



The MetaCoq Project

and a bit about CertiCoq

Matthieu Sozeau, $\pi.r^2$, Inria Paris & IRIF

A. Anand, G. Malecha (BedRock Systems Inc.)

S. Boulrier, C. Cohen, N. Tabareau and T. Winterhalter (Inria)

Y. Forster and F. Kunze (Saarland University)

May 23rd 2019

Nomadic Labs, Paris, France

The MetaCoq Project Goals

Meta-programming support for Coq ([Anand et al., 2018](#)):

- 1 **Reification and denotation** of terms (Template-Coq)
- 2 **Monadic** interpreter for scripting vernacular commands
- 3 **Specification** of Coq's typing and operational semantics
- 4 **Correctness proof** of a functional type-checker for Coq

This is all work-in-progress, but documented ([Sozeau et al., 2019](#))!

Hopefully tractable enough to eventually prove:

- ▶ A verified unification algorithm
- ▶ A verified refiner
- ▶ A verified tactic language

Already good enough to **embed** DSLs.

It already provides enough expressivity **inside Coq** for correctness proofs of metaprograms such as:

- ▶ The CertiCoq compiler ([Anand et al., 2017](#))
- ▶ Parametricity, forcing translations ([Boulier et al., 2017](#))
- ▶ An extensional-to-intensional type theory translation ([Winterhalter et al., 2018](#))
- ▶ Translations from/to functional languages ([Zaliva and Sozeau, 2019](#))

- ▶ Started as an extension of Template-Coq by Malecha.
Malecha and Bengtson (2016) studied fully reflective tactics in Coq for the System F fragment.

- ▶ Started as an extension of Template-Coq by Malecha. Malecha and Bengtson (2016) studied fully reflective tactics in Coq for the System F fragment.
- ▶ MTac/MTac2 (Ziliani et al., 2015; Kaiser et al., 2018): A typed metaprogramming environment geared towards writing typed tactics.

- ▶ Started as an extension of Template-Coq by Malecha. Malecha and Bengtson (2016) studied fully reflective tactics in Coq for the System F fragment.
- ▶ MTac/MTac2 (Ziliani et al., 2015; Kaiser et al., 2018): A **typed** metaprogramming environment geared towards writing typed **tactics**.
- ▶ Coq in Coq (Barras, 1999): Metatheoretical proofs for a **model** of Coq, rather than the current **implementation**.
⇒ Hope for reuse

- ▶ Started as an extension of Template-Coq by Malecha. Malecha and Bengtson (2016) studied fully reflective tactics in Coq for the System F fragment.
- ▶ MTac/MTac2 (Ziliani et al., 2015; Kaiser et al., 2018): A typed metaprogramming environment geared towards writing typed tactics.
- ▶ Coq in Coq (Barras, 1999): Metatheoretical proofs for a model of Coq, rather than the current implementation.
⇒ Hope for reuse
- ▶ Idris, Agda and Lean have similar meta-programming frameworks.

- 1 Introduction
- 2 Reification and denotation: Template-Coq
 - Core language reification
 - Reifying commands: The Template Monad
- 3 Specifying Coq: Typing and the Checker
- 4 The calculus behind Coq: PCUIC
 - Formalizing an Erasure procedure
- 5 Applications
 - CertiCoq
 - Extensional to Intensional Type Theory revisited
- 6 Conclusion

- ▶ Initially developed by G. Malecha
- ▶ Quoting and unquoting of terms and declarations:
Quote Definition `quoted_t : Ast.t := t.`
Make Definition `denoted_t := quoted_t.`
- ▶ Ideally **faithful** representation of COQ terms
- ▶ Differences: Strings for `global_reference` and lists instead of arrays. But native integers and arrays are coming soon to Coq.

- ▶ Coq data structures: [Ast](#)
- ▶ [Demonstration](#)

- ▶ Terms using de Bruijn indices, with all constructors including Case, Fix, CoFix and polymorphic universes.
- ▶ Data structures to push definitions and inductive declarations to the kernel and retrieve information about constants and inductives from the kernel.
- ▶ **Missing:** the module system.

We need a way to communicate with the kernel, e.g. to add new definitions etc. Instead of special purpose commands we use a general monad.

