Verified meta-theory at scale for a certified proof assistant

Meven Lennon-Bertrand

INI Big Specification Program – 17/10/24

University of Cambridge

Very useful!

Very useful! But what makes them usable?

Very useful! But what makes them usable?

Pattern-matching

(Computational) univalence

(Strong) records

Termination checking

(Co)Inductive types

Universes

Proof irrelevance

Very useful! But what makes them usable?

Pattern-matching

(Computational) univalence

(Strong) records

Termination checking

(Co)Inductive types

Universes

Proof irrelevance

Gradual typing

Modalities

Observational equality

Subtyping

Very useful! But what makes them usable?

Pattern-matching

(Computational) univalence

(Strong) records

Termination checking

(Co)Inductive types

Universes

Proof irrelevance

Gradual typing

Modalities

Observational equality

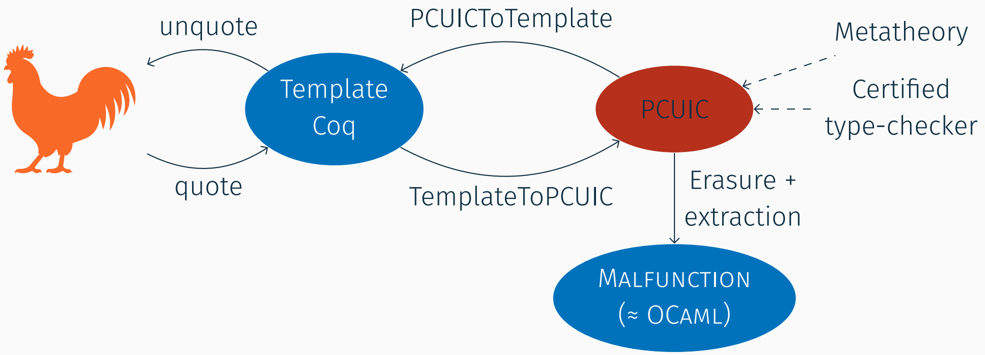
Subtyping

Real proof assistants are complicated!

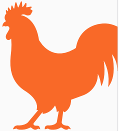
The MetaCoq project

MetaCoq is developed by (left to right) Abhishek Anand, Danil Annenkov, Simon Boulier, Cyril Cohen, Yannick Forster, Jason Gross, Meven Lennon-Bertrand, Kenji Maillard, Gregory Malecha, Jakob Botsch Nielsen, Matthieu Sozeau, Nicolas Tabareau and Théo Winterhalter.

The picture


unquote
PCUICToTemplate
Metatheory
Template Coq
PCUIC
Certified type-checker
quote
TemplateToPCUIC
Erasure + extraction
Malfunction (≈ OCaml)

THE PICTURE



1

Correct and Complete Type Checking and Certified Erasure for Coq, in Coq

MATTHIEU SOZEAU, Inria, France
YANNICK FORSTER, Inria, France
MEVEN LENNON-BERTRAND, University of Cambridge, United Kingdom
JAKOB BOTSCH NIELSEN, Concordium Blockchain Research Center, Denmark
NICOLAS TABAREAU, Inria, France
THÉO WINTERHALTER, Inria, France

Coq is built around a well-delimited kernel that performs type checking for definitions in a variant of the Calculus of Inductive Constructions (CIC). Although the metatheory of CIC is very stable and reliable, the correctness of its implementation in Coq is less clear. Indeed, implementing an efficient type checker for CIC is a rather complex task, and many parts of the code rely on implicit invariants which can easily be broken by further evolution of the code. Therefore, on average, one critical bug has been found every year in Coq. This paper presents the first implementation of a type checker for the kernel of Coq (without the module system, template polymorphism and η-conversion), which is proven sound and complete in Coq with respect to its formal specification. Note that because of Gödel's second incompleteness theorem, there is no hope to prove completely the soundness of the specification of Coq inside Coq (in particular strong normalization), but it is possible to prove the correctness and completeness of the implementation assuming soundness of the specification, thus moving from a trusted code base (TCB) to a trusted theory base (TTB) paradigm. Our work is based on the MetaCoq project which provides meta-programming facilities to work with terms and declarations at the level of the kernel. We verify a relatively efficient type checker based on the specification of the typing relation of the Polymorphic, Cumulative Calculus of Inductive Constructions (PCUIC) at the basis of Coq. It is worth mentioning that during the verification process, we have found a source of incompleteness in Coq's official type checker, which has then been fixed in Coq 8.14 thanks to our work. In addition to the kernel implementation, another essential feature of Coq is the so-called *extraction* mechanism: the production of executable code in functional languages from Coq definitions. We present a verified version of this subtle type and proof erasure step, therefore enabling the verified extraction of a safe type checker for Coq in the future.

