Formalizing Coq Modules in the MetaCoq project

Bachelor's Thesis

Yee-Jian Tan

October 20, 2022

Try not to focus on the logical details, instead, try to focus on the simplicity behind the details.

— Leung Man Chun

Abstract

This is an abstract. I am formalizing Coq modules in MetaCoq, it is a missing piece of the already established MetaCoq project. In the second part, as a requirement for the mathematics requirement, I study some basic properties of lambda calculus, hopefully culminating in the decidability for conversion in real-world type systems such as the Martin-Lof Type Theory (MLTT).

Acknowledgements

I wouldn't come this far with the help of many people along the way. I would like to thank especially Nicolas Tabareau for having me in Nantes for the summer of 2022, which changed my life. Thank you professors Yang Yue and Martin Henz of the National University of Singapore for being my advisors for this project, even though it a rare one — externally proposed, cross-faculty, joint final year project.

I would also like to thank the people who inadvertently made this possible — Thomas Tan Chee Kun who gave me a foray into the field of Type Theory, then Rodolphe Lepigre who was an amazing mentor landing me into a summer of internship. Not forgetting the valuable friendship and advice I received in Gallinette team, for the people whom I interacted with: Pierre, Tomas, Yann, Arthur, Iwan, Kenji, Meven, Assia, Pierre-Marie, Hamza, Enzo, Yannick and everyone else that appeared during my stay in Gallinette. I truly loved the time there and can't wait to be back again.

Reading this paper

This paper is written with screen readers in mind. I will add links and references wherever possible, especially in any upcoming definitions. Click on symbols to jump to its definition, if I figure out how to make it work.

Contents

Part I: Formalization of Coq Modules

1. Types and Proof Assistants

1.1. What are proof assistants?

Proof assistants are a special kind of computer program whose job is to verify the correctness of a mathematical proof, that is, verify if a conclusion can be made by applying a fixed set of logical deduction steps on a set of fixed assumptions. Modern implementations of proof assistants usually utilizes theories such as Higher Order Logic (eg. Isabelle, HOL Light), or some form of Type Theory.

In order to have a computer program that can do such powerful logical reasoning to contain sophisticated mathematical results, the first question that arises is how to represent a theorem and what qualifies as a valid proof of a theorem. Fortunately, Type Theory gives a simple and direct relationship between Logic and Types.

1.2. Type Theory

Types were first invented by Bertrand Russell as a way to provide higher-order structures than sets to overcome his famous Russell's paradox. However, it was Alonzo's Church's Lambda Calculus which when used with types (Simply Typed Lambda Calculus) exhibited many desirable properties as not just a consistent formal system, also as a rewriting system that exhibits strong normalization. The arguably most important characteristic of Type Theories to qualify as a useful theory for the implementation of proof assistants, lies in the Curry-Howard Correspondence.

1.3. The Curry-Howard Correspondence

The Curry-Howard Correspondence establishes an one-to-one correspondence between two distinct disciplines — Logic and Types. It asserts that there is a one-to-one correspondences between Types and Propositions, and between Terms and Proofs. More precisely, it states that every proposition in intuitionistic propositional logic (constructive 0-th order logic) can be expressed as a Type in Simply Typed Lambda Calculus, and correspondingly, a term of a specific type is viewed as a proof for the proposition represented by the type.

Intuitioinistic Propositional Logic	Type Theory
Proposition	Type
Proof	Term

What are those exactly?

1.3.1. Intuitionistic Propositional Logic

Propositional logic, also known as *0-th order logic* is a logical system where propositions (sentences) are constructed with variables and logical connectors, such as $\rightarrow$ (implication), $\vee$ (disjuction), $\neg$ (negation), $\wedge$ (conjunction) and more. For example, propositions in Proprositional logic looks like:

1. $p \vee q$
2. $p \wedge \neg p$
3. $(p \rightarrow q) \rightarrow (\neg q \rightarrow \neg p)$

One can assign a valuation to each variable, either 1 or 0, which represents "True" or "False" informally. If there is some valuation that result in the propostion result in 1, then we say the proposition is *satisfiable*; if there is none, i.e. all valuations will result in the proposition being 0, we say the proposition is a *contradiction*; on the other hand, if any valuation will result in 1, the proposition is a *tautology*.

