Formalizing Coq Modules in the MetaCoq project

XFC4101 CA Report

Yee-Jian Tan

March 6, 2023

Contents

Introduction 1.

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

DBLP:journals/corr/BauerGLSSS16

jung2018iris

The MetaCoq project [3] is therefore started by the Gallinette Team in INRIA, to "formalize Coq in Coq" and acts as a platform to interact with Coq's terms directly, in a verified manner; an immediate application is to verify the correctness of the implementation of Coq. This also reduces the trust in the implementation of Coq to the correctness of the theories underlying Coq, moving from a "trusted code base" to a "trusted theory base". In 2020, the core language of Coq, minus a few features are already successfully verified in Coq [**coqcoqcorrect**]. However, there are still a few missing pieces not yet verified, among them the Module system of Coq. The Module system, although not part of the core calculus of Coq, is an important feature for Coq developers to develop in a modular fashion, providing massive abstraction and a suitable interface for reusing definitions and theorems.

coqcoqcorrect

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

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

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

[1] `https://awards.acm.org/software-system`

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

[3] `https://metacoq.github.io`

2. Types and Proof Assistants

2.1. What are proof assistants?

Proof assistants are a special kind of computer program whose job is to verify the correctness of a mathematical proof. Informally, it has to verify if a conclusion can be made by applying a fixed set of logical deduction steps on a set of fixed assumptions. There are several ways to implement this; modern implementations of proof assistants usually utilizes theories such as Higher Order Logic (e.g. Isabelle, HOL Light), or some form of Type Theory.

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

2.2. Type Theory

Types were first invented by Bertrand Russell as a way to provide higher-order structures than sets to overcome his famous Russell's paradox. However, it was Alonzo Church's Lambda Calculus which when used with types (Simply Typed Lambda Calculus, or STLC) exhibited many desirable properties. STLC turned out to be a consistent formal system, and a rewriting system that exhibits strong normalization, which means the simplification of STLC terms will always terminate, and always tends towards a unique normal form.

Most importantly, type systems such as STLC exhibits the Curry-Howard Correspondence, which is the crucial observation that allows Type Theories to be useful for

2.3. The Curry-Howard Correspondence

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

Intuitionistic Propositional Logic	Type Theory
Proposition	Type
Proof	Term

What are those exactly?

2.3.1. Intuitionistic Propositional Logic

There are two parts: Intuitionistic, and Propositional Logic. Let us start with Propositional Logic.

Propositional Logic

Propositional logic, (sometimes *0-th order logic*) is a logical system where propositions are constructed with variables and logical connectors, such as → (implication), ∨ (disjunction), ¬ (negation), ∧ (conjunction) and more, but not including quantification (∃ (exists), ∀ (for all)). For example, propositions in Propositional Logic might look like the following:

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

In **Classical** Propositional Logic, one can assign a valuation to each variable, either 1 or 0, which represents ”True” or ”False” informally, and extend the definition of valuation to propositions in a “natural” fashion:

$$\begin{aligned} \tilde{v}(p) &= v(p) && \text{if } p \text{ is a variable} \\ \tilde{v}(p \wedge q) &= 1 && \text{iff } \tilde{v}(p) = 1 \text{ and } \tilde{v}(q) = 1 \\ \tilde{v}(p \vee q) &= 1 && \text{iff } \tilde{v}(p) = 1 \text{ or } \tilde{v}(q) = 1 \end{aligned}$$

And so on. If there is some valuation that result in the proposition result in 1, then we say the proposition is *satisfiable*; if there is none, i.e. all valuations will result in the proposition being 0, we say the proposition is a *contradiction*; on the other hand, if any valuation will result in 1, the proposition is a *tautology*.

In particular, in the above example,

1. $p \vee q$ is satisfiable by $v(p) = 1, v(q) = 0$,
2. $p \wedge \neg p$ is a contradiction, and
3. $(p \to q) \to (\neg q \to \neg p)$ is a tautology.

