My Document

Tan Yee Jian

March 26, 2023

Abstract

This is an abstract.

Contents

Chapter 1

Introduction

1.1 Why Theorem Provers?

Deductive, logical proofs have long been a way for humans to deduce facts and truths. However, after several centuries of development, mathematics have 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 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 (CHECK Voevodsky) to different extents, the field is now in need of a 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, testing can only show the presence of bugs, but not the absence of it, to ensure our programs are absolutely 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 theorem provers are thus born. In its core, it takes a mathematical claim, and checks the user's proofs 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 a ITP that implements the Polymorphic Cumulative Calculus of Inductive Constructions (PCUIC), 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 and the Feit-Thompson theorem, and was awarded the ACM Software System Award in 2014. In 2021, the CompCert C compiler, a industrial-strength compiler verified in Coq has been awarded the

ACM Software award, 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? While proof assistants check for the correctness of proofs and programs (yes, they are equivalent under the Curry-Howard Correspondence), 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 mathematically ”nice”, such as strongly normalizing (and hence decidable), 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, meaning that it is possible to prove *False* in Coq!

One way to remedy this situation is to have a verified implementation for Coq itself - possibly in Coq! 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 for a perfect environment to argue about various properties of Coq terms, specifically all the correctness properties of the underlying PCUIC system. Then, an 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 definitely 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?

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 a conservative one), 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, Module System, Template Polymorphism and Proof-Irrelevant Propositions. 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 My Contributions

In this project, I have implemented a non-parametrized module system in the MetaCoq project and verified various properties about 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 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: I will start by a review of previous related works about the implementation of Modules in Coq and relevant systems. Then, I will explain the syntax and semantics of Coq modules, before describing my implementation of Modules and the verification of its properties, the core of this project. After that, I will describe a second implementation of Modules in Coq, which I call the Modular Environment rewrite that solves various problems surfaced from the initial implementation. Finally, I will document the Coq-specific, interactive-proving related skills I learnt during the project. This report will end with possible future work; while some technical details, such as concrete typing rules, and some type-theoretic results I studied as a joint thesis with the Mathematics Department, will be put in the appendix.

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, 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.

Building on the idea of Courant, Jacek Chrząszcz 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 Chrząszcz 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 Chrząszcz's implementation, and the expression of modules in Coq is restricted only to module paths. The core of Chrząszcz'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.

[1]PTS can be seen as a generalization of Barendregt's Lambda Cube, by defining type systems using a triple $(\mathscr{S}, \mathscr{A}, \mathscr{R})$ representing 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.

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[2] and Harper et. al. in 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 [1] 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[3]. Another related project is CakeML [5], a verified subset of the SML language, but unfortunately does not include the verification of the module system.

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.

Chapter 3

The Module System of Coq

This chapter aims to explain the syntax and semantics of the module system via a generic description. Since the module system is well extended to integrate with other extra-logical parts of Coq such as pretty printing, notation, and hint databases, which are beyond the scope of this project, I will explain only a relevant subset of the module language in this section. For a more precise definition of modules and related structures, please refer to Coq: Modules.

3.1 Abstract Syntax of Coq Modules

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
  - an assumption,
    ```
    Definition a: bool.
    ```
  - a constant definition,
    ```
    Definition b: bool := true.
    ```
  - an inductive definition,
    ```
    Inductive nat :=
    | O
    | S (n: nat).
    ```
  - or a **module**, a **module type**, or a **functor**, recursively.
- A **module type** is a structure given a name. A **module** is also structure given a name, but all assumptions must be proven valid; that is, an element of the declared type is inhabited. When declaring a **module**, one can explicitly give a **module type**.

- A **module alias** is the association of a short name to an existing module.
- A **functor** is a module defined with a parameter with a binder and a required type for the module supplied as an argument.

3.1.1 Conversion of Coq Terms

To understand Coq Modules, we need to first understand the basic structure of Coq. The core object in the language of Coq are terms. Terms of a type correspond to a proof for a theorem as in the Curry-Howard correspondence. The syntax and semantics of Coq terms are as explained by the syntax, [1] conversion (including reduction and expansion) [2] and typing [3] rules 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).

3.1.2 Modules as second-class objects

However, Coq modules are not first class objects of the language; meaning that they are on another axis of the language and interacting with the core language (consisting of terms) in limited ways only. This means that . 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 semantics of the current implementation of Coq Modules and formalize them at the level of TemplateCoq. Section 3.2 describes the semantics behind plain modules and aliased modules, without functors.

