

# Coq Coq CODET!

Towards a Verified Toolchain for Coq in METACOQ

MATTHIEU SOZEAU, *π.r*<sup>2</sup>, Inria Paris, France

YANNICK FORSTER, Saarland University, Germany

SIMON BOULIER, NICOLAS TABAREAU, and THÉO WINTERHALTER, Gallinette, Inria Nantes, France

At the heart of Coq’s kernel is an implementation of a typechecker for the (Polymorphic, Cumulative) Calculus of Inductive Constructions (PCUIC), a well defined type-theoretic system. However, in its entirety, the kernel consists of 20k lines of relatively complicated OCaml code that has to be trusted. This is due on one hand to the complexity of Coq’s full calculus, including modules and sections, and on the other hand to the inclusion of high-performance conversion tests (bytecode and native compilation) along with user-friendly features like delayed checking of opaque proofs and support for nested global declarations. We report on our ongoing work on verifying a reasonably large subset of Coq’s kernel in Coq, as part of the METACOQ project [Anand et al. 2018; Sozeau et al. 2019].

The METACOQ project provides metaprogramming facilities to turn terms and declarations in Coq into a reified inductive datatype of CIC terms. We specify typing and reduction relations for PCUIC and prove meta-theoretical properties like confluence and subject reduction. Based on this specification, we then implement a verified, sound type checker.

In addition to the kernel implementation, an essential feature of Coq is so-called *extraction* [Letouzey 2004]: the production of executable code in functional languages from Coq definitions. We will present a verified version of this subtle type-and-proof erasure step, which links to the CERTICOQ verified compiler going from erased terms to COMPCERT’s C-light language. During the presentation, we hope to demonstrate uses of the checker and extraction procedure and provide empirical evaluation results.<sup>1</sup> This work also provides insights into possible simplifications or generalizations of the calculus and its implementation.

DEFINITION (COT COT CODET<sup>2</sup>). **French, Interjection:**

(1) *cackle* (the cry of a hen, especially one that has laid an egg)

METACOQ provides a reification of the syntax of the core language of Coq along with global environments made from inductive, constant and universe constraint declarations. The initial version of METACOQ also included an inductive definition of the typing and cumulativity relations for this term language, and a partially implemented type checker for it. These specifications and implementations were however not verified or tested: it turns out they were incomplete or simply wrong in many places. In this work, we complete the definitions and exercise them by verifying the basic metatheory of an alternative, equivalent presentation of the system (§1), define a sound typechecker for it (§2) and an erasure procedure (§3).

## 1 METATHEORY

The whole development starts from a representation of terms as used in the Coq kernel, called TEMPLATE-COQ. The only difference to the actual OCaml implementation is the use of lists and unbounded natural numbers in TEMPLATE-COQ whereas arrays and modular integers are used in Coq’s implementation. However, this representation is not well-suited for proofs: n-ary applications have to be kept in a canonical form (no left-nesting, non-empty list of arguments) and there is an inessential cast constructor that is only used to control which conversion machine to use while we only support one at the moment.

This motivates us to switch to an equivalent presentation of the calculus named PCUIC with binary applications and no casts. We can devise a type and reduction-preserving translation from TEMPLATE-COQ to PCUIC terms, by translating casts to  $\beta$ -redexes and forget about them in the metatheoretical study. Composition with the trivial reverse direction from PCUIC to TEMPLATE-COQ produces  $\beta$ -equivalent terms (up-to reducing trivial  $(\text{fun } x : T \Rightarrow x) t$   $\beta$ -redexes).

The typing relation enjoys the usual principles of **weakening** and **substitution**, along with weakening w.r.t well-formed inductive or constant declarations in the global environment. All these properties are derived using a general induction principle on typing derivations giving appropriate induction hypotheses on declarations in the local or global environment: this can be systematically derived using a well-founded induction on the size of derivations. We formally capture this using a proposition `env_prop P`, stating that the property  $P$  holds for well-typed terms and all entries in well-formed contexts.

We define **conversion and cumulativity** using a 1-step reduction relation and a syntactic check for  $\alpha$ -equality up to universes. We can prove **validity** of typing: `env_prop (fun  $\Sigma \Gamma t T \Rightarrow \text{isWfArity\_or\_Type } \Sigma \Gamma T$ )`.

There is a catch here: validity in this system does not say that the type itself is typeable, but rather that it is a well-formed arity  $\forall \Gamma, s$  for  $s$  a sort and  $\Gamma$  a well-typed context, or is typeable. This infelicity explains why plugin writers must “refresh” algebraic universes to fresh universe level variables when putting back an inferred type in a term position.

An important property of the calculus, necessary for Subject Reduction (SR), is **confluence**. We devise an original extension of the usual Tait/Martin-Löf parallel-reduction proof to handle the dependent `let in` of CIC, which cannot be simply modeled

<sup>1</sup> *Disclaimer:* at the time of writing, the remaining proofs to finish are Validity, Subject Reduction and Principality, along with admitted lemmas on the metatheory (file PCUICAdmittedLemmas in pcuic/theories summarizes them). Erasure and the verified typechecker are otherwise proven correct, but their extraction to ML or Haskell produces untypable terms, we are investigating extraction to Scheme.

<sup>2</sup> [https://en.wiktionary.org/wiki/cot\\_cot\\_codet](https://en.wiktionary.org/wiki/cot_cot_codet), see french version [https://fr.wiktionary.org/wiki/cot\\_cot\\_codet](https://fr.wiktionary.org/wiki/cot_cot_codet) for etymology.

as a  $\beta$  redex, as is generally done in the literature. Our generalized confluence lemma extends the so-called “triangle method” due to Takahashi [1989]<sup>3</sup> to talk about contexts of assumptions *and* definitions.

**Corollary confluence**  $\Sigma \Gamma \Delta \Delta' t u v : wf \Sigma \rightarrow pred1 \Sigma \Gamma \Delta t u \rightarrow pred1 \Sigma \Gamma \Delta' t v \rightarrow$

$let \Gamma' := rho\_ctx \Sigma \Gamma in let t' := rho \Sigma \Gamma' t in pred1 \Sigma \Delta \Gamma' u t' \times pred1 \Sigma \Delta' \Gamma' v t'.$

The contexts  $\Delta$  and  $\Delta'$  resulting from the “top” parallel reductions are joined using an explicit  $\rho$  function, similarly to the terms. Confluence of the transitive closure of 1-step reduction follows, from which we get injectivity of type constructors w.r.t. definitional equality. This in turn enables us to prove **Subject Reduction**, except for co-fixpoints on “positive” co-inductive types, which are known to break the property [Giménez 1996].

Finally, we *assume* the **Strong Normalization** (SN) of well-typed terms in this system (Gödel’s incompleteness theorem prevents a fully formal proof). We include a characterization of strict positivity for (co-)inductive types that any well-formed inductive must validate. However, we did not implement the (rather complex) guard-checking algorithm of Coq. To avoid introducing a trivially false assumption due to non-terminating fixpoints, we add another axiom `guarded : global_context  $\rightarrow$  context  $\rightarrow$  term  $\rightarrow$  bool` and premises `guarded  $\Sigma \Gamma t = true$`  to the (co-)fixpoint typing rules. We only assume that `guarded` is preserved by lifting and substitution, nothing else. This prevents deriving an inconsistency from these rules as there is no way to build a closed typing derivations involving (co-)fixpoints without further assumptions on `guarded`.

