Code Generation via Meta-programming in Dependently Typed Proof Assistants

Mathis Bouverot-Dupuis[1,2] and Yannick Forster[1]

[1] Inria Paris, France
[2] ENS Paris

Abstract

Dependently typed proof assistants offer powerful meta-programming features, which users can take advantage of to implement proof automation or compile-time code generation. This paper surveys meta-programming frameworks in Rocq, Agda, and Lean, with seven implementations of a running example: deriving instances for the `Functor` type class. This example is fairly simple, but sufficiently realistic to highlight recurring difficulties with meta-programming: conceptual limitations of frameworks such as term representation – and in particular binder representation –, meta-language expressiveness, and verifiability as well as current limitations such as API completeness, learning curve, and prover state management, which could in principle be remedied. We conclude with insights regarding features an ideal meta-programming framework should provide. A full experience report in paper form is accessible at https://hal.science/hal-05024207.

All proof assistants support user-extensible tactics and code generation through different meta-programming frameworks. Meta-programs are programs which produce or manipulate other programs as data. They can in particular be used to generate boilerplate code, i.e. code that can be mechanically derived from definitions, thereby increasing the productivity of proof assistant users. Common examples are induction principles [25, 12], equality deciders [25, 10], finiteness proofs [5], countability proofs [5], or substitution functions for syntax [22]. Naturally, the default meta-programming language of a proof assistant is its implementation language, and several proof assistants even come with multiple independent meta-programming frameworks. However, we can observe that meta-programming is not wide-spread on the example of boilerplate generation tools which often fall into one of the following. Either proof assistants come with built-in boilerplate generation support (such as induction principles or type class instances) which is widely used, or tools for generating boilerplate are developed, but not adopted by the community [25, 10, 3]. Lastly, many papers remark that automatic boilerplate generation would be feasible and interesting, but do not carry it out [27, 29, 9, 7]. Furthermore, subcommunities often seem to be split into silos regarding frameworks and we are not aware of scientific comparative work between different frameworks and proof assistants. The notable exception is Dubois de Prisque's PhD thesis [6], using several meta-programming frameworks in Rocq, but not coming with one central example implemented in different frameworks and focusing solely on Rocq.

An additional barrier for adoption is that most frameworks are organically grown and documentation is not accessible to non-experts: pros and cons are often implicitly known by developers but not readily accessible. In fact the situation is so chaotic that, at times, in order to generate boilerplate code authors create ad-hoc meta-programming facilities from scratch [22, 13, 23, 11] instead of taking advantage of existing meta-programming facilities.

On the other hand, the vast choice of meta-programming frameworks also hints that we are at the point where enough evidence is available to evaluate the state of the art and suggest future developments. In this paper, we focus on three major dependently typed proof assistants based on the Calculus of Inductive Constructions (CIC) [4, 17]: Rocq [26], Agda [15, 16], and Lean 4 [14]. We survey their respective meta-programming frameworks: for Rocq we cover OCaml plugins, MetaRocq [20, 21], Ltac2 [19], Elpi [24, 25], and furthermore Agda's Reflection API [28] and Lean 4's meta-programming API.[1] We furthermore explain a novel approach to use Rocq's OCaml API with locally nameless syntax. This means that we focus on systems which are designed as proof assistants with consistent meta-theory, rather than dependently typed programming languages, and focus on those with conceptual similarity and shared foundations. In particular, we do not consider Idris [1, 2], HOL-based systems such as Isabelle, HOL4, or HOL light, or LF-based systems such as Beluga.

We evaluate the different meta-programming frameworks on a simple yet realistic example: automatically deriving instances of the `Functor` type class for a simple family of inductives, covering,

[1] We are not aware of a dedicated publication about meta-programming in Lean 4, but there exists a collaborative book draft [18]. Lean 3's meta-programming was surveyed by Ebner et al. [8].

amongst many other types, options, lists, and trees. For Rocq we e.g. want to generate the following for the list type:

```
Fixpoint fmap {A B : Type} (f : A -> B) (l : list A) : list B :=
  match l with [] => [] | x :: l => f x :: fmap f l end.
```

Many tasks involving automatic boilerplate generation follow the same model as this example: take an inductive as input and produce a term as output. We choose this example because it is simple enough for code to be readable and explainable, yet complex enough to expose issues that arise in more realistic meta-programs, and makes use of common meta-programming features such as type-class search or the ability to extend the global environment.

Paper summary Our paper at https://hal.science/hal-05024207 can be seen as a first step towards a suggested rosetta stone project for meta-programming in Rocq project:

https://github.com/coq-community/metaprogramming-rosetta-stone.