- ▶ Coq data structures: [TemplateMonad](#)
- ▶ **Demonstration**

The Template Monad

- ▶ Allows to crawl the environment and modify it.
- ▶ Different from MTac's monad (shallow vs. deep embedding)
- ▶ **Extracted version** for compilation of plugins to OCaml (much much faster).
- ▶ Could be used to justify MTac2 programs and run them without oracles.

- 1 Introduction
- 2 Reification and denotation: Template-Coq
- 3 Specifying Coq: Typing and the Checker**
- 4 The calculus behind Coq: PCUIC
- 5 Applications
- 6 Conclusion

Typing

- ▶ Inductive specifications of typing, conversion and reduction on terms.
- ▶ Includes global environments and universes. An enhanced elimination principle transfers properties to local and global environments, defined using a measure on derivations.
- ▶ WIP: strict positivity and guard condition.
- ▶ Missing: existential variables and local named variables.
- ▶ Modules: P. M. Pédrot, Derek Dreyer and I have a plan for elaborating them (involving ω -universes)

- ▶ Reduction is specified as the reflexive, symmetric transitive closure of 1-step reduction.
- ▶ Cumulativity just compares the normal forms up-to the subtyping relation on universes.

Cumulativity

- ▶ Reduction is specified as the reflexive, symmetric transitive closure of 1-step reduction.
- ▶ Cumulativity just compares the normal forms up-to the subtyping relation on universes.
- ▶ Standard proof of Church-Rosser (defining 1-step parallel reduction first) would be a nightmare in the implemented representation.

- ▶ Reduction is specified as the reflexive, symmetric transitive closure of 1-step reduction.
- ▶ Cumulativity just compares the normal forms up-to the subtyping relation on universes.
- ▶ Standard proof of Church-Rosser (defining 1-step parallel reduction first) would be a nightmare in the implemented representation.
 - ▶ Need a well-formedness predicate to be maintained everywhere for applications: $\text{tApp} : \text{term} \rightarrow \text{list term} \rightarrow \text{term}$. Invariant: no nested applications and no empty list of arguments. Coq's ML code uses a smart constructor to ensure that.
 - ▶ Interaction with tCast is non-trivial: e.g. term equality must be up-to casts, which might appear at heads of applications. It is **not** structurally recursive.

Cumulativity

- ▶ Reduction is specified as the reflexive, symmetric transitive closure of 1-step reduction.
- ▶ Cumulativity just compares the normal forms up-to the subtyping relation on universes.
- ▶ Standard proof of Church-Rosser (defining 1-step parallel reduction first) would be a nightmare in the implemented representation.
 - ▶ Need a well-formedness predicate to be maintained everywhere for applications: $\text{tApp} : \text{term} \rightarrow \text{list term} \rightarrow \text{term}$. Invariant: no nested applications and no empty list of arguments. Coq's ML code uses a smart constructor to ensure that.
 - ▶ Interaction with tCast is non-trivial: e.g. term equality must be up-to casts, which might appear at heads of applications. It is **not** structurally recursive.

Solution: transfer metatheoretical proofs from a cleaned-up representation: PCUIC.

Weak head call-by-name reduction, conversion and type inference are implemented.

- ▶ Using a stack machine (without sharing) for head reduction.
- ▶ Uses fuel to run inside Coq (**upcoming**: fuel-free reduction assuming SN for the 1-step reduction relation).
- ▶ Partial **correctness** proof w.r.t. the typing specification. (**upcoming**: total correctness and completeness using fuel-free conversion).

Showing that the reduction/conversion implementation is correct w.r.t. small or big step operational semantics requires a few refinement steps ([Kunze et al., 2018](#)).

- ▶ Using Coq's extraction, we get an alternative, partially certified checker for Coq definitions \Rightarrow **more trust!**
- ▶ Runs in reasonable time on medium sized definitions (e.g. recursive definitions by well-founded recursion).

- 1 Refinements to efficient implementations closer to Coq 's implementation
 - ▶ Verifying the universe constraint algorithm (A. Guéneau and J.H. Jourdan).
 - ▶ Link to a Rust checker developed at MPI-SWS, implementing sharing in reduction.
- 2 Link to more ideal type theories like the calculus used in Coq in Coq for which SN is proved:
 - ▶ Simpler presentations of inductive types (W-types, containers)
 - ▶ Removing the global environment/delta reduction
 - ▶ Simpler universe systems without polymorphism or cumulative inductive types.

- 1 Introduction
- 2 Reification and denotation: Template-Coq
 - Core language reification
 - Reifying commands: The Template Monad
- 3 Specifying Coq: Typing and the Checker
- 4 The calculus behind Coq: PCUIC
 - Formalizing an Erasure procedure
- 5 Applications
 - CertiCoq
 - Extensional to Intensional Type Theory revisited
- 6 Conclusion

We need a cleaner version of the calculus for metatheoretical proofs. PCUIC has the same features except:

- ▶ Applications are binary and all terms are well-formed (for substitution and lifting for example)

We need a cleaner version of the calculus for metatheoretical proofs. PCUIC has the same features except:

- ▶ Applications are binary and all terms are well-formed (for substitution and lifting for example)
- ▶ Casts are removed: replaced by identity function applications to preserve reductions.
- ▶ Typing derivations can be transferred from Template-Coq to PCUIC.

We need a cleaner version of the calculus for metatheoretical proofs. PCUIC has the same features except:

- ▶ Applications are binary and all terms are well-formed (for substitution and lifting for example)
- ▶ Casts are removed: replaced by identity function applications to preserve reductions.
- ▶ Typing derivations can be transferred from Template-Coq to PCUIC.
- ▶ PCUIC's typing and reduction relations are simpler and enjoy weakening (for global and local environments) and substitution.
- ▶ WIP: confluence (using a novel parallel reduction notion handling lets) and subject reduction (standard except for cofixpoints)

PCUIC is close to the calculi we have shown consistency for in [Timany and Sozeau \(2018\)](#), except for the presentation of eliminators of inductives.

- ▶ Can we formally show an equivalence of presentation between `fix+match` and eliminators?
- ▶ Working idea: use a translation to well-founded definitions on the subterm relation (e.g. as done in Paulin-Mohring's HDR).

Terminology: “Extraction” should be reserved to extraction of programs from proofs (e.g. classical realisability interpretations).

In Coq, most of the time, we “extract” programs from programs (mainly a pretty-printing operation).

Erasure removes proofs (terms of type $P : \mathbf{Prop}$) and types (terms of type \mathbf{Type}).

- ▶ Easy to formalize as a translation from PCUIC to an untyped call-by-value lambda-calculus, relying on the certified checker for type inference: need to decide if something is a type or proof.
- ▶ Goal: prove it preserves observational equivalence, for erasure of closed terms of informative inductive types.

```
∀ sigma t T v : Ast.term,  
  sigma ;; [] |- t : T -> axiom_free sigma ->  
  t ~>_wcbv v →  
  ∃ v' : ErasedAst.term,  
    erase sigma t ~>_wcbv v' ∧ v ~_T v'
```

We assume canonicity. Observational equivalence at:

- ▶ propositional types is the full relation
- ▶ inductive types relates the same constructors applied to related arguments.
- ▶ functions types preserves relatedness from arguments to applications.

WIP Formalization of the proof of [Letouzey \(2004\)](#), without the $\text{Prop} \leq \text{Type}$ rule.

- ▶ Mullen et al. (2018) have a verified compiler “ $\mathcal{C}\epsilon\text{uf}$ ” for the System F fragment + eliminators of simple inductive types (no `fix` + `match` support).
- ▶ Forster and Kunze (2019) verify a simpler translation from the non-polymorphic fragment of Coq **with time bounds**, to CBV lambda-calculus, interpreting inductives using impredicative encodings. Aside: they show this model of calculus is polynomially equivalent to Turing machines.
- ▶ None profit from the **Prop/Type** separation!
- ▶ Essential to get small and fast erased terms: just the computational content.

- 1 Introduction
- 2 Reification and denotation: Template-Coq
 - Core language reification
 - Reifying commands: The Template Monad
- 3 Specifying Coq: Typing and the Checker
- 4 The calculus behind Coq: PCUIC
 - Formalizing an Erasure procedure
- 5 Applications
 - CertiCoq
 - Extensional to Intensional Type Theory revisited
- 6 Conclusion

CertiCoq is a certified compiler from **erased** terms to CompCert's C-light (or simply C). The verified compiler phases include:

- ▶ Eta-expansion of constructors
- ▶ Removal of constructor parameters
- ▶ Transformation of the global environment to local let-ins.
- ▶ CPS conversion
- ▶ Closure conversion, defunctionalization (safe-for-space)
- ▶ Shrink reduction (removes administrative redexes)
- ▶ Link to a verified garbage collector.
- ▶ Resulting code can be compiled to assembly via CompCert or gcc.