3.2 Semantics of Modules

From now onwards, we consider only non-parametrized modules.

There are two operations involving modules: how to define a module and how to use a module. We will specify the behavior, implementation and proof obligations below.

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

Modules are containers for definitions that allow reuse. Definitions in Coq are stored in a Global Environment. We first look at the structure of Global Environment:

3.2.1 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 kername, second a pathname, and finally, the definition object. Canonical kernames can be though of as unique labels, and for the ease of understanding, as natural numbers 1, 2, 3 etc. The pathname is a name which the user gives to the definition; it is of the form of a dot-separated string, such as $M.N.a$. Finally, the definition object can be:

- A **constant definition** to a Coq term, such as a lambda term, application term, etc..
- An **inductive definition** of a type.
- A **module definition**. We consider a module to be inductively defined as a list of constant, inductive, module or module signature definitions. Alternatively, it can also be an alias to a previously defined module (which may be an alias).
- A **module signature definition** has the same structure as a module definition, but instead of concrete definitions, it only specifies a name and a type for each entry. It can be also an alias to a previously defined module signature (which may be an alias).

The terms "module type" and "module signature" are used interchangeably.

3.2.2 Plain Modules

Behavior and Implementation

Modules can be thought of as a named global environment where the definitions within it are namespaced by the module name. Therefore, 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 pathname and a kername.

Therefore, implementation wise, one need to ensure the correctness of "referring to definition"; that is, when a definition within a module is referred to by its pathname $M.N.a$, it will be fetched correctly from the table.

Proof obligation

We say the implementation of such a module is correct if the meta-theory 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) during its definition.
2. Define the behavior 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. The details of (2) are described in Section 3.2.4.

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.

3.2.3 Aliased Modules

Aliased modules are just a renaming of existing modules, which can be seen as syntactic sugar for modules. Therefore, the correctness depends only on implementing this internal referencing correctly.

Behavior and Implementation

Suppose we have

```
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* $M.X$). In the OCaml implementation, all definitions in M can now be referred to by the pathnames $N.X$ in addition to $M.X$, while still having the same kername.

Proof Obligations

1. Well-definedness: aliasing can only occur for well-defined modules. There cannot be self-alias and forward aliasing (aliasing 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.

3.2.4 Using Modules

As mentioned, the only way modules are used is during reduction or conversion of a Coq term. In Coq, reduction and conversion are made up of smaller reduction rules, such as $\beta, \delta, \zeta, \eta, \iota$ reductions. In particular, Modules are related only to δ reductions, which "replaces a defined variable with its definition" [4].

The correctness of δ-reduction and conversion is a meta-theoretic property, which is already shown to be correct and have properties such as normalization, confluence etc. in PCUIC [4]. I will contribute by expanding the definition of delta-conversion and expand the existing proofs in the MetaCoq project that such properties continue to hold.

[4]Coq: Conversion

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 first entry an identifier, and second entry a `structure_field`. A `structure_field` can then be declarations of the following form: for a constant, a mutually inductive type, a module, and a module type.

```
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.
```

`module_implementation` here represents the four ways of defining a module, respectively,

1. an abstract module declaration usually used in module signatures for nested modules;
2. an algebraic module expression, most commonly a form of functor application. Here, we consider only module aliasing (nullary functor application).

3. a module declaration given with an explicit module type. The `structure_body` argument will contain the concrete implementation given.
4. a module declaration without an explicit module type; in other words, the module has the 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:

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

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`:

```
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.
```

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 I will merely mention it.

```
Definition global_declarations := list (kername * global_decl).

Record global_env := mk_global_env
  { universes : ContextSet.t;
    declarations : global_declarations;
    retroknowledge : Retroknowledge.t }.
```

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.

```
Lemma declared_module_lookup {Σ mp mdecl} :
  declared_module Σ mp mdecl ->
  lookup_module Σ mp = Some mdecl.
Proof.
  unfold declared_module, lookup_module. now intros ->.
```

```
Qed.

Lemma lookup_module_declared {Σ kn mdecl} :
  lookup_module Σ kn = Some mdecl ->
  declared_module Σ kn mdecl.
Proof.
  unfold declared_module, lookup_module.
  destruct lookup_env as [[]|] => //. congruence.
Qed.
```

The same is done for Module Types as well.

4.2 Typing Modules

To ensure a module, or a module type is well-defined, 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)
```

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. Since the checking of well-typedness 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_*` here 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 on 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 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.
```

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, and as we will see later, this will be the main challenge of this project.

