Formalizing Coq Modules in the MetaCoq project

XFC4101 Final Report

Yee-Jian Tan

March 22, 2023

For Rodolphe and Nicolas, who changed my life

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.

Contents

1 Introduction

The Coq Proof Assistant is one of the most prominent proof assistants, with applications ranging from formalizing a mathematical theory with the Homotopy Type Theory (HoTT)'s univalent foundation of mathematics [**DBLP:journals/corr/BauerGLSSS16**], to a framework to verify a computational theory via the Iris proof mode for Concurrent Separation Logic [**jung2018iris**], to a practical large-scale verification project: the recent recipient of the 2021 ACM Software System Award — the CompCert compiler [1]. Although the Coq Proof Assistant is the frameworking making these projects possible, one might ask: "Who watches the watchers?" [2] Even though the theory behind Coq, the Polymorphic, Cumulative Calculus of Inductive Constructions is known be consistent and strongly normalizing (hence proof-checking 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.

DBLP:journals/corr/BauerGLSSS16

jung2018iris

1: https://awards.acm.org/software-system

2: From latin: *Quis Custodiet ipsos custodes?*

The MetaCoq project [3] is therefore started by the Gallinette Team in INRIA, to "formalize Coq in Coq" and acts as a platform to interact with Coq's terms directly, in a verified manner; an immediate application is to verify the correctness of the implementation of Coq. This also reduces the trust in the implementation of Coq to the correctness of the theories underlying Coq, moving from a "trusted code base" to a "trusted theory base". In 2020, the core language of Coq, minus a few features are already successfully verified in Coq [**coqcoqcorrect**]. 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 in a modular fashion, providing massive abstraction and a suitable interface for reusing definitions and theorems.

3: https://metacoq.github.io

coqcoqcorrect

In this project, I aim to extend the MetaCoq formalization of Coq by formalizing the Module system of Coq in 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. I study under the supervision of Professor Yang Yue some results of Type Theory; alongside the formalization of Modules jointly supervised by Professor Nicolas Tabareau and Professor Martin Henz.

The first section of this report will give a brief mathematical overview of the relationship between Type Theory and Theorem Proving. Following that, I will give an introduction to the MetaCoq project, the language of Coq and Coq Modules, before describing some related works and finally a specification for Coq Modules which I will implement.

Previous Works 2

Since Coq is influenced by the ML family of languages, the specification for modules in Coq is very similar to that of OCaml, and Standard ML (SML). in this chapter, we review previous implementation of Coq modules, as well as relevant module systems in other ML languages that this project can refer to.

2.1 Coq Module Implementations

The earliest exploration of adding a module system to Pure Type System (PTS) [1], a generalized type system, was done by Judicaël Courant. He designed the *MC* Module Calculus system that includes modules, signatures, and functors for PTS and proved that it is a conservative extension. [2] Modules in *MC* are anonymous, second-class objects with a specific set of reduction rules, and Courant has proven the resulting system to have decidable type inference and the principal type property.

1: PTS can be seen as a generalization of Barendregt's Lambda Cube, by defining type systems using a triple $(\mathscr{S}, \mathscr{A}, \mathscr{R})$ represnting sorts, axioms and rules of the type system respectively.

2: In proof theory, a conservative extension of a formal language is one that cannot prove statements that are not already provable in the base language.

Building on the idea of Courant, Jacek Chrąszcz designed the earliest implementation of a module system in Coq in his PhD thesis, and was released with Coq version 7.4. The module system by Chrąszcz was a subset of that of Courant with some changes. Similar to *MC*, modules, signatures and functors are implemented together with specialized reduction rules, but he argued that an anonymous module system does not work well with the definition and rewriting system of Coq. Therefore, all modules are named in Chraszcz's implementation, and the expression of modules in Coq is restricted only to module paths. The core of Chrąszcz's PhD thesis is the conservativity proof about the module system extension over Coq, together with the syntax, typing rules, and rewriting rules of the module system in Coq.

The most recent work on Coq's module system is Elie Soubrian's PhD thesis. He proposed many improvements on the system, among which, a unified notion of structure for modules and signatures, and the availability of higher-order functors are already in today's OCaml implementation of Coq. However, other features mentioned in the thesis, such as applicative functors, and a notion of namespace that allow a separate, dynamic naming scope for modules, are not yet implemented in the module system in Coq today.

2.2 Modules in ML Dialects

The two main dialects ML today, OCaml and Standard ML have interestingly different semantics for modules. Modules are by default applicative (generative functors are possible) in OCaml while generative in Standard ML.

The module system of SML has evolved over the years, from the earliest account by MacQueen[**macqueen1984modules**] and Harper et. al. in

macqueen1984modules

terms of "strong sum" types, to the "transparent" approach by Lillibridge. Harper and Lillibridge also developed first-order modules in SML eventually using standard notions from type theory. Meanwhile, Leroy made progress on applicative functors, modular module systems, and mutually recursive modules in SML, then OCaml.

On this note, Derek Dreyer wrote his PhD thesis [**dreyerphd**] on understanding and extending ML modules, and subsequently on implementing ML modules in its most desirable form, applicative and first-order as a subset of a small type system, F_ω[**f-ing**]. Another related project is CakeML [**cakeml**], a verified subset of the SML language, but unfortunately does not include the verification of the module system.

dreyerphd

f-ing

cakeml

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.

2.3 Knowledge Management in Proof Assistants