# Camltac: OCaml as a Tactic Language

**Dario Halilovic**
EPFL
Lausanne, Switzerland
dario.halilovic@epfl.ch

Clément Pit-Claudel
EPFL
Lausanne, Switzerland
clement.pit-claudel@epfl.ch

## Abstract

We present Camltac, a Rocq plugin that allows OCaml to be written directly within Rocq scripts. Camltac blurs the line between Rocq plugins and tactics, letting users leverage the expressivity, performance, and rich ecosystem of OCaml to write tactics and metaprograms, without compromising on simplicity and convenience.

## 1 Introduction

The following snippet shows how to write a tactic reifying operations on std++'s `gmap` [1] in Camltac:

```
Camltac Run ocaml:{{
  let rec reify x : constr tactic =
    match%rocq x with
    | "empty" ->
      {%open_constr| MapEmpty |}
    | "insert ?k ?v ?m" ->
      let* m' = reify m in
      {%constr| MapInsert %{k} %{v} %{m'} |}
    | "union ?m1 ?m2" ->
      let* m1' = reify m1 in
      let* m2' = reify m2 in
      {%constr| MapUnion %{m1'} %{m2'} |}

  let _ = FFI.(define "reify"
    (constr @-> tac constr) reify)
}}.
```

Backtracking Ltac2 `match!` — Quotation — Monadic `let` — Ltac2 match pattern — Antiquotation — Ltac2 interoperability

In this example, `match%rocq` is the backtracking pattern-matching from Ltac2; pattern strings and quotations use Rocq's usual syntax, with pattern variables (`?a`) and OCaml antiquotations (`%{k}`) respectively, and details such as qualified names and implicit arguments are inferred by Rocq.

Rocq benefits from a rich ecosystem of metaprogramming frameworks [2], including Ltac2 [3], Mtac2 [4], Elpi [5], and MetaRocq [6]. Each of these has unique strengths and favored use-cases: lightweight scripting (Ltac2), code generation (e.g. HierarchyBuilder [7] for Elpi), strongly typed tactic programming (Mtac2) or verified metaprogramming (MetaRocq). All of them share one characteristic: they are used within Rocq files, without leaving the proof assistant.

***OCaml plugins are powerful*** As the implementation language of Rocq, OCaml itself can be used to define metaprograms, and Rocq supports loading OCaml plugins at runtime. OCaml plugins have many benefits: they are more flexible than other metaprogramming solutions, since the user can directly access Rocq's internal APIs [8]; they can define new vernacular commands [5]; use memoization [9]; use parallelism and I/O operations [10]; and benefit from OCaml's performance [11].

***But OCaml plugins are inconvenient*** In practice however, writing a plugin is the *last* resort. We believe this unfortunate status mainly comes from three compounding issues that significantly worsen the experience of writing plugins:

*Stability.* Rocq makes no guarantees about the stability of its internal APIs, meaning plugins often break between versions of Rocq. Keeping up with the latest Rocq APIs requires a non-trivial maintenance effort.[1]

*Boilerplate.* Setting up an OCaml plugin requires heavy boilerplate: writing a single "Hello world" tactic can require up to 10 files [12].

*Friction.* Writing a plugin for Rocq is inconvenient: OCaml tactics are written, compiled and edited separately from Rocq scripts; plugins cannot access definitions from the environment easily; and one cannot create terms using Rocq's natural syntax. (Instead, one has to manipulate de Bruijn indices by hand, which is quite tricky [2].)

Camltac integrates OCaml as a tactic language inside Rocq.[2] It provides a complete metaprogramming experience without compromise:

- We include standard Ltac2 APIs to guarantee stability;
- We eliminate boilerplate by writing and running OCaml code directly within Rocq scripts;
- We provide convenience by integrating tightly with Rocq, including a proper quotation mechanism to naturally write Rocq terms in OCaml.

In the rest of this extended abstract, we present the various features of Camltac. Note that Camltac is far from a finished product: one of our goals in this workshop is to obtain early feedback from the community. Our implementation is available for installation at https://github.com/epfl-systemf/camltac.

