B.Comp. & B.Sc. Dissertation

Formalizing Coq Modules in the MetaCoq Project

By

Tan Yee Jian

A joint thesis of

Department of Computer Science Department of Mathematics

School of Computing Faculty of Science

National University of Singapore

2022/2023

B.Comp. & B.Sc. Dissertation

Formalizing Coq Modules in the MetaCoq Project

By

Tan Yee Jian

A joint thesis of

Department of Computer Science Department of Mathematics

School of Computing Faculty of Science

National University of Singapore

2022/2023

Project No.	: H0411180
Advisor (SoC)	: Assoc. Prof. Martin Henz
Advisor (FoS)	: Prof. Yang Yue
Advisor (INRIA)	: Prof. Nicolas Tabareau
Main Evaluator	: Assoc. Prof. Seth Lewis Gilbert

Deliverables: Report (1 Volume)

Abstract

The MetaCoq project provides a verified implementation of a huge subset of Coq, however, does not include several features, such as Modules. This project aims to formalize and implement a subset of non-parametrized modules in the MetaCoq project and show that this implementation enjoys nice type-theoretic properties such as confluence, and principal typing. This project provides two different approaches to formalizing modules and global environments in Coq, and summarizes some proof-theoretic skills that are useful in constructing formal proofs.

Subject Descriptors:

Theory of computation – Logic – Type Theory
Security and privacy – Formal methods and theory of security – Logic and verification

Contents

Chapter 1

Introduction

1.1 Why Theorem Provers?

Deductive, logical proofs have long been a way for humans to deduce mathematical facts. However, after several centuries of development, mathematics has grown so huge that mathematicians have no choice but to depend on previous results to develop new theorems. This requires a form of trust in the correctness of pen-and-paper proofs of previous results, which in turn might be dependent on previously proved, or even facts dismissed as ”obvious”. Theoretically, mathematicians know that every theorem can be boiled down to merely logical consequences from pre-determined axioms, but as more and more recent proofs are found to be erroneous (Voevodsky 2018) to different extents, the field is now in need of mechanical verification.

In the realm of computer programming, another problem arises, which similarly cries for a similar mechanical verification - the correctness of computer programs. As Dijkstra said, “Program testing can be used to show the presence of bugs, but never to show their absence!” (Dijkstra et al. 1970) To ensure our programs are true to our intentions, a formal verification or proof is needed. Being the tool for automation itself, the act of automatically checking the correctness of programs and the lack of bugs is, in turn, another task solvable by automation, by programming.

Therefore, proof assistants (or sometimes known as theorem provers) are thus born.

At its core, a proof assistant takes a mathematical claim and checks the user's proof against a logical system with axioms and deductive rules. However, there are two approaches to writing such proofs - one can have it automated, thus the family of Automated Theorem Provers (ATP) and the family of Interactive Theorem Provers (ITP). Coq is an ITP that implements the Polymorphic Cumulative Calculus of Inductive Constructions (PCUIC) (Timany and Sozeau 2017), which can prove facts in higher-order logic. It is among the earliest ITPs of its time and has been used to complete large proofs such as the Four Color Theorem (Gonthier 2008) and the Feit-Thompson Theorem (Gonthier et al. 2013), and was awarded the 2013 ACM Software System Award (Herbelin 2014). In 2021, the CompCert C compiler, an industrial-strength compiler verified in Coq has been awarded the ACM Software award (*Software system award goes to three for pioneering open source initiatives* n.d.), recognizing the potential of ITPs in the real of verifying software correctness as well.

1.2 Why MetaCoq?

As the ancient saying goes: who watches the watchers[1]? While proof assistants check for the correctness of mathematical proofs and computer programs (which are equivalent as per the Curry-Howard Correspondence (Howard 1980; Curry 1934)), who checks the correctness of the proof assistants, which are in turn programs themselves? Indeed, although the theory behind Coq, PCUIC is known to be "correct", such as being strongly normalizing (and hence decidable in type checking); however, the current implementation of Coq in OCaml is known to be not entirely bug-free – at least one critical bug that threatens the correctness of Coq is found per year, meaning that it is possible to prove *False* in Coq (*Consistency Bugs in Coq* 2023)!

One way to remedy this situation is to have a verified implementation for Coq itself. The MetaCoq project is in general a metaprogramming platform for Coq, where users can reify Coq terms in Coq and manipulate them, and thus giving rise to a perfect environment to argue about various properties of Coq terms, specifically all the correctness

[1] Originally in latin, *Quis custodiet ipsos custodes?*, literally "Who will guard the guards themselves?

properties of the underlying PCUIC system (Sozeau, Anand, et al. 2020). Then, a verified implementation of Coq can be implemented in the MetaCoq project, and after passing through a series of verified components, give rise to a compiled, verified version of Coq. Of course, without the low-level optimizations in OCaml, the MetaCoq implementation is slower than the original Coq, but it can be an implementation that one runs once every month, for the highest level of security and assurance.

The careful reader here might have already noticed a problem in verifying a Coq implementation in Coq - Gödel's Second Incompleteness stands in the way of proving a formal system's consistency with itself. Noticing this problem, the approach by the MetaCoq implementation is to assume the consistency of PCUIC as an axiom, moving the trust from the correctness of the OCaml code to the correctness of the theory of PCUIC itself.

1.3 Why Modules?

The verified implementation of Coq under the MetaCoq project has been done for a substantial, working subset of Coq, excluding the Module system. Why would implementing modules be a worthwhile effort, or rather, why is the module system important for proof assistants like Coq, or programming in general?

Organizing and generalizing human knowledge is one of the most natural things humans do when trying to understand this world. From the distinction of different "subjects" ranging from history to engineering, or in the case of mathematics, from algebra to statistics. The use of modules is for a similar purpose in proof assistants: to categorize, package and organize knowledge, and is deeply related to the idea of modular programming and "programming in-the-large". Therefore, even though adding the idea of modules to Coq or any other formal logic systems should not increase the "power" of the system (in fact, when extending a formal system with modules, one should actively ensure that the extension is conservative), but it is almost essential for the users and developers to better organize their proofs.

The MetaCoq project has successfully implemented and verified a large subset of the Coq language in 2020, with the exception with a few features such as η-expansion, Mod-

ule System, Template Polymorphism and Proof-Irrelevant Propositions(Sozeau, Anand, et al. 2020). Therefore, when I interned at the birthplace of the MetaCoq project, the *Gallinette* team in Nantes, France, I chose to work on implementing Modules in the MetaCoq project.

1.4 Contributions

In this project, I have implemented a non-parametrized module system in the MetaCoq project and verified various properties of modules and its interaction with the global environment of Coq, the typing of Coq terms, η-expansion and more. Furthermore, I have written a translation between the language with (non-parametrized) modules to PCUIC without modules, hence the use of non-parametrized modules is a subset and hence a conservative extension of the original language.

In this report, I detail my implementation of non-parametrized modules and the considerations behind the design choices made. I will also explain the related correctness properties related to the modules which I proved. As a learning project, I will also detail Coq-specific skills that I learnt during the project.

The rest of the report is structured as follows: Chapter 2 will start with a review of previous related works about the implementation of Modules in Coq and relevant systems. Then, Chapter 3 will introduce the MetaCoq project and explain the syntax and semantics of Coq modules, before describing the implementation of Modules and the verification of their properties in Chapter 4, the core of this project. After that, Chapter 5 will describe a second possible implementation of Modules in Coq, called the Modular Environment rewrite that solves various problems that surfaced from the initial implementation. Finally, Chapter 6 will document the skill acquired in writing formal proofs in Coq during the implementation of the project. This report will end with possible future work in Chapter 7.

Chapter 2

Previous Works

Since Coq is influenced by the ML family of languages, the specification for modules in Coq is very similar to that of OCaml. in this chapter, we review the previous implementations of Coq modules, as well as relevant module systems in other ML languages that this project can refer to. However, since the type system of Coq is much stronger and more sophisticated compared to ML languages, the implementations also vary wildly and one can only refer to them as inspirations.

2.1 Coq Module Implementations

The earliest exploration of adding a module system to a Pure Type System (PTS) , a generalized type system subsuming Coq, was done by Judicaël Courant in his Ph.D. thesis (Courant 1997). He designed the *MC* Module Calculus system which included modules, signatures, and functors for PTS and proved that the extension is conservative[1]. 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.

Building on the idea of Courant, Jacek Chrąszcz designed the earliest implementation of a module system in Coq in his Ph.D. thesis (Chrząszcz 2004), released with Coq version

[1]In proof theory, an extension of a formal language is conservative if it cannot prove statements that are not already provable in the base language.

