Formalizing Coq Modules in the MetaCoq Project

XFC4101 Final Report

Tan Yee Jian

April 17, 2023

National University of Singapore

Outline

Introduction

Contents

Motivation

Coq is a proof assistant for formal verification in Intuitionistic (Constructive) Logic.

Motivation

Coq is a proof assistant for formal verification in Intuitionistic (Constructive) Logic.

Widely used for mathematical proofs (such as the Four-Color Theorem and Feit-Thompson Theorem) and program verification (CompCert C Compiler).

Motivation

Coq is a proof assistant for formal verification in Intuitionistic (Constructive) Logic.

Widely used for mathematical proofs (such as the Four-Color Theorem and Feit-Thompson Theorem) and program verification (CompCert C Compiler).

Implementation of Coq has consistency-threatening bugs!
Who watches the watchers?

Motivation

Coq is a proof assistant for formal verification in Intuitionistic (Constructive) Logic.

Widely used for mathematical proofs (such as the Four-Color Theorem and Feit-Thompson Theorem) and program verification (CompCert C Compiler).

Implementation of Coq has consistency-threatening bugs! Who watches the watchers?

Or can Coq verify itself?

The MetaCoq Project

A metaprogramming platform for Coq (TemplateCoq), turned into a verified implementation of Coq in Coq.

The MetaCoq Project

A metaprogramming platform for Coq (TemplateCoq), turned into a verified implementation of Coq in Coq.

In 2020, Sozeau et. al. completed the formalization for a large subset of Coq in the MetaCoq project: ”Coq Coq Correct! Verification of Type Checking and Erasure for Coq, in Coq”.

The MetaCoq Project

A metaprogramming platform for Coq (TemplateCoq), turned into a verified implementation of Coq in Coq.

In 2020, Sozeau et. al. completed the formalization for a large subset of Coq in the MetaCoq project: ”Coq Coq Correct! Verification of Type Checking and Erasure for Coq, in Coq”.

However, a few features such as Modules are missing from the project. Modules are important for almost all large Coq projects!

The MetaCoq Project

A metaprogramming platform for Coq (TemplateCoq), turned into a verified implementation of Coq in Coq.

In 2020, Sozeau et. al. completed the formalization for a large subset of Coq in the MetaCoq project: ”Coq Coq Correct! Verification of Type Checking and Erasure for Coq, in Coq”.

However, a few features such as Modules are missing from the project. Modules are important for almost all large Coq projects!

Therefore we are here!

Contributions of This Project

1. A Coq implementation of non-parametrized Coq modules within the MetaCoq framework, at the TemplateCoq level.

Contributions of This Project

1. A Coq implementation of non-parametrized Coq modules within the MetaCoq framework, at the TemplateCoq level.
2. Verification of correctness properties related to modules, environment, and typing.

Contributions of This Project

1. A Coq implementation of non-parametrized Coq modules within the MetaCoq framework, at the TemplateCoq level.
2. Verification of correctness properties related to modules, environment, and typing.
3. Translation of modules from TemplateCoq to PCUIC.

Contributions of This Project

1. A Coq implementation of non-parametrized Coq modules within the MetaCoq framework, at the TemplateCoq level.
2. Verification of correctness properties related to modules, environment, and typing.
3. Translation of modules from TemplateCoq to PCUIC.
4. A second implementation of Coq modules unifying Modules and the Global Environment.

Contributions of This Project

1. A Coq implementation of non-parametrized Coq modules within the MetaCoq framework, at the TemplateCoq level.
2. Verification of correctness properties related to modules, environment, and typing.
3. Translation of modules from TemplateCoq to PCUIC.
4. A second implementation of Coq modules unifying Modules and the Global Environment.
5. A summary of three recursion-related formal proof techniques.

Contributions of This Project

1. A Coq implementation of non-parametrized Coq modules within the MetaCoq framework, at the TemplateCoq level.
2. Verification of correctness properties related to modules, environment, and typing.
3. Translation of modules from TemplateCoq to PCUIC.
4. A second implementation of Coq modules unifying Modules and the Global Environment.
5. A summary of three recursion-related formal proof techniques.

Contents

The Coq Proof Assistant

In its core, Coq is a strongly typed lambda calculus, basically calculus of constructions + co-inductive types + universe polymorphism + cumulativity.

The Coq Proof Assistant

In its core, Coq is a strongly typed lambda calculus, basically calculus of constructions + co-inductive types + universe polymorphism + cumulativity.

Core objects are terms. Operational semantics – reduction.

$$(\lambda x.x)\ y \to_\beta y$$

The Coq Proof Assistant

In its core, Coq is a strongly typed lambda calculus, basically calculus of constructions + co-inductive types + universe polymorphism + cumulativity.