## 2 Features

At its core, Camltac provides the ability to interpret any OCaml code placed inside specific delimiters (using a mechanism similar to Elpi's). Camltac can compile and execute

[1] As a service to the community, Rocq's developers test most popular plugins in Rocq's CI, and resolve compatibility issues through overlays.

[2] A. Charguéraud proposed a similar idea in 2009 [13]. With Camltac, OCaml joins the many *meta-languages* in its family, from the original ML for the Edinburgh LCF prover [14] to F* [15].

OCaml code using the `Camltac Run` vernacular or in the tactic-in-term modality `ocaml:(...)`. Using OCaml has several immediate advantages:

*Type-safety and compile-time errors.* The OCaml compiler performs a number of checks, catching many errors before running the program (pattern-matching exhausitivity, absence of dead code or unbound variables, type-checking, ...).

*Access to the OCaml ecosystem.* OCaml is a mature programming language, with a rich set of libraries and tools, including formatters, linters, preprocessors, build systems, and language servers.

*Familiarity.* OCaml is a popular language among users of Rocq. Moreover, there is a large community around OCaml, with numerous learning resources and help available online.

*Effects, IO and parallelism.* OCaml 5 has built-in support for algebraic effects and domain parallelism, and battle-tested libraries such as Lwt [16] and Eio [17] can handle concurrent computations and IO.

*Native compilation.* Camltac snippets are natively compiled ahead-of-time, benefiting from the excellent performance of OCaml's compiler.

## 3 Writing tactics

Camltac tactics are written in the *tactic monad*: the type `'a tactic` denotes tactics yielding a value of type `'a`. While the monad is implicit in Ltac2 (i.e. all functions are tactics), there is a clear distinction between tactics and functions in OCaml. We therefore include dedicated syntax for tactics:

- A monadic `let* x = t in f` binding for sequencing tactics, which desugars to `bind t (fun x -> f)`.
- *Tacticals* from Ltac2, such as `repeat` or `progress`, are included as regular higher-order functions.
- Quotations and pattern matching (§ 3.2) also come with their own syntax.

### 3.1 APIs

Camltac does not define its own APIs: instead, it reuses the standard Ltac2 APIs, accessible in the `Ltac2` module. Using Ltac2 APIs guarantees stability and semantic correctness, since the APIs are provided and maintained by Rocq developers. We provide Ltac2 APIs for OCaml in a separate library, MLtac2 [18], that is included in Rocq's CI for stability.

Additionally, we offer extra APIs for features not covered by Ltac2, such as hint databases, with the hope of eventually upstreaming them to Rocq.

### 3.2 Metaprogramming

Camltac provides a quotation mechanism similar to Ltac2 quotations. Quotations are enabled by a Pre-Processor eXtension (PPX) named `ppx_rocq`. The Ltac2 `constr:(...)` quotation is written as `[%constr "..."]` instead, or equivalently `{%constr|...|}`, and expanded at definition time to a function call that uses Rocq's parser and converts the obtained term to Rocq's `constr` representation.

Quotations in `ppx_rocq` can contain *antiquoted* values, i.e., parts of the term that result from evaluating an OCaml expression. For example, the Camltac equivalent of the Ltac2 expression `constr:($x + $y)` is `[%constr "%{x} + %{y}"]`, for two arbitrary OCaml expressions `x` and `y` (note one difference with Ltac2 antiquotations, where `x` and `y` must be variables). Since the expansion is done at compile time, invalid or ill-typed expressions are immediately reported to the user.

Pattern-matching on terms and goals is also supported by `ppx_rocq`, with backtracking, non-backtracking, or multi-backtracking semantics (`match%rocq`, `match%lazy`, and `match%multi`). Goal and term matching both use the same syntax: matching on goals is written `match%rocq goal with`, while matching on terms is written `match%rocq t with`. In a pattern-matching branch, each pattern variable generates an OCaml variable of the same name, which allows one to write pattern matching in a natural way.