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 and the expression for modules in Coq is restricted only to module paths. The core of Chrąszcz's thesis is the conservativity proof of the module system extension over Coq, together with its associated syntax, typing rules, and rewriting rules.

The most recent work on Coq's module system is Elie Soubrian's Ph.D. thesis (Soubiran 2010). He proposed many improvements to the system, among which a few are already implemented, such as unifying modules and signatures with structures, and higher-order functors. However, other features mentioned in the thesis such as applicative functors, and namespaces allowing separate, dynamic naming scopes for modules, are not yet implemented in the module system of Coq.

2.2 Modules in ML Dialects

Interestingly, the two main ML dialects today, OCaml and SML have different approaches and semantics for modules. Modules are by default applicative (generative functors are possible) in OCaml while generative in SML.

The module system of SML has evolved over the years, from the earliest account by MacQueen (MacQueen 1984) and Harper et. al. in terms of "strong sum" types, to the "transparent" approach by Lillibridge (Harper and Lillibridge 1994). 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 (Leroy 1994; Leroy 1995; Leroy 2000). On this note, Derek Dreyer wrote his Ph.D. thesis (Dreyer, Harper, and Crary 2005) on understanding and extending ML modules, and subsequently implementing ML modules in its most desirable form, applicative and first-order as a subset of a relatively small type system, F_ω (Rossberg, Russo, and Dreyer 2010). Another related project is CakeML (Tan, Owens, and Kumar 2015) which verifies a subset of SML, however, modules are unverified.

Chapter 3

MetaCoq, Coq, and the Module System

This chapter aims to provide sufficient background knowledge for explaining the implementations of Chapters 4 and 5. We will explain the following in order:

1. First, the organization of the MetaCoq project as a context of where the implementation of modules will occur.
2. Then, the abstract syntax of the Coq Module system, some information on the semantics of Coq, and the scope of the implementation of this project;
3. Finally in simple terms, the semantics of the subset of Module System that this project will implement, along with a brief list of some relevant properties to verify.

Note that this chapter does not intend to give a full description of modules in Coq, but to only explain the subset that is relevant to this project; the actual module system of Coq is well extended with the other extra-logical parts of Coq such as pretty printing, notation, and hint databases, which are beyond the scope of this project. For a more precise definition of modules and related structures, please refer to Coq: Modules.

3.1 Structure of the MetaCoq Project

The MetaCoq project is a project that provides a toolchain for verified metaprogramming in Coq (Sozeau, Anand, et al. 2020). With the ability to manipulate Coq terms in Coq, the MetaCoq is a perfect avenue to implement a verified implementation of Coq, in Coq, by writing verified Coq functions for the reduction of Coq terms, together with the proof of its correctness properties, such as the strong normalization of Coq terms (and therefore, decidable type-checking and evaluation), confluence on the rewriting of Coq terms, and so on.

To better formalize the semantics of Coq, the MetaCoq is split into a few main components. From the layer closest to the Coq language to the layer closest to machine code, we have: TemplateCoq (Section 3.1.1), PCUIC (Section 3.1.2), followed by Safe Checker, Erasure and beyond(Section 3.1.3). Each of the translations from one component (or type theory) to another is verified to preserve the correctness of reduction and conversion. The effort was completed for a large part of the core language of Coq (Sozeau, Boulier, et al. 2019), with only a few missing pieces :

- Eta-conversion
- Template Polymorphism
- Proof-irrelevant propositions (SProps)
- Modules

I will be tackling the last.

Let us remind ourselves of the task of MetaCoq project: we would like to have an implementation of Coq that is verified the desired properties of the underlying theory. For a better user experience, the terms of Coq are much more complex than its underlying, "Platonic" type-theoretical form, with many user-friendly, extra-logical features such as hint databases, pretty printing and more. Therefore, MetaCoq has several stages for a Coq term to go through, first stripping the internal representation (TemplateCoq) down to a form simple enough for proving correctness properties (PCUIC), then through a few stages, compiling the Coq terms to machine code.

3.1.1 TemplateCoq

TemplateCoq is a quoting library for Coq: a Coq program that takes a Coq term, and constructs its kernel representation as a new TemplateCoq term. This is the first layer of the stripping of a Coq term, where the structures associated with a term such as the global environment are quoted and represented faithfully as in the kernel, except that there are no Coq Modules present in the global environment of TemplateCoq (and the rest of the MetaCoq project).

This allows one to turn a Coq program into a Coq internal representation along with its associated environment structures, such as the definitions and declarations in the environment. Then, typing, reduction, and conversion rules can be defined, and its properties verified. Since this faithful representation of Coq terms might not yet be the easiest to prove things on, because of reasons such as n-ary parallel application terms instead of the unary, curried form, the important properties will be proved at the next level of PCUIC, and the definition of typing, reduction, and conversion rules defined here are shown to be preserved under translation to PCUIC.

3.1.2 PCUIC

PCUIC is the Polymorphic Cumulative Calculus of Inductive Constructions. It is a ”cleaned up version of the term language of Coq and its associated type system, shown equivalent to the one in Coq”(*Metacoq* n.d.). In other words, it is a type theory that is as powerful as Coq can express, having good properties such as weakening, confluence, principality (that every term has a principal type) etc. (Sozeau, Boulier, et al. 2019).

A term generated in TemplateCoq can be converted into a PCUIC term via a verified process. Since the theory of PCUIC is then proven to have all the “nice” properties in Coq, by the equivalence, the verified translation of TemplateCoq terms into PCUIC terms propagates these properties to the language of Coq.

3.1.3 Safe Checker, Erasure and Beyond

The core semantic operation of type theories is their reductions. The safechecker is a verified ”reduction machine, conversion checker and type checker” for PCUIC terms. At this point, we already have the tools to start with a Coq term, first quoting into TemplateCoq, then converting into a PCUIC term, and eventually having its type checked in the Safe Checker via a fully verified process. As far as correctness is concerned, this has already formed a verified end-to-end process of Coq's correctness.

The MetaCoq has further provided a verified Type and Proof erasure process from PCUIC to untyped Lambda Calculus. This erased language is can be evaluated in *C-light* semantics, the subset of C accepted by the CompCert verified compiler, which completes a maximally safe evaluation toolchain for the language of Coq, all the way to machine code (Sozeau, Boulier, et al. 2019).

3.1.4 Implementing Modules

The implementation of modules is easiest done at the level of TemplateCoq, since it has a faithful representation of the global environment of Coq, which can be extended with Modules. However, it is a choice on what to do with the modules upon translation to the next level of PCUIC. In this project, the modules in the global environment will not be added to PCUIC, instead modules will be elaborated away to the pre-existing, moduleless implementation of Global Environment upon translation to PCUIC. The benefits of doing so include:

1. We can enjoy the nice properties of PCUIC listed above automatically, via a verified translation process, and

2. Avoid extending the proof-theoretic strength of PCUIC calculus accidentally and possibly violating the niceness properties of PCUIC, or requiring more work necessary to re-establish the properties of PCUIC.

3.2 Abstract Syntax of Coq Modules

Now that we know the plan of implementing Coq Modules within the MetaCoq project, let us understand more about Coq modules before diving into their implementation.

The module system in Coq can be defined abstractly below, via a mutually recursive definition:

- A **structure** is an anonymous collection of definitions, and is the underlying construct of modules. They contain **structure elements**, which can be
  - A **constant definition** of a Coq term, including lambda term, application term, etc..

    ```
    Definition b: bool := true.
    ```

  - An **inductive definition** of a type.

    ```
    Inductive nat :=
    | O
    | S (n: nat).
    ```

  - or a **module**, a **module type**, or a **functor**, recursively.
- A **module** is a structure given a name.
- A **module type**, which is also a structure given a name, is a signature for modules. Within a module type, not all definitions must be concrete (e.g. just declaring

  ```
  Definition b: bool.
  ```

  is sufficient).
- A **functor** can be thought of as a function that accepts modules as arguments (and possibly functors as well) and returns modules. Additionally, a functor, like a function in functional programming languages, can return functions as well; functors that return functors are thus known as higher-order functors.

- A **module alias** is the association of a new name to an existing module. It can be thought of as a special case of functor application – one can think of it as assigning a name to the result of a nullary functor application.

For the remaining of the project, we call non-functor modules *plain modules*, or sometimes by its more proper name, *non-parametrized modules*.

3.3 Semantics of Modules

3.3.1 Conversion of Coq Terms

To understand the semantics of Coq Modules, we need to first understand the basic semantics of Coq. Since in its core, Coq is an extension of Calculus of Construction, a kind of typed lambda calculus, the core objects of Coq are simply its (lambda-calculus) terms. Every term is strongly typed; a term being well-typed is equivalent to saying that it is valid. Terms of a type correspond to proofs for a theorem as in the Curry-Howard correspondence; if a type is inhabited, the corresponding theorem thus has a proof, and that is how proof assistants work.

The syntax and semantics of Coq terms are as explained by the syntax, [1] conversion (including reduction and expansion) [2] and typing [3] rules respectively in the Coq documentation. One should note that in Coq, types are terms as well and therefore have types of their own, which in turn have types, and so on. To avoid Girard's Paradox, the analog of Russell's Paradox in Type Theory induced by this typing hierarchy, the type of all types lives in another **universe**, which is not required to understand in this project.

Every Coq term comes with a global environment Σ containing definitions, and a local context Γ containing assumptions during a proof. The operational semantics of Coq is given by its **reduction** relations between terms ($\beta, \delta, \zeta, \eta, \iota$ to name a few); of which includes the most well-known β-reduction (function application). The denotational semantics of Coq is then given by the **conversion** relations, which are the reflexive, transitive

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

closure of the reduction relations.

The typing relation, that declares a term t is well-typed with a term T is written as

$$\Sigma; ; \Gamma \vdash t \ : \ T.$$

3.3.2 Modules as second-class objects

Coq modules are *not* first-class objects of the language; they are on another axis of the language and interact with the core language (of terms) in limited ways only, and have their separate semantics. These semantics include their own β-reduction, such as during functor application.

Non-parametrized modules in Coq can be treated as a named container of constant and inductive definitions, including possibly nested modules. The names of terms and types defined within the module are implicitly prefaced by the sequence of modules containing it (which create separate namespaces). This abstraction allows users to reuse definitions from another file without name conflicts. To further expand this possibility, functors define a whole family of modules that are instantiated and specialized by supplying module arguments to them. Functors are therefore opaque second-class objects which are only useful when a module is generated.

3.3.3 The exclusion of functors in this project

Functors, and especially higher-order functors, are powerful in massive generalizations. However, the semantics of functors is complex. In Coq, the functors are generative, that is, if we do a functor application twice with the exact same arguments, the two resulting modules are deemed distinct. However, that is not the case for the function application of lambda calculus terms, thus deemed by some as not natural, as seen in the debate between applicative and generative functors within the literature of ML module implementations (Section 2.2). Furthermore, it is not entirely clear if the project's planned approach of elaborating Modules away as described in Section 3.1.4 will trivially extend to the case of functors, if at all. For the advantages listed in Section 3.1.4, it is wise to first try the

”elaborating away” approach for modules, and leave the extension to functors for future work.

3.3.4 Global Environment

The global environment in Coq can be understood as a table or a map. There are three columns in the map: first is a canonical kernel name (kername for short) second a pathname, and finally, the definition object.

A pathname is a name given to a definition by the user, including the ambient path (created by possibly nested modules) to that definition, in a dot-separated string such as *M.N.a*. Canonical kernames are the pathnames modulo aliasing, and can be thought of as unique labels.

Finally, the definition object can be:

- A **constant definition** to a Coq term.
- An **inductive definition** of a type.
- A **module definition**, or
- A **module type definition**.

Implementation

The implementation of the global environment in TemplateCoq is merely a list of definitions (Listing 4.1.4), while the looking up of names is via a linear search function through the list of declarations. Although this is less efficient than the map implementation in PCUIC, this does not affect the speed of the MetaCoq implementation, since the checker of MetaCoq operates at the level of PCUIC.

3.3.5 Plain Modules

Behavior and Implementation

To implement this, we need to design a data structure to represent modules, such as an inductive datatype that directly encodes the structure-based definition in section 3.2.

Properties

We say the implementation of such a module is correct if the meta-theory of the original system is 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. Define the data structure associated with modules and give modules typing rules to determine what it means for modules to be correct.
2. Show that the module interacts with the environment well by proving theorems about Environment Typing.
3. Since modules change the structure of environments, and environments are essential in the typing of Coq terms, verify the typing properties of Coq under the addition of modules.

Concretely, if a module as below is defined while the global environment, which stores definitions is denoted as Σ:

```
Module M.
    Definition a: nat := 0.
End M.
```

Then the environment must have a new declaration added:

$$\Sigma := \Sigma :: \text{ModuleDeclaration}(M, [\text{ConstantDeclaration}(M.a, nat, 0)])$$

So when $M.a$ is called, it must refer to the definition in the Global Environment correctly.

3.3.6 Aliased Modules

Aliasing modules can be treated as a special case of functor application of nullary functors (plain modules). Alternatively and more intuitively, aliased modules are just a renaming of existing modules, which can be seen as syntactic sugar for modules with alternative names. Therefore, the correctness depends only on implementing this internal referencing correctly.

Behavior and Implementation

We implement aliased modules with only one piece of data – the kername it is referencing to. Then, by the typing rule of modules that enforce that such a declared kername must already be defined in the environment, we prevent creating forward references and therefore cyclic aliasing. The lookup operation in the environment will then traverse this tree of aliasing back to the root to find its concrete definition, thus reducing the problem to a simple lookup correctness problem (Listing 4.1.5).

```
Module N := M.
```

Aliasing N to M and M is a previously defined module (or an alias), then any access path $N.X$ should be resolved similarly to $M.X$ (note that since M is possibly an alias as well, we do not require $N.X$ to resolve *to* $M.X$).

Proof Obligations

1. Well-definedness: aliasing can only occur for well-defined modules. There cannot be self-aliasing and forward aliasing (to something not yet defined).

2. The resolution of aliased modules is done at definition. If N is aliased to M, then N will immediately inherit the same kername as M. We will show this resolution is decidable and results in correct aliasing.

These properties depend on the correctness of a lookup function to determine if the aliasing is valid (already existing in the environment, and of the correct type).

3.3.7 Translation to PCUIC

Once the above is done, we can be sure that a Coq program with Modules has all its terms are well-defined, and it enjoys the nice properties of conversion, at the level of TemplateCoq. Then, as our definition of Modules is eventually elaborated away into PCUIC calculus and the definitions in modules "flattened" into the corresponding global environment, we will have to show the correctness of this translation.

Behavior and Implementation

The translation of modules, since it is implemented inductively, amounts to translating the base cases of the induction, which are the constant and inductive definitions are given with unique translated kernames. This is via a straightforward recursive function using the sub-procedures of translating constant and inductive procedures.

Proof Obligations

Once that is done, it suffices to check that this extended translation procedure still preserves typing, reduction, and conversion, then the niceness properties will follow.

Chapter 4

Implementation of MetaCoq Modules

4.1 The Module Data Structure

First, we define the underlying *structure* of a module. A `structure_field` contains a list of pairs with the first entry an identifier, and the second entry a `structure_field`. A `structure_field` can then be declarations for the following: constants, mutually inductive types, modules, and module types.

```
Inductive structure_field :=
| sfconst : constant_body -> structure_field
| sfmind : mutual_inductive_body -> structure_field
| sfmod : module_implementation -> structure_body -> structure_field
| sfmodtype : structure_body -> structure_field
with module_implementation :=
| mi_abstract : module_implementation (** Declare Module M: T. *)
| mi_algebraic : kername -> module_implementation (** Module M [:T] := N. *)
| mi_struct : structure_body -> module_implementation (** Module M:T. ... End M.*)
| mi_fullstruct : module_implementation (** Module M. ... End M.*)
with structure_body :=
| sb_nil
| sb_cons : ident -> structure_field -> structure_body -> structure_body.
```

Listing 4.1.1: Definition of a structure field.

The type `module_implementation` here represents the four ways of defining a module respectively:

1. `mi_abstract`: an abstract module declaration usually used in module signatures for nested modules;
2. `mi_algebraic`: an algebraic module expression, most commonly functor application. Here, we consider only module aliasing (nullary functor application).
3. `mi_struct`: a module declaration with an explicit module type. The `structure_body` argument will contain the concrete implementation given.
4. `mi_fullstruct`: a module declaration without an explicit module type; in other words, the module has a type exactly equal to itself, so the implementation will be stored within the second argument to the `sfmod` constructor.

Now, we can define proper Modules and Module Types as follows:

```
344 Definition module_type_decl := structure_body.
345 Definition module_decl := module_implementation × module_type_decl.
```

Listing 4.1.2: Definition of the Module and Module Type structures.

Since a module type is a module without an implementation. Note that from the code from now on, structure body (`structure_body`) and module type `module_type_decl` are equivalent.

Finally, the global declaration can be added with two new constructors, `ModuleDecl` and `ModuleTypeDecl`:

```
347 Inductive global_decl :=
348 | ConstantDecl : constant_body -> global_decl
349 | InductiveDecl : mutual_inductive_body -> global_decl
350 | ModuleDecl : module_decl -> global_decl
351 | ModuleTypeDecl : module_type_decl -> global_decl.
```

Listing 4.1.3: Extending the global declarations with modules.

A global environment is defined as a list of global declarations with some other bookeeping data, most importantly information about type universes, which is outside of the scope of this paper, so we will merely mention it.

```
354 Definition global_declarations := list (kername * global_decl).
355
356 Record global_env := mk_global_env
357   { universes : ContextSet.t;
358     declarations : global_declarations;
359     retroknowledge : Retroknowledge.t }.
```

Listing 4.1.4: Definition of a global environment.

Verified Properties

Even before any information on the well-typedness of a defined module, we can already assert that whenever a module is defined, we should be able to look it up in the global environment Σ, and the lookup result is the exact module we defined. Vice versa, if we found some module via a kername kn, it must have been defined with the same name.

```
202 Lemma declared_module_lookup {Σ mp mdecl} :
203   declared_module Σ mp mdecl ->
204   lookup_module Σ mp = Some mdecl.
205 Proof.
206   unfold declared_module, lookup_module. now intros ->.
207 Qed.
208
209 Lemma lookup_module_declared {Σ kn mdecl} :
210   lookup_module Σ kn = Some mdecl ->
211   declared_module Σ kn mdecl.
212 Proof.
213   unfold declared_module, lookup_module.
214   destruct lookup_env as [[]|] ⇒ //. congruence.
215 Qed.
```

Listing 4.1.5: Proof of lookup of modules.

The same is done for Module Types as well.

4.2 Typing Modules

To ensure a module or a module type is well-typed, we need to define typing rules on modules. They are defined in terms of inductive types:

```
Inductive on_structure_field Σ : structure_field -> Type :=
  | on_sfconst c : on_constant_decl Σ c -> on_structure_field Σ (sfconst c)
  | on_sfmind kn inds : on_inductive Σ kn inds -> on_structure_field Σ (sfmind inds)
  | on_sfmod mi sb : on_module_impl Σ mi
                   -> on_structure_body Σ sb
                   -> on_structure_field Σ (sfmod mi sb)
  | on_sfmodtype mtd : on_structure_body Σ mtd -> on_structure_field Σ (sfmodtype mtd)

with on_structure_body Σ : structure_body -> Type :=
  | on_sb_nil : on_structure_body Σ sb_nil
  | on_sb_cons kn sf sb : on_structure_field Σ sf
                        -> on_structure_body Σ sb
                        -> on_structure_body Σ (sb_cons kn sf sb)
with on_module_impl Σ : module_implementation -> Type :=
  | on_mi_abstract : on_module_impl Σ mi_abstract
  | on_mi_algebraic kn : on_module_impl Σ (mi_algebraic kn)
```

Listing 4.2.1: Typing rules for structure fields.

For example, the constructor `on_sfmod` says that for a structure field containing a module to be well-defined, we require the module implementation and module type (here written as structure body) to be recursively well-typed, via `on_module_impl` and `on_structure_body` respectively. Since the checking of well-typedness also depends on the global environment (eg. reference to predefined constants etc.), the argument Σ is passed around everywhere in these typing rules.

One might further notice that the type constructors `on_*` in 4.2.1 are largely self-contained, except for `on_constant_decl` and `on_inductive` in the `on_sfconst` and `on_sfmind` constructors, respectively. This should give the reader some insight into the intuition that non-parametrized modules are largely tree-like containers with actual content supplied by Constant Declarations and (Mutually) Inductive Declarations. If we follow the path of the definition of, say, `on_constant_decl`:

```
Definition on_constant_decl Σ d :=
  match d.(cst_body) with
  | Some trm ⇒ P Σ [] trm (Typ d.(cst_type))
  | None ⇒ on_type Σ [] d.(cst_type)
  end.
```

Listing 4.2.2: Typing rule for constant declarations.

The type-checking is now at the level of terms - if the constant body is `None`, ie. it is a declaration without a body, then one checks the well-typedness of the type; on the other hand, if the constant has a body of `Some trm`, we will check the well-typedness of the term `trm` and its type, using the predicate `P` that will be supplied later.

4.3 Typing the Global Environment

Since modules do not interact with terms, it thus mainly lives in the global environment and provides namespaced definitions to the user. It is thus important to make sure that the global environment is well-typed.

```
Definition on_global_decl Σ kn decl :=
  match decl with
  | ConstantDecl d ⇒ on_constant_decl Σ d
  | InductiveDecl inds ⇒ on_inductive Σ kn inds
  | ModuleDecl mb ⇒ on_module_decl Σ mb
  | ModuleTypeDecl mtd ⇒ on_structure_body Σ mtd
  end.

Inductive on_global_decls (univs : ContextSet.t) (retro : Retroknowledge.t)
  : global_declarations -> Type :=
| globenv_nil : on_global_decls univs retro []
| globenv_decl Σ kn d :
    on_global_decls univs retro Σ ->
    on_global_decls_data univs retro Σ kn d ->
    on_global_decls univs retro (Σ ,, (kn, d)).
```

Listing 4.3.1: Typing rule for global declarations.

To make sure that a list of global declarations is well-typed, we need to check that:

1. the prefix global environment Σ is well-defined; and
2. the current global declaration given by kername `kername` and declaration `d` consists of well-typed data (`on_global_decls_data`) concerning the prefix Σ:

```
Definition fresh_global (s : kername) (g : global_declarations) : Prop :=
  Forall (fun g ⇒ g.1 <> s) g.

Record on_global_decls_data (univs : ContextSet.t)
  retro (Σ : global_declarations)
  (kn : kername) (d : global_decl) :=
    {
      kn_fresh :  fresh_global kn Σ ;
      udecl := universes_decl_of_decl d ;
      on_udecl_udecl : on_udecl univs udecl ;
      on_global_decl_d : on_global_decl (mk_global_env univs Σ retro, udecl) kn d
    }.
```

Listing 4.3.2: Freshness requirement for global declarations.

The global environment is well-typed if it has well-typed global declarations and well-typed universes.

```
Definition on_global_env (g : global_env) : Type :=
  on_global_univs g.(universes)
  × on_global_decls g.(universes) g.(retroknowledge) g.(declarations).
```

Listing 4.3.3: Typing the global environment.

Verified Properties

A few things can be said about typing rules. First thing is that it is artificially defined to ”carve out” a subset of terms that we deem as well-typed; and other than a few sanity properties such as being consistent (if there exists a proof tree showing the well-typedness of a term, there doesn't exist another proof tree that shows otherwise), it is entirely what it is made to be. Therefore, the main place to verify these properties is at the typing of terms, which we will explain later. Also, the definition of environment is

parallel to the calculus of the terms, so instead of checking the correctness of the typing rules itself, we can check its behavior when we change a set of typing rules for the terms.

Recall that the in the example of chasing the definition of `on_constant_decl`, we found that the well-typedness predicate `P` is parametrized, therefore allowing us to investigate the following functoriality property concerning different predicates:

```
1426 Lemma on_global_decl_impl {cf : checker_flags} Pcmp P Q Σ kn d :
1427   (forall Γ t T,
1428     on_global_env Pcmp P Σ.1 ->
1429     P Σ Γ t T -> Q Σ Γ t T) ->
1430   on_global_env Pcmp P Σ.1 ->
1431   on_global_decl Pcmp P Σ kn d -> on_global_decl Pcmp Q Σ kn d.
```

Listing 4.3.4: Functoriality of typing of a global declaration.

This lemma says that: fix a global environment Σ. Let well-typeness predicates `P, Q` be given. Then for all global declarations with kername `kn` and declaration `d`, if we assume

- the predicate `P` implies the predicate `Q` over all local contexts, terms, and types; and
- the global environment is well typed against `P`; and
- the declaration `kn, d` is well typed against `P`,

then the declaration `kn, d` is well typed against `Q`.

The key of the proof here is to do a case analysis on the type of the declaration `d`. Since any global declaration can only be a constant, inductive, module or module type, we first clear the first two cases using lemmas `on_{constant,inductive}_decl_impl`. The remaining two are mutually inductive, so we solve them using the mutual induction principle `on_mi_sf_sb_mutrect`, and reduce to the base cases, which are again the constant and the inductive case.

Since the above lemma is true across all declarations, it is natural to think that this functoriality should extend to lists of global declarations as well – that is indeed the case.

```
1454  Lemma on_global_env_impl {cf : checker_flags} Pcmp P Q :
1455    (forall Σ Γ t T,
1456        on_global_env Pcmp P Σ.1 ->
1457        on_global_env Pcmp Q Σ.1 ->
1458        P Σ Γ t T -> Q Σ Γ t T) ->
1459    forall Σ, on_global_env Pcmp P Σ -> on_global_env Pcmp Q Σ.
```

Listing 4.3.5: Functoriality of typing of the global environment.

4.4 Typing of Terms

Since we modified the environment to include module definitions, this also directly affects the properties of the typing of Coq terms, since every term is typed under a global environment (and a local context). Terms in TemplateCoq are of an inductive type with 18 constructors:

```
401 Inductive term : Type :=
402 | tRel (n : nat)
```

...

```
420 | tFloat (f : PrimFloat.float).
```

Listing 4.4.1: Definition of TemplateCoq terms.

each having its own typing rule (relation between a term and its type). The typing relation is too long (around 122 lines) to show in its entirety, thus we show its signature and a few constructors:

```
741 Inductive typing `{checker_flags} (Σ : global_env_ext) (Γ : context)
742 : term -> term -> Type
743 :=
744 | type_Rel n decl :
745     wf_local Σ Γ ->
746     nth_error Γ n = Some decl ->
747     Σ ;;; Γ |- tRel n : lift0 (S n) decl.(decl_type)

    ...

864 where " Σ ;;; Γ |- t : T " := (typing Σ Γ t T) : type_scope
```

Listing 4.4.2: Typing of TemplateCoq terms.

where the constructors are typing rules for each kind of term, such as a variable, a sort, a lambda abstraction, an application and so on.

Verified Properties

A key lemma about the typing of terms under well-formed environments, `env_prop`, is given as follows:

```
1020 Definition env_prop `{checker_flags} (P : forall Σ Γ t T, Type)
1021 (PΓ : forall Σ Γ (wfΓ : wf_local Σ Γ), Type) :=
1022   forall (Σ : global_env_ext) (wfΣ : wf Σ) Γ (wfΓ : wf_local Σ Γ) t T
1023   (ty : Σ ;;; Γ |- t : T),
1024     on_global_env cumul_gen (lift_typing P) Σ
1025     * (PΓ Σ Γ (typing_wf_local ty) * P Σ Γ t T).
```

Listing 4.4.3: Definition of key lemma in typing.

Which says the following: fix any two predicates `P` and `PΓ` that about a term `t` and a type `T`. Suppose we are given global environment Σ and local context Γ which are well-

formed, and that the following typing relation holds:

$$\Sigma;;\Gamma \vdash t : T,$$

then P holds on the global environment Σ, and PΓ holds on the local context.

Once we supply such predicates P and PΓ, the above will be turned into a statement that can be proven (or disproved). This is a strong statement because, for instance, it implies the following property about typing:

```
Lemma env_prop_typing `{checker_flags} {P PΓ} : env_prop P PΓ ->
  forall (Σ: global_env_ext) (wfΣ: wf Σ) (Γ: context) (wfΓ: wf_local Σ Γ) (t T: term),
    Σ ;;; Γ |- t : T -> P Σ Γ t T.
```

Listing 4.4.4: A consequence of env prop.

Implying that if we know that `env_prop P PΓ` is true, then P does not only holds on the global environment Σ, it holds on the terms as well.

To proof `env_prop` is difficult since, mechanically, it is a verification across the inductive cases of the global environment, as well as the 18 kinds of terms and their corresponding typing rules. It would be immensely useful to first prove an intermediary lemma leading to it, or in a way, an induction step:

```
Lemma typing_ind_env `{cf : checker_flags} :
  forall (P : global_env_ext -> context -> term -> term -> Type)
         (Pdecl := fun Σ Γ wfΓ t T tyT ⇒ P Σ Γ t T)
         (PΓ : forall Σ Γ, wf_local Σ Γ -> Type),

...

         P Σ Γ t B) ->

        env_prop P PΓ.
```

Listing 4.4.5: Induction hypothesis for proving env prop.

The module typing rules we defined earlier are used extensively since we are showing properties of global environments as well. We omit almost 400 lines of verified proof from this report and leave the details to the interested reader.

4.5 Translation to PCUIC

At this point, we have already successfully defined modules and their typing rules in TemplateCoq, as well as verifying their correctness properties. Since TemplateCoq has modules while PCUIC does not, we need to elaborate modules away during the translation. We do this by storing all definitions within modules as if they are in the (flat) global environment. Of course, the key here is to choose an algorithm for the elaboration of modules that preserves the freshness (no name clash) of the translated environment, preserves the correct ”look-up” properties, and also being well-typed.

The global environment for PCUIC is without modules:

```
278 Inductive global_decl :=
279 | ConstantDecl : constant_body -> global_decl
280 | InductiveDecl : mutual_inductive_body -> global_decl.
281 Derive NoConfusion for global_decl.
282
283 Definition global_declarations := list (kername * global_decl).
284
285 Record global_env := mk_global_env
286   { universes : ContextSet.t;
287     declarations : global_declarations;
288     retroknowledge : Retroknowledge.t }.
```

Listing 4.5.1: Definition of the global environment for PCUIC.

Having seen definitions of global environments on both sides, we can now look at the translation function. This is similarly a mutually recursive function (since structures

are defined in a mutual inductive manner), but the heavy lifting is done in the translation of structure fields:

```
314  Fixpoint trans_structure_field kn id (sf : Ast.Env.structure_field) :=
315    let kn' := kn_append kn id in
316    match sf with
317    | Ast.Env.sfconst c ⇒ [(kn', ConstantDecl (trans_constant_body c))]
318    | Ast.Env.sfmind m ⇒ [(kn', InductiveDecl (trans_minductive_body m))]
319    | Ast.Env.sfmod mi sb ⇒ match mi with
320      | Ast.Env.mi_fullstruct ⇒ trans_structure_body kn' sb
321      | Ast.Env.mi_struct s ⇒ trans_structure_body kn' s
322      | _ ⇒ trans_module_impl kn' mi
323      end
324    | Ast.Env.sfmodtype _ ⇒ []
325    end
```

Listing 4.5.2: Translation of structure fields to PCUIC.

There are a few things to note:

1. The body of this mutually recursive branch of the fixpoint is mainly a match expression describing the translation for each kind of structure field.
    - Constant declarations and inductive type declarations are translated as is, with a new kername `kn'`;
    - Module declarations have the declarations in its implementation translated recursively;
    - Module types are removed entirely since there is no more module in PCUIC, and hence the signature of modules would be meaningless.
2. The new kername `kn'` is done by appending the prefix (module) name `kn` with the identifier `id` associated with that structure field entry.

```
150  Definition kn_append (kn: kername) (id: ident) : kername := ((MPdot kn.1 kn.2), id).
```

Listing 4.5.3: Appending kernames.

Once we have the above, we can translate modules by folding through the structure body.

```
334  with trans_structure_body kn (sb: Ast.Env.structure_body) :=
335    match sb with
336    | Ast.Env.sb_nil ⇒ []
337    | Ast.Env.sb_cons id sf tl ⇒
338      trans_structure_field kn id sf ++ trans_structure_body kn tl
339    end.

508  Definition trans_global_decl (d : kername × Ast.Env.global_decl) :=
509    let (kn, decl) := d in match decl with
510    | Ast.Env.ConstantDecl bd ⇒ [(kn, ConstantDecl (trans_constant_body bd))]
511    | Ast.Env.InductiveDecl bd ⇒ [(kn, InductiveDecl (trans_minductive_body bd))]
512    | Ast.Env.ModuleDecl bd ⇒ trans_module_decl kn bd
513    | Ast.Env.ModuleTypeDecl _ ⇒ []
514    end.
```

Listing 4.5.4: Translating structure body and global declarations.

To translate global declarations, we need to flatmap across the global environment with `trans_global_decl`, since now each TemplateCoq global declaration can translate into multiple PCUIC global declarations due to the tree structure of modules.

```
525  Definition trans_global_decls env (d : Ast.Env.global_declarations)
526    : global_env_map :=
527    fold_right
528    (fun decl Σ ⇒ fold_right add_global_decl Σ (trans_global_decl Σ decl)) env d.
```

Listing 4.5.5: Translating global declarations.

Properties on Translation

The properties in this part can be phrased in the form of "translation preserves [property]". The ultimate property here is that under a well-formed environment Σ, "translation preserves typing/well-typedness":

```
Theorem template_to_pcuic_typing {cf} {Σ : Ast.Env.global_env_ext} Γ t T :
  ST.wf Σ ->
  ST.typing Σ Γ t T ->
  let Σ' := trans_global Σ in
  typing Σ' (trans_local Σ' Γ) (trans Σ' t) (trans Σ' T).
```

Listing 4.5.6: Translation preserves typing.

The main lemma proved here is `template_to_pcuic`, which is of the following form:

```
Theorem template_to_pcuic {cf} :
  ST.env_prop (fun Σ Γ t T ⇒
    let Σ' := trans_global Σ in
    wf Σ' ->
    typing Σ' (trans_local Σ' Γ) (trans Σ' t) (trans Σ' T))
    (fun Σ Γ _ ⇒
      let Σ' := trans_global Σ in
      wf Σ' ->
      wf_local Σ' (trans_local Σ' Γ)).
```

Listing 4.5.7: Template to PCUIC.

The statement here is abstracted by the `env_prop` (4.4.3), which in its totality, assigns `P` to the first argument, asserting that

$$\Sigma;;\Gamma \vdash t : T \implies \Sigma';;\Gamma' \vdash t' : T'$$

where Σ', Γ', t', T' represents the translated global environment, local context, term

and type Σ, Γ, t, T respectively; and sets the second argument to assert that the translated local context is also well-formed.

Before showing these big theories about typing, we have "sanity" checks on translation to ensure that environments are well-translated. This property can be formulated as

1. "translation preserves non-existence", that is, the translated environment should only contain the intended translation and nothing more; and
2. its dual, "translation preserves existence", that is, nothing is lost in translation.

In Coq, this can be formulated as the following lemma:

```
Lemma trans_lookup_env {cf} {Σ : Ast.Env.global_env} cst {wfΣ : Typing.wf Σ} :
  match Ast.Env.lookup_env Σ cst with
  | None ⇒ lookup_env (trans_global_env Σ) cst = None
  | Some d ⇒ match d with
    | Ast.Env.ConstantDecl _ | Ast.Env.InductiveDecl _ ⇒
      ∑ (Σ' : Ast.Env.global_env) (d': global_decl),
        [× Ast.Env.extends_decls Σ' Σ,
          Typing.wf Σ',
          wf_global_decl (Σ', Ast.universes_decl_of_decl d) cst d,
          extends_decls (trans_global_env Σ') (trans_global_env Σ),
          trans_global_decl (trans_global_env Σ') (cst, d) = (cst, d')::[] &
          lookup_env (trans_global_env Σ) cst = Some d']
    (** Modules are elaborated away. *)
    | Ast.Env.ModuleDecl _ | Ast.Env.ModuleTypeDecl _ ⇒
      lookup_env (trans_global_env Σ) cst = None
    end
  end.
```

Listing 4.5.8: Translation preserves (non-)existence.

The two cases of the outermost `match` statement on the return value of `lookup_env Σ cst` correspond to the two properties above:

1. (non-existence) the case of `None` returned, which means that the original environment Σ does not contain the declaration `cst`, it should not appear in the translated environment. Contrastingly,

2. (existence) the case where `cst` is the declaration `d` in the original environment Σ, we can say the following:

    - if `d` is a constant or an inductive declaration, it should be translated as if, therefore it should be found in the translated, well-formed environment Σ′ (lines 228-233 of 4.5.8). However,
    - if `d` is a module or a module type, then it shouldn't exist in the translated environment Σ′. Furthermore, its constituent structure fields should appear as declarations in Σ′, which is exactly the definition of the translation.

Chapter 5

A Modular Environment

After the completion of my implementation in the previous chapter, the last mile of verifying the TemplateCoq to PCUIC translation proved to be time-consuming, resulting in weeks without any `QED`s. In the process of verifying those properties, many technical challenges surfaced in constructing formal proofs in Coq for concepts that are easily explained in natural language. Fortunately, two beneficial events ensued:

1. After identifying the pain points in the proofs, I found out the design decision that caused the trouble. In tackling it, a new, more natural solution surfaced, but it requires a complete rewrite. As an experimental effort, I detailed the rewrite in this chapter and up to the environment typing rules, as a possible second take at the project. I will discuss this implementation in the first section.

2. Due to the new design choices in the rewrite and their complexity, some non-trivial proof machinery was required to complete the formal proofs, such as nested inductive types with their induction hypotheses, well-founded recursion, and strengthening of induction hypotheses. Since a number of them are direct consequences of this modular environment rewrite, I will include them in the next chapter (Chapter 6) as a summary of my learning.

5.1 Difficulties on the Global Declaration List

The proof for `trans_lookup_env` (4.5.8) proved to be tedious due to the double fold. In each fold, one adds definitions to the environment using an opaque `add_global_decl` function, which changes the accumulator Σ during each fold. A nested fold makes this even more tedious with proof levels going up to four or five levels.

```
239 Proof.
240   destruct Σ as [univs Σ retro]. induction Σ.
241   - cbn; auto.

307       --- (** a.2 is a *)
308         unfold trans_global_env. subst Σmap'; simpl.

316         (** proving assertion by mutual induction *)
317         * subst P P0 P1. apply Ast.Env.sf_mi_sb_mutind ⇒ //=.
318           ** cbn. intros c id.

326             *** simpl in *. subst.
```

...

Listing 5.1.1: Tedious nested proofs.

After spending weeks trying different methods to make this work, including an elaborate argument defining a strict partial order on kernames (6.3), I decided to venture for a new solution. The source of the problem here is the double-fold (4.5.8), which obscures the meaning of expressions after repeated applications of `add_global_decl` changing the accumulator environment. To break down the two folds:

- The first fold is done over the *list* of global declarations, while
- the second fold is done over the *module* structure.

and the definition for structure fields is identical to that of global declarations:

```
Inductive structure_field :=
| sfconst : constant_body -> structure_field
| sfmind : mutual_inductive_body -> structure_field
| sfmod : module_implementation -> structure_body -> structure_field
| sfmodtype : structure_body -> structure_field

Inductive global_decl :=
| ConstantDecl : constant_body -> global_decl
| InductiveDecl : mutual_inductive_body -> global_decl
| ModuleDecl : module_decl -> global_decl
| ModuleTypeDecl : module_type_decl -> global_decl.
```

Listing 5.1.2: Identical definitions of structures and declarations

5.2 The Modular Environment

It would be natural to unify modules and environments into a single structure — modules. The list of global declarations itself can be seen as an anonymous, ambient, top-level module, thus just a special case of a module. Under this generalization, any well-formedness properties or well-typedness properties about environments should be subsumed under modules, thus giving us a good sense when defining the typing rules for modules.

5.2.1 Definition of the Data Structure

Let us begin by defining modules, then specialize into global declarations.

```
Inductive structure_field :=
| ConstantDecl : constant_body -> structure_field
| InductiveDecl : mutual_inductive_body -> structure_field
| ModuleDecl :
    module_implementation
    -> list (ident × structure_field)
    -> structure_field
| ModuleTypeDecl : list (ident × structure_field) -> structure_field
with module_implementation :=
| mi_abstract : module_implementation (** Declare Module M: T. *)
| mi_algebraic : kername -> module_implementation (** Module M [:T] := N. *)
| mi_struct : (** Module M:T. ... End M.*)
    list (ident × structure_field) -> module_implementation
| mi_fullstruct : module_implementation (** Module M. ... End M.*).
```

Listing 5.2.1: Definition of structure fields.

Structure fields now subsume global declarations, and a structure body is associated list of structure fields and identifiers. Similar to the previous definition (4.1.1), the possible nesting of structure bodies inside structure fields gives rise to the tree structure of modules.

```
Definition module_type_decl := structure_body.
Definition module_decl := module_implementation × module_type_decl.
Notation global_decl := structure_field.
Notation global_declarations := structure_body.
```

Listing 5.2.2: Definition of global declarations.

Similarly, we define module types to be structure bodies, and a module to be a module type with an implementation. Since global environments are anonymous modules, they do not have a possibility of reuse and thus signature is insignificant here - it is of the same structure as a structure body.

So, how do we tell apart a global environment from a structure body? We differentiate it by an additional directory path to the current file.

```
478  Record global_env := mk_global_env
479    { universes : ContextSet.t;
480      declarations : structure_body;
481      retroknowledge : Retroknowledge.t;
482      path : dirpath }.
```

Listing 5.2.3: Definition of global declarations.

5.2.2 Definition of Typing Rules

The typing rules of this section are still experimental and not yet finalized. However, they illustrate a possible set of rules that can be implemented and then verified against various properties.

```
1271 Inductive on_structure_field Σ : structure_field -> Type :=
1272   | on_ConstantDecl c :
1273       on_constant_body Σ c -> on_structure_field Σ (ConstantDecl c)
1274   | on_InductiveDecl kn inds :
1275       on_inductive Σ kn inds -> on_structure_field Σ (InductiveDecl inds)
1276   | on_ModuleDecl mi mt :
1277       on_module_impl Σ mi ->
1278       on_structure_body on_structure_field mt ->
1279       on_structure_field Σ (ModuleDecl mi mt)
1280   | on_ModuleTypeDecl mtd :
1281       on_structure_body Σ mtd ->
1282       on_structure_field Σ (ModuleTypeDecl mtd)
```

Listing 5.2.4: Typing rules for structure fields.