Classical Logic

Logical systems can be either intuitionistic or classical, based on one fact: whether they accept the *Law of Excluded Middle* as an axiom. Precisely, it says

Axiom 2.3.1 (Law of Excluded Middle) For any proposition p, either p or $\neg p$ holds.

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

Theorem 2.3.1 *There exists irrational numbers a, b where a^b is rational.*

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

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

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

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

is rational. □

Even with the proof, we still don't have a concrete pair of irrational numbers (a, b) fulfilling the property, but the conclusion is declared to be logically sound in classical logic. Intuitionistic logic, on the other hand, rejects this rule, leading to a logical system where to prove something exists, it requires an algorithm for the explicit construction of such object.

2.3.2. The Correspondence

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

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

2.3.3. From STLC to PCUIC

barendregt1992lambda
girard1989proofs

coquand1986calculus

The Simply Typed Lambda Calculus (STLC) is a simple construction; we do not define it here but refer to many classical textbooks such as [**barendregt1992lambda**] and [**girard1989proofs**]. It is shown to be equivalent to Intuitionistic Propositional Logic under the Curry-Howard Correspondence. Furthermore, stronger type systems such as the Calculus of Constructions (CoC) [**coquand1986calculus**] extend the Curry-Howard correspondence to proofs in the full intuitionistic predicate logic (or *1-st order logic*), allowing proofs of quantified statements. The Polymorphic Cumulative Calculus of co-Inductive Constructions (PCUIC) is a further extension of the CoC to include co-inductive types, polymorphic universes and cumulativity, which we will not explain in this paper and instead refer interested readers to the paper [**pcuic2017timany**].

pcuic2017timany

3. Introduction to the MetaCoq Project

3.1. The MetaCoq Project

sozeau2020metacoq

coqcoqcorrect

MetaCoq is a project to formalize the core calculus, PCUIC, in Coq, and become a platform to write tools that can manipulate Coq terms [**sozeau2020metacoq**]. The effort was complete for a large part of the core language of Coq [**coqcoqcorrect**], with a few missing pieces :

- Eta-conversion
- Template Polymorphism
- SProps
- Modules

I will be tackling the last.

3.2. Structure of the MetaCoq Project

The MetaCoq project is an ambitious project aiming to provide a verified implementation of Coq, and for its size, it is reasonably split into a few main components. From the layer closest to the Coq language to layer closest to machine code, we have: TemplateCoq (Section 3.2.1), PCUIC (Section 3.2.2), followed by Safe Checker, Erasure and beyond(Section 3.2.3).

Let us remind ourselves of the task of MetaCoq: we would like to see that the OCaml representation of Coq is indeed correct and preserves the desired properties of the underlying theory. Since Coq has added many “bells and whistles” for its users, the terms of Coq definitely is much more complex than its underlying, Platonic type theoretical form. Therefore, MetaCoq has several stages for a Coq term to go through, stripping down to the bare minimum through the following few stages.

3.2.1. TemplateCoq

TemplateCoq is a quoting library for Coq: a Coq program that takes a Coq term, and constructs an inductive data type that correspond to its kernel representation in the OCaml implementation. This is the first layer of the stripping of a Coq term, where the structures associated with a term, such as the Global Environment under which it is defined, are preserved properly as in the kernel.

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.

3.2.2. 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.” [1]. In other words, it is a type theory that is as powerful as Coq can express, having the good properties such as weakening, confluence, principality (that every term has a principal type) etc. [**coqcoqcorrect**].

[1] from https://metacoq.github.io

coqcoqcorrect

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.

3.2.3. Safe Checker, Erasure and Beyond

The core semantic operation of type theories are the 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 converted into a PCUIC term, and eventually has 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 [**coqcoqcorrect**].

coqcoqcorrect

4. The Coq Proof Assistant

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

4.1. An example of a Coq program

This is a simple example featuring the definition of a type "nat", a definition of a constant of type "nat", namely "zero", and a function "plus" which takes two "nat"s and returns its sum.

Finally, thanks to Curry-Howard Correspondence, a proposition is a type; and here a named proposition is postulated. A simple proof is also given, which constructs a term inhabiting that type, hence proving the correctness of this proposition.