CCS Concepts: • **Theory of computation → Type theory**.

etatheory

ertified
e-checker

Certifying Coq's type-checker

- Smaller spec than what the people here typically do
- But still a real-life system!
- The challenge is to **prove** things

Terms

```
Inductive term : Type :=
  | tRel (n : nat)
  | tVar (id : ident) | tEvar (ev : nat) (args : list term)
  | tLetIn (na : aname) (def : term) (def_ty : term) (body : term)
  | tSort (s : sort)
  | tProd (na : aname) (ty : term) (body : term)
  | tLambda (na : aname) (ty : term) (body : term)
  | tApp (u v : term)
  | tConst (c : kername) (u : Instance.t)
  | tInd (ind : inductive) (u : Instance.t)
  | tConstruct (ind : inductive) (idx : nat) (u : Instance.t)
  | tCase (ci : case_info) (type_info : predicate term)
    (discr : term) (branches : list (branch term))
  | tProj (proj : projection) (t : term)
  | tFix (mfix : mfixpoint term) (idx : nat)
  | tCoFix (mfix : mfixpoint term) (idx : nat)
  | tPrim (prim : prim_val term).
```

Typing

A few 100 lines of Coq:

Typing

A few 100 lines of Coq:

```
Inductive typing `{checker_flags} (Σ : global_env_ext) (Γ : context)
  : term → term → Type :=
  …
  | type_Lambda (na A t B) : lift_typing typing Σ Γ (j_vass na A) →
    Σ ;;; Γ ,, vass na A ⊢ t : B →
    Σ ;;; Γ ⊢ tLambda na A t : tProd na A B
```

A few 100 lines of Coq:

```
Inductive typing `{checker_flags} (Σ : global_env_ext) (Γ : context)
  : term → term → Type :=
  …
  | type_Lambda (na A t B) : lift_typing typing Σ Γ (j_vass na A) →
    Σ ;;; Γ ,, vass na A ⊢ t : B →
    Σ ;;; Γ ⊢ tLambda na A t : tProd na A B

| type_Case : forall ci p c brs indices ps mdecl idecl,
    let predctx := case_predicate_context ci.(ci_ind) mdecl idecl p in
    let ptm := it_mkLambda_or_LetIn predctx p.(preturn) in
    declared_inductive Σ ci.(ci_ind) mdecl idecl →
    Σ ;;; Γ ,,, predctx ⊢ p.(preturn) : tSort ps →
    Σ ;;; Γ ⊢ c : mkApps (tInd ci.(ci_ind) p.(puinst)) (p.(pparams) ++ indices) →
    case_side_conditions (fun Σ Γ ⇒ wf_local Σ Γ) typing Σ Γ ci p ps
                          mdecl idecl indices predctx  →
    case_branch_typing (fun Σ Γ ⇒ wf_local Σ Γ) typing Σ Γ ci p ps
                         mdecl idecl ptm brs →
    Σ ;;; Γ ⊢ tCase ci p c brs : mkApps ptm (indices ++ [c])
```

Where's the catch?

We can write a (minimalistic) kernel for Coq in a few kLoC of pure functional code.

Surely it can't be that hard to certify?

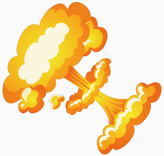
Where's the catch?

We can write a (minimalistic) kernel for Coq in a few kLoC of pure functional code.

Surely it can't be that hard to certify?

Dependent type theory + Invariants

=



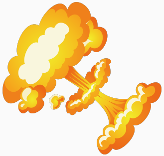
Where's the catch?

We can write a (minimalistic) kernel for Coq in a few kLoC of pure functional code.

