedge Q$ conjunction
sums $\sigma + \tau$	$P \vee Q$ disjunction
type `unit` (1 constructor)	$\top$ triviality
type `empty` (0 constructors)	$\bot$ absurdity
polymorphism $\forall\alpha.\ \tau$	$\forall X.P$ for all
type abstraction $\exists\alpha.\ \tau$	$\exists X..P$ there exists

Deduction rules = typing rules

Simply-typed lambda-calculus

$$\Gamma_1, x : A, \Gamma_2 \vdash x : A$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.\, M : A \to B} \qquad \frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}$$

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B}{\Gamma \vdash (M, N) : A \times B} \qquad \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \pi_1\, M : A} \qquad \frac{\Gamma \vdash M : A \times B}{\Gamma \vdash \pi_2\, M : B}$$

$$\frac{\Gamma \vdash M : \texttt{empty}}{\Gamma \vdash \texttt{match}\ M\ \texttt{with end} : A}$$

Deduction rules = typing rules

Intuitionistic logic

$$\overline{\Gamma}_1, A, \overline{\Gamma}_2 \vdash A$$

$$\frac{\overline{\Gamma}, A \vdash B}{\overline{\Gamma} \vdash A \Rightarrow B} \qquad \frac{\overline{\Gamma} \vdash A \Rightarrow B \qquad \overline{\Gamma} \vdash A}{\overline{\Gamma} \vdash B}$$

$$\frac{\overline{\Gamma} \vdash A \qquad \overline{\Gamma} \vdash B}{\overline{\Gamma} \vdash A \wedge B} \qquad \frac{\overline{\Gamma} \vdash A \wedge B}{\overline{\Gamma} \vdash A} \qquad \frac{\overline{\Gamma} \vdash A \wedge B}{\overline{\Gamma} \vdash B}$$

$$\frac{\overline{\Gamma} \vdash \bot}{\overline{\Gamma} \vdash A}$$

$\overline{\Gamma}$ is Γ without variable names, e.g. $\overline{x : A, y : A} = A, A$.

Logical consistency = uninhabitated type

Typed language	Logic
Inhabitated type τ $(\exists M.\ \emptyset \vdash M : \tau)$	Provable proposition P
There exists one non-inhabitated type	The logic is consistent

A proof sketch that a type is not inhabited

(Extends the proof of soundness from lecture #7.)

Theorem (Canonical forms)

Let v be a value. If $\emptyset \vdash v : \sigma \to \tau$*, then v is an abstraction* $\lambda x.M$*.*
If $\emptyset \vdash v : \sigma \times \tau$*, then v is a pair* (v_1, v_2)*.*
It is impossible that $\emptyset \vdash v : \texttt{empty}$*.*

Theorem (Preservation)

If $\Gamma \vdash M : \tau$ *and* $M \to N$*, then* $\Gamma \vdash N : \tau$*.*

Theorem (Progress)

If $\emptyset \vdash M : \tau$*, either M is a value or M reduces.*

Theorem (Normalization)

Every typable term has a normal form:
if $\Gamma \vdash M : \tau$*, there exists N such that* $M \xrightarrow{*} N \not\to$

A proof sketch that a type is not inhabited

Corollary (Logical consistency)

The `empty` *type is not inhabited.*

Proof.

Assume there exists M such that $\emptyset \vdash M : \mathtt{empty}$.

By normalization we have N such that $M \xrightarrow{*} N \not\to$.

By preservation we have $\emptyset \vdash N : \mathtt{empty}$.

By progress we have that N is a value.

By canonical forms, we have a contradiction. $\square$

Divergence and inconsistency

Most language features that make a programming language Turing-complete make logics inconsistent.

Example: general recursion

`let rec f x = f x in f ()` has type τ for any τ.

As a proof principle, it is $(P \Rightarrow P) \Rightarrow P \ldots$

Example: algebraic types with negative occurrences

`Inductive t : Type := Lam: (t -> t) -> t`
encodes pure lambda-calculus, including divergence.

`Inductive P : Prop := Hyp: (P -> False) -> P`
is such that `P <-> (P -> False)`, from which `False` follows.

Proving normalization

Proving the normalization property

An approach introduced by Tait (1967) for simple types, extended to system *F* by Girard (1972). A special case of logical relation (Plotkin, 1973; Statman, 1985).