Listing 1: A simple Coq program.

```
Inductive nat: Set :=
| O
| S : nat -> nat.

Definition zero : nat := O.

Fixpoint plus (n m: nat) : nat :=
    match n with
    | S n' => S (plus n' m)
    | O => m
    end.

Proposition zero_is_left_additive_identity : Prop:
    forall n: nat, plus zero n = n.
Proof.
    intro n.
    simpl.
    reflexivity.
Qed.
```

4.2. Coq Modules

Modules in Coq not only allows the reuse of code, it also provides parametrized theories or data structures in the form of functors (or parametrized modules).

In the following example, we show how one can package definitions into named modules for reuse; we also show how we can create parametrized generic data structures. In the final line, a new Magma was created based on the Magma "Nat" by the functor "DoubleMagma".

Listing 2: An example of Modules.

```
Inductive nat :=
| O
| S : nat -> nat.

Fixpoint plus (n m: nat) :=
    match n with
    | S n' => S (plus n' m)
    | O => m
    end.

(* A magma is a set with a binary (closed) operation. *)
Module Type Magma.
    Parameter T: Set.
    Parameter op: T -> T -> T.
End Magma.

Module Type M := Magma.

(* Natural numbers with plus form a magma. *)
Module Nat: Magma.
    Definition T := nat.
    Definition op := plus.
End Nat.

(* A functor transforming a magma into another magma. *)
Module DoubleMagma (M: Magma): Magma.
    Definition T := M.T.
    Definition op x y := M.op (M.op x y) (M.op x y).
End DoubleMagma.

Module NatWithDoublePlus := DoubleMagma Nat.
```

4.3. Towards Specifying the Meaning of Coq Modules

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

There are a few keywords when it comes to Coq Modules:

- A **structure** is an anynomous collection of definitions, and is the underlying construct of modules. They contain **structure elements**, which can be a
  - constant definition
    ```
    Definition a: bool := true.
    ```
  - assumption
    ```
    Axiom inconsistent: forall p: Prop, p.
    ```
  - **module**, **module type**, **functors** recursively.
- A **module** is a structure given a name. It can be defined explicitly to be of a certain **module type**, which a named structure with possibly empty definitions.
- 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.

For a more precise definition of modules and related structures, please refer to Coq: Modules.

4.3.1. 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] 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).

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

4.3.2. Modules as second-class objects

However, Coq modules are not first class objects of the language and do not participate in conversion themselves; i.e. there is no notion to ”reduce” a module. Plain modules in Coq can be treated as a named container of constant and inductive definitions, including possibly nested

[1] Coq: Essential Vocabulary

[2] Coq: Conversion

[3] Coq: Typing

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 4.4 describes the semantics behind plain modules and aliased modules, without functors.

4.3.3. Related Works

Jacek Chrzaszcz’s article in TPHOLs 2003 [**jacek2003**] explains the motivation and choices for the implementation of modules as a second class object and its interaction with terms as described above. Elie Soubiran’s PhD Thesis on Coq Modules [**soubiran**] describes an extension of the Module system which only some features are implemented. The ModuleSystem Wiki Page on Coq’s official Github repository contains valuable information on the usage and design decision of the Module system of Coq, including a list of open issues with Modules. In general, the issues do not compromise the correctness of the type system implemented by Coq; instead, they should be viewed as possible areas of improvements for Coq users.

jacek2003

soubiran

Other slightly useful references include papers that explain modules in the ML family of languages, specifically OCaml and to a certain extent, Standard ML. On this note, Derek Dreyer wrote his PhD thesis [**dreyerphd**] on understanding and extending ML modules, and subsequently on implementing ML modules in its most desirable form, applicative and first-order as a subset of a small type system, F_ω[**f-ing**]. Other notable implementations include that of CakeML [**cakeml**], a verified ML language and Standard ML modules by MacQueen[**macqueen1984modules**]. 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.

dreyerphd

f-ing

cakeml

macqueen1984modules

4.4. 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 behaviour, implementation and proof obligations below.

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:

4.4.1. 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.

