Towards Formalizing the Guard Condition of Coq

Yee-Jian Tan

MPRI M1 Internship Project
Advisor: Yannick Forster, Cambium team

Goal for Today

1. Consistency of a Type Theory
2. Coq's Guard Checker
3. Towards a Formalization

Consistency of a Type Theory

Three ingredients for consistency

Consistency of a Type Theory

1. Strong normalization: reduction terminates and every term has a unique normal form.
2. Subject reduction: reduction preserves typing.
3. Canonicity: normal form of terms in the empty context must begin with a constructor.

Together, we can prove consistency:

Proof of Consistency

Any term of the Empty type has a normal form (1) of the same type (2), which, in the empty context, must begin with a constructor (3). But the type False has no constructor.

Inductive Types in Coq: Constructors

We want to show Coq's consistency with the same scheme.

Inductive types in Coq:

```
Inductive nat :=
  | O : nat
  | S : nat → nat.

Inductive list (A : Set) :=
  | nil  : list A
  | cons : A → list A → list A.

Inductive vec (A : Set) : nat → Set :=
  | vnil              : vec A O
  | vcons (n : nat) : A → vec A n → vec A (S n).
```

Which are defined using constructors.

More examples:

```
Inductive Acc (A : Set) (R : A → A → Prop) (a : A) : Prop :=
  | acc : (forall b : A,  (R b a → Acc A R b)) → Acc A R a.

Inductive rtree :=
  | node : list rtree (* nested *) → rtree.

Inductive rtree' :=
  | node' : list_rtree → rtree'
with list_rtree :=
  | rtree_nil : list_rtree
  | rtree_cons : rtree' → list_rtree → list_rtree.
```

We can also have nested and/or mutual inductive types.

The dual of a constructor is an eliminator, whose type is known as the induction principle.

```
About nat_rec.
(** nat_rec : forall P : nat → Set,
  P 0 → (forall n : nat, P n → P (S n)) → forall n : nat, P n
*)
About list_rec.
(** list_rec : forall (A : Set) (P : list A → Set),
  P (nil A) →
  (forall (a : A) (l : list A), P l → P (cons A a l)) →
  forall l : list A, P l
*)
About vec_rec.
(** vec_rec: forall (A : Set) (P : forall n : nat, vec A n → Set),
  P 0 (vnil A) →
  (forall (n : nat) (a : A) (v : vec A n), P n v → P (S n) (vcons A n a v)) →
  forall (n : nat) (v : vec A n), P n v
*)
```

Eliminators vs Match

Coq was designed to extract to OCaml, so `match` operators are used instead of eliminators.

Eliminators can be defined using `match` and `fixpoints`.

```
About nat_rec.
(** nat_rec : forall P : nat -> Set,
  P O -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n
*)
```

```
Fixpoint nat_rec (P : nat -> Set)
  (p0 : P O) (ps : forall (m: nat), P m -> P (S m)) (n : nat) : P n :=
match n with
  | O => p0
  | S m => ps m (nat_rec P p0 ps m)
end.
End M.
```

Eliminators vs Case

For example, the plus operation on ℕ in both styles:

```
Definition plus_elim (a b : nat) := nat_rec (fun _ => nat) b (fun _ p => S p) a.

Fixpoint plus (a b : nat) {struct a} := match a with O => b | S a' => S (plus a' b) end.
```

which are equivalent.

```
Theorem plus_equiv : forall (a b : nat), plus a b = plus_elim a b.
Proof.
  induction a as [|a Ha].
  - simpl. reflexivity.
  - cbn. intro b. f_equal. exact (Ha b).
Qed.
```

Fixpoints in Coq

However, non-terminating fixpoints can break strong normalization!

```
Fixpoint plus' (a b : nat) {struct a} := match a with
 | O => b
 | S _ => S (plus' a b)
end.
```

Fixpoints in Coq

However, non-terminating fixpoints can break strong normalization!

```
Fixpoint plus' (a b : nat) {struct a} := match a with
 | O ⇒ b
 | S _ ⇒ S (plus' a b)
end.
```

```
Definition one := plus' (S O) O.
Theorem one_equals_two : one = S one.
Proof. unfold one at 1. rewrite alt. rewrite ← alt. unfold id. apply f_equal. symmetry. reflexivity. Qed.

Theorem n_not_succ_n: forall (n : nat), n = S n → False.
Proof. induction n as [|n Hn]; intro H; now inversion H. Qed.

Goal False. exact (n_not_succ_n one one_equals_two). Qed.
```