Surely it can't be that hard to certify?

Dependent type theory + Invariants

=

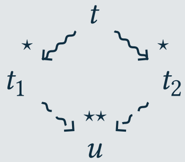


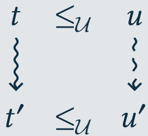
Similar issue if you try to prove safety = progress + preservation

Substitution

- substitution calculus
- universe and term substitution for cumulativity, typing, etc.

Confluence & Simulation





Injectivity (and no-confusion) of type constructors

- If $\Pi\, x{:}\, A.B \cong \Pi\, x{:}\, A'.B'$ then $A \cong A'$ and $B \cong B'$
- If $\Pi\, x{:}\, A.B \cong \mathbb{N}$ then $\bot$

What we have – Metatheory (II)

Subject reduction/Preservation

```
Theorem subject_reduction Σ Γ t u T : wf Σ →
  Σ ;;; Γ ⊢ t : T → Σ ;;; Γ ⊢ t ⇝ u → Σ ;;; Γ ⊢ u : T.
```

Progress

```
Lemma whnf_progress : ∀ Σ t T, axiom_free Σ → wf Σ →
  Σ ; [] ⊢ t : T →
  { t' & Σ ; [] ⊢ t ⇝ t' } ∨ whnf Σ [] t.
```

What we have – Metatheory (II)

Subject reduction/Preservation

```
Theorem subject_reduction Σ Γ t u T : wf Σ →
  Σ ;;; Γ ⊢ t : T → Σ ;;; Γ ⊢ t ⇝ u → Σ ;;; Γ ⊢ u : T.
```

Progress

```
Lemma whnf_progress : ∀ Σ t T, axiom_free Σ → wf Σ →
  Σ ; [] ⊢ t : T →
  { t' & Σ ; [] ⊢ t ⇝ t' } ∨ whnf Σ [] t.
```

+ normalisation ⟹

Canonicity

Every closed term of an inductive type evaluates to a constructor of that type.

Consistency

There are no closed proofs of an empty inductive type.

What we cannot have – normalisation

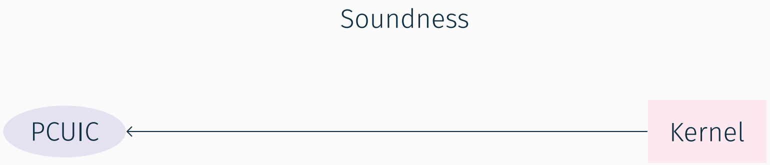
Normalisation is axiomatized

Normalisation is axiomatized

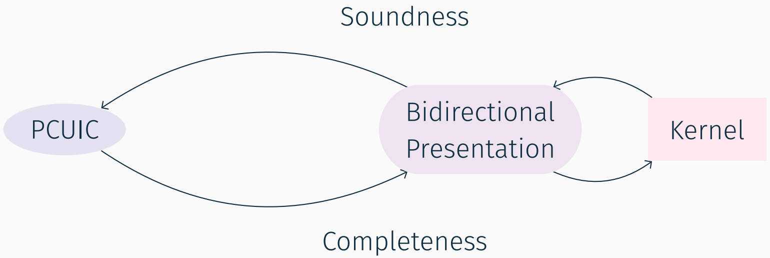
```
Class GuardCheckerCorrect :=
{
  guard_red1 b Σ Γ mfix mfix' idx :
    Σ ;;; Γ ⊢ tFixCoFix b mfix idx ⇝
      tFixCoFix b mfix' idx →
    guard b Σ Γ mfix → guard b Σ Γ mfix' ;
  …
}.
Axiom guard_checking_correct : GuardCheckerCorrect.

Axiom Normalization : forall Σ Γ t,
  wf_ext Σ → welltyped Σ Γ t → Acc (cored Σ Γ) t.
```