## 4 Examples

We have included several more tutorials and examples of Camltac in our `examples` directory[3]. We briefly describe them here:

***Reification*** We implement several reification procedures, including reification on Boolean expressions and the example from the introduction.

***Running tactics in parallel*** `Eio` [17] is an effects-based library for parallelism and concurrency. Using `Eio`, we implement a simple proof search procedure that launches several tactics concurrently and uses the first successful result.

***Tracing tactics*** `ppx_minidebug` [19] is a small utility for OCaml that instruments function calls, generating an explorable trace. We used `ppx_minidebug` to trace a simple backtracking tactic and reason about its execution.

## 5 Comparison with other systems

***Ltac2*** Camltac tactics map almost one-to-one with Ltac2: indeed, Camltac shares a lot of design choices with Ltac2, and includes a large part of its APIs. For example, here's the reification tactic from the introduction, written in Ltac2:

```
Ltac2 rec reify x :=
  match! x with
  | empty =>
    open_constr:(MapEmpty)
  | insert ?k ?v ?m =>
    let m' := reify m in
    constr:(MapInsert $k $v $m')
  | union ?m1 ?m2 =>
```

[3] https://github.com/epfl-systemf/camltac/

```
  let m1' := reify m1 in
  let m2' := reify m2 in
  constr:(MapUnion $m1' $m2')
end.
```

***OCaml plugins*** In the framework of Bouverot-Dupuis and Forster [2], Camltac conserves the pros of OCaml plugins, since Camltac code has access to Rocq's full implementation (P1), but also resolves or mitigates nearly all criticisms made of plugins:

- (C1) Quotations hide de Bruijn indices when constructing terms (they are still present when traversing terms).
- (C2) Quotations and antiquotations are supported through `ppx_rocq`.
- (C3) Camltac does not require any additional setup to run OCaml code.
- (C4) The MLtac2 library provides an entry point to meta-programming in Rocq.
- (C5) State management is handled by the tactic monad.

***Proof scripting*** We do not intend to use Camltac as a proof scripting language (between `Proof` and `Qed`), but rather as a meta-programming and as a tactic definition language:

- As a meta-programming language, we provide a complete access to Rocq's internal APIs and runtime state, hence Rocq plugins can be reimplemented in Camltac.
- As a tactic definition language, we provide quotations, antiquotations, and pattern-matching using `ppx_rocq`, as well as Ltac2 APIs through the MLtac2 library.

## 6 Limitations

***Learning curve*** Using Camltac requires familiarizing oneself with a new language (OCaml) and APIs. Learning OCaml has a non-zero cost, but we believe it to be reasonable, given the similarity (in spirit, at least) with Gallina and Ltac2.

***Lack of tactic notations*** OCaml's syntax is quite constrained: apart from extension nodes, there is not much room for custom notations. We have found this to be only a minor annoyance, and we think that the OCaml syntax should be sufficient in most cases.

***Binder representation*** Camltac uses Rocq's internal representation of binders that uses de Bruijn indices. Quotations partially hide de Bruijn indices when constructing terms, but they are still present when traversing them. An interesting approach in that regard would be to integrate the locally nameless API proposed by Bouverot-Dupuis and Forster [2].

***Vernacular commands*** While Camltac can register new vernacular commands through Rocq's OCaml APIs, doing so is not particularly pleasant, since we do not support Camlp5's syntax to extend the grammar (i.e. the syntax of `.mlg` files).

***Limited exports*** While `Require` correctly loads Camltac modules from the current project, they currently cannot be exported outside of the current project, since Camltac files are not installed by Dune.

***Future work*** Camltac is a work in progress, with some features missing. These include the Ltac2 `pattern` and `reference` quotations, missing APIs defined in pure Ltac2 code, and other desirable features such as general IDE support. We have not yet ported tactic libraries or large plugins to Camltac. We also hope to improve the interaction between Camltac and MetaRocq's extracted tactics, to achieve the same seamless integration between MetaRocq and Camltac tactics as between Rocq and OCaml code.

## Conclusion

Camltac enables a myriad of new applications by integrating OCaml more closely into Rocq. Relying on the standard Ltac2 APