The MetaCoq Project

Matthieu Sozeau, $\pi.r^2$, Inria Paris & IRIF
j.w.w. A. Anand (Cornell), S. Boulier, N. Tabareau and T. Winterhalter (Gallinette) and C. Cohen (Marelle)

VALS Seminar
October 26th 2018
LRI, Gif-sur-Yvette, 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!

Hopefully tractable enough to eventually prove:

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

The MetaCoq Project Users

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)

History

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

History

- 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):
  A typed metaprogramming environment geared towards writting typed tactics.

History

- 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):
  A typed metaprogramming environment geared towards writting typed tactics.
- Coq in Coq (Barras, 1999):
  Metatheoretical proofs for a model of COQ, rather than the current implementation.
  ⇒ Hope for reuse

History

- 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):
  A typed metaprogramming environment geared towards writting 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.

The MetaCoq Project

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 Extraction procedure
5. Applications
   - CertiCoq
   - Extensional to Intensional Type Theory revisited
6. Conclusion

Template-Coq

- 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.

Template-Coq

- Coq data structures: Ast
- Demonstration

Template-Coq

- 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.

The Template Monad

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)
- WIP: extracted version for compilation of plugins to OCaml
- Could be used to justify MTac2 programs and run them without oracles.

The MetaCoq Project

Typing, Conversion and the Checker

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: PMP, Derek Dreyer, Joshua Yanovski and I have a plan for elaborating them (involving ω-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.

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.

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: `tApp : term -> list term -> 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: `tApp : term -> list term -> 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.

Checker

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.
- Partial correctness proof w.r.t. the typing specification.

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).

Running the Checker

- Using extraction, we can get an alternative checker for COQ definitions.
- Runs in reasonable time on medium sized definitions (e.g. recursive definitions by well-founded recursion).

Two directions

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.

The MetaCoq Project

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)

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 TemplateCoq to PCUIC.

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 TemplateCoq to PCUIC.
- PCUIC's typing and reduction relations are simpler and enjoy weakening (for global and local environments) and substitution.
- WIP: confluence and subject reduction (standard except for cofixpoints)

PCUIC

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).

Extraction

Extraction removes proofs and types.

- Easy to formalize as a translation from PCUIC to a call-by-value lambda-calculus.
- Goal: prove it preserves observational equivalence, for extraction 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,
   extract sigma t ~>_wcbv v' ∧ v ~_T v'
```

Extraction

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.

Formalization of the proof of Letouzey (2004), without the `Prop` $\leq$ `Type` rule.

The MetaCoq Project

CertiCoq

CertiCoq is a certified compiler from extracted terms to CompCert's C-light. 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.
- Shrink reduction (removes administrative redexes)
- Link to a verified garbage collector.
- Resulting code can be compiled to assembly via CompCert or gcc.

CertiCoq

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

```
∀ G t T v : Ast.term,
 G ;; [] |- t : T -> axiom_free G ->
 t ~>_wcbv v →
 ∃ v' : CLight.syntax, extract t ~>_wcbv v' ∧ v ~_T v'
```

- Proofs are relatively straightforward thanks to forward simulations only, starting from a strongly normalizing calculus.
- We moved extraction at the start of the compilation pipeline to avoid size explosions.
- Issues with COQ's representation: mutual fixpoints as blocks (duplication), lambdas in `match` branches.
- The extracted version of the compiler is reasonably fast otherwise. Impractical inside COQ mainly due to string representation of references.

ETT to ITT

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

- 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.
- Template-Coq is used to produce derivations in ETT, by quoting a partially typed term defined in Coq.
- The ITT theory can be interpeted in Template-Coq. 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 Template-Coq
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 Template-Coq a Coq term to denote along with obligations.
5. Run a template program asking the user to prove the obligations and defining the completed term.

ETT to ITT Result

```
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.

Other applications

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

Summary

- 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 or through extraction to ML.
- 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:

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

The End

Write your plugins in Coq!

Certify them in Coq!

Run them natively using a certified compiler!

http://template-coq.github.io/template-coq

Bibliography

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 Pédrot, 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 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.

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.

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. Submitted.

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.