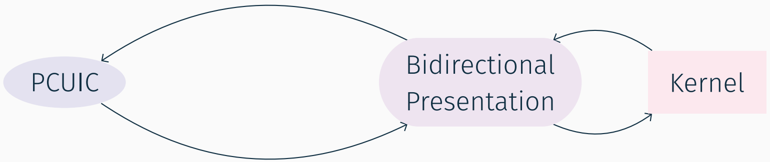
What we have – Type-checker



What we have – Type-checker

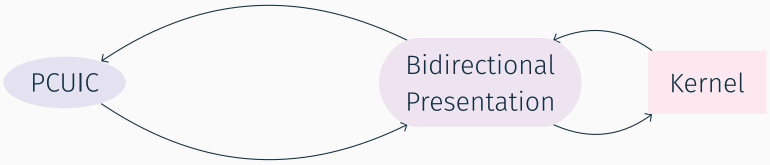


What we have – Type-checker

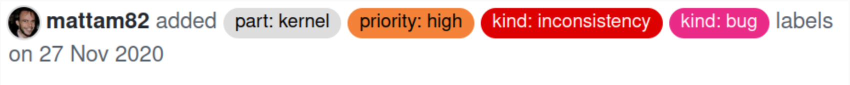


Deep in the proof, we realized… it was false!

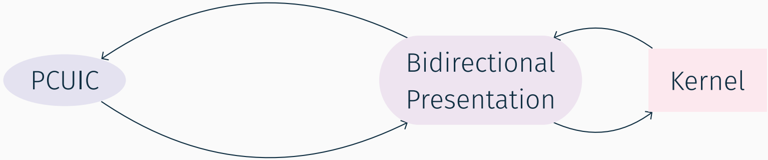
What we have – Type-checker



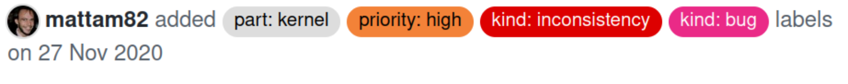
Deep in the proof, we realized… it was false!



What we have – Type-checker



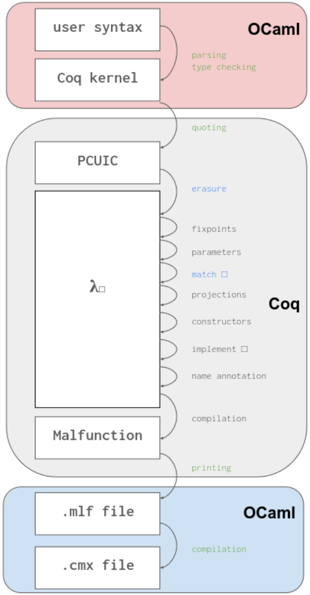
Deep in the proof, we realized... it was false!



→ re-design of pattern-matching in Coq, backed by MetaCoq.

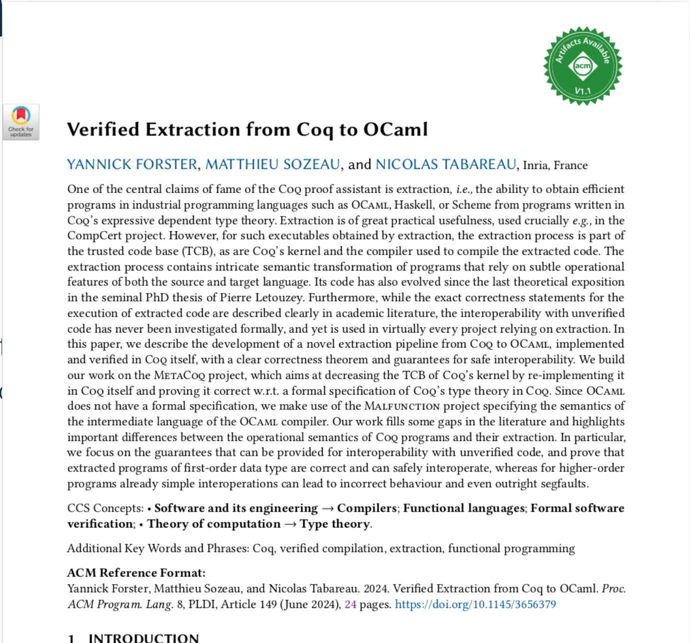
Extraction:

1. Erase proofs from programs: PCUIC → λ□
2. Compile λ□ to your favourite language (OCaml)



Extraction:

1. Erase proofs t
2. Compile λ☐ t



Verified Extraction from Coq to OCaml

YANNICK FORSTER, MATTHIEU SOZEAU, and NICOLAS TABAREAU, Inria, France

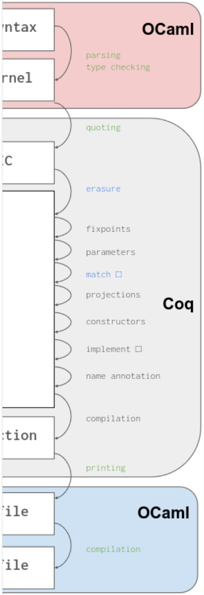
One of the central claims of fame of the Coq proof assistant is extraction, *i.e.*, the ability to obtain efficient programs in industrial programming languages such as OCaml, Haskell, or Scheme from programs written in Coq's expressive dependent type theory. Extraction is of great practical usefulness, used crucially *e.g.*, in the CompCert project. However, for such executables obtained by extraction, the extraction process is part of the trusted code base (TCB), as are Coq's kernel and the compiler used to compile the extracted code. The extraction process contains intricate semantic transformation of programs that rely on subtle operational features of both the source and target language. Its code has also evolved since the last theoretical exposition in the seminal PhD thesis of Pierre Letouzey. Furthermore, while the exact correctness statements for the execution of extracted code are described clearly in academic literature, the interoperability with unverified code has never been investigated formally, and yet is used in virtually every project relying on extraction. In this paper, we describe the development of a novel extraction pipeline from Coq to OCaml, implemented and verified in Coq itself, with a clear correctness theorem and guarantees for safe interoperability. We build our work on the MetaCoq project, which aims at decreasing the TCB of Coq's kernel by re-implementing it in Coq itself and proving it correct w.r.t. a formal specification of Coq's type theory in Coq. Since OCaml does not have a formal specification, we make use of the Malfunction project specifying the semantics of the intermediate language of the OCaml compiler. Our work fills some gaps in the literature and highlights important differences between the operational semantics of Coq programs and their extraction. In particular, we focus on the guarantees that can be provided for interoperability with unverified code, and prove that extracted programs of first-order data type are correct and can safely interoperate, whereas for higher-order programs already simple interoperations can lead to incorrect behaviour and even outright segfaults.

CCS Concepts: • **Software and its engineering** → **Compilers**; **Functional languages**; **Formal software verification**; • **Theory of computation** → **Type theory**.

Additional Key Words and Phrases: Coq, verified compilation, extraction, functional programming

ACM Reference Format:
Yannick Forster, Matthieu Sozeau, and Nicolas Tabareau. 2024. Verified Extraction from Coq to OCaml. *Proc. ACM Program. Lang.* 8, PLDI, Article 149 (June 2024), 24 pages. https://doi.org/10.1145/3656379

1 INTRODUCTION



And now?

We have a fully certified, extracted kernel!

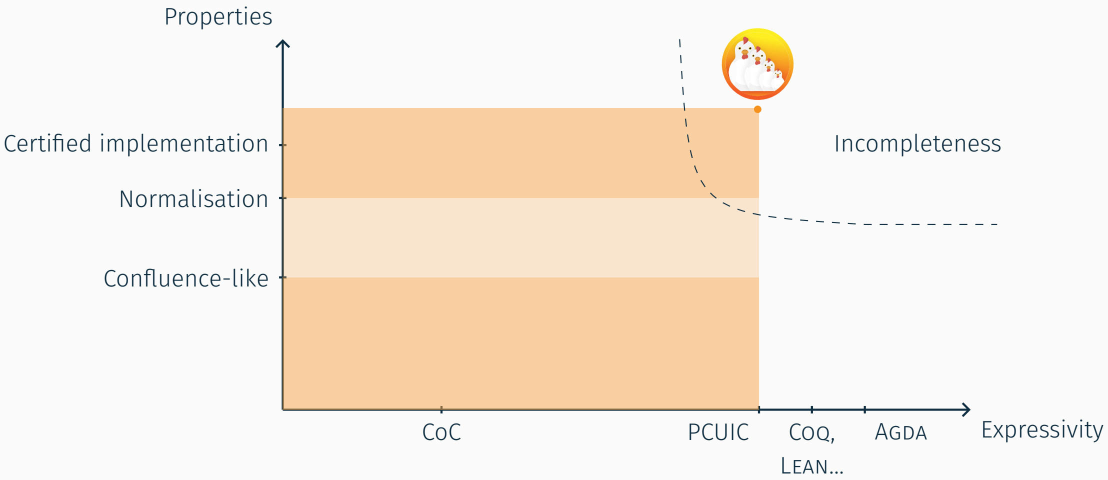
And now?

