Towards Formalising the Guard Checker of Coq

Yee-Jian Tan, Yannick Forster

14 September, 2024

Cambium, Inria Paris

Eliminators and Fixpionts

- Coq is based on the Calculus of *Inductive* Constructions (CIC) [1].
- To construct: constructors

  To eliminate: eliminators (aka recursors, destructors), or fixpoints + match.

  ```
  Fixpoint add (m n : nat) {struct m} := match m with O => n | S m' => add m' (S n) end.
  ```
- Advantage: extracted code to e.g. OCaml is more idiomatic

Eliminators and Fixpionts

Unrestricted fixpoints can be non-terminating...

```
#[bypass_check(guard)]
Fixpoint boom (n : nat) : False := boom n.
```

and **break consistency**!

```
Check (boom 0). (* False *)
```

Eliminators and Fixpionts

Unrestricted fixpoints can be non-terminating…

```
#[bypass_check(guard)]
Fixpoint boom (n : nat) : False := boom n.
```

and **break consistency**!

```
Check (boom 0). (* False *)
```

How does Coq avoid non-termination?

The guard checker! It checks for **structural recursion**.

```
Fixpoint add (m n : nat)
  {struct m} : nat :=
  match m with
  | O    => n
  | S m' => add m' (S n)
  end.
```

How does Coq avoid non-termination?

The guard checker! It checks for **structural recursion**.

```
Fixpoint add (m n : nat)
  {struct m} : nat :=
  match m with
  | O    => n
  | S m' => add m' (S n)
  end.
```

How does Coq avoid non-termination?

The guard checker! It checks for **structural recursion**.

```
Fixpoint add (m n : nat)
  {struct m} : nat :=
  match m with
  | O    => n
  | S m' => add m' (S n)
  end.
```

Simple!

Structural Recursion

Another example:

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

Structural Recursion

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

```
Fixpoint div (m n : nat) {struct m} :=
  match m with
  | O   => O
  | S k => S (div (minus k n) n)
  end.
```

Structural Recursion

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

```
Fixpoint div (m n : nat) {struct m} :=
  match m with
  | O    => O
  | S k => S (div (minus k n) n)
  end.
```

`div` is not guarded! Why?

Structural Recursion

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

```
Fixpoint div (m n : nat) {struct m} :=
  match m with
  | O    => O
  | S k => S (div (minus k n) n)
  end.
```

Because `O` is not a subterm of `m`!

Structural Recursion

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

```
Fixpoint div (m n : nat) {struct m} :=
  match m with
  | O   => O
  | S k => S (div (minus k n) n)
  end.
```

This is structural!

Things are not as simple as they seem.

The Guard Checker of Coq

The Guard Checker of Coq

- About 1,000 lines of **unspecified, unexplained** OCaml code
- Iterated by different authors over 30 years
- Multiple dimensions of complexity

Consistency: there is no term of Empty type in the empty context.

Ingredients for consistency

1. **(Weak) Normalisation**: every term has a normal form.
2. **Subject Reduction**: reduction preserves typing.
3. **Canonicity**: for inductive types, normal forms begin with a constructor in the empty context.

Proof: In the empty context, any term of the Empty type must have a normal form (1) of the same type (2). Since the context is empty, it must begin with a constructor (3), but the Empty type has none.

Consistency Proofs

Consistency: there is no term of Empty type in the empty context.

Ingredients for consistency

1. **(Weak) Normalisation**: every term has a normal form.
2. **Subject Reduction**: reduction preserves typing.
3. **Canonicity**: for inductive types, normal forms begin with a constructor in the empty context.

Proof: In the empty context, any term of the Empty type must have a normal form (1) of the same type (2). Since the context is empty, it must begin with a constructor (3), but the Empty type has none.

Contribution

Two main contributions of this project:

Implementation

A full implementation of Coq's Guard Checker in Coq, using the MetaCoq project.

Extending previous work by Lennard Gäher [2].

Documentation

In the report: examples (Chapter 2, Appendix) and explanations (Chapter 3).