Core objects are terms. Operational semantics – reduction.

$$(\lambda x.x)\ y \to_\beta y$$

Denotational semantics – conversion.

$$\lambda x.x \equiv_\alpha \lambda y.y$$

Curry-Howard Correspondance – Types are Theorems, Programs are Proofs.

A metaprogramming platform for Coq.

Originally **TemplateCoq**, a Coq program that reifies/quotes terms in Coq.

A metaprogramming platform for Coq.

Originally **TemplateCoq**, a Coq program that reifies/quotes terms in Coq.

Since we have terms, why not state some properties about them?

A metaprogramming platform for Coq.

Originally **TemplateCoq**, a Coq program that reifies/quotes terms in Coq.

Since we have terms, why not state some properties about them?

Added **PCUIC** (Polymorphic Cumulative Calculus of Inductive Constructions) and **Safechecker**, a fuel-free, verified reduction machine.

The MetaCoq Project - History

A metaprogramming platform for Coq.

Originally **TemplateCoq**, a Coq program that reifies/quotes terms in Coq.

Since we have terms, why not state some properties about them?

Added **PCUIC** (Polymorphic Cumulative Calculus of Inductive Constructions) and **Safechecker**, a fuel-free, verified reduction machine.

Proof erasure to untyped calculus, ready for translation into ”usual” programming languages.

The MetaCoq Project - History

A metaprogramming platform for Coq.

Originally **TemplateCoq**, a Coq program that reifies/quotes terms in Coq.

Since we have terms, why not state some properties about them?

Added **PCUIC** (Polymorphic Cumulative Calculus of Inductive Constructions) and **Safechecker**, a fuel-free, verified reduction machine.

Proof erasure to untyped calculus, ready for translation into ”usual” programming languages.

Where is the implementation?

- (Coq) – TemplateCoq – PCUIC – Checker – Erasure – (Machine Code)
- Actual data structure of modules live in TemplateCoq.
- Verification of properties of modules live in TemplateCoq.
- Translation from TemplateCoq to PCUIC.

Contents

Example - Definitions

Modules as “collections of definitions”.

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

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

Example - Modules

“Packaging” definitions into a Module (Type).

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


Module Nat: Magma.
    Definition T := nat.
    Definition op := plus.
End Nat.
```

Example - Aliasing

Modules can be aliased for ease of reference.

```
Module Type M := Magma.
Module MyNat: M := Nat.
```

Example - Functors

Higher-order modules - Functors.

```
(* 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.
```

Abstract Syntax of Coq Modules

A **structure** is an ordered list of declarations of the following kinds:

- A **constant** declaration.
- An **inductive** declaration.
- A **module** declaration.
- A **module type** declaration.

Abstract Syntax of Coq Modules

A **structure** is an ordered list of declarations of the following kinds:

- A **constant** declaration.
- An **inductive** declaration.
- A **module** declaration.
- A **module type** declaration.

A **module** is a **structure** with a name and possibly a **module type**, where all definitions are concrete.

Abstract Syntax of Coq Modules

A **structure** is an ordered list of declarations of the following kinds:

- A **constant** declaration.
- An **inductive** declaration.
- A **module** declaration.
- A **module type** declaration.

A **module** is a **structure** with a name and possibly a **module type**, where all definitions are concrete.

A **module type** is a **structure** with a name.

Abstract Syntax of Coq Modules

A **structure** is an ordered list of declarations of the following kinds:

- A **constant** declaration.
- An **inductive** declaration.
- A **module** declaration.
- A **module type** declaration.

A **module** is a **structure** with a name and possibly a **module type**, where all definitions are concrete.

A **module type** is a **structure** with a name.

A **functor** is a parametrized module, by another module or functor.

Semantics of Coq Modules

Modules are declarations, and they live in an **environment**. An environment is an ordered list of declarations:

- A **constant** declaration.
- An **inductive** declaration.
- A **module** declaration.
- A **module type** declaration.

Semantics of Coq Modules

Coq Modules are second-class objects and have separate semantics from that of terms. Lives on another plane and have limited interactions.

Semantics are given by typing rules. Formation rules and access rules.

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

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

Implementation

Outline

Roadmap

Implementation	Verification
1. Definition of Modules	2. Lookup of definitions
3. Typing rules for Modules	4. Functoriality of Typing Rules
(Term typing rules)	5. Typing of terms
6. Translation to PCUIC	(Correctness of translation)
7. Modular Environment	(Correctness of implementation)

8. Three Formal Proof Techniques

Contents

1. Definition of Modules

Definition of Structures.

```
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
| mi_algebraic : kername -> module_implementation
| mi_struct : structure_body -> module_implementation
| mi_fullstruct : module_implementation
with structure_body :=
| sb_nil
| sb_cons : ident -> structure_field -> structure_body -> structure_body.
```

Listing 1: TemplateCoq/theories/Environment.v

1. Definition of Modules

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.

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 2: TemplateCoq/theories/Environment.v

Theorem (Lookup)
*Looking up **kn** yields **mdecl** iff **mdecl** is declared with **kn**.*

```
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.
```

Listing 3: TemplateCoq/theories/EnvironmentTyping.v

The core is the structure fields.

