Formal Verification of Computer Programs

A Primer. (part 2)

Vadim Zaliva [1] Nika Pona [2]

[1] Carnegie Mellon University

[2] Digamma.ai

Abstract

In this presentation we will introduce Coq's basic functionality and explain how Coq works internally.

Table of contents

Coq mini-intro

Coq mini-intro

We did our proofs in a formal language called *Gallina*, a mechanized version of **Calculus of Inductive Constructions**, which is a very expressive type theory well studied in mathematical logic. We write the specifications, model our programs and do the proofs in this language.

We could do all of the above on paper, but it would quickly get out of hand. Moreover, we want to be sure that there are no mistakes in the proofs. So we use a tool called **proof assistant**: a program that checks that your proof is correct. It also provides an environment to make the construction of proofs easier.

In particular, we will talk about the Coq proof assistant:

`https://coq.inria.fr/`.

What Coq does?

In Coq you can:

- define functions and predicates
- state mathematical theorems and software specifications
- interactively develop formal proofs of theorems
- machine-check these proofs by a relatively small trusted kernel
- extract certified programs to languages like OCaml, Haskell or Scheme.

Inductive definitions

In Coq everything is either a *term* or a *type*. You can define basic inductive types using the `Inductive` command.

```
1  Inductive 𝔹 : Set :=
2    | true : 𝔹
3    | false : 𝔹.
```

The boolean type has two **constructors**: `true` and `false`.

```
1  Inductive ℕ : Set :=
2    | O : ℕ
3    | S : ℕ → ℕ.
```

Natural numbers are defined by two constructors as well: `O` (zero) and `S` (successor function). Any *term* of type $\mathbb{N}$ is constructed using these two. E.g., `S O` : $\mathbb{N}$, `S (S O)` : $\mathbb{N}$, `S (S (S O))` : $\mathbb{N}$ ...

Recursive definitions

You can define recursive functions using the `Fixpoint` command.

Since we know that by construction any term of type $\mathbb{N}$ is either `O` or `S n` for some $n : \mathbb{N}$, we can use *pattern-matching*.

```
1  Fixpoint plus (n m : ℕ) : ℕ :=
2  match n with
3  | O ⇒ m
4  | S p ⇒ S (plus p m)
5  end
```

Note: Coq only accepts definitions that terminate[1]. Here we are recursing on a direct subterm of n thus we are guaranteed to terminate and Coq is able to automatically ensure this. Sometimes you have to do a termination proof by hand.

[1] This limitation is needed to ensure consistency of the system, as well as decidability of type-checking.

Proofs

Finally, you can state and prove theorems about the objects you defined. Let + be notation for `plus`.

Theorem

2 + 3 = 5.

How would you prove this, if you were to justify each step of the proof?

```
1  Theorem plus_2_3 : (S (S O)) + (S (S (S O))) = (S (S (S (S (S O ))))).
2   Proof.
3    unfold plus. (* apply definition of plus *)
4    reflexivity. (* apply definition of equality *)
5   Qed.
```

In Coq you constructs a proof using so-called tactics. E.g., `simpl` is a tactic that performs basic application of definitions, `reflexivity` proves equality between two syntactically equal terms (modulo some reductions).

Demo

More Proofs

In Coq you have to justify every step of the proof and the `Qed` command only succeeds on correct proofs. But you don't have to do every proof from scratch, since there are extensive libraries covering lemmas about basic mathematics as well as several decision procedures that automatize proof search.

- Using SMT and SAT solvers: `https://smtcoq.github.io/`
- First-order decision procedures (CoqHammer): `https://github.com/lukaszcz/coqhammer`
- Tactics for solving arithmetic goals over ordered rings (`Micromega`)
- And much more, cf. `https://coq.inria.fr/opam/www/`

How does Coq work?

How does Coq work?

We can formalize programs, properties and proofs in the same language, as well as efficiently and reliably check proof correctness due to the so-called **Curry-Howard Isomorphism**.

Curry-Howard isomorphism is a correspondence between programs and terms on one hand and proofs and types on the other. All logical statements in Coq are typing judgments and thus checking the correctness of proofs amounts to type checking. Let's see what it means precisely on a small example.

Curry-Howard Isomorphism (CHI)

We said that *"Curry-Howard Isomorphism is a correspondence between programs and terms on one hand and proofs and types on the other"*. To make this precise, we need to specify what kind of programs we have on one side and what kind of proofs (that is, what kind of logic) we have on the other side.