The compiler is shown to preserve observational equivalence for weak call-by-value reduction.

$$\forall G \ t \ T \ v : \text{Ast.term},$$

$$G \ ; \ ; \ [] \ |- \ t : T \ \rightarrow \ \text{axiom_free } G \ \rightarrow$$

$$t \ \sim\>_wcbv \ v \ \rightarrow$$

$$\exists v' : \text{CLight.syntax}, \ \text{erase } t \ \sim\>_wcbv \ v' \ \wedge \ v \ \sim_T \ v'$$

- ▶ “Simple” proofs: Forward simulations only, starting from a strongly normalizing calculus.
- ▶ Erasure at the start of the pipeline: drastically reduces term size.
- ▶ Issues with Coq’s representation: mutual fixpoints as blocks (duplication), lambdas in `match` branches (P. M. Pédrot is on it).
- ▶ The **extracted** (via Coq’s extraction) version of the compiler is reasonably fast otherwise. Impractical inside Coq mainly due to string representation of references.

- ▶ Together with extraction correctness and CompCert's correctness:
The value you compute in COQ is the one computed by the assembly, if CompCert can produce it. Extraction and CertiCoq are total functions on well-typed COQ terms.
- ▶ Termination is guaranteed.
- ▶ If you can interpret your language in COQ (e.g. using a monadic encoding for effects), you get a certified, relatively efficient interpreter for free.

Future work:

- ▶ Reflexive tactics you can trust: replace `vm_compute` and `native_compute` with a trustable implementation.
- ▶ Interface with VST for linking with verified C implementations (e.g. an efficient union-find implementation, or OS API).

$$\frac{\Gamma \vdash p : t = u}{\Gamma \vdash t \equiv u}$$

- ▶ Idea: take a derivation in ETT (with the reflection rule) and translate it to a decorated derivation in ITT.
- ▶ We verified a variant of Oury's translation, using ideas from the parametricity translation. It assumes uniqueness of identity proofs and functional extensionality in the target theory.
- ▶ MetaCoq is used to produce derivations in ETT, by quoting a partially typed term defined in Coq.
- ▶ The ITT theory can be interpreted in MetaCoq. We denote the result of the translation + obligations corresponding to uses of the reflection rule.

ETT to ITT Example

```
Definition vrev {A n m} (v : vec A n) (acc : vec A m) : vec A (n + m) :=
vec_rect A (fun n _ => forall m, vec A m -> vec A (n + m))
  (fun m acc => acc)
  (fun a n _ rv m acc => {!rv _ (vcons a m acc)!})
  n v m acc.
```

- 1 Quote to MetaCoq
- 2 Translate to ETT adding type annotations using a retyping algorithm. ETT supports eliminators only (no fix+match).
- 3 Apply translation to ITT, generating obligations for conversions
- 4 Extract to MetaCoq a Coq term to denote along with obligations.
- 5 Run a meta-program asking the user to prove the obligations and defining the completed term.

```
Definition vrev {A n m} (v : vec A n) (acc : vec A m)
  : vec A (n + m) :=
vec_rect A (fun n _ => forall m, vec A m -> vec A (n + m))
  (fun m acc => acc)
  (fun a n _ rv m acc =>
    transport
      (vrev_obligation3 A n m v acc a n0 v0 rv m0 acc0)
      (rv (S m0) (vcons a m0 acc0)))
  n v m acc.
```

- ▶ Programmed and evaluated entirely in Coq!
- ▶ ETT to ITT is a translation of **derivations** and not only **terms**: computationally intensive.

- ▶ Translation to a CBV lambda-calculus with translation validation ([Forster and Smolka, 2017](#)).
- ▶ Parametricity translation with stronger free theorems for Prop ([Anand and Morrisett, 2017](#)).
- ▶ Formalization of syntactic models ([Boulier et al., 2017](#)).
- ▶ ...

- ▶ MetaCoq provides meta-programming features on top of Coq
- ▶ It allows implementation, verification of those meta-programs w.r.t. operational or typing semantics, and their evaluation inside Coq, through extraction to ML, or through compilation with CertiCoq.
- ▶ It includes a (partially verified) checker and extraction.
- ▶ It has interesting applications!