```
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)).
```

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 kernel name `kername` and declaration `d` consists of well-typed data (`on_global_decls_data`) with respect to the prefix Σ:

```
(** Well-formed global environments have no name clash. *)
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
    }.
```

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).
```

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 exist a proof tree showing the well-typedness of a term, there doesn't exist another proof tree that show otherwise), it is entirely what it is made to be. Therefore, the main place to verify these properties are at the typing of terms, which I 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 with respect to different predicates:

```
Lemma on_global_decl_impl {cf : checker_flags} Pcmp P Q Σ kn d :
  (forall Γ t T,
    on_global_env Pcmp P Σ.1 ->
    P Σ Γ t T -> Q Σ Γ t T) ->
  on_global_env Pcmp P Σ.1 ->
  on_global_decl Pcmp P Σ kn d -> on_global_decl Pcmp Q Σ kn d.
Proof.
intros H HP.
cut ((forall (m : module_implementation),
        on_module_impl Pcmp P Σ m -> on_module_impl Pcmp Q Σ m)
  × (forall (p : structure_field),
        on_structure_field Pcmp P Σ p -> on_structure_field Pcmp Q Σ p)
  × (forall (s : structure_body),
        on_structure_body Pcmp P Σ s -> on_structure_body Pcmp Q Σ s)).
intro md_sf_sb.
destruct d; simpl.
- now eapply on_constant_decl_impl.
- now eapply on_inductive_decl_impl.
- unfold on_module_decl. destruct m as [mi mt]; simpl.
  intros [onmip onmtp]. split; now apply md_sf_sb.
- apply md_sf_sb.
- apply (on_mi_sf_sb_mutrect); try now constructor.
  -- intros c oncp. constructor.
     now apply on_constant_decl_impl with (Pcmp := Pcmp) (P := P).
  -- intros kn' mind onip. apply on_sfmind with (kn := kn').
     now apply on_inductive_decl_impl with (Pcmp := Pcmp) (P := P).
Qed.
```

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 case, which are again the constant and the inductive case.

Since the above lemma is true across all declarations, it is natural to think that that this functoriality should extend to lists of global declarations as well. That is indeed the case, this time we omit the proof, which uses the above lemma in its core.

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

4.4 Typing of Terms

Since I modified the environment to include module definitions, this also directly affects the properties about 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:

```
Inductive term : Type :=
| tRel (n : nat)
| tVar (id : ident) (* For free variables (e.g. in a goal) *)
| tEvar (ev : nat) (args : list term)
| tSort (s : Universe.t)
| tCast (t : term) (kind : cast_kind) (v : term)
| tProd (na : aname) (ty : term) (body : term)
| tLambda (na : aname) (ty : term) (body : term)
| tLetIn (na : aname) (def : term) (def_ty : term) (body : term)
| tApp (f : term) (args : list term)
| tConst (c : kername) (u : Instance.t)
| tInd (ind : inductive) (u : Instance.t)
| tConstruct (ind : inductive) (idx : nat) (u : Instance.t)
| tCase (ci : case_info) (type_info:predicate term)
        (discr:term) (branches : list (branch term))
| tProj (proj : projection) (t : term)
| tFix (mfix : mfixpoint term) (idx : nat)
| tCoFix (mfix : mfixpoint term) (idx : nat)
| tInt (i : PrimInt63.int)
| tFloat (f : PrimFloat.float).
```

each having its own typing rule (relation between a term and its type). The typing relation can be found as a 122-line inductive type in the project. For brevity, we show here its ”signature” and notation:

```
Inductive typing `{checker_flags} (Σ : global_env_ext) (Γ : context)
: term -> term -> Type
```

...

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

Where its 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

One of the key lemma here, `env_prop`, is given as follows:

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

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.
```

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 show such a strong statement 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 hypothesis:

```
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),

   (forall Σ (wfΣ : wf Σ) (Γ : context) (wfΓ : wf_local Σ Γ),
    All_local_env_over typing Pdecl Σ Γ wfΓ -> PΓ Σ Γ wfΓ) ->

   ...

