Introducing MetaCoq: A Safe Tactic Language for Coq

Beta Ziliani
FAMAF, Universidad Nacional de Córdoba and CONICET
bziliani@famaf.unc.edu.ar

Interactive proof assistants like Coq are now common tools employed by researchers worldwide to inspire confidence in their results. Noteworthy examples include major milestones in the verification of large algebraic proofs (Gonthier et al. 2013; Hales et al. 2015), and in the verification of large software systems (Klein et al. 2010; Leroy 2009).

These assistants owe their success partially to the rich higher-order logics they encode, which allow for the specification and verification of sophisticated theorems such as the ones listed above. However, what is their gain is also their curse: such highly undecidable logics do not allow for the same level of automation SMT solvers provide for fragments of first-order logic. As a consequence, the proof developer must often write several lines of proofs to solve goals, even trivial ones that do not appear in their traditional pen-and-paper counterparts.

To accommodate for this, proof assistants are equipped with languages to build *tactics*: programs that decompose a given goal into smaller *subgoals* until their truth is self-evident. Coq, in particular, includes two languages: OCaml and Ltac. The former is the language used to implement Coq itself. Tactics written in OCaml are compiled and can make use of imperative data structures—making them very efficient. At the same time, developers are exposed the low level details of Coq's internals, including potentially unsafe operations. Moreover, the tactic development process is slow. For one, the proof developer must reason at the level of *de Bruijn* indices, carefully making sure that such indices refers to the right binders. In addition, OCaml tactics must be compiled and linked to Coq every time the tactic is modified.

Ltac, on the other hand, is a dynamic language that allows for a rapid high-level development of tactics, directly within the Coq environment and without requiring compilation and linking. Moreover, it provides a convenient representation of proof terms using the concrete syntax of Coq: it frees the developer to think to the level of de Bruijn indices. However, this language grew "one hack at a time", and despite its many years in the wild, it still lacks many basic language constructs required for proper tactic development—a fact reflected by a growing number of domain-specific tactic languages that are written either as plugins in OCaml (*e.g.*, Gonthier and Mahboubi 2010), or in sophisticated patterns using Coq's overloading mechanisms (*e.g.*, Krebbers et al. 2017).

Another issue with languages like OCaml and Ltac is that they offer almost no static guarantees: a tactic that is successfully defined—that is, accepted by the OCaml compiler or Ltac's interpreter—may construct an ill-typed term that is only rejected when it is too late to understand where the problem originated from.

Recently, Ziliani et al. (2015) devised Mtac, a new language providing the static guarantees tactic languages are currently missing. Mtac is based on the key realization that a tactic language is just a functional language with certain *effects*, like non-termination and syntax manipulation. These effects can be typed in Gallina, the language in which definitions are written in Coq, by means of a monad—not unlike Haskell's `IOMonad`. As a result, a program of type $\mathsf{M}\ \tau$, where M is the monad and τ is any type in Gallina, has the guarantee that, if it terminates, the resulting term will have type τ.

While Mtac allows for the construction of more principled programs than Ltac, it still suffers from a problem shared with Ltac: it does not include enough language constructs to enable the construction of primitive tactics and tactic combinators.

In this talk: The goal for the talk is to present and discuss MetaCoq, a new framework—based on an improved version of Mtac—for writing *typed* tactics and tactic combinators. Tactics in MetaCoq are easy to write, to combine, and to modularize.

In its current form, MetaCoq consists of:

1. Mtac2, a new version of Mtac with a richer set of language constructs and revised semantics.
2. A novel extendable interface for manipulating goals.
3. Several basic tactics (proving MetaCoq's versatility).
4. A new proof mode to write scripts directly in MetaCoq.

MetaCoq is a plugin for Coq developed in collaboration with Yann Régis-Gianas and Jan-Oliver Kaiser. It is downloadable from:

`http://github.com/MetaCoq/MetaCoq`

References

G. Gonthier and A. Mahboubi. An introduction to small scale reflection in Coq. *JFR*, 3(2):95–152, 2010.

G. Gonthier, A. Asperti, J. Avigad, Y. Bertot, C. Cohen, F. Garillot, S. Le Roux, A. Mahboubi, R. O'Connor, S. Ould Biha, I. Pasca, L. Rideau, A. Solovyev, E. Tassi, and L. Théry. A machine-checked proof of the odd order theorem. In *ITP*. Springer, 2013.

T. C. Hales, M. Adams, G. Bauer, D. T. Dang, J. Harrison, T. L. Hoang, C. Kaliszyk, V. Magron, S. McLaughlin, T. T. Nguyen, T. Q. Nguyen, T. Nipkow, S. Obua, J. Pleso, J. Rute, A. Solovyev, A. H. T. Ta, T. N. Tran, D. T. Trieu, J. Urban, K. K. Vu, and R. Zumkeller. A formal proof of the kepler conjecture. *arXiv*, 1501.02155, 2015.

G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an operating-system kernel. *CACM*, 53(6):107–115, 2010.

R. Krebbers, A. Timany, and L. Birkedal. Interactive proofs in higher-order concurrent separation logic. In *POPL*, 2017.

X. Leroy. Formal verification of a realistic compiler. *CACM*, 52 (7):107–115, 2009.

B. Ziliani, D. Dreyer, N. R. Krishnaswami, A. Nanevski, and V. Vafeiadis. Mtac: A monad for typed tactic programming in Coq. *JFP*, 25, 2015.