We conclude in the paper that there are conceptual and current limitations to existing frameworks. Conceptually, we observe that:

- *Binder representation* was a recurring issue in the paper,
- *Term quotations* allow to use user syntax directly when constructing and pattern matching on terms, thereby removing the need to spell out fully qualified constant names, implicit arguments, type-class instances, or universe levels,
- *Recursive function generation* is crucial, which became especially apparent for the example in Lean 4, where the need to fall back on primitive recursors prevented us from implementing a plugin with the same features as in the other systems,
- *State* is inherent to meta-programs, which can read and modify the global environment, local environment, and unification state,
- *Verifiability* is a desirable property of frameworks, where ideally formal verification is possible. Verification is not so attractive for users of meta-programs because properties such as well-typedness can be checked a posteriori by the kernel, but implementors of meta-programs might be interested in (partial) correctness guarantees. Indeed, formal specifications can partly replace documentation – which is lacking for all considered frameworks anyways – and can help in writing correct meta-programs, which is far from an easy task considering the complexity of the underlying systems.

Currently, we observe that:

- *The learning curve* of a meta-programming framework is crucial, and writing meta-programs in a different language than that of the underlying proof assistant leads to a steeper curve.
- A proper meta-programming language benefits from adequate tooling, such as a language server, a documentation generator, a package manager, an optimising compiler or an efficient interpreter, and a good integration with the proof assistant's build system.
- Precise performance considerations are outside the scope of this paper, although we do comment on performance when relevant.

Overall, two meta-programming approaches emerge: meta-programming directly in the proof assistant, or using a domain specific language (DSL).

The first option allows the possibility of verifying meta-programs, which is relevant for authors of meta-programs. Verifying meta-programs requires both a specification of the basic meta-programming operations provided by the framework, and adequate means to use these basic specifications in order to derive guarantees about complex meta-programs. The second option does not allow certifying meta-programs, but enables using domain-specific programming language features.

In both cases, one needs a feature-complete meta-programming API, which stays up to date with the evolution of the proof assistant. As implementors of a meta-programming framework, *bootstrapping* the proof assistant gives a feature-complete API for free, but requires significant work a priori (for instance Lean 4's elaborator is bootstrapped). An alternate approach, which MetaRocq and Agda follow, is to do meta-programming directly in the proof assistant, without bootstrapping. Interfacing with the elaborator and kernel is done using a meta-programming monad, which from a user's point of view is very similar to bootstrapping, but requires more work from the implementors.

References

[1] Edwin C. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Program.*, 23(5):552–593, 2013. doi:10.1017/S095679681300018X.

[2] Edwin C. Brady. Idris 2: Quantitative type theory in practice. In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference)*, volume 194 of *LIPIcs*, pages 9:1–9:26. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. URL: https://doi.org/10.4230/LIPIcs.ECOOP.2021.9, doi:10.4230/LIPICS.ECOOP.2021.9.

[3] Tej Chajed. Record updates in coq. In *CoqPL 2021: The Seventh International Workshop on Coq for Programming Languages*, 2021. Extended Abstract. URL: https://popl21.sigplan.org/details/CoqPL-2021-papers/3/Record-Updates-in-Coq.

[4] Thierry Coquand and Gérard P Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, 1988. doi:10.1016/0890-5401(88)90005-3.

[5] Arthur Azevedo de Amorim. Deriving instances with dependent types. In *Proceedings of the Sixth International Workshop on Coq for Programming Languages (CoqPL 2020)*, 2020. URL: https://popl20.sigplan.org/details/CoqPL-2020-papers/1/Deriving-Instances-with-Dependent-Types.

[6] Louise Dubois de Prisque. *Prétraitement compositionnel en Coq. (Compositional preprocessing in Coq)*. PhD thesis, University of Paris-Saclay, France, 2024. URL: https://tel.archives-ouvertes.fr/tel-04696909.

[7] Catherine Dubois, Nicolas Magaud, and Alain Giorgetti. Pragmatic isomorphism proofs between coq representations: Application to lambda-term families. In Delia Kesner and Pierre-Marie Pédrot, editors, *28th International Conference on Types for Proofs and Programs, TYPES 2022, June 20-25, 2022, LS2N, University of Nantes, France*, volume 269 of *LIPIcs*, pages 11:1–11:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. URL: https://doi.org/10.4230/LIPIcs.TYPES.2022.11, doi:10.4230/LIPICS.TYPES.2022.11.

[8] Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. A metaprogramming framework for formal verification. *Proc. ACM Program. Lang.*, 1(ICFP), August 2017. doi:10.1145/3110278.

[9] João Paulo Pizani Flor, Wouter Swierstra, and Yorick Sijsling. Pi-ware: Hardware description and verification in agda. In Tarmo Uustalu, editor, *21st International Conference on Types for Proofs and Programs, TYPES 2015, May 18-21, 2015, Tallinn, Estonia*, volume 69 of *LIPIcs*, pages 9:1–9:27. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015. URL: https://doi.org/10.4230/LIPIcs.TYPES.2015.9, doi:10.4230/LIPICS.TYPES.2015.9.

[10] Benjamin Grégoire, Jean-Christophe Léchenet, and Enrico Tassi. Practical and sound equality tests, automatically – Deriving eqType instances for Jasmin's data types with Coq-Elpi. In *CPP 2023: Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2023: Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs, pages 167–181, Boston MA USA, France, January 2023. ACM. URL: https://inria.hal.science/hal-03800154, doi:10.1145/3573105.3575683.