The typing for structure fields are natural: constants and inductive types are typed

as-if, while modules and module types are typed recursively. The interesting part that is different from the previous implementation is the combination of the information when typing a structure body:

```
1284 with on_structure_body (Σ: global_env_ext) : structure_body -> Type :=
1285   | on_sb_nil : on_structure_body Σ nil
1286   | on_sb_cons Σ sb i sf :
1287       on_structure_body Σ sb ->
1288       fresh_structure_body Σ i sb ->
1289       on_udecl Σ (universes_decl_of_decl sf) ->
1290       on_structure_field Σ sf ->
1291       on_structure_body Σ (sb ,, (i, sf))
```

Listing 5.2.5: Typing rules of structure body.

This time, the typing rules for a structure body include typical checks as in the global environment. If we focus on the `on_sb_cons` constructor for the inductive case, the checks include

1. the correctness of the prefix structure body against the environment,
2. the freshness of identifier `i` against the prefix structure body and the environment,
3. the universe declaration of the structure field,
4. and the recursive check to the structure field itself.

We define naturally the below:

```
1301 Definition on_module_type_decl := on_structure_body.
1302 Definition on_module_decl Σ m := on_module_impl Σ m.1 × on_module_type_decl Σ m.2.

1312 Definition on_global_decl := on_structure_field.
```

Listing 5.2.6: Typing rules of modules, and this global declarations.

Properties to Verify

Here, we list some properties to modify and then verify, following the changes. We give the sketch in mathematical form only, with references to existing code where possible.

The uniqueness of names in the environment is important - shadowing is not allowed in Coq. Therefore, it would be important to check that our freshness rule in `on_structure_body` implies no duplication of names:

Lemma 5.2.1. *If a structure body Σ is well-formed, then no two structure fields in Σ have the same kername.*

We then have the following corollary:

Collorary 5.2.1. *If the global declarations Σ are well-formed, then no two global declarations in Σ have the same kername.*

The corresponding previous formulation is as follows:

```
1361 Lemma NoDup_on_global_decls univs retro decls
1362   : on_global_decls univs retro decls -> NoDup (List.map fst decls).
```

Listing 5.2.7: Well-formed implies no duplicated names.

Other than that, we can have the similar functoriality result on typing global environments, this time following directly from the case of a structure body:

```
1361 Lemma NoDup_on_global_decls univs retro decls
1362   : on_global_decls univs retro decls -> NoDup (List.map fst decls).
```

Listing 5.2.8: Functoriality of typing of global environments.

For a possible extension of this approach, please refer to the chapter on future work: Section 7.2.1.

Chapter 6

Tactics and Proof Machineries

Now we discuss some of the main learning points gained during the implementation of this project. Although these might be common knowledge for the Coq or proof theory expert, they contain some non-trivial proof-theoretic machinery that is useful in formal proofs involving inductive data types, even though they might be dismissed in typical pen-and-paper proofs by their ”apparent” correctness. They are listed below in no particular order.

- nested induction, mutual induction, (can always reduce to the case of a simple inductive type, but...)
- well-founded recursion and measures.
- definition of strict partial orders and strengthening of induction hypotheses

6.1 Nested Inductive Types

Inductive types are a way to define things recursively. A typical example is the inductive definition of a list:

```
Inductive list (T: Type) :=
| nil
| cons (t: T) (tl: list).
```

Listing 6.1.1: Definition of a polymorphic list.

To argue about lists, we have the following induction principle:

$$\forall P, (P\ nil) \implies (\forall t\ tl, (P\ tl) \implies P(cons\ t\ tl)) \implies \forall l, (P\ l)$$

Nested inductive types are inductive types that have nested induction in them, one simple example being a rose tree (Meertens 1988):

```
Inductive roseTree :=
| node (xs: list roseTree).
```

Listing 6.1.2: Definition of a rose tree.

where the nested induction occurs at the list of rose trees within a node. Intuitively, this is a tree structure in which every node contains a "forest" of trees. Unfortunately, Coq does not generate a strong enough induction principle for nested inductive types such as the below:

$$\forall P, (\forall xs, P(node\ xs)) \implies \forall rt, (P\ rt)$$

We focus on the induction hypothesis, the condition before the top-level implication. It omitted the fact that xs is a list, and we need to check each rose tree within the list with predicate P first. Here is a stronger induction principle that is generally used:

$$\forall P, (\forall xs, (\forall x \in xs, P\ x) \implies P(node\ xs)) \implies \forall rt, (P\ rt)$$

Notice the weakening of the induction hypothesis, and thus the strengthening of the induction principle. In my case, it is the definition of structure fields of the modular rewrite:

```
325  Inductive structure_field :=
326  | ConstantDecl : constant_body -> structure_field
327  | InductiveDecl : mutual_inductive_body -> structure_field
328  | ModuleDecl :
329      module_implementation
330      -> list (ident × structure_field)
331      -> structure_field
332  | ModuleTypeDecl : list (ident × structure_field) -> structure_field
```

Listing 6.1.3: Definition of structure fields.

where the nested `list (ident × structure_field)` is a doubly-nested inductive, since the product (or pair) type itself is defined inductively, in addition to the inductively defined polymorphic lists (6.1.1). In this case, we need to manually define a stronger induction principle for use. This can be done by noticing that nested inductive types have an equivalent rewrite by expanding the definition explicitly. For example, in the case of rose trees, Coq will generate the correct (sufficiently strong) induction principle for the following equivalent definition:

```
Inductive roseTree :=
| node (xs: listRoseTree)
where listRoseTree :=
| nil
| cons (r: roseTree) (tl: listRoseTree).
```

Listing 6.1.4: Non-nested inductive definition of rose trees.

The reason why we prefer to use readily-available abstractions such as lists is that one can then rely on facts that are proven already on the polymorphic lists, and properties

on their operations such as `map` and `fold`. Therefore, the operation here is twofold:

1. generate induction principle for the non-nested formulation and remember both the induction principle (a type in Coq, due to Curry-Howard correspondence) and the term that inhabits it.

2. now redefine the type in a nested inductive manner, then manually define the induction principle and prove it. The corresponding action in Coq is to define the type corresponding to the inductive principle (which is similar to the one extracted above) while replacing the explicitly inductive parts with its relevant counterparts. For example, replace the explicit weakening described in (REFME) with the `Coq.Lists.List.Forall` inductive (higher-order) predicate. Finally, prove this newly defined induction principle correct by supplying the term we obtained in the previous step with suitable amendments.

We illustrate this with an actual implementation of the strengthened induction principle on structure fields. First, we define the fixpoint `F1` that switch cases on the constructor of a typical list, `nil` and `cons`:

```
360  Section Nested.
361    Variable F : forall (s: structure_field), P s.
362    Fixpoint F1 (s : structure_body) : P1 s :=
363      match s as s0 return (P1 s0) with
364      | nil ⇒ f7
365      | cons (i,s0) s1 ⇒ f8 i s0 (F s0) s1 (F1 s1)
366      end.
367  End Nested.
```

Listing 6.1.5: Manually proving IH for the nested list.

Then, we can integrate `F1` into the induction principle terms for structure fields (including its mutually inductive counterpart, module implementation), therefore forming the proofs for the individual strengthened induction principles:

```
369     Fixpoint F (s : structure_field) : P s :=
370     match s as s0 return (P s0) with
371     | ConstantDecl c ⇒ f c
372     | InductiveDecl m ⇒ f0 m
373     | ModuleDecl m s0 ⇒ f1 m (F0 m) s0 (F1 F s0)
374     | ModuleTypeDecl s0 ⇒ f2 s0 (F1 F s0)
375     end
376   with F0 (m : module_implementation) : P0 m :=
377     match m as m0 return (P0 m0) with
378     | mi_abstract ⇒ f3
379     | mi_algebraic k ⇒ f4 k
380     | mi_struct s ⇒ f5 s (F1 F s)
381     | mi_fullstruct ⇒ f6
382     end.
383
384   Definition structureField_rect := F.
385   Definition moduleImpl_rect := F0.
386   Definition structureBody_rect := F1 F.
```

Listing 6.1.6: Induction principles with explicit names.