Various **lambda calculi** were designed to describe programs.If you never heard of lambda calculi, think of functional programming languages such as **LISP**, **OCaml**, and **Haskell**.

The untyped lambda calculus

Consider the simplest example of lambda calculus[2]: the untyped lambda calculus. The terms are:

Var variables

Abs $(\lambda x.M)$, if x a variable and M is a term

App (MN), if M and N are terms

Think of $(\lambda x.M)$ as a function with argument x and body M and of (MN) as applying function M to the argument N[3].

[2] Or a rudimentary functional programming language.

[3] For more details see [Srensen and Urzyczyn, 1998].

The untyped lambda calculus (cont.)

In lambda calculus one could encode basic constructs such as natural numbers and lists, as well as functions on them and logical constants and operations. Using the *Y combinator* recursive functions could be encoded as well.

Examples of terms:

$id = \lambda x.x$ (identity function)

Church Booleans

We can encode boolean algebra using our calculus.

The *True* and *False* values:

$$\text{true} = \lambda t.\lambda f.t$$
$$\text{false} = \lambda t.\lambda f.f$$

And some basic boolean algebra operations:

$$\text{and} = \lambda b.\lambda c.(b\ c\ \text{false})$$
$$\text{or} = \lambda x.\lambda y.(x\ \text{true}\ (y\ \text{true}\ \text{false}))$$
$$\text{not} = \lambda b.(b\ \text{false}\ \text{true})$$

Church Numerals

Similarly, we can encode *natural numbers*:

$$
\begin{aligned}
c_0 &= \lambda s.\lambda z.z \\
c_1 &= \lambda s.\lambda z.sz \\
c_2 &= \lambda s.\lambda z.s(sz) \\
c_3 &= \lambda s.\lambda z.s(s(sz)) \\
&\textit{etc.}
\end{aligned}
$$

And some basic arithmetic operations:

$$
\begin{aligned}
\mathrm{succ} &= \lambda n.\lambda s.\lambda z.\ s\ (n\ s\ z) \\
\mathrm{plus} &= \lambda m.\lambda n.\lambda s.\lambda z.\ m\ s\ (n\ s\ z) \\
\mathrm{times} &= \lambda m.\lambda n.\ m\ (\mathrm{plus}\ n)\ c_0
\end{aligned}
$$

Simply typed lambda-calculus

We are interested in a **typed** lambda calculus. The types of **ST** are:

Var type variables

Arrow $\tau \to \sigma$, if τ and σ are types

Types are assigned to terms according to certain rules. Let Γ be a set of simply-typed terms (also called context).

$$\Gamma, x : \tau \vdash x : \tau \ (\textbf{Ax})$$

$$\frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash (\lambda x.M) : \sigma \to \tau} \ (\textbf{Abs})$$

$$\frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash (MN) : \tau} \ (\textbf{App})$$

Type derivation

Here is a simple derivation that shows that the term $\lambda x.x$ (identity) has type $\sigma \to \sigma$, with σ being any type of **ST**:

$$\frac{x : \sigma \vdash x : \sigma \ (\textbf{Ax})}{\vdash (\lambda x.x) : \sigma \to \sigma} \ (\textbf{Abs})$$

And if you apply identity to a term of type σ, the output is also of type σ:

$$\frac{\dfrac{y : \sigma, x : \sigma \vdash x : \sigma \ (\textbf{Ax})}{y : \sigma, \vdash (\lambda x.x) : \sigma \to \sigma} \ (\textbf{Abs}) \qquad y : \sigma \vdash y : \sigma \ (\textbf{Ax})}{y : \sigma \vdash ((\lambda x.x)y) : \sigma} \ (\textbf{App})$$

Back to CHI

We made one side of the Curry-Howard Correspondence more precise: the simply typed lambda calculus, which corresponds to some rudimenatry functional programming language. Now: what does it correspond to and how?

In general programs (or computations) will correspond to certain kind of logics. Namely, **intuitionistic logics**. These logics were created to formalize what is called constructive mathematics: that is, mathematics in which we are only interested in objects that can be effectively constructed.

Implicational propositional logic

Consider the simplest intuitionistic logic **Impl**. The formulae of **Impl** are the following:

Var propositional variables

Impl $\phi \to \psi$, for ϕ, ψ propositional formulae

Examples of formulae: $p \to p$, $p \to (q \to p)$.