Define the sets $\mathit{RED}(\tau)$ by induction on type τ:

$$\begin{aligned} \mathit{RED}(\iota) &= \{M \mid M \text{ terminates, i.e. } \exists N, M \xrightarrow{*} N \not\to\} \\ \mathit{RED}(\sigma \to \tau) &= \{M \mid \forall N \in \mathit{RED}(\sigma),\ M\,N \in \mathit{RED}(\tau)\} \end{aligned}$$

(We write ι for any base type: `bool`, `nat`, etc)

Normalization for simple types

$$\begin{aligned} RED(\iota) &= \{M \mid M \text{ terminates, i.e. } \exists N, M \xrightarrow{*} N \not\to\} \\ RED(\sigma \to \tau) &= \{M \mid \forall N \in RED(\sigma),\ M\,N \in RED(\tau)\} \end{aligned}$$

We then show:

1. If $M \in RED(\tau)$ then M terminates.
2. If $\emptyset \vdash M : \tau$ then $M \in RED(\tau)$, or, more generally:

   If $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash M : \tau$ and $M_i \in RED(\tau_i)$ for every i, then $M\{x_1 \leftarrow M_1, \ldots, x_n \leftarrow M_n\} \in RED(\tau)$.

Extension to polymorphism

In a predicative type system such as ML, or Martin Löf type theory, ou Agda, we can take

$$\mathit{RED}(\forall\alpha.\tau) = \{M \mid \forall\sigma, M[\sigma] \in \mathit{RED}(\tau\{\alpha \leftarrow \sigma\})\}$$

This definition remains well founded because α can only be instantiated by types σ that are “smaller” than $\forall\alpha.\tau$.

In an impredicative system such as system F or Coq, α can be instantiated by any type, including $\forall\alpha.\tau$. Example:

$$\text{if} \quad \mathit{id} : \forall\alpha.\,\alpha \to \alpha \quad \text{then} \quad \mathit{id}\ [\forall\alpha.\,\alpha \to \alpha]\ \mathit{id} : \forall\alpha.\,\alpha \to \alpha$$

The definition of *RED* is therefore incorrect.

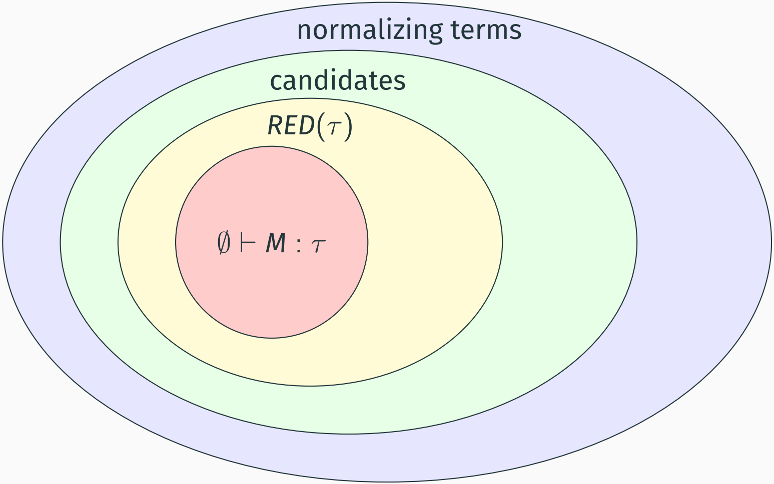
Reducibility candidates

Girard's idea: interpret type variables α not just by the sets *RED*(σ) for some types σ, but by a larger class of sets: the reducibility candidates (*candidats de réductibilité*).

A set *U* of terms is a reducibility candidate if

1. every $M \in U$ terminates;
2. *U* is closed under expansion: if $M \to M'$ and $M' \in U$ then $M \in U$
3. *U* is closed under certain reductions.
   (See Girard, *The blind spot*, vol. 1 ch. 6)

Reducibility candidates, visually



Normalization for system F

Reducibility: (Φ : type variable $\rightarrow$ candidate)

$$
\begin{aligned}
RED(\iota, \Phi) &= \{M \mid M \text{ terminates}\} \\
RED(\sigma \rightarrow \tau, \Phi) &= \{M \mid \forall N \in RED(\sigma, \Phi), M\ N \in RED(\tau, \Phi)\} \\
RED(\alpha, \Phi) &= \Phi(\alpha) \\
RED(\forall \alpha. \tau, \Phi) &= \{M \mid \forall \sigma, \forall U \in CAND(\sigma), M[\sigma] \in RED(\tau,\ \Phi + \alpha \mapsto U)\}
\end{aligned}
$$

We then prove:

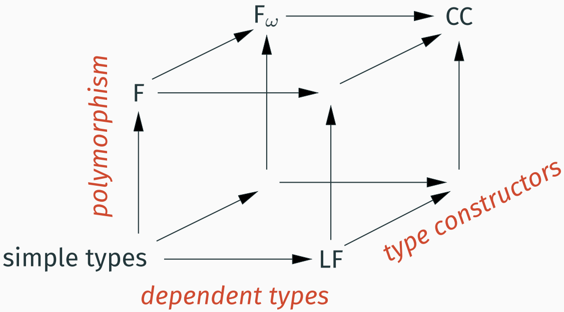
1. $RED(\tau, \Phi)$ is a reducibility candidate.
2. If $\emptyset \vdash M : \tau$ then $M \in RED(\tau, \Phi)$.

Formalizing and mechanizing Coq

From simple types to the Calculus of Constructions

Simple types	$\texttt{neg} : \texttt{bool} \to \texttt{bool}$	term $\mapsto$ term
+ polymorphism	$\texttt{id} : \forall\alpha.\, \alpha \to \alpha$	type $\mapsto$ term
+ type operators	$\texttt{list} : \texttt{Type} \to \texttt{Type}$	type $\mapsto$ type
+ dependent types	$\texttt{vec} : \texttt{nat} \to \texttt{Type}$	term $\mapsto$ type

= Calculus of Constructions



From the Calculus of Constructions to Coq

Calculus of Constructions

+ universe hierarchy	$0 : \mathtt{nat} : \mathtt{Type}_0 : \mathtt{Type}_1$
+ inductive types	$\mathtt{nat}, \mathtt{list}, \wedge, \vee, \exists$
+ coinductive types	$\mathtt{stream}, \mathtt{delay}$
+ universe cumulativity	
+ universe polymorphism	

$\approx$ Coq

A formalization of the Calculus of Constructions with universes

In the style of *Pure Type Systems*:

- No syntactic distinction between terms and types.
- A single λ for all the kinds of functions (term $\mapsto$ term, type $\mapsto$ term, type $\mapsto$ type, etc)
- A single Π representing function types and $\forall$ types.
- Universes to stratify into terms, types, kinds, etc.

Abstract syntax

$$
\begin{array}{lrll}
\text{Universe:} & U ::= & \mathtt{Prop} \mid \mathtt{Type}_i & \\
\text{Terms, types:} & A, B ::= & x & \text{variables} \\
 & & \mid \lambda x : A.\, B & \text{abstractions} \\
 & & \mid A\ B & \text{applications} \\
 & & \mid U & \text{universe name} \\
 & & \mid \Pi x : A.\, B & \text{dependent function type}
\end{array}
$$

Notation: $A \to B \overset{def}{=} \Pi x : A.\, B$ if x not free in B.

Typing rules

$$\frac{(U, U') \in \mathcal{A}}{\emptyset \vdash U : U'}\ \text{(ax)} \qquad \frac{\Gamma \vdash A : U}{\Gamma, x : A \vdash x : A}\ \text{(var)} \qquad \frac{\Gamma \vdash A : B \quad \Gamma \vdash C : U}{\Gamma, x : C \vdash A : B}\ \text{(wk)}$$

$$\frac{\Gamma \vdash A : U_1 \quad \Gamma, x : A \vdash B : U_2 \quad (U_1, U_2, U_3) \in \mathcal{R}}{\Gamma \vdash \Pi x : A.B : U_3}\ \text{(pi)}$$

$$\frac{\Gamma, x : A \vdash B : C \quad \Gamma \vdash \Pi x : A.C : U}{\Gamma \vdash \lambda x : A.\, B : \Pi x : A.\, C}\ \text{(abstr)}$$

$$\frac{\Gamma \vdash f : \Pi x : A.\, B \quad \Gamma \vdash a : A}{\Gamma \vdash f\ a : B\{x \leftarrow a\}}\ \text{(app)}$$

$$\frac{\Gamma \vdash A : B \quad \Gamma \vdash B' : U \quad B \overset{*}{\rightarrow}\overset{*}{\leftarrow} B'}{\Gamma \vdash A : B'}\ \text{(conv)}$$

The conversion rule: typing modulo reductions

$$\frac{\Gamma \vdash A : B \qquad \Gamma \vdash B' : U \qquad B \overset{*}{\rightarrow}\overset{*}{\leftarrow} B'}{\Gamma \vdash A : B'} \;\text{(conv)}$$

Types are identified up to reductions (computations).

Example 1: the type `dtype (Fun Bool Bool)` contains the same values as the type `bool` → `bool`, because these two types are equal modulo computation of the `dtype` function.