4.4.2. Plain Modules

Behaviour 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 behaviour of access of definitions within Modules.

Once these two are done, we can be sure that a Coq program with Modules has all its terms well-defined (by (1)) and enjoys the nice properties of conversion, since the additional terms defined in Modules fulfill (2). This follows as our definition of Modules on the TemplateCoq level, is eventually elaborated down into the PCUIC calculus the idea of modules and aliasing do not exist anymore, they are flattened into the corresponding global environment. The details of (2) are described in Section 4.4.4.

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:

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

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

4.4.3. 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.

Behaviour 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.

4.4.4. 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 [**coqcoqcorrect**]. 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

coqcoqcorrect

5. Project Plan for The Next Semester

There are a few steps to this formalization of Coq Modules that I can foresee:

1. Study the structure and conventions of the MetaCoq project. Since this is a huge project, many common theorems have been abstracted into multiple functors which can generate a theorem given a corresponding object, which I can make use of in my formalization.
2. Add the data structure of Plain Modules into the global environment which is quoted by TemplateCoq. This requires the modification of the current Global Environment data structure containing declarations (minus modules).
3. Ensure the Global Environment remains well-defined with the current typing rules and correctness assertions. The modification in the previous step would require modifications to the existing proofs on the Global Environment (well-formedness, well-typedness) and its related properties.
4. Extend the TemplateCoq plugin (in OCaml) which transforms a Coq term into a TemplateCoq representation, to translate a Coq module into the internal representation of Modules I implemented above. This requires OCaml knowledge specifically in the framework of Coq plugins.
5. Now that syntactical transformation from Coq to TemplateCoq is complete, I need to define how these modules that live in the Global Environment will be used. In particular, I need to implement the ”canonical kername” model as implemented in the Coq kernel to deal with module aliasing and usage of definitions.
6. The modification to the semantics in the previous step will again induce changes to the theorems related to δ-reductions (as explained in 4.4.4), so I have to extend the proofs to fit the new definitions. This should ensure the TemplateCoq formalization of modules is sound.
7. Translate Coq Modules from TemplateCoq to PCUIC by elaboration. In PCUIC, modules will cease to exist and references to definitions in modules will be treated as direct references to plain definitions outside modules. This is done by referring module paths to their canonical paths.
8. The last step in this project is to verify that the translation from TemplateCoq to PCUIC is correct; that is, a PCUIC term is well-formed and well-typed if and only if its translation is well-formed and well-typed in TemplateCoq.

APPENDIX

A. Typing Rules for Coq Modules

Typing rules here are given by Coq: Typing Modules.

A.1. Overview

This appendix is divided into 5 parts. Section A.1 will first define the necessary terminology. Then, sections A.2 to A.5 will define the typing rules of Coq Modules. They correspond to the 4 operations defined on modules:

1. **Formation Rules** (A.2) describes how to form modules.
2. **Evaluation Rules** (A.3) describes how to evaluate specific terms involving modules: with-expressions and path-expressions.
3. **Access Rules** (A.4) describes how to access contents of modules (i.e. definitions in modules) using pathnames.
4. **Typing Rules** (A.5) describes how a module is inductively typed.

A.1.1. Definitions

A structure is … A module type is … FIXME:

A.1.2. Judgement Rules

A.1.3. Variables

E represents environment, S represents structures, …

A.2. Formation Rules

A structure can be formed from the tail of a well-formed environment.

WF-Struct:

$$\frac{\mathrm{WF}(E, E')[]}{E[] \vdash \mathrm{WF}(\mathrm{Struct}\ E'\ \mathrm{End})}$$

If structure S is well-formed under the well-formed environment E, then the environment containing the Module X of type S is well-formed.

WF-Mod1

$$\frac{\mathrm{WF}(E)[] \qquad E[] \vdash \mathrm{WF}(S)}{\mathrm{WF}(E; \mathrm{Mod}(X : S))[]}$$

The previous also works if X is of type S_2, a subtype of S_1.

WF-Mod2

$$\frac{E[] \vdash S_2 <: S_1 \qquad \mathrm{WF}(E)[] \qquad E[] \vdash \mathrm{WF}(S_1) \qquad E[] \vdash \mathrm{WF}(S_2)}{\mathrm{WF}(E; \mathrm{Mod}(X : S_1 := S_2))[]}$$

If the module pointed by p has type S in the global environment E, then we can create an aliased module X to p. This implies that $p \in E$.

WF-Alias

$$\frac{\mathrm{WF}(E)[] \qquad E[] \vdash p : S}{\mathrm{WF}(E; \mathrm{ModA}(X == p))[]}$$

WF-ModType

A ModType is a named well-formed structure.

$$\frac{\mathrm{WF}(E)[] \qquad E[] \vdash \mathrm{WF}(S)}{\mathrm{WF}(E; \mathrm{ModType}(Y := S))[]}$$

WF-Ind

An inductive definition strenghtened by p is well-formed if it is already well-formed in E and it is within p. This can be generalized with subtyping.

$$\frac{\begin{array}{c} \mathrm{WF}(E; \mathrm{Ind}[r](\Gamma_I := \Gamma_C))[] \\ E[] \vdash p : \mathrm{Struct}\ e_1; \ldots; e_n; \mathrm{Ind}[r](\Gamma_I' := \Gamma_C'); \ldots\ \mathrm{End} \\ E[] \vdash \mathrm{Ind}[r](\Gamma_I' := \Gamma_C') <: \mathrm{Ind}[r](\Gamma_I := \Gamma_C) \end{array}}{\mathrm{WF}(E; \mathrm{Ind}_p[r](\Gamma_I := \Gamma_C))[]}$$

A.3. Evaluation Rules

A.3.1. Evaluating `with` expressions

WEval-With-Mod:

$$\frac{\begin{array}{c} E[] \vdash S \to \mathrm{Struct}\ e_1; \ldots; e_i; \mathrm{Mod}(X : S_1); e_{i+2}; \ldots; e_n\ \mathrm{End} \\ E; e_1; \ldots; e_i[] \vdash S_1 \to \overline{S_1} \qquad E[] \vdash p : S_2 \qquad E; e_1; \ldots; e_i[] \vdash S_2 <: \overline{S_1} \end{array}}{\begin{array}{c} E[] \vdash S\ \mathrm{with}\ X := p \to \\ \mathrm{Struct}\ e_1; \ldots; e_i; \mathrm{ModA}(X == p); e_{i+2}\{X/p\}; \ldots; e_n\{X/p\}\ \mathrm{End} \end{array}}$$

WEval-With-Mod-Rec:

$$\frac{\begin{array}{c} E[] \vdash S \to \mathrm{Struct}\ e_1; \ldots; e_i; \mathrm{Mod}(X : S_1); e_{i+2}; \ldots; e_n\ \mathrm{End} \\ E; e_1; \ldots; e_i[] \vdash S_1\ \mathrm{with}\ p := p_1 \to \overline{S_2} \end{array}}{\begin{array}{c} E[] \vdash S\ \mathrm{with}\ X_1.p := p_1 \to \\ \mathrm{Struct}\ e_1; \ldots; e_i; \quad \mathrm{mod}\ X : \overline{S_2}; e_{i+2}\{X.p/p_1\}; \ldots; e_n\{X.p/p_1\}\ \mathrm{End} \end{array}}$$

WEval-With-Def

This is the base case.

$$\frac{\begin{array}{c} E[] \vdash S \to \mathrm{Struct}\ e_1; \ldots; e_i; (c : T_1); e_{i+2}; \ldots; e_n\ \mathrm{End} \\ E; e_1; \ldots; e_i[] \vdash (c := t : T) <: (c : T_1) \end{array}}{E[] \vdash S\ \mathrm{with}\ c := t : T \to \mathrm{Struct}\ e_1; \ldots; e_i; (c := t : T); e_{i+2}; \ldots; e_n\ \mathrm{End}}$$