Finally, the last 3 lines give suggestive names to the fixpoints corresponding to the induction hypothesis for the respective structures. Their types are already inductive hypotheses about only one particular branch of the mutually inductive definition, and we combine them into a huge induction principle:

```
Definition sf_mi_sb_mutrect
  (P : structure_field -> Type) (P0 : module_implementation -> Type)
  (P1 : structure_body -> Type := All (fun x ⇒ P (snd x)))
  (f : forall c : constant_body, P (ConstantDecl c))
  (f0 : forall m : mutual_inductive_body, P (InductiveDecl m))
  (f1 : forall m : module_implementation,
      P0 m -> forall s : structure_body, P1 s -> P (ModuleDecl m s))
  (f2 : forall s : structure_body, P1 s -> P (ModuleTypeDecl s))
  (f3 : P0 mi_abstract) (f4 : forall k : kername, P0 (mi_algebraic k))
  (f5 : forall s : structure_body, P1 s -> P0 (mi_struct s))
  (f6 : P0 mi_fullstruct) :
  (forall s: structure_field, P s) * (forall m: module_implementation, P0 m)
  * (forall s: structure_body, P1 s).
Proof.
repeat split.
eapply structureField_rect; eauto.
eapply moduleImpl_rect; eauto.
eapply structureBody_rect; eauto.
Defined.
```

Listing 6.1.7: The strengthened mutual induction principle.

To define a type, one has to show that the type is inhabited, and the purpose of the proof is exactly to construct such a term for the type. Till here, we have seen a non-trivial example of how to define an induction principle for a nested (and in addition, mutually) inductive type.

There has been some metaprogramming effort to generate nested induction hypotheses in Coq automatically using the MetaCoq platform (Liesnikov, Ullrich, and Forster 2020), however, the tool was not mature enough and has not been updated since some previous versions.

6.2 Well-founded recursion

Recursion with a single recursive constructor and one (or more) base cases are well-known, such as that of factorial or Fibonacci numbers. One can naturally extend the definition to structural recursions, such as that on an abstract syntax tree when writing programming language implementations. However, when the recursive case is not a strict ”predecessor” of the current case, we need a stronger form of recursion. For the case of linearly ordered datatypes such as natural numbers, we can argue using strong induction; however, for other datatypes where there isn't a canonical linear order, such as, again, rose trees (6.1.2).

Recursing on the items in the list of rose trees is essential, but to Coq they are not direct subterms of the tree while the argument `xs` is. We are left with the difficult choice of convincing the positivity check for fixpoints in Coq that our recursion terminates.

Fortunately, this can be solved by well-founded recursion, a more general form of recursion that determines termination by the following:

1. defining a linear order on the recursive term, called a *measure*.
2. showing that this measure is lower-bounded.

Then the recursion is well-founded and will terminate. The use case of this in the project is to recurse through a module and resolve the fully qualified name (e.g. `M.N.a`) from the nested module structure and their identifier names (e.g. `M, N, a`). For tree-like structures, a typical measure is the *height* of the tree:

```
415 Equations alt_size_sf (sf: structure_field) : nat :=
416   | ConstantDecl _ := 1;
417   | InductiveDecl _ := 1;
418   | ModuleDecl mi mt := 1 + (max (alt_size_mi mi) (alt_size_sb mt));
419   | ModuleTypeDecl mt := 1 + (alt_size_sb mt);
420 where alt_size_sb (sb: structure_body) : nat :=
421   | nil := 0;
422   | (hd::tl) := alt_size_sf hd.2 + alt_size_sb tl;
423 where alt_size_mi (mi: module_implementation) : nat :=
424   | mi_struct s := alt_size_sb s;
425   | _ := 0.
```

Listing 6.2.1: Height defined on structure body.

Then we show this measure is lower-bounded:

```
427 Lemma alt_size_sf_ge_one: (forall sf: structure_field, 0 < alt_size_sf sf).
428 Proof.
429   destruct sf; simp alt_size_sf; lia.
430 Qed.
```

Listing 6.2.2: Proof of lower bound of the height measure.

Therefore, we can define a well-founded recursion using this measure, specifically mentioned on line 433:

```
Equations paths_of_structure_field (sf: structure_field) (prefix: list ident)
  : list (list ident) by wf (alt_size_sf sf) lt :=
  | ConstantDecl _, p := [p];
```

...

```
  | ModuleTypeDecl (hd::tl), p :=
    (paths_of_structure_field hd.2 (p++[hd.1])) ++
    (paths_of_structure_field (ModuleTypeDecl tl) p).
```

```
Defined. Next Obligation.
  pose proof (alt_size_sf_ge_one s).
  simp alt_size_sf. simpl. lia.
Defined.
```

Listing 6.2.3: Defining a well-founded recursive function using the new measure.

6.3 Strengthening the induction hypothesis

Sometimes, the induction hypothesis of a statement is too weak and entirely useless. For example, suppose we wish to show the following:

Theorem 6.3.1. *Fix a natural number x. Then for all natural numbers $n \geq 1$, $x + n \neq x$.*

Wrong proof. Let us attempt a direct induction.

Base case: suppose $n = 1$, $x + n = n + 1 = succ(x) \neq x$ by definition.

Inductive case: suppose any $n \geq 1$, $x + n \neq x$. Then

$$x + (n + 1) = (x + n) + 1 = succ(x + n)$$

Now, we wish to say

$$x + n \neq x \implies succ(x + n) \neq x$$

which is not true in general! ■

The key here is that the induction hypothesis of "not equal" is too weak and does not help with proving the inductive case. Should it have been the strict inequality instead, $x + n > x$ which implies $x + n \neq x$, we would have been fine!

Put in context, we wanted to show that for every declaration with kername `kn'` in a module named `kn`, can not have the same name as the module after translation.

```
502 Lemma translated_module_decl_all_kn_neq:
503 forall m kn, Forall (fun '(kn', _) ⇒ kn <> kn') (trans_module_decl kn m).
```

Listing 6.3.1: A module cannot have the same kername as its contents.

The direct induction proved to be a failure. However, we can prove instead a stronger statement:

```
350 Lemma translated_structure_field_all_kn_extends:
351 forall sf kn id, Forall (fun '(kn', _) ⇒ kn_extends kn kn')
352   (trans_structure_field kn id sf).
```

Listing 6.3.2: Every declaration within a module must have a kername extending that of the module.

Since the binary relation on kernames, `kn_extends kn kn'` decides whether `kn'` is an extension of `kn` and is a strict partial order, it would imply the inequality in 6.3.1. Then, it remains to define it in Coq and show the asymmetry and transitivity of this relation. We omit the proof here.

```
152 Inductive kn_extends : kername -> kername -> Prop :=
153 | kn_extends_one kn id : kn_extends kn (kn_append kn id)
154 | kn_extends_step id kn kn' (H: kn_extends kn kn') : kn_extends kn (kn_append kn' id).

190 Lemma kn_extends_trans : forall kn kn' kn'',
191 kn_extends kn kn' -> kn_extends kn' kn'' -> kn_extends kn kn''.

239 Lemma kn_extends_irrefl : forall kn, ~(kn_extends kn kn).
```

Listing 6.3.3: Definition, asymmetry and transitivity of the relation.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this paper, we looked at Coq's module system and gave two possible implementations of non-parametrized Modules in the MetaCoq project, by the means of elaborating the module structures away when translating from TemplateCoq to PCUIC. In the first implementation, we verified several niceness properties about the implementation. Inspired by the difficulty of verifying the translation from TemplateCoq to PCUIC, we have proposed another implementation that unifies the Coq global environment with modules with initial implementations and ideas. Along the way, we also noted down the proof-theoretic skills acquired with nested inductive types, well-formed recursion, and the strengthening of induction hypotheses for future references.

7.2 Future Work

We list a few directions for possible future work:

7.2.1 Modular Environment Rewrite

The (experimental) modular rewrite of the environment is incomplete due to its complexity and the time required to troubleshoot Coq errors and acquire the machineries

mentioned in Chapter 6. An immediate, possible future work is to continue the work done in Chapter 5, with implementation and verification order similar to that of Chapter 4. Verification of the TemplateCoq to PCUIC translation under the new implementation, which should be easier due to the lack of the double-fold problem (4.5.5).

7.2.2 Functors, and higher-order functors

In Section 2.2, we briefly studied the history of module implementations, and the behavior of functors and their semantics have been a non-trivial topic. Even in Coq, the OCaml implementation has encountered consistency-threatening bugs from functors [1], a sign of complexity in this area. However, functors are arguably the most interesting part of modules that allow massive abstraction and generalization that is worth looking into.

7.2.3 More Typing Rules

More of a documentation effort, the state of typing rules specifically for modular environments (as implemented in the OCaml implementation) perhaps needs to be re-studied with the recent paper of Soubrian after the improvements since his Ph.D. There are some differences between the typing rules of the official documentation, Chrąszsz's and Soubrian's Ph.D. thesis, which might all be different from that of the official implementation. This is not only important for contributors and maintainers of the future, but will also facilitate potential academic explorations of module implementations.

[1] such as `https://github.com/coq/coq/issues/15838`