```
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)
```

Listing 4: Typing rules for structure fields.

3. Typing rules for modules

Subsequently, the typing rule for structures, and modules.

```
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)
  | on_mi_struct sb : on_structure_body Σ sb
                      -> on_module_impl Σ (mi_struct sb)
  | on_mi_fullstruct : on_module_impl Σ mi_fullstruct.
Definition on_module_type_decl := on_structure_body.
Definition on_module_decl Σ m := on_module_impl Σ m.1
                                 × on_module_type_decl Σ m.2.
```

Listing 5: Typing rules for structure, and modules.

4. Functoriality of Typing Rules

Lemma (Global declaration)
Fix term typing rules P, Q such that if the environment is P-well-formed if P types term t with type T, then Q types term t with type T as well.

Let Σ be a P-well-formed environment. If the definition (kn, d) is well-formed, then (kn, d) is Q-well-formed.

```
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.
```

Listing 6: Functoriality of typing of a global declaration.

Theorem (Global Environment)
Fix term typing rules P, Q such that they type terms in the same way for all terms $t : T$.

Let Σ be a P-well-formed environment. Then Σ is Q-well-formed.

```
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.
```

Listing 7: Functoriality of the typing of global environments.

5. Typing of terms

Theorem

Fix any two predicates P and P_Γ 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 P_Γ holds on the local context.

```
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).
```

Listing 8: Definition of key lemma in typing.

Checkpoint 1!

This marks the end of the TemplateCoq part of the First Implementation. We have seen

1. The definition of Modules.
2. Proof of lookup iff declared.
3. The definition of Typing Rules.
4. Functoriality.
5. Typing properties of terms.

We will show the translation to PCUIC and motivate the Second Implementation.

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 }.
```

Listing 9: Definition of the global environment for PCUIC.

So we translate by … removing modules!

The engine of the translation of modules.

```
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
```

Listing 10: Translation of structure fields to PCUIC.

Run the field-by-field translation over the 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.
```

Listing 11: Translating structure body.

Now we can translate a global declaration...

```
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.
```

Listing 12: Translating a global declaration.

And finally global declarations!

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

Listing 13: Translating global declarations.

And finally global declarations!

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

Listing 14: Translating global declarations.

Uh-oh… notice the double fold.

Theorem (Translated iff Exists)
”Translation preserves non-existence”, that is, the translated environment should only contain the intended translation and nothing more; and its dual, ”Translation preserves existence”, that is, nothing is lost in translation.

6.5. Verification of translation

6.9. Motivation for Second Implementation

```
Proof.
  destruct Σ as [univs Σ retro]. induction Σ.
  - cbn; auto.
      --- (** a.2 is a *)
        unfold trans_global_env. subst Σmap'; simpl.
        (** proving assertion by mutual induction *)
        * subst P P0 P1. apply Ast.Env.sf_mi_sb_mutind ⇒ //=.
          ** cbn. intros c id.
            *** simpl in *. subst.
```

...

Listing 15: Tedious nested proofs.

The first case takes 200 lines and counting!

6.9. Motivation for Second Implementation

```
Proof.
  destruct Σ as [univs Σ retro]. induction Σ.
  - cbn; auto.
      --- (** a.2 is a *)
        unfold trans_global_env. subst Σmap'; simpl.
        (** proving assertion by mutual induction *)
        * subst P P0 P1. apply Ast.Env.sf_mi_sb_mutind ⇒ //=.
          ** cbn. intros c id.
            *** simpl in *. subst.
```

…

Listing 16: Tedious nested proofs.

The first case takes 200 lines and counting! Too many repeated proofs.

6.9. Motivation for Second Implementation

Culprit!

```
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 17: An opportunity for abstraction!

Contents

As the name suggests, we combine the concepts of structures (modules) and environments.

As the name suggests, we combine the concepts of structures (modules) and environments.

An environment is just a module

As the name suggests, we combine the concepts of structures (modules) and environments.

An environment is just a module named by its directory path (eg. `/metacoq/template-coq/theories/Environment.v`).

As the name suggests, we combine the concepts of structures (modules) and environments.

An environment is just a module named by its directory path (eg. `/metacoq/template-coq/theories/Environment.v`).

All theorems on the typing of environment follow from that of modules!

Let us define modules, then specialize into environments.

```
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
```

Listing 18: Definition of structure fields.

"Globalization"!

```
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 19: Definition of global declarations.

Implemented but unverified typing rules. The interesting part follows...