Let Γ be a set of formulae. The theorems of **Impl** are proved according to the following rules of inference:

$$\Gamma, \phi \vdash \phi \ (\textbf{Ax})$$

$$\frac{\Gamma, \phi \vdash \psi}{\Gamma \vdash \phi \to \psi} \ (\to\text{-I})$$

$$\frac{\Gamma \vdash \phi \to \psi \qquad \Gamma \vdash \phi}{\Gamma \vdash \psi} \ (\to\text{-E})$$

Curry-Howard Isomorphism (Intuition)

You may notice a parallelism between the rules of inference of **Impl** and typing rules of simply typed lambda-calculus, this is the core of Curry-Howard Isomorphism.

$$\Gamma, \phi \vdash \phi \ (\textbf{Ax}) \qquad \Gamma, x : \tau \vdash x : \tau \ (\textbf{Ax})$$

$$\frac{\Gamma, \phi \vdash \psi}{\Gamma \vdash \phi \to \psi} \ (\to\text{-I}) \qquad \frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash (\lambda x.M) : \sigma \to \tau} \ (\textbf{Abs})$$

$$\frac{\Gamma \vdash \phi \to \psi \qquad \Gamma \vdash \phi}{\Gamma \vdash \psi} \ (\to\text{-E}) \qquad \frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash (MN) : \tau} \ (\textbf{App})$$

Curry-Howard Isomorphism between ST and Imp

Consider **ST** with type variables being propositional variables; then types are formulae of **Impl**. We can show that:

Theorem (CHI)

A simple lambda term M has a type τ in **ST** *if and only if τ is provable in* **Impl**.

Curry-Howard Isomorphism (conclusion)

You've seen the simplest type theory and propositional logic and how Curry-Howard Isomorphism works.

Now we can expand **ST** to include more terms and types (you can allow sum and product types, polymorphism, recursors - which allows you to formulate more functions) or alternatively expand the logic to be more expressible and complex (add other connectives, quantifiers, inductive definitions etc).

But the principle stays the same and you have Curry-Howard Isomorphism for very complex type systems, such as Calculus of Inductive Constructions, on which Coq is based.

Curry-Howard Isomorphism (terms)

logic	**lambda-calculus**
formula	type
proof	term
propositional variable	type variable
implication	function space
conjunction	product
disjunction	disjoint sum
absurdity	empty type
normalization	reduction
provability	type inhabitation

Proofs as functional programs

```
1  Theorem plus_n_O : (∀ n, n + O = n).
2  Proof.
3    induction n; simpl.
4    - reflexivity.
5    - rewrite IHn.
6    reflexivity.
7  Qed.
```

By Curry-Howard Isomorphism, the proof of this statement corresponds to constructing a term, that is, to writing a functional program.