Implementation in Coq

MetaCoq project

- Formalises Coq's type theory in Coq (faithful) [3]
- A verified implementation of type checker [4]
- A verified extraction function to OCaml [5]

Proved:

- Subject Reduction and Canonicity,
- **parameterised** by a guard checker
- assumed Normalisation

Implementation of the Guard Checker

```
From MetaCoq.Guarded Require Import plugin.

(* define your fixpoint *)
Fixpoint add (m n : nat) : nat :=
  match m with
  | O => n
  | S m' => add m' (S n)
  end.

MetaCoq Quote add_syntax := add.
Check check_fix.
Compute (check_fix add_syntax).
```

Implementation of the Guard Checker

```
From MetaCoq.Guarded Require Import plugin.

(* define your fixpoint *)
Fixpoint add (m n : nat) : nat :=
  match m with
  | O => n
  | S m' => add m' (S n)
  end.
```

```
MetaCoq Quote add_syntax := add.
Check check_fix.
Compute (check_fix add_syntax).
```

Implementation of the Guard Checker

```
From MetaCoq.Guarded Require Import plugin.

(* define your fixpoint *)
Fixpoint add (m n : nat) : nat :=
  match m with
  | O => n
  | S m' => add m' (S n)
  end.

MetaCoq Quote add_syntax := add.
Check check_fix.
Compute (check_fix add_syntax).
```

```
add_syntax : Ast.term :=
  (Ast.tFix [{|
    dname := {| binder_name := nNamed "add" |};
    dtype := Ast.tProd {| binder_name := nNamed "m" |}
      (Ast.tInd {| inductive_mind := "nat" |} [])
      (...);
    dbody :=
      Ast.tLambda
        {| binder_name := nNamed "m" |}
        (Ast.tInd {| inductive_mind := "nat"; inductive_ind := 0 |} [])
        (Ast.tLambda
          {| binder_name := nNamed "n" |}
          (Ast.tInd {...} [])
          (Ast.tCase
              {| ci_ind := {| inductive_mind := "nat" |}; |}
              {| Ast.pcontext := [{| binder_name := nNamed "m"; |}];
                 Ast.preturn := Ast.tInd {| inductive_mind := "nat" |} []
              |}
              (Ast.tRel 1)
              [ {| Ast.bcontext := []; Ast.bbody := Ast.tRel 0 |};
                {| Ast.bcontext := [{| binder_name := nNamed "m'"; |}];
                   Ast.bbody :=
                     Ast.tApp (Ast.tRel 3)
                       [Ast.tRel 0;
                        Ast.tApp
                          (Ast.tConstruct {| inductive_mind := "nat"; inductive_ind := 0 |})
                          [Ast.tRel 1]]
                 |}]));
          rarg := 0
        |}] 0)
```

Implementation of the Guard Checker

```
From MetaCoq.Guarded Require Import plugin.

(* define your fixpoint *)
Fixpoint add (m n : nat) : nat :=
  match m with
  | O => n
  | S m' => add m' (S n)
  end.

MetaCoq Quote add_syntax := add.
Check check_fix.
Compute (check_fix add_syntax).
```

```
check_fix : Ast.term -> bool
```

Implementation of the Guard Checker

```
From MetaCoq.Guarded Require Import plugin.

(* define your fixpoint *)
Fixpoint add (m n : nat) : nat :=
  match m with
  | O => n
  | S m' => add m' (S n)
  end.

MetaCoq Quote add_syntax := add.
Check check_fix.
Compute (check_fix add_syntax).
```

```
= true : bool
```

History of the Guard Checker

Phase 1: Beginnings

- Inductive + CoC = CIC (Frank Pfenning and Christine Paulin-Mohring, 1989) [1]
- Pattern Matching with Dependent Types (Thierry Coquand, 1992) [6]
- The first Guard Checker in Coq v5.10.2 (Christine Paulin-Mohring, 1994) [7]

Phase 1: Beginnings

- Inductive + CoC = CIC (Frank Pfenning and Christine Paulin-Mohring, 1989) [1]
- Pattern Matching with Dependent Types (Thierry Coquand, 1992) [6]
- The first Guard Checker in Coq v5.10.2 (Christine Paulin-Mohring, 1994) [7]

Phase 1: Beginnings

- Inductive + CoC = CIC (Frank Pfenning and Christine Paulin-Mohring, 1989) [1]
- Pattern Matching with Dependent Types (Thierry Coquand, 1992) [6]
- The first Guard Checker in Coq v5.10.2 (Christine Paulin-Mohring, 1994) [7]

Phase 2: Specifications

- Inductive + CoC = CIC (Frank Pfenning and Christine Paulin-Mohring, 1989) [1]
- Pattern Matching with Dependent Types (Thierry Coquand, 1992) [6]
- The first Guard Checker in Coq v5.10.2 (Christine Paulin-Mohring, 1994) [7]
- *Codifying Recursive Definition with Recursive Schemes* (Eduardo Gimenez, 1994) [8]
- *Inductive Definitions for Type Theory* (Christine Paulin-Mohring, 1996) [9]
- *Un Calcul de Constructions Infinies et son application à la vérification de systèmes communicants* (Eduardo Gimenez, 1996) [10]

Phase 3: Big Changes

- Inductive + CoC = CIC (Frank Pfenning and Christine Paulin-Mohring, 1989) [1]
- Pattern Matching with Dependent Types (Thierry Coquand, 1992) [6]
- The first Guard Checker in Coq v5.10.2 (Christine Paulin-Mohring, 1994) [7]
- *Codifying Recursive Definition with Recursive Schemes* (Eduardo Gimenez, 1994) [8]
- *Inductive Definitions for Type Theory* (Christine Paulin-Mohring, 1996) [9]
- *Un Calcul de Constructions Infinies et son application à la vérification de systèmes communicants* (Eduardo Gimenez, 1996) [10]
- β-ι commutative cuts subterm rule (Pierre Boutillier, 2010) [11]

```
match v2 in with
| nil => (fun _ => nil C)
| cons h2 t2 => (fun t1' => cons (f h1 h2) (map2 f t1' t2))
end t1
```

Two Weeks before Christmas, 2013

```
From: Daniel Schepler <dschepler AT gmail.com>
To: Coq Club <coq-club AT inria.fr>
Subject: [Coq-Club] bijective function implies equal types is provably inconsistent with functional extensionality in Coq
Date: Thu, 12 Dec 2013 11:02:00 -0800
```

```
Section bijective_impl_eq.
Hypothesis functional_extensionality :
  forall (A B:Type) (f g:A->B),
  (forall x:A, f x = g x) -> f = g.

...

Definition not_bijective_impl_eq : False := func_unit_discr unit_eq_False_False_funs.
End bijective_impl_eq.
```

```
--
Daniel Schepler
```

Phase 3: Big Changes

- Inductive + CoC = CIC (Frank Pfenning and Christine Paulin-Mohring, 1989) [1]
- Pattern Matching with Dependent Types (Thierry Coquand, 1992) [6]
- The first Guard Checker in Coq v5.10.2 (Christine Paulin-Mohring, 1994) [7]
- *Codifying Recursive Definition with Recursive Schemes* (Eduardo Gimenez, 1994) [8]
- *Inductive Definitions for Type Theory* (Christine Paulin-Mohring, 1996) [9]
- *Un Calcul de Constructions Infinies et son application à la vérification de systèmes communicants* (Eduardo Gimenez, 1996) [10]
- β-ι commutative cuts subterm rule (Pierre Boutillier, 2010) [11]
- Restore compatibility with Propositional Extensionality (Maxime Dénès, 2014) [12]
- Restore strong normalisation (Hugo Herbelin, 2022) [13]
- Extrude uniform parameters (Hugo Herbelin, 2024) [14]

Phase 3: Big Changes

- Inductive + CoC = CIC (Frank Pfenning and Christine Paulin-Mohring, 1989) [1]
- Pattern Matching with Dependent Types (Thierry Coquand, 1992) [6]
- The first Guard Checker in Coq v5.10.2 (Christine Paulin-Mohring, 1994) [7]
- *Codifying Recursive Definition with Recursive Schemes* (Eduardo Gimenez, 1994) [8]
- *Inductive Definitions for Type Theory* (Christine Paulin-Mohring, 1996) [9]
- *Un Calcul de Constructions Infinies et son application à la vérification de systèmes communicants* (Eduardo Gimenez, 1996) [10]
- β-ι commutative cuts subterm rule (Pierre Boutillier, 2010) [11]
- Restore compatibility with Propositional Extensionality (Maxime Dénès, 2014) [12]
- Restore strong normalisation (Hugo Herbelin, 2022) [13]
- Extrude uniform parameters (Hugo Herbelin, 2024) [14]

A Taste of the Guard Checker

Example: add

```
Fixpoint add (m n : nat)
  {struct m} : nat :=
  match m with
  | O    => n
  | S m' => add m' (S n)
  end.
```

Goal: check that add is guarded.

Guarded: *All* recursive calls have a **strict subterm** as the **recursive argument**.

Example: add

```
Fixpoint add (m n : nat)
  {struct m} : nat :=
  match m with
  | O    => n
  | S m' => add m' (S n)
  end.
```

Subterm Specification

With respect to the **recursive parameter** `m`, terms can be a

- Large Subterm (e.g. `m`)
-
-

Example: `add`

```
Fixpoint add (m n : nat)
  {struct m} : nat :=
  match m with
  | O    => n
  | S m' => add m' (S n)
  end.
```

Subterm Specification

With respect to the **recursive parameter** `m`, terms can be a

- Large Subterm (e.g. `m`)
- Strict Subterm (e.g. `m'`)
-

Example: `add`

```
Fixpoint add (m n : nat)
  {struct m} : nat :=
  match m with
  | O    => n
  | S m' => add m' (S n)
  end.
```

Subterm Specification

With respect to the **recursive parameter** `m`, terms can be a

- Large Subterm (e.g. `m`)
- Strict Subterm (e.g. `m'`)
- Not Subterm (e.g. `n`)

Example: add

```
Fixpoint add (m n : nat)
  {struct m} : nat :=
  match m with
  | O    => n
  | S m' => add m' (S n)
  end.
```

```
Guard Env : [n:Bound{1}|m:Large|add]
```

Guard Environment

Subterm specifications of terms in the local context are stored.

Example: add

```
Fixpoint add (m n : nat)
  {struct m} : nat :=
  match m with
  | O    => n
  | S m' => add m' (S n)
  end.
```

```
Guard Env : [...]
Stack     : [Closure m'|Closure(S n)]
```

Stack of subterm specifications

The subterm information of arguments are stored on a stack when checking the head of an application.

Example: add

```
Fixpoint add (m : nat) :=
  fun (n : nat) =>
  match m return nat with
  | O    => n
  | S m' => add m' (S n)
  end.
```

```
Guard env: [m:Large|add]
Stack:     []
```

Initial state. Parameters after the recursive parameter are turned into lambdas.

Example: add

```
Fixpoint add (m : nat) :=
  fun (n : nat) =>
  match m return nat with
  | O    => n
  | S m' => add m' (S n)
  end.
```

```
Guard env: [m:Large|add]
Stack:     []
```

For a lambda to be guarded, its

- binder type must be guarded, and
- body must be guarded.

Example: add

```
Fixpoint add (m : nat) :=
  fun (n : nat) =>
  match m return nat with
  | O    => n
  | S m' => add m' (S n)
  end.

Guard env: [m:Large|add]
Stack:     []
```

Binder type is guarded.

Example: add

```
Fixpoint add (m : nat) :=
  fun (n : nat) =>
  match m return nat with
  | O    => n
  | S m' => add m' (S n)
  end.
```

```
Guard env: [n:Bound{1}|m:Large|add]
Stack:     []
```

The body is checked with a updated guard environment.

Example: add

```
Fixpoint add (m : nat) :=
  fun (n : nat) =>
  match m return nat with
  | O    => n
  | S m' => add m' (S n)
  end.
```

```
Guard env: [n:Bound{1}|m:Large|add]
Stack:     []
```

For a match to be guarded,

- discriminant,
- return type, and
- every branch

must be guarded.