```
Inductive on_structure_field Σ : structure_field -> Type :=
  | on_ConstantDecl c :
      on_constant_body Σ c -> on_structure_field Σ (ConstantDecl c)
  | on_InductiveDecl kn inds :
      on_inductive Σ kn inds -> on_structure_field Σ (InductiveDecl inds)
  | on_ModuleDecl mi mt :
      on_module_impl Σ mi ->
      on_structure_body on_structure_field mt ->
      on_structure_field Σ (ModuleDecl mi mt)
  | on_ModuleTypeDecl mtd :
      on_structure_body Σ mtd ->
      on_structure_field Σ (ModuleTypeDecl mtd)
```

Listing 20: Typing rules for structure fields.

7.5. Typing Rules

Now structure bodies encompass the typing of environments, such as the freshness of names.

```
with on_structure_body (Σ: global_env_ext) : structure_body -> Type :=
  | on_sb_nil : on_structure_body Σ nil
  | on_sb_cons Σ sb i sf :
      on_structure_body Σ sb ->
      fresh_structure_body Σ i sb ->
      on_udecl Σ (universes_decl_of_decl sf) ->
      on_structure_field Σ sf ->
      on_structure_body Σ (sb ,, (i, sf))
```

Listing 21: Typing rules of structure body.

Contents

All three techniques are related to recursion and were investigated during the modular environment rewrite.

1. Stronger Induction Principle for Nested Inductive Types
2. Well-formed Recursion
3. Strengthening of Induction Hypothesis (omitted)

Inductive type within an inductive type.

Rose tree (Meertens 1998):

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

Listing 22: Definition of a rose tree.

Unfortunately, Coq does not generate a strong enough induction principle for nested inductive types, only the below:

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

We need to check each rose tree within the list with predicate P first.

Unfortunately, Coq does not generate a strong enough induction principle for nested inductive types, only the below:

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

We need to check each rose tree within the list with predicate P first. Here is a stronger induction principle that is generally used:

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

The induction hypothesis is weakened, and the induction principle is strengthened!

In the modular rewrite - definition of structures!

```
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
```

Listing 23: Definition of structure fields.

Typical recursion: predecessor. What if this is not obvious?

Typical recursion: predecessor. What if this is not obvious?

Define a measure, and show that it is

- bounded below, and
- strictly decreasing at every recursive step.

Typical recursion: predecessor. What if this is not obvious?

Define a measure, and show that it is

- bounded below, and
- strictly decreasing at every recursive step.

8.2. Where is this used?

To recurse through the nested inductive structure body! Here is a measure:

```
Equations alt_size_sf (sf: structure_field) : nat :=
  | ConstantDecl _ := 1;
  | InductiveDecl _ := 1;
  | ModuleDecl mi mt := 1 + (max (alt_size_mi mi) (alt_size_sb mt));
  | ModuleTypeDecl mt := 1 + (alt_size_sb mt);
where alt_size_sb (sb: structure_body) : nat :=
  | nil := 0;
  | (hd::tl) := alt_size_sf hd.2 + alt_size_sb tl;
where alt_size_mi (mi: module_implementation) : nat :=
  | mi_struct s := alt_size_sb s;
  | _ := 0.
```

Listing 24: Height defined on structure body.

```
Lemma alt_size_sf_ge_one: (forall sf: structure_field, 0 < alt_size_sf
Proof.
  destruct sf; simp alt_size_sf; lia.
Qed.
```

Listing 25: Proof of lower bound of the height measure.

Conclusion

We have seen...

1. Implementation of modules, typing rules, translation/elaboration.

We have seen...

1. Implementation of modules, typing rules, translation/elaboration.
2. Verification of the properties.

We have seen...

1. Implementation of modules, typing rules, translation/elaboration.
2. Verification of the properties.
3. Second implementation that combines environments and modules.

We have seen...

1. Implementation of modules, typing rules, translation/elaboration.
2. Verification of the properties.
3. Second implementation that combines environments and modules.
4. Formal proof techniques necessary for verification.

We have seen...

1. Implementation of modules, typing rules, translation/elaboration.
2. Verification of the properties.
3. Second implementation that combines environments and modules.
4. Formal proof techniques necessary for verification.

Future work

1. Complete the modular environment rewrite.
2. Functors (and higher-order functors)
3. Document typing rules.

Previous implementations of Coq Modules: Courant, Chrąszcz, and Soubrian:

1. Courant added (second-class) modules, signature, and functors to Pure Type System (PTS).
2. Chrąszcz implemented modules, signature, and functors in mainline Coq, and proved the conservativity of his extension.
3. Soubrian implemented higher-order functions and unified modules and signatures with structures, and proposed dynamic naming scopes for modules.

Related work - ML Modules

1. SML by Lillibridge, Harper et. al..
2. OCaml by Leroy: applicative functors.
3. CakeML came closest in verifying modules.

Thank you!

Questions?