In particular, in the above example, the propositions are satisfiable, a contradiction and a tautology in that order.

Logical systems can be either intuitionistic or classical, based on one fact: whether they accept the *rule of the excluded middle*. Precisely, it says

Axiom 1.3.1 (Excluded Middle) For any proposition p, either p or $\neg p$ holds.

Much of modern mathematics is built on classical logic, which asserts the axiom of excluded middle. This leads to the existence of multitude of non-constructive proofs, which shows existence without giving a witness. A classical non-constructive proof using the law of excluded middle is as follows:

Theorem 1.3.1 *There exists irrational numbers a, b where a^b is rational.*

Proof. By the law of excluded middle, $\sqrt{2}^{\sqrt{2}}$ is either rational or irrational.

Suppose it is rational, then take $a = b = \sqrt{2}$. We have found such a, b.

Suppose it is irrational, then take $a = \sqrt{2}^{\sqrt{2}}$, $b = \sqrt{2}$. Then

$$a^b = \left(\sqrt{2}^{\sqrt{2}}\right)^{\sqrt{2}} = \sqrt{2}^{\sqrt{2}\times\sqrt{2}} = 2$$

is rational. □

Even with the proof, we still don't have a concrete pair of irrational numbers (a, b) fulfilling the property, but the conclusion is declared to be logically sound.

Intuitionistic logic, on the other hand, rejects this rule, leading to a logical system where to prove something exists, it requires an algorithm for the explicit construction of such object.

1.3.2. Simply Typed Lambda Calculus

The Simply Typed Lambda Calculus (STLC) is a simple construction:

1.3.3. The Correspondence

This correspondence is two-way: in particular, we can encode a proposition as a type, and to write a proof is equivalent to finding a term that inhabit that particular type. This reduced the problem of proof-checking to the problem of finding the type of a term, or under the Curry-Howard correspondence, whether a proof truly proves a given proposition.

This fundamental correspondence is elegant, but only limited to propositional logic. More complex type systems, such as the Calculus of Constructions by Theiry Coquand, extend the Curry-Howard correspondence to higher-order logic, finally allowing the encoding of most mathematical truth and specifying proofs for theories in those systems.

1.4. The Coq Proof Assistant

The Coq Proof Assistant (or Coq for short) is a proof assistant based on the Calculus of Inductive Constructions, which added co-inductive types into the Calculus of Constructions.

2. Introduction to the MetaCoq Project

MetaCoq is a project to formalize the core calculus, PCUIC, in Coq, and become a platform to write tools that can manipulate Coq terms. The effort was complete for a large part of the core language of Coq, with a few missing pieces:

- Eta
- Template Polymorphism
- SProps
- Modules

I will be tackling the last.

3. The Module System in Coq

Module systems are a feature of the ML family of languages; it allows for massive abstraction and the reuse of code. In particular, Coq also has a module system that is influenced by ML modules, first implemented by Jacek Chrząszcz in 2003, then modified by Elie Soubrian in 2010.

3.1. A specification of Coq Modules

3.1.1. Conversion of Coq Terms

To understand Coq Modules, we need to first understand the basic structure of Coq. The core object in the language of Coq are terms. Terms of a type correspond to a proof for a theorem as in the Curry-Howard correspondence. The syntax and semantics of Coq terms are as explained by the syntax [insert link] and conversion rules [insert link] of Coq. The evaluation of Coq terms are done under a Global Environment Σ containing definitions, and a local context Γ containing assumptions. Evaluation in Coq is known as conversion, the reflexive, transitive closure of the various reduction rules that is defined, including the famous β-reduction (function application).