## 2 CHECKER

In order to write a sound *type checker*, we need to be able to decide untyped *conversion* and for this we need to implement a *reduction* machine. We implement weak-head reduction using a stack machine, mostly following the kernel implementation (leaving out sharing of reductions which requires an effect). Termination of such a reduction machine is non-trivial and relies on the strong normalization of the system: one needs to provide a certificate of welltypedness of the term to reduce. Once extracted, this certificate goes away, and it yields a partial function that is proven terminating on well-typed terms only. Correctness of the reduction machine itself w.r.t. the transitive closure of 1-step reduction is necessary to show termination and requires dealing with the notion of positions in the stack.

For conversion, we need to rely on an even more complex measure. In particular, we deal with mutually recursive functions to implement the reduction to weak head normal forms and comparisons of stacks or terms. To express the termination order depending on the situation and to “tie the knot”, we define a single function taking an argument specifying which subfunction it is supposed to act as. Conversion of terms works by putting its arguments in weak-head normal form (which may itself not change the arguments at all and must be followed by a call that does progress), then compares the heads of the terms, and then their arguments in case the heads do match.

Type checking and type inference are more straightforward and can be implemented by structural recursion on the syntax of terms. For checking, we simply perform type inference before comparing the two types for cumulativity.

## 3 ERASURE

The *extraction* procedure implemented in Coq translates PCUIC terms to functional programming languages (e.g. OCaml, Haskell or Scheme). Coq’s current erasure was implemented and justified on paper by Letouzey [2004]. We implement his erasure function in Coq, which translates from PCUIC to a similar, but untyped calculus which omits parts of terms that can only contain types and has no constructors to build e.g. function types.

The original correctness proof shows a small-step simulation result between a term and its erased form, using a simulation relation closed under substitution. We instead favor big-step semantics for the formalization, slightly simplifying the proof, while keeping the same simulation relation. We have to elaborate all cases in detail (dealing with the peculiar representation of fixpoints for example) and formalize many auxiliary lemmas not surfacing on paper.

The target type of erasure is in turn the starting point for CERTICOQ’s extraction to C-light code. This part of our work can also be seen as a first step towards building verified realizability models for Coq, which could be used to justify the consistency of logical extensions by axioms such as Markov’s principle.

## REFERENCES

- Abhishek Anand, Simon Boulier, Cyril Cohen, Matthieu Sozeau, and Nicolas Tabareau. 2018. Towards Certified Meta-Programming with Typed Template-Coq. In *ITP 2018 (Lecture Notes in Computer Science)*, Jeremy Avigad and Assia Mahboubi (Eds.), Vol. 10895. Springer, 20–39. [https://doi.org/10.1007/978-3-319-94821-8\\_2](https://doi.org/10.1007/978-3-319-94821-8_2)
- Carlos Eduardo Giménez. 1996. *Un calcul de constructions infinies et son application à la vérification de systèmes communicants*. Ph.D. Dissertation. Ecole Normale Supérieure de Lyon. <ftp://ftp.inria.fr/INRIA/LogiCal/Eduardo.Gimenez/thesis.ps.gz>
- Pierre Letouzey. 2004. *Programmation fonctionnelle certifiée: l’extraction de programmes dans l’assistant Coq*. Thèse de Doctorat. Université Paris-Sud. [http://www.pps.jussieu.fr/~letouzey/download/these\\_letouzey.pdf](http://www.pps.jussieu.fr/~letouzey/download/these_letouzey.pdf)
- Gert Smolka. 2015. Confluence and Normalization in Reduction Systems. (2015). <https://www.ps.uni-saarland.de/courses/sem-ws15/ars.pdf> Lecture Notes.
- Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. 2019. The MetaCoq Project. (April 2019). [http://www.irif.fr/~sozeau/research/publications/drafts/The\\_MetaCoq\\_Project.pdf](http://www.irif.fr/~sozeau/research/publications/drafts/The_MetaCoq_Project.pdf) Submitted.
- Masako Takahashi. 1989. Parallel reductions in  $\lambda$ -calculus. *Journal of Symbolic Computation* 7, 2 (1989), 113–123.

<sup>3</sup>Smolka [2015] provides a neat exposition of this technique