[11] Jason Gross, Théo Zimmermann, Miraya Poddar-Agrawal, and Adam Chlipala. Automatic test-case reduction in proof assistants: A case study in coq. In June Andronick and Leonardo de Moura, editors, *13th International Conference on Interactive Theorem Proving, ITP 2022, August 7-10, 2022, Haifa, Israel*, volume 237 of *LIPIcs*, pages 18:1–18:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. URL: https://doi.org/10.4230/LIPIcs.ITP.2022.18, doi:10.4230/LIPICS.ITP.2022.18.

[12] Bohdan Liesnikov, Marcel Ullrich, and Yannick Forster. Generating induction principles and subterm relations for inductive types using metacoq. *CoRR*, abs/2006.15135, 2020. URL: https://arxiv.org/abs/2006.15135, arXiv:2006.15135.

[13] Nicolas Magaud. Towards automatic transformations of coq proof scripts. In Pedro Quaresma and Zoltán Kovács, editors, *Proceedings 14th International Conference on Automated Deduction in Geometry, ADG 2023, Belgrade, Serbia, 20-22th September 2023*, volume 398 of *EPTCS*, pages 4–10, 2023. doi:10.4204/EPTCS.398.4.

[14] Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In *Lecture Notes in Computer Science*, pages 625–635. 2021. doi:10.1007/978-3-030-79876-5_37.

[15] Ulf Norell. *Towards a practical programming language based on dependent type theory*, volume 32. Chalmers University of Technology, 2007.

[16] Ulf Norell, Nils Anders Danielsson, Jesper Cockx, and Andreas Abel. Agda wiki. http://wiki.portal.

chalmers.se/agda/pmwiki.php.

[17] Christine Paulin-Mohring. Inductive definitions in the system Coq rules and properties. In *International Conference on Typed Lambda Calculi and Applications*, pages 328–345. Springer, 1993. doi:10.1007/BFb0037116.

[18] Arthur Paulino, D Testa, E Ayers, H Böving, J Limperg, S Gadgil, and S Bhat. Metaprogramming in lean 4. *Online Book. https://github. com/arthurpaulino/lean4-metaprogramming-book*, 2024.

[19] Pierre-Marie Pédrot. Ltac2: Tactical warfare. In *The 5th International Workshop on Coq for Programming Languages (CoqPL 2019)*, 2019. Talk at CoqPL 2019, affiliated with POPL 2019. URL: https://popl19.sigplan.org/details/CoqPL-2019/8/Ltac2-Tactical-Warfare.

[20] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The MetaCoq Project. *Journal of Automated Reasoning*, February 2020. URL: https://inria.hal.science/hal-02167423, doi:10.1007/s10817-019-09540-0.

[21] Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. Coq coq correct! verification of type checking and erasure for coq, in coq. *Proc. ACM Program. Lang.*, 4(POPL), December 2019. doi:10.1145/3371076.

[22] Kathrin Stark, Steven Schäfer, and Jonas Kaiser. Autosubst 2: reasoning with multi-sorted de bruijn terms and vector substitutions. In Assia Mahboubi and Magnus O. Myreen, editors, *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, pages 166–180. ACM, 2019. doi:10.1145/3293880.3294101.

[23] Carst Tankink, Herman Geuvers, James McKinna, and Freek Wiedijk. Proviola: A tool for proof reanimation. *CoRR*, abs/1005.2672, 2010. URL: http://arxiv.org/abs/1005.2672, arXiv:1005.2672.

[24] Enrico Tassi. Elpi: an extension language for Coq (Metaprogramming Coq in the Elpi λProlog dialect). In *The Fourth International Workshop on Coq for Programming Languages*, Los Angeles (CA), United States, January 2018. URL: https://inria.hal.science/hal-01637063.

[25] Enrico Tassi. Deriving Proved Equality Tests in Coq-Elpi: Stronger Induction Principles for Containers in Coq. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving (ITP 2019)*, volume 141 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 29:1–29:18, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ITP.2019.29, doi:10.4230/LIPIcs.ITP.2019.29.

[26] The Coq Development Team. The coq proof assistant, September 2024. doi:10.5281/zenodo.14542673.

[27] Cas van der Rest and Wouter Swierstra. A completely unique account of enumeration. *Proc. ACM Program. Lang.*, 6(ICFP):411–437, 2022. doi:10.1145/3547636.

[28] Paul van der Walt and Wouter Swierstra. Engineering proof by reflection in agda. In *International Symposium on Implementation and Application of Functional Languages*, 2012. URL: https://api.semanticscholar.org/CorpusID:18658569.

[29] Marcell van Geest and Wouter Swierstra. Generic packet descriptions: verified parsing and pretty printing of low-level data. In Sam Lindley and Brent A. Yorgey, editors, *Proceedings of the 2nd ACM SIGPLAN International Workshop on Type-Driven Development, TyDe@ICFP 2017, Oxford, UK, September 3, 2017*, pages 30–40. ACM, 2017. doi:10.1145/3122975.3122979.