   (forall Σ (wfΣ : wf Σ) (Γ : context) (wfΓ : wf_local Σ Γ) (t A B : term) s,
      PΓ Σ Γ wfΓ ->
      Σ ;;; Γ |- t : A ->
      P Σ Γ t A ->
      Σ ;;; Γ |- B : tSort s ->
      P Σ Γ B (tSort s) ->
      Σ ;;; Γ |- A ≤ B ->
      P Σ Γ t B) ->

     env_prop P PΓ.
Proof.
```

```
    ...
Qed.
```

In its proof, the module typing rules are used extensively since we are showing properties of global environments as well. We omit the almost 400 lines of proofs and leave the details to the interested reader.

4.5 Translation to PCUIC

Since TemplateCoq has modules while PCUIC does not, to translate to PCUIC, we need to elaborate modules away via the natural elaboration - store 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:

```
Inductive global_decl :=
| ConstantDecl : constant_body -> global_decl
| InductiveDecl : mutual_inductive_body -> global_decl.
Derive NoConfusion for global_decl.

Definition global_declarations := list (kername * global_decl).

Record global_env := mk_global_env
  { universes : ContextSet.t;
    declarations : global_declarations;
    retroknowledge : Retroknowledge.t }.
```

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:

```
Fixpoint trans_structure_field kn id (sf : Ast.Env.structure_field) :=
  let kn' := kn_append kn id in
  match sf with
  | Ast.Env.sfconst c ⇒ [(kn', ConstantDecl (trans_constant_body c))]
  | Ast.Env.sfmind m ⇒ [(kn', InductiveDecl (trans_minductive_body m))]
  | Ast.Env.sfmod mi sb ⇒ match mi with
    | Ast.Env.mi_fullstruct ⇒ trans_structure_body kn' sb
    | Ast.Env.mi_struct s ⇒ trans_structure_body kn' s
    | _ ⇒ trans_module_impl kn' mi
    end
  | Ast.Env.sfmodtype _ ⇒ []
  end
```

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. For example, in the example below:

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

   We have `kn := "M", id := "a"`, and thus `kn' := "M.a"`.

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

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

Definition trans_modtype_decl := trans_structure_body.

Definition trans_module_decl kn (m: Ast.Env.module_decl) : list(kername × global_decl) :=
  match m with
  | (Ast.Env.mi_fullstruct, mt) ⇒ trans_modtype_decl kn mt
  | (mi, _) ⇒ trans_module_impl kn mi
  end.
```

Once we are done with the translation of modules, we can translate the global declaration(s):

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

Which is analogous to that of structure fields. The interesting part is the double fold in the translation of global declarations, which is actually a flatmap across `trans_global_decl`, since now each TemplateCoq global declaration can translate into multiple PCUIC global declarations, thanks to the tree structure of modules.

```
Definition trans_global_decls env (d : Ast.Env.global_declarations) : global_env_map :=
  fold_right (fun decl Σ' ⇒
    let decls := (trans_global_decl Σ' decl) in
    fold_right add_global_decl Σ' decls) env d.
```

Properties on Translation

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

```
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).
Proof.
  intros wf ty.
  apply (ST.env_prop_typing template_to_pcuic); auto.
  now eapply ST.typing_wf_local.
  now apply template_to_pcuic_env.
Qed.
```

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)).
```

The statement here is abstracted by the `env_prop` (TODO: link me to section 4.4), 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 we have need to "sanity" checks on translation to ensure that environments are well-translated. This property can be formulated as

1. "non-existence is preserved under translation", that is, the translated environment should only contain the intended translation and nothing more, and its dual;

2. "existence is preserved under translation", that is, nothing is lost in translation.

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