The conversion relation is then defined with these parameters, from which we can conclude nice properties on conversion, such as strong normalization, confluence and decidability.

3.1.2. Modules as second-class objects

However, Coq modules are not first class objects of the language and do not participate in conversion themselves; i.e. there is no notion to "reduce" a module. Plain modules in Coq can be treated as a named container of constant and inductive definitions, including possibly nested modules; namespaced by a dot-separated string called a "path". This abstraction allows users to reuse definitions, essentially importing another "global environment" into the current one.

To further expand this possibility, module functors exist to be interfaces which users can provide definitions for, by supplying a module definition. Functors are therefore opaque second-class objects which is only useful when a module is generated.

In this chapter, we formalize the syntax and semantics of the current implementation of Coq and implement in the level of TemplateCoq. Section 3.3 describes the implementation of plain modules without functors.

3.2. Related Works

Elie Soubrian's PhD Thesis on Coq Modules is the latest reference on Coq Modules specifically. Since ML modules are the source of inspiration for Coq's Modules, it is also worthwhile to study Derek Dreyer's PhD Thesis on "Understanding and Evolving the ML Module System". Recently, F-ing modules by Dreyer et. al. provides an implementation of the ML module system in plain F_ω, a strict subset of CIC and hence PCUIC used in modern Coq.

3.3. Plain Modules

Plain modules are just saving declarations and definitions in a structure, in the global context. Its contents are not modified during conversion/reduction.

In Coq, modules are second-class objects; in other words, a module is not a term. Its definition is stored and referred to by a canonical kername. We say the implementation of such a module is correct if the metatheory of the original system are unchanged and remains correct; that is the proofs go through when terms can be defined within modules. Since the MetaCoq project has proven various nice properties about conversion in Coq, our project on plain modules is two-fold:

1. Ensure the correctness of the static semantics of Coq Modules (well-typedness).
2. Define the behaviour of access of definitions within Modules.

Once these two are done, we can be sure that a Coq program with Modules has all its terms well-defined (by (1)) and enjoys the nice properties of conversion, since the additional terms defined in Modules fulfill (2). This follows as our definition of Modules on the TemplateCoq level, is eventually elaborated down into the PCUIC calculus the idea of modules and aliasing do not exist anymore, they are flattened into the corresponding global environment.

Part II: Decidability of Conversion in MLTT

The Untyped Lambda Calculus | 4.

Here, I present some properties of the untyped lambda calculus, as part of the study of Type Systems with Professor Yang Yue. We follow the text "Proof and Types" by Girard. Here, I present some fundamental results.

4.1. The Church-Rosser Confluence

We study the paper by Takahashi's successors on a proof for Church-Rosser theorem. It states that:

Theorem 4.1.1 (Church-Rosser) *If* $M \to_\beta M_1$, $M \to_\beta M_2$, *then there exists a term* N *such that* $M_1 \to_\beta N$ *and* $M_2 \to_\beta N$.

This property asserts the uniqueness of a normal form of a rewriting system, if there exists one. This is especially important for systems that are known to normalize; in particular, the Simply Typed Lambda Calculus, the Martin Lof Type Theory, and Calculus of Constructions are all known to be strongly normalizing. Together with the decidability of conversion, we have a viable theory for the implementation of proof assistants.

The crucial step of the proof is to define a notion of parallel reduction, and the key lemma is

Lemma 4.1.2 *If* $M \to_{\beta n} N$, *then* $M \to_\beta M^{n*}$.

This provides a candidate of confluence just based on the original term M, instead of the intermediate steps during $M \to_\beta M_1, M_2$ or even M_1, M_2 itself.

(Hopefully I have a Coq proof.)

4.2. Cut elimination

Appendix

Specification of Coq Modules A.

A.1. Typing Rules

Should I include a link, or type them all here? Only the relevant ones?