Formalizing Coq Modules in the MetaCoq project

XFC4101 CA Report

Yee-Jian Tan

October 22, 2022

Contents

1. Introduction

The Coq Proof Assistant is one of the prominent proof assistants, with applications from the Homotopy Type Theory (HoTT)'s univalent foundation of mathematics, to the Iris proof mode for Concurrent Separation Logic, or the recent receipient of the 2021 ACM Software System Award — the CompCert compiler. However, even though the theory behind Coq, the Calculus of Inductive Constructions is known be consistent and strongly normalizing (hence proof-writing is decidable), but the OCaml implementation of Coq is known to have an average of one critical bug per year which allows one to prove False statements in Coq.

The MetaCoq project is therefore started by the Galinette Team in INRIA, to "formalize Coq in Coq" and acts as a platform to interact with Coq's terms directly, in a verified manner. This also reduces the trust in the implementation of Coq to the correctness of the theories underlying Coq. In 2020, the core language of Coq, minus a few features are already successfully verified in Coq. However, there are still a few missing pieces not yet verified, among them the Module system of Coq.

The Module system, although not part of the core calculus of Coq, is an important feature for Coq developers to develop modularly, providing massive abstraction and a suitable interface for reusing definitions and theorems.

In this project, I aim to formalize the Module system of Coq using the MetaCoq project framework, by first understanding the implementation of Modules, and then providing a specification of the implementation of Coq modules, finally writing proofs to show the correctness of the current implementation. In particular, I will focus on the formalization of non-parametrized, plain modules.

In order to tackle this project, it is important to understand the theory behind the implementation of Coq; more explicitly, how Type Theory helps in theorem proving. This report is therefore structured from the bottom up, from some results of Type Theory which I study in under the supervision of Professor Yang Yue; to the formalization of Modules jointly supervised by Professor Nicolas Tabareau and Professor Martin Henz.

The first section will give an overview of Type Theory and Theorem Proving. The second section will cover some important results from Type Theory that I studied. Following that, I will give an introduction to the language of Coq and the core problem of formalizing Coq Modules, before describing some related works and finally a specification for Coq Modules which I will implement.

2. Types and Proof Assistants

2.1. 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.

2.2. 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.

2.3. 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?

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 2.1 (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 2.1 *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.

Simply Typed Lambda Calculus

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

2.4. 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.

3. Introduction to the MetaCoq Project

3.1. 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.

3.2. 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.

4. 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.

4.1. A specification of Coq Modules

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, [1] conversion (including reduction and expansion) [2] and typing [3] respectively. 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).

1: Coq: Essential Vocabulary
2: Coq: Conversion
3: Coq: Typing

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.

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 4.3 describes the implementation of plain modules without functors.

4.2. Related Works

Jacek Chrzaszcz's article in TPHOLs 2003 [1] explains the motivation and choices for the implementation of modules as a second class object and its interaction with terms as described above. Elie Soubiran's PhD Thesis on Coq Modules [2] describes an extension of the Module system which only some features are implemented. The ModuleSystem Wiki Page on Coq's official Github repository contains valuable information on the usage and design decision of the Module system of Coq, including a list of open issues with Modules. In general, the issues do not compromise the correctness of the type system implemented by Coq; instead, they should be viewed as possible areas of improvements for Coq users.

[1]: Chrzszcz (2003), *Implementing Modules in the Coq System*

[2]: Soubiran (2010), *Développement modulaire de théories et gestion de l'espace de nom pour l'assistant de preuve Coq.*

Other slightly useful references include papers that explain modules in the ML family of languages, specifically OCaml and to a certain extent, Standard ML. On this note, Derek Dreyer wrote his PhD thesis [3] on understanding and extending ML modules, and subsequently a on implementing ML modules in its most desirable form, applicative and first-order as a subset of a small type system, F_ω[4]. Other notable implementations include that of CakeML [5], a verified ML language and Standard ML modules by MacQueen[6]. However, since the type system of Coq is much stronger and sophisticated compared to ML languages, the implementations also vary wildly and one can only refer to them for inspirations.

[3]: Dreyer et al. (2005), *Understanding and Evolving the Ml Module System*

[4]: Rossberg et al. (2010), *F-Ing Modules*

[5]: Tan et al. (2015), *A Verified Type System for CakeML*

[6]: MacQueen (1984), *Modules for standard ML*

4.3. 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.

Appendix

A. Specification of Coq Modules

A.1. Typing Rules

I will be typesetting the rules here, taken from Coq: Typing Modules.

B. The Untyped Lambda Calculus

Here, I present some properties of the untyped lambda calculus I studied. We follow the text "Proof and Types" by Girard. Here, I present some fundamental results.

B.1. The Church-Rosser Confluence

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

Theorem B.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 B.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.)

B.2. Cut elimination

Sequent Calculus is a logical system capable of doing deduction. One of the axioms, the Cut rule, is actually eliminable. This can be proven by induction.

(I will insert the proof here).

Bibliography

Here are the references in citation order.

[1] Jacek Chrzszcz. 'Implementing Modules in the Coq System'. In: (Dec. 2003) (cited on page 6).

[2] Elie Soubiran. 'Développement modulaire de théories et gestion de l'espace de nom pour l'assistant de preuve Coq.' Theses. Ecole Polytechnique X, Sept. 2010 (cited on page 6).

[3] Derek Dreyer, Robert Harper, and Karl Crary. 'Understanding and Evolving the Ml Module System'. AAI3166274. PhD thesis. USA, 2005 (cited on page 7).

[4] Andreas Rossberg, Claudio V. Russo, and Derek Dreyer. 'F-Ing Modules'. In: *Proceedings of the 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation*. TLDI '10. Madrid, Spain: Association for Computing Machinery, 2010, pp. 89–102. DOI: 10.1145/1708016.1708028 (cited on page 7).

[5] Yong Kiam Tan, Scott Owens, and Ramana Kumar. 'A Verified Type System for CakeML'. In: *Proceedings of the 27th Symposium on the Implementation and Application of Functional Programming Languages*. IFL '15. Koblenz, Germany: Association for Computing Machinery, 2015. DOI: 10.1145/2897336.2897344 (cited on page 7).

[6] David MacQueen. 'Modules for standard ML'. In: *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*. 1984, pp. 198–207 (cited on page 7).