```
222 Lemma trans_lookup_env {cf} {Σ : Ast.Env.global_env} cst {wfΣ : Typing.wf Σ} :
223   match Ast.Env.lookup_env Σ cst with
224   | None ⇒ lookup_env (trans_global_env Σ) cst = None
225   | Some d ⇒ match d with
226     | Ast.Env.ConstantDecl _ | Ast.Env.InductiveDecl _ ⇒
227       ∑ (Σ' : Ast.Env.global_env) (d': global_decl),
228         [× Ast.Env.extends_decls Σ' Σ,
229           Typing.wf Σ',
230           wf_global_decl (Σ', Ast.universes_decl_of_decl d) cst d,
231           extends_decls (trans_global_env Σ') (trans_global_env Σ),
232           trans_global_decl (trans_global_env Σ') (cst, d) = (cst, d')::[] &
233           lookup_env (trans_global_env Σ) cst = Some d']
234     (** Modules are elaborated away. *)
235     | Ast.Env.ModuleDecl _ | Ast.Env.ModuleTypeDecl _ ⇒ lookup_env (trans_global_env Σ) cst = N
236     end
237   end.
```

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

1. 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. 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 228-234). 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 process of verification the only remaining part, the TemplateCoq to PCUIC translation proved to be time consuming, resulting me in spending weeks without writing any `QED`s. This is because in the process of verifying those properties, I faced many technical challenges in constructing their proofs. Fortunately, two beneficial events ensued:

1. After identifying the pain points in the proofs, I found out the decision that led to this situation and in tackling it, gave rise to this more natural solution which requires a complete rewrite. Although incomplete, I have designed this rewrite and implemented a part of it as far as writing out the environment typing rules. I will discuss this implementation in the firs section.

2. In the process of rewrite, due to the design choices and its complexity, some non-trivial proof machineries such as nested mutual inductive types with their induction hypotheses, well-founded recursion, strengthening induction hypothesis were required. Since a number of them are direct consequence of this modular environment rewrite, I will include them as a section at the end of this chapter as a summary.

5.1 Difficulties on the Global Declaration List

The proof for (TODO: link `trans_lookup_env`) proved to be tedious due to the double fold. In each fold, one adds definitions to the environment using an (rightfully) 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.

```
Proof.
  destruct Σ as [univs Σ retro].
```

```
240   induction Σ.
241   - cbn; auto.
242   - unfold Ast.Env.lookup_env. cbn -[trans_global_env].
243     destruct eq_kername eqn:eqk.

    ...

307       --- (** a.2 is a *)
308         unfold trans_global_env. subst Σmap'; simpl.
309         destruct a as [kn d]; simpl in *.

    ...

316         (** proving assertion by mutual induction *)
317         * subst P P0 P1. apply Ast.Env.sf_mi_sb_mutind => //=.
318           ** cbn. intros c id.
319             pose proof (kn_appended_not_eq kn id).

    ...

326             *** simpl in *. subst.
327               destruct o. apply ST.fresh_global_iff_lookup_global_None in kn_fresh.
328               rewrite kn_fresh in h. inversion h.
```

After spending weeks trying different methods to make this work, including an elaborate argument defining a strict partial order on kernames (such as the result used on line 319), I decided to venture for a new solution. The source of the problem here is the double-fold, 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:

```
324   Inductive structure_field :=
325   | sfconst : constant_body -> structure_field
326   | sfmind : mutual_inductive_body -> structure_field
327   | sfmod : module_implementation -> structure_body -> structure_field
328   | sfmodtype : structure_body -> structure_field

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.
```

it is thus natural to link both of them together.

5.2 The Modular Environment

It would be beneficial to unify these two with the same 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 that of modules, and thus giving us a good sense in defining the typing rules for modules.

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 : list (ident × structure_field) -> module_implementation (** Module M:T. ... End
| mi_fullstruct : module_implementation (** Module M. ... End M.*).

Notation structure_body := (list (ident × structure_field))%type.
```

Structure field now subsumes global declaration, and we note that structure body is just a convenient name for the list of structure fields indexed by its name - an identifier. The possible nested structure body inside structure fields are what gives 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.
```

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 of a structure body.

5.3 Tactics and Proof Machineries

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

Bibliography

[1] Derek Dreyer, Robert Harper, and Karl Crary. "Understanding and Evolving the Ml Module System". AAI3166274. PhD thesis. USA, 2005. ISBN: 0542015501.

[2] David MacQueen. "Modules for standard ML". In: *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming.* 1984, pp. 198–207.

[3] 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. ISBN: 9781605588919. DOI: `10.1145/1708016.1708028`. URL: `https://doi.org/10.1145/1708016.1708028`.

[4] Matthieu Sozeau et al. "Coq Coq Correct! Verification of Type Checking and Erasure for Coq, in Coq". In: *Proc. ACM Program. Lang.* 4.POPL (2019). DOI: `10.1145/3371076`. URL: `https://doi.org/10.1145/3371076`.

[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. ISBN: 9781450342735. DOI: `10.1145/2897336.2897344`. URL: `https://doi.org/10.1145/2897336.2897344`.