Fixpoints in Coq

However, non-terminating fixpoints can break strong normalization!

```
Fixpoint plus' (a b : nat) {struct a} := match a with
  | O => b
  | S _ => S (plus' a b)
end.
```

```
Definition one := plus' (S O) O.
Theorem one_equals_two : one = S one.
Proof. unfold one at 1. rewrite alt. rewrite <- alt. unfold id. apply f_equal. symmetry. reflexivity. Qed.

Theorem n_not_succ_n: forall (n : nat), n = S n -> False.
Proof. induction n as [|n Hn]; intro H; now inversion H. Qed.

Goal False. exact (n_not_succ_n one one_equals_two). Qed.
```

Fixpoints in Coq

However, non-terminating fixpoints can break strong normalization!

```
Fixpoint plus' (a b : nat) {struct a} := match a with
  | O => b
  | S _ => S (plus' a b)
end.
```

```
Definition one := plus' (S O) O.
Theorem one_equals_two : one = S one.
Proof. unfold one at 1. rewrite alt. rewrite <- alt. unfold id. apply f_equal. symmetry. reflexivity. Qed.

Theorem n_not_succ_n: forall (n : nat), n = S n -> False.
Proof. induction n as [|n Hn]; intro H; now inversion H. Qed.

Goal False. exact (n_not_succ_n one one_equals_two). Qed.
```

Fixpoints in Coq

However, non-terminating fixpoints can break strong normalization!

```
Fixpoint plus' (a b : nat) {struct a} := match a with
 | O ⇒ b
 | S _ ⇒ S (plus' a b)
end.
```

```
Definition one := plus' (S O) O.
Theorem one_equals_two : one = S one.
Proof. unfold one at 1. rewrite alt. rewrite ← alt. unfold id. apply f_equal. symmetry. reflexivity. Qed.

Theorem n_not_succ_n: forall (n : nat), n = S n → False.
Proof. induction n as [|n Hn]; intro H; now inversion H. Qed.

Goal False. exact (n_not_succ_n one one_equals_two). Qed.
```

... and consistency!

Coq’s Guard Checker

Coq's guard checker

- sufficient condition for termination
- based on a syntactical check for **structural recursion**
- the condition it imposes is known as the **guard condition**.

  In short: it checks that the recursive argument is structurally smaller.

```
Fixpoint plus (a b : nat) {struct a} := match a with O ⇒ b | S a' ⇒ S (plus a' b) end.
```

Other guard conditions

- well-foundedness in `Program Fixpoint`
- sized types in Agda
- type-based conditions

Coq’s guard checker

An oversimplification of how the guardchecker works:

```
Fixpoint f (n : nat) := match n with
  (* strict subterms of n : [] *)
  | O ⇒ O
  | S n1 ⇒ (* strict subterms of n : [n1] *)
  match n1 with (* strict subterms of n : [n1, n2] *)
    | O ⇒ n1
    | S n2 ⇒ ((fun x ⇒ x) f) n1
  end
end.
```

- Internally, the subterms are deduced from a (regular) tree representing `nat`.
- In real life: mutual, nested inductive types (and fixpoints) that complicate matter...

Is that the end of the story?

Of course not! Many things happened since the guard checker's birth.

- remains crucial for the correctness of Coq
- at the heart of multiple consistency-threatening bugs.
- bugfixes and optimizations → about 1k LOC of OCaml (2k including data structures)

Coq's Guard Checker: a Timeline

Many others have contributed to the guard checker, sorry if I missed your names!

1990s

- Eduardo Gimenez : "Codifying Recursive Definitions with Recursive Schemes".
- Christine Paulin-Mohring : Inductive types in Coq.

2000s

- Bruno Barras : first commit of the Guard Checker in Coq by Bruno Barras.

2010s

- Pierre Boutiller : relaxation of the guard condition via $\beta - \iota$ cuts
- Maxime Dénès : Propositional Extensionality bug + fixes

2020s

- Hugo Herbelin : restored strong normalization, extracted uniform parameters, …

Towards a Formalization

Why understand the Guard Checker of Coq?

User POV

Fighting the guard checker is common in formalization projects. We need an accurate understanding of it.

Theoretical POV

We want to know that Coq's metatheory is consistent.

Immediate Goals

- Understand the guard checker and produce a specification/paper/document
- Lay the groundwork for formalization: we do it in MetaCoq.

Introduction to MetaCoq

Definition: implementation details

Distinction must be made between

Guard Condition

A **predicate** on whether a term is guarded.

```
Inductive Guard Σ Γ : term → Prop :=
| Guard_tFix (f : tFix) : "f is structurally recursive" → Guard Σ Γ f
| ...  end.
```

Guard Checker

Guard Checker: a **function** that computes/decides the guardedness of a term.

```
Definition guard Γ Σ t A → (Γ ; Σ ⊢ t : A) → Bool.
Theorem guard_ok := guard t = true iff Guard t.
```

Guard Condition in MetaCoq: Current State

Did MetaCoq prove consistency? Not yet, but there is hope. See Meven's talk later!

3 ingredients:

1. Strong normalization -
2. Subject reduction -
3. Canonicity -

Guard Condition in MetaCoq: Current State

Did MetaCoq prove consistency? Not yet, but there is hope. See Meven’s talk later!

3 ingredients:

1. Strong normalization - postulated. Requires a notion of guardedness.
2. Subject reduction -
3. Canonicity -

Guard Condition in MetaCoq: Current State

Did MetaCoq prove consistency? Not yet, but there is hope. See Meven's talk later!

3 ingredients:

1. Strong normalization - postulated. Requires a notion of guardedness.
2. Subject reduction - proved, assuming the guard checker exists.
3. Canonicity -

Guard Condition in MetaCoq: Current State

Did MetaCoq prove consistency? Not yet, but there is hope. See Meven's talk later!

3 ingredients:

1. Strong normalization - postulated. Requires a notion of guardedness.
2. Subject reduction - proved, assuming the guard checker exists.
3. Canonicity - proved, assuming the guard checker exists.

First Contribution: an issue in the current setup

The Wrong Way to Guard Check

The current order of proofs:

The Wrong Way to Guard Check

The current order of proofs:

1. **Assume** a guard checker (function)

The Wrong Way to Guard Check

The current order of proofs:

1. **Assume** a guard checker (function)
2. Define typing relation + 1-step reduction relation

The Wrong Way to Guard Check

The current order of proofs:

1. **Assume** a guard checker (function)
2. Define typing relation + 1-step reduction relation
3. **Assume** Strong Normalization

The Wrong Way to Guard Check

The current order of proofs:

1. **Assume** a guard checker (function)
2. Define typing relation + 1-step reduction relation
3. **Assume** Strong Normalization
4. Define reduction function (and show it respects the reduction relation)

The Wrong Way to Guard Check

The current order of proofs:

1. **Assume** a guard checker (function)
2. Define typing relation + 1-step reduction relation
3. **Assume** Strong Normalization
4. Define reduction function (and show it respects the reduction relation)
5. Define a guard checker that replaces 1...?

The Wrong Way to Guard Check

The current order of proofs:

1. **Assume** a guard checker (function)
2. Define typing relation + 1-step reduction relation
3. **Assume** Strong Normalization
4. Define reduction function (and show it respects the reduction relation)
5. Define a guard checker that replaces 1...?

Circular dependency! Any way to break the loop?

The **Correct** Way to Guard Check

The **correct** order of proofs:

1. Define guard condition (predicate)

The **Correct** Way to Guard Check

The **correct** order of proofs:

1. Define guard condition (predicate)
2. Define typing relation + 1-step reduction relation

The **Correct** Way to Guard Check

The **correct** order of proofs:

1. Define guard condition (predicate)
2. Define typing relation + 1-step reduction relation
3. **Assume** Strong Normalization

The **Correct** Way to Guard Check

The **correct** order of proofs:

1. Define guard condition (predicate)
2. Define typing relation + 1-step reduction relation
3. **Assume** Strong Normalization
4. Define reduction function (and show it respects the reduction relation)

The **Correct** Way to Guard Check

The **correct** order of proofs:

1. Define guard condition (predicate)
2. Define typing relation + 1-step reduction relation
3. **Assume** Strong Normalization
4. Define reduction function (and show it respects the reduction relation)
5. Define guard checker (and show it respects the guard condition)

No more circular dependency!

Plan and Future Work

Plan for Current Work

Do bullet points 1 (define guard predicate) and 5 (port guard checker to Coq) concurrently.

Faithful to current OCaml implementation.

Future work

- Move trust to a new guard condition that
  - is simpler to understand, thus easier to trust, and
  - implies the old guard condition.

  $$\text{Old Guard Condition} \xrightarrow{\text{reduces}} \text{New Guard Condition}$$

  by doing a (verified) translation.
- Ideally, Coq's guard checker will be extracted from a verified implementation in MetaCoq.

Conclusion

We have seen today

- Three ingredients to prove consistency:
  1. Strong Normalization (Guard Condition!)
  2. Subject Reduction
  3. Canonicity
- Inductive types; eliminators vs fixpoints (and danger)
- Introduction to MetaCoq
- "First predicate, then function"

Conclusion

We have seen today

- Three ingredients to prove consistency:
  1. Strong Normalization (Guard Condition!)
  2. Subject Reduction
  3. Canonicity
- Inductive types; eliminators vs fixpoints (and danger)
- Introduction to MetaCoq
- “First predicate, then function”

Thank you! Questions?