Example: add

```
Fixpoint add (m : nat) :=
  fun (n : nat) =>
  match m return nat with
  | O    => n
  | S m' => add m' (S n)
  end.
```

```
Guard env: [n:Bound{1}|m:Large|add]
Stack:     []
```

Discriminant (`m`) and the return type (`nat`) are guarded.

Example: add

```
Fixpoint add (m : nat) :=
  fun (n : nat) =>
  match m return nat with
  | O    => n
  | S m' => add m' (S n)
  end.

Guard env: [n:Bound{1}|m:Large|add]
Stack:     []
```

To check a branch:

- expand into a lambda
- specify parameters
- check the lambda

Example: add

```
Fixpoint add (m : nat) :=
  fun (n : nat) =>
  match m return nat with
  | O    => n
  | S m' => add m' (S n)
  end.

Guard env: [n:Bound{1}|m:Large|add]
Stack:     []
```

0-th branch has no parameter.

- ~~expand into a lambda~~
- ~~specify parameters~~
- check the “lambda”: guarded.

Example: add

```
Fixpoint add (m : nat) :=
  fun (n : nat) =>
  match m return nat with
  | O    => n
  | S m' => add m' (S n)
  end.
```

```
Guard env: [n:Bound{1}|m:Large|add]
Stack:     [m':Strict]
```

1-st branch:

- expand into a lambda
-
-

Example: add

```
Fixpoint add (m : nat) :=
  fun (n : nat) =>
  match m return nat with
  | O    => n
  | S m' => add m' (S n)
  end.

Guard env: [n:Bound{1}|m:Large|add]
Stack:     [m':Strict]
```

1-st branch:

- expand into a lambda
- specify parameters: **strict!**
-

Example: add

```
Fixpoint add (m : nat) :=
  fun (n : nat) =>
  match m return nat with
  | O    => n
  | S m' => add m' (S n)
  end.
```

```
Guard env: [n:Bound{1}|m:Large|add]
Stack:     [m':Strict]
```

1-st branch:

- expand into a lambda
- specify parameters: **strict!**
- check the lambda

Example: add

```
Fixpoint add (m : nat) :=
  fun (n : nat) =>
  match m return nat with
  | O    => n
  | S m' => add m' (S n)
  end.

Guard env: [m':Strict|n  :Bound{1}|
            m :Large |add:Not     ]
Stack:      []
```

1-st branch:

- expand into a lambda
- specify parameters: **strict!**
- check the lambda

Example: add

```
Fixpoint add (m : nat) :=
  fun (n : nat) =>
  match m return nat with
  | O     => n
  | S m'  => add m' (S n)
  end.
```

```
Guard env: [m':Strict|n  :Bound{1}|
            m :Large |add:Not     ]
Stack:     []
```

Application with the recursive call is guarded if

- arguments are all guarded, and
- **key case**: the recursive argument is a strict subterm (on the stack)

Example: add

```
Fixpoint add (m : nat) :=
  fun (n : nat) =>
  match m return nat with
  | O    => n
  | S m' => add m' (S n)
  end.
```

```
Guard env: [m':Strict|n  :Bound{1}|
            m :Large |add:Not     ]
Stack:     [Closure m'|Closure(S n)]
```

Arguments are checked from right to left: both guarded.

Stack is populated with closures.

Example: add

```
Fixpoint add (m : nat) :=
  fun (n : nat) =>
  match m return nat with
  | O    => n
  | S m' => add m' (S n)
  end.
```

```
Guard env: [m':Strict|n   :Bound{1}|
            m  :Large |add:Not     ]
Stack:     [Strict    |Closure(S n)]
```

Since the recursive parameter of `add` is at position 0, specify the 0-th element of the stack.

`m'` is a **strict** subterm according to the Guard Environment.

Done!