Example 2: the trivial proof for the proposition `4 = 4` is also a proof for the proposition `2 + 2 = 4`, because these two propositions are equal modulo computation of the + function.

The conversion rule: typing modulo reductions

$$\frac{\Gamma \vdash A : B \qquad \Gamma \vdash B' : U \qquad B \xrightarrow{*}\xleftarrow{*} B'}{\Gamma \vdash A : B'}\ \text{(conv)}$$

Types are identified up to reductions (computations).

- Enables new ways for programming and proving, such as “proofs by reflection”, where computation replaces logical deduction.
- A challenge for the metatheory: typing depends on computation.
- A challenge for the implementation of the type-checker: need an efficient evaluation mechanism during type-checking.

Universe management

$$\frac{(U, U') \in \mathcal{A}}{\emptyset \vdash U : U'}\ \text{(ax)} \qquad \frac{\Gamma \vdash A : U_1 \quad \Gamma, x : A \vdash B : U_2 \quad (U_1, U_2, U_3) \in \mathcal{R}}{\Gamma \vdash \Pi x : A.B : U_3}\ \text{(pi)}$$

The $\mathcal{A}$ relation determines which universe belong to which universe. In Coq:

$$\mathcal{A} = \{(\texttt{Prop}, \texttt{Type}_0), (\texttt{Type}_i, \texttt{Type}_{i+1})\}$$

The $\mathcal{R}$ relation determines the universe for $\Pi x : A.B$. In Coq:

$$\mathcal{R} = \{(U, \texttt{Prop}, \texttt{Prop}), (\texttt{Type}_i, \texttt{Type}_j, \texttt{Type}_{\max(i,j)})\}$$

Crucial for logical consistency! For instance, $\texttt{Type} : \texttt{Type}$ or Girard's system U can encode the Burali-Forti paradox…

Towards a Coq mechanization of the logic of Coq

B. Barras, *Coq en Coq*, 1996.
B. Barras et B. Werner, *Coq in Coq*, 1997.
A complete formalization of CC in Coq version 6.
Includes normalization, logical consistency, and proof + extraction of a type checker.

B. Barras, *Auto-validation d'un système de preuves avec familles inductives*, PhD, 1999.
Extension to CC + inductive types.
Normalization is not proved.

Towards a Coq mechanization of the logic of Coq

A. Charguéraud, *Locally nameless tutorial*, vers 2010.

`https://www.chargueraud.org/softs/ln/`

A simple formalization of CC + universes in Coq.

Stops just after the preservation theorem.

M. Sozeau, S. Boulier, Y. Forster, N. Tabareau, Th. Winterhalter, *Coq Coq Correct! Verification of type checking and erasure for Coq, in Coq*, 2020.

A formalization of PCUIC (Polymorphic Cumulative Calculus of Inductive Constructions). Normalization is admitted. Verifies all other parts of the metatheory, an efficient type-checker, and an extraction algorithm.

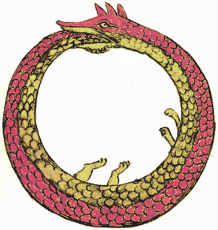
Towards an Agda mechanization of the logic of Agda

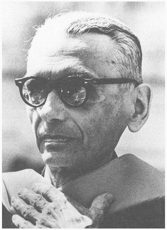
J. Chapman, *Type theory should eat itself*, 2008
Towards a normalization algorithm for MLTT, using intrinsically-typed syntax.

T. Altenkirch, A. Kaposi, *Type theory in type theory using quotient inductive types*, 2016
A specification of MLTT in intrinsically-typed syntax, using the quotient types from HoTT.

A. Abel, J. Öhman, A. Vezzosi, *Decidability of conversion for type theory in type theory*, 2018
An algorithm to test convertibility in the presence of dependent types (one universe), in Agda (MLTT + induction-recursion).

Proof assistants should eat themselves?





Can we mechanize a good fragment of the logic of a proof assistant in a barely bigger fragment?

References

References

Proofs of normalization:

- Simple types, in Coq: B. Pierce et al, *Software Foundations*, volume 2, chapter “Norm”.
- System F: J.-Y. Girard, *The blind spot: Lectures on logic.* European Mathematical Society, 2011, chapter 6.
- Calculus of Constructions: H. Geuvers, *A short and flexible proof of Strong Normalization for the Calculus of Constructions*, 1995.

Cut elimination and logical consistency:

- J.-Y. Girard, *The blind spot: Lectures on logic.* European Mathematical Society, 2011, chapters 4 and 5.