We are working on completing the formalization of the metatheory and existing translations. Research problems:

- ▶ Treatment of inductive types and recursion during type-checking.
- ▶ Proper support of meta-programming constructs like splicing, staging?

Write your plugins in Coq!

Certify them in Coq!

Run them natively using a certified compiler!

<http://metacoq.github.io/metacoq>

- A. Anand and G. Morrisett. Revisiting Parametricity: Inductives and Uniformity of Propositions. *ArXiv e-prints*, May 2017.
- Abhishek Anand, Andrew Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Belanger, Matthieu Sozeau, and Matthew Weaver. [CertiCoq: A verified compiler for Coq](#). In *CoqPL*, Paris, France, 2017.
- Abhishek Anand, Simon Boulier, Cyril Cohen, Matthieu Sozeau, and Nicolas Tabareau. [Towards Certified Meta-Programming with Typed Template-Coq](#). In Jeremy Avigad and Assia Mahboubi, editors, *ITP 2018*, volume 10895 of *Lecture Notes in Computer Science*, pages 20–39. Springer, 2018.
- Bruno Barras. [Auto-validation d'un système de preuves avec familles inductives](#). Thèse de doctorat, Université Paris 7, November 1999.
- Simon Boulier, Pierre-Marie Pédro, and Nicolas Tabareau. [The next 700 syntactical models of type theory](#). In *Certified Programs and Proofs (CPP 2017)*, pages 182 – 194, Paris, France, January 2017.
- Yannick Forster and Fabian Kunze. [A certifying extraction with time bounds from Coq to call-by-value \$\lambda\$ -calculus](#). April 2019.

- Yannick Forster and Gert Smolka. [Weak Call-by-Value Lambda Calculus as a Model of Computation in Coq](#). In Mauricio Ayala-Rincón and César A. Muñoz, editors, *Interactive Theorem Proving - 8th International Conference, ITP 2017, Brasília, Brazil, September 26-29, 2017, Proceedings*, volume 10499 of *Lecture Notes in Computer Science*, pages 189–206. Springer, 2017.
- Jan-Oliver Kaiser, Beta Ziliani, Robbert Krebbers, Yann Régis-Gianas, and Derek Dreyer. [Mtac2: typed tactics for backward reasoning in Coq](#). *PACMPL*, 2(ICFP):78:1–78:31, 2018.
- Fabian Kunze, Gert Smolka, and Yannick Forster. [Formal Small-step Verification of a Call-by-value Lambda Calculus Machine](#). *CoRR*, abs/1806.03205, 2018.
- Pierre Letouzey. [Programmation fonctionnelle certifiée: l'extraction de programmes dans l'assistant Coq](#). Thèse de doctorat, Université Paris-Sud, July 2004.
- Gregory Malecha and Jesper Bengtson. [Extensible and Efficient Automation Through Reflective Tactics](#). In Peter Thiemann, editor, *ESOP*, pages 532–559, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

- Eric Mullen, Stuart Pernsteiner, James R. Wilcox, Zachary Tatlock, and Dan Grossman. [Cœuf: minimizing the Coq extraction TCB](#). In June Andronick and Amy P. Felty, editors, *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018, Los Angeles, CA, USA, January 8-9, 2018*, pages 172–185. ACM, 2018.
- Matthieu Sozeau, Abhishek Anand, Simon Boulter, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. [The MetaCoq Project](#). Submitted, May 2019.
- Amin Timany and Matthieu Sozeau. [Cumulative Inductive Types In Coq](#). In Hélène Kirchner, editor, *3rd International Conference on Formal Structures for Computation and Deduction, FSCD 2018, July 9-12, 2018, Oxford, UK*, volume 108 of *LIPICs*, pages 29:1–29:16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018.
- Théo Winterhalter, Matthieu Sozeau, and Nicolas Tabareau. [Eliminating Reflection from Type Theory](#). 2018. Accepted at CPP'19.
- Vadim Zaliva and Matthieu Sozeau. [Reification of shallow-embedded DSLs in Coq with automated verification](#). In *CoqPL*, Cascais, Portugal, 2019.
- Beta Ziliani, Derek Dreyer, Neelakantan R. Krishnaswami, Aleksandar Nanevski, and Viktor Vafeiadis. [Mtac: A monad for typed tactic programming in Coq](#). *J. Funct. Program.*, 25, 2015.