WEval-With-Def-Rec

$$\frac{\begin{array}{c} E[] \vdash S \to \mathrm{Struct}\ e_1; \ldots; e_i; \mathrm{Mod}(X_1 : S_1); e_{i+2}; \ldots; e_n\ \mathrm{End} \\ E; e_1; \ldots; e_i[] \vdash S_1\ \mathrm{with}\ p := p_1 \to \overline{S_2} \end{array}}{\begin{array}{c} E[] \vdash S\ \mathrm{with}\ X_1.p := t : T \to \\ \mathrm{Struct}\ e_1; \ldots; e_i; \mathrm{Mod}(X : \overline{S_2}); e_{i+2}; \ldots; e_n\ \mathrm{End} \end{array}}$$

A.3.2. Evaluating paths ($p.X$)

WEval-Path-Mod1

$$\frac{\begin{array}{c}E[] \vdash p \rightarrow \text{Struct } e_1; \ldots; e_i; \text{Mod}(X : S[:= S_1]); e_{i+2}; \ldots; e_n \text{ End} \\ E; e_1; \ldots; e_i[] \vdash S \rightarrow \overline{S}\end{array}}{E[] \vdash p.X \rightarrow \overline{S}}$$

WEval-Path-Mod2

$$\frac{\text{WF}(E)[] \qquad \text{Mod}(X : S[:= S_1]) \in E \qquad E[] \vdash S \rightarrow \overline{S}}{E[] \vdash X \rightarrow \overline{S}}$$

WEval-Path-Alias1

$$\frac{\begin{array}{c}E[] \vdash p \rightarrow \text{Struct } e_1; \ldots; e_i; \text{ModA}(X == p_1); e_{i+2}; \ldots; e_n \text{ End} \\ E; e_1; \ldots; e_i[] \vdash p_1 \rightarrow \overline{S}\end{array}}{E[] \vdash p.X \rightarrow \overline{S}}$$

WEval-Path-Alias2

$$\frac{\text{WF}(E)[] \qquad \text{ModA}(X == p_1) \in E \qquad E[] \vdash p_1 \rightarrow \overline{S}}{E[] \vdash X \rightarrow \overline{S}}$$

WEval-Path-Type1

$$\frac{\begin{array}{c}E[] \vdash p \rightarrow \text{Struct } e_1; \ldots; e_i; \text{ModType}(Y := S); e_{i+2}; \ldots; e_n \text{ End} \\ E; e_1; \ldots; e_i[] \vdash S \rightarrow \overline{S}\end{array}}{E[] \vdash p.Y \rightarrow \overline{S}}$$

WEval-Path-Type2

$$\frac{\text{WF}(E)[] \qquad \text{ModType}(Y := S) \in E \qquad E[] \vdash S \rightarrow \overline{S}}{E[] \vdash Y \rightarrow \overline{S}}$$

A.4. Access Rules

A.5. Typing Rules

MT-Eval

$$\frac{E[] \vdash p \rightarrow \overline{S}}{E[] \vdash p : \overline{S}}$$

MT-Str

$$\frac{E[] \vdash p \rightarrow \overline{S}}{E[] \vdash p : S/p}$$

Where S/p is the strengthening operation, defined on S where $S \to \text{Struct } e_1; \ldots ; e_n \text{ End}$ as:

$$\begin{aligned}
(c := t : T)/p &= (c := t : T)\\
(c : U)/p &= (c := p.c : U)\\
\text{Mod}(X : S)/p &= \text{ModA}(X == p.X)\\
\text{ModA}(X == p')/p &= \text{ModA}(X == p')\\
\text{Ind}[r](\Gamma_I := \Gamma_C)/p &= \text{Ind}_p[r](\Gamma_I := \Gamma_C)\\
\text{Ind}_{p'}[r](\Gamma_I := \Gamma_C)/p &= \text{Ind}_{p'}[r](\Gamma_I := \Gamma_C)
\end{aligned}$$

It strengthens the definition of inductives to equal to the current path only, solving the issue of equality of defined inductive types in modules.