We have a **fully certified, extracted kernel**!

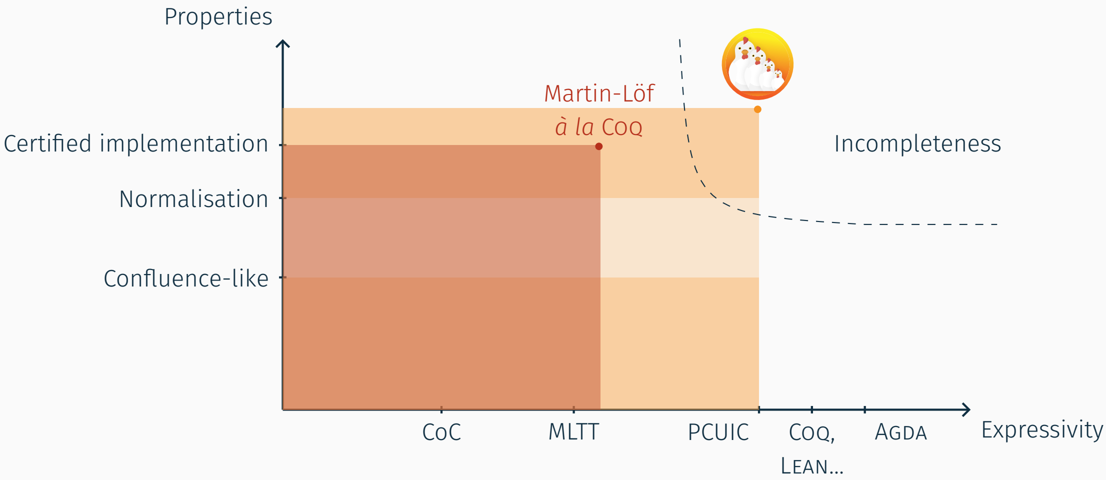
But:

- axiomatized normalisation → no guarantees on the guard;
- untyped conversion (not what semanticists like);
- missing some fancy features of Coq:
  - η rules
  - Sort/template polymorphism
  - Modules
  - …

Martin-Löf à la Coq


Properties
Certified implementation
Normalisation
Confluence-like
Incompleteness
CoC
PCUIC
Coq,
Lean...
Agda
Expressivity

A DIFFERENT ANGLE



The project in two words

- Even smaller system (not real life any more)
- But fancier proofs!
- Already miserable...

Some lessons we learned
(or we should have learned)

The good

- it's doable, now!
- we even found a bug in Coq: it's apparently useful to do the proofs
- starting to drive the design of the kernel

The good

- it's doable, now!
- we even found a bug in Coq: it's apparently useful to do the proofs
- starting to drive the design of the kernel

The bad

- still very heroic (>1y to change pattern-matching...)
- terrible proof engineering
    - too little automation
    - too many features of Coq
- very difficult to experiment (no modularity)
- how hard is the last yard going to be?

Tooling

- AutoSubst 2 (OCaml implementation)
- Winterhalter's PartialFun library for partial functions
- fancy induction stuff
- We could do so much more…

Tooling

- AutoSubst 2 (OCaml implementation)
- Winterhalter's PartialFun library for partial functions
- fancy induction stuff
- We could do so much more…

Meta-theory ≠ certification

- two very different problems
- ongoing: separating them cleanly

Tooling

- AutoSubst 2 (OCaml implementation)
- Winterhalter's PartialFun library for partial functions
- fancy induction stuff
- We could do so much more…

Meta-theory ≠ certification

- two very different problems
- ongoing: separating them cleanly

There must be a better way

- Iris-style embedded logic?
- quotient inductive-inductive types, second order generalized algebraic theory, synthetic Tait computability…
- Modularity, modularity, modularity

Thank you!