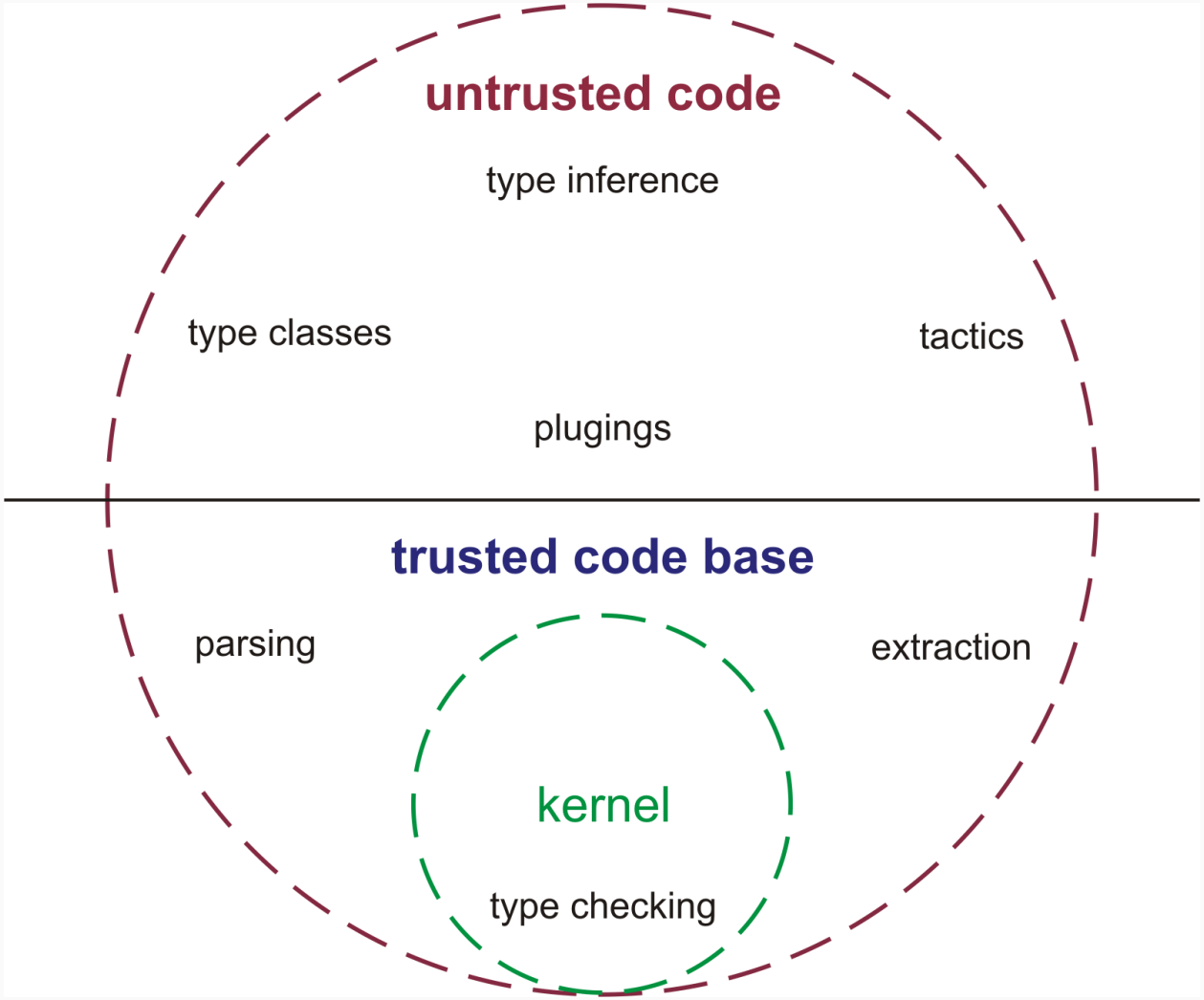
```
1  plus_n_O = λ n : ℕ ⇒
2              ℕ_ind (λ n0 : ℕ ⇒ n0 + O = n0) eq_refl
3              (λ (n0 : ℕ) (IHn : n0 + O = n0) ⇒
4               eq_ind_r (λ n1 : ℕ ⇒ S n1 = S n0) eq_refl IHn) n
5  : ∀ n : ℕ, n + O = n
```

where `nat_ind` is a term with type that corresponds to induction on natural numbers and `eq_refl` being a term of type `x = x`.

Coq's kernel

Since the logic we are dealing with is more complex than **Impl**, the functional programs corresponding to these types are also way more complex. However, checking whether a given term t has a given type σ is a decidable problem even for Calculus of Inductive Constructions. By Curry-Howard Isomorphism, this also yields a procedure for checking the correctness of proofs written in Coq's logic. This algorithm constitutes Coq's trusted **kernel**.

Coq's architecture



[Sozeau and Forster, 2019]

Coq applications

Coq is used to formalize and reason about many things. Some applications include:

- Properties of programming languages (e.g. the *CompCert* compiler certification project, or the *Bedrock* verified low-level programming library)
- Formalization of mathematics (e.g. the full formalization of the *Feit-Thompson theorem*, *Fundamental theorem of calculus*, *Four color theorem*), or *homotopy type theory*.
- Verification of distributed protocols (e.g. *blockchain consensus*)
- Verification of cryptography (e.g. encryption and digital signature schemes, zero-knowledge protocols, and hash functions)

Questions?

Examples from this presentation:

`https://github.com/digamma-ai/formal-verification-intro`

Contact:

- Vadim Zaliva, ✉ vzaliva@cmu.edu, @vzaliva
- Nika Pona, ✉ npona@digamma.ai

References i


Sozeau, M. and Forster, Y. (2019).
Coq Coq Codet! Towards a Verified Toolchain for Coq in MetaCoq.


Srensen, M. H. B. and Urzyczyn, P. (1998).
Lectures on the Curry-Howard Isomorphism.