Example: add

```
Fixpoint add (m : nat) :=
  fun (n : nat) =>
  match m return nat with
  | O    => n
  | S m' => add m' (S n)
  end.

Guard env: [m':Strict|n  :Bound{1}|
            m :Large |add:Not     ]
Stack:     [Strict   |Closure(S n)]
```

Since the recursive parameter of `add` is at position 0, specify the 0-th element of the stack.

`m'` is a **strict** subterm according to the Guard Environment.

Done!

Example: add

```
Fixpoint add (m : nat) :=
  fun (n : nat) =>
  match m return nat with
  | O    => n
  | S m' => add m' (S n)
  end.

Guard env: [m':Strict|n  :Bound{1}|
            m :Large |add:Not     ]
Stack:     [Strict   |Closure(S n)]
```

Since the recursive parameter of `add` is at position 0, specify the 0-th element of the stack.

`m'` is a **strict** subterm according to the Guard Environment.

Done!

More features

What if...

Delayed? Answer: Stack handles this well.

```
Fixpoint add (m n : nat) {struct m} : nat :=
  (fun k => match k with
  | O    => n
  | S m' => add m' (S n)
  end) m.
```

More features

What if...

Obfuscated? Answer: weak-head reduction **only** when checking subterm specification.

```
Fixpoint add (m n : nat) {struct m} : nat :=
  (fun k => match (id k) with
  | O    => n
  | S m' => add (pred (S m')) (S n)
  end) m.
```

More features

What if...

Not guarded in erasable subterms?

Answer: strong normalisation (reduction only when needed).

```
Fail Fixpoint add (m n : nat) {struct m} : nat :=
  let _ := add m (add m m) in
  (fun k => match (id k) with
  | O    => n
  | S m' => add (pred (S m')) (S n)
  end) m.
```

More features

What if...

Not guarded in erasable subterms?

Answer: strong normalisation (reduction only when needed).

```
Fixpoint add (m n : nat) {struct m} : nat :=
  let _ := add m (add m m) in
  (fun k => match (id k) with
  | O    => n
  | S m' => add (id m') (S n)
  end) m.
```

Not covered in example: β-ι cuts, redex stack, nested fix, ...

The (at least) 4 Dimensions of Complexity

Dimensions of Complexity

- The stack of subterm specifications for β-ι commutative cuts
- Strong normalisation:
  - a redex stack
  - only reduce terms to weak-head normal form when needed
- Support for mutual and nested fixpoints
  - regular trees
- OCaml `lazy` for efficiency

Resulting in 1,000 lines of OCaml code.

Full Implementation in Coq

- Complete, available as a MetaCoq (TemplateCoq) plugin.
- Feature parity with the kernel
- Test parity* with the kernel
- Intentionally kept as close as possible to Coq’s guard checker
- Available at: https://github.com/inria-cambium/m1-tan/tree/v1.0.0

Conclusion and Future Work

Summary

Conclusion

- implemented the Guard Checker in Coq
- documented its features
- gave examples of its behaviour

Future Work

- verify that the guard checker itself is a terminating program
- specification of an abstract **guard condition** of the checker
- verify that the guard checker implements the guard condition
- relative consistency proofs for its soundness

Well-Founded Recursion

- An alternative to structural recursion
- Coq: structural by default; well-founded using `Program Fixpoint` or `Equations`

  Lean: structural by default; well-founded attempted otherwise (`termination_by`)

  Agda: structural by default; well-founded using `Induction.WellFounded`

Agda: Semantic Termination Checking

	Syntactic	Semantic
Example	Coq	Agda
Reduction	Minimal	Full
Mechanism	Guard	Sized Types
Advantage	Fast	Accurate

- Chan, Li, and Bowman [15] attempted Sized Types in Coq in 2019, compilation time increased as much as 5-15x on the Coq Standard Library.
- New algorithm in Agda by Nisht and Abel [16] is linear on input, but not yet proven complete.