A.5.1. Subtyping rules

MSub-Str

$$\frac{\begin{array}{c} E; e_1; \ldots ; e_n[] \vdash e_{\sigma(i)} <: e_i' \text{ for } i = 1, \ldots , m \\ \sigma : \{1, \ldots , m\} \to \{1, \ldots , n\} \text{ injective}\end{array}}{E[] \vdash \text{Struct } e_1; \ldots ; e_n \text{ End} <: \text{Struct } {e'}_1; \ldots ; {e'}_m \text{ End}}$$

Mod-Mod:

$$\frac{E[] \vdash S_1 <: S_2}{E[] \vdash \text{Mod}(X : S_1) <: \text{Mod}(X : S_2)}$$

Alias-Mod:

$$\frac{E[] \vdash p : S_1 E[] \vdash S_1 <: S_2}{E[] \vdash \text{ModA}(X == p) <: \text{Mod}(X : S_2)}$$

Mod-Alias:

$$\frac{E[] \vdash p : S_2 E[] \vdash S_1 <: S_2 E[] \vdash X =_{\beta\delta\iota\zeta\eta} p}{E[] \vdash \text{Mod}(X : S_1) <: \text{ModA}(X == p)}$$

Alias-Alias

$$\frac{E[] \vdash p_1 =_{\beta\delta\iota\zeta\eta} p_2}{E[] \vdash \text{ModA}(X == p_1) <: \text{ModA}(X == p_2)}$$

Modtype-Modtype

$$\frac{E[] \vdash S_1 <: S_2 E[] \vdash S_2 <: S_1}{E[] \vdash \text{ModType}(Y := S_1) <: \text{ModType}(Y := S_2)}$$

Structure element subtyping rules

Assum-Assum

$$\frac{E[] \vdash T_1 \leq_{\beta\delta\iota\zeta\eta} T_2}{E[] \vdash (c\ :\ T_1) <: (c\ :\ T_2)}$$

Def-Assum

$$\frac{E[] \vdash T_1 \leq_{\beta\delta\iota\zeta\eta} T_2}{E[] \vdash (c\ := t\ :\ T_1) <: (c\ :\ T_2)}$$

Assum-Def

$$\frac{E[] \vdash T_1 \leq_{\beta\delta\iota\zeta\eta} T_2 \qquad E[] \vdash c =_{\beta\delta\iota\zeta\eta} t_2}{E[] \vdash (c\ :\ T_1) <: (c\ := t_2\ :\ T_2)}$$

Def-Def

$$\frac{E[] \vdash T_1 \leq_{\beta\delta\iota\zeta\eta} T_2 \qquad E[] \vdash t_1 =_{\beta\delta\iota\zeta\eta} t_2}{E[] \vdash (c\ := t_1\ :\ T_1) <: (c\ := t_2\ :\ T_2)}$$

Ind-Ind

$$\frac{E[] \vdash \Gamma_I =_{\beta\delta\iota\zeta\eta} \Gamma'_I \qquad E[\Gamma_I] \vdash \Gamma_C =_{\beta\delta\iota\zeta\eta} \Gamma'_C}{\mathrm{Ind}[r](\Gamma_I\ := \Gamma_C) <: \mathrm{Ind}[r](\Gamma'_I\ := \Gamma'_C)}$$

Indp-Ind

$$\frac{E[] \vdash \Gamma_I =_{\beta\delta\iota\zeta\eta} \Gamma'_I \qquad E[\Gamma_I] \vdash \Gamma_C =_{\beta\delta\iota\zeta\eta} \Gamma'_C}{\mathrm{Ind}_p[r](\Gamma_I\ := \Gamma_C) <: \mathrm{Ind}[r](\Gamma'_I\ := \Gamma'_C)}$$

Indp-Indp

$$\frac{E[] \vdash \Gamma_I =_{\beta\delta\iota\zeta\eta} \Gamma'_I \qquad E[\Gamma_I] \vdash \Gamma_C =_{\beta\delta\iota\zeta\eta} \Gamma'_C \qquad E[] \vdash p = \beta\delta\iota\zeta\eta p'}{\mathrm{Ind}_p[r](\Gamma_I\ := \Gamma_C) <: \mathrm{Ind}_{p'}[r](\Gamma'_I\ := \Gamma'_C)}$$