Lean: Native Eliminators

- Lean is the opposite of Coq: eliminators are native in the kernel, recursive functions only exist in the surface syntax
- Type Checking:
  1. Eliminators are generated for Inductive Types
  2. A strong (aka course-of-values) induction principle is defined using the said eliminators
  3. Recursive functions are translated into an encoding by the strong induction principle
- Extraction (Code Generation/Compilation) to C: the syntax gets extracted as-is
- Advantage: eliminators are simpler for the theory
  Disadvantage: hard to prove extraction correct, possible suprising behaviour

Bibliography

[1] F. Pfenning and C. Paulin-Mohring, "Inductively Defined Types in the Calculus of Constructions," in *Mathematical Foundations of Programming Semantics, 5th International Conference, Tulane University, New Orleans, Louisiana, USA, March 29 - April 1, 1989, Proceedings*, M. G. Main, A. Melton, M. W. Mislove, and D. A. Schmidt, Eds., in Lecture Notes in Computer Science, vol. 442. Springer, 1989, pp. 209–228. doi: 10.1007/BFB0040259.

[2] L. Gäher, "Guard Checker in MetaCoq." GitHub, 2021.

[3] M. Sozeau *et al.*, "The MetaCoq Project," *J. Autom. Reason.*, vol. 64, no. 5, pp. 947–999, 2020, doi: 10.1007/S10817-019-09540-0.

[4] M. Sozeau, S. Boulier, Y. Forster, N. Tabareau, and T. Winterhalter, "Coq Coq correct! verification of type checking and erasure for Coq, in Coq," *Proc. ACM Program. Lang.*, vol. 4, no. POPL, pp. 1–28, 2020, doi: 10.1145/3371076.

Bibliography

[5] Y. Forster, M. Sozeau, and N. Tabareau, “Verified Extraction from Coq to OCaml,” *Proc. ACM Program. Lang.*, vol. 8, no. PLDI, Jun. 2024, doi: 10.1145/3656379.

[6] T. Coquand, “Pattern matching with dependent types,” in *Informal proceedings of Logical Frameworks*, 1992, pp. 66–79.

[7] C. Cornes *et al.*, “The Coq Proof Assistant-Reference Manual,” *INRIA Rocquencourt and ENS Lyon, version*, vol. 5, 1996.

[8] E. Giménez, “Codifying Guarded Definitions with Recursive Schemes,” in *Types for Proofs and Programs, International Workshop TYPES'94, Båstad, Sweden, June 6-10, 1994, Selected Papers*, P. Dybjer, B. Nordström, and J. M. Smith, Eds., in Lecture Notes in Computer Science, vol. 996. Springer, 1994, pp. 39–59. doi: 10.1007/3-540-60579-7_3.

Bibliography

[9] C. Paulin-Mohring, *Définitions Inductives en Théorie des Types. (Inductive Definitions in Type Theory)*. 1996. [Online]. Available: https://tel.archives-ouvertes.fr/tel-00431817

[10] E. Giménez, "Un Calcul de Constructions Infinies et son application a la vérification de systemes communicants," 1996.

[11] P. Boutillier, "A relaxation of Coq's guard condition," in *JFLA - Journées Francophones des langages applicatifs - 2012*, Feb. 2012, pp. 1–14. [Online]. Available: https://hal.science/hal-00651780

[12] M. Dénès, "Tentative fix for the commutative cut subterm rule." [Online]. Available: https://github.com/coq/coq/commit/9b272a861bc3263c69b699cd2ac40ab2606543fa

[13] H. Herbelin, "Check guardedness of fixpoints also in erasable subterms." [Online]. Available: https://github.com/coq/coq/pull/15434

[14] H. Herbelin, “Extrude uniform parameters of inner fixpoints in guard condition check.” [Online]. Available: https://github.com/coq/coq/pull/17986

[15] J. Chan, Y. Li, and W. J. Bowman, “Is sized typing for Coq practical?,” *J. Funct. Program.*, vol. 33, p. e1, 2023, doi: 10.1017/S0956796822000120.

[16] K. Nisht and A. Abel, “Type-Based Termination Checking in Agda,” 30th International Conference on Types for Proofs, Programs, TYPES 2024, pp. 32–33, 2024. [Online]. Available: https://types2024.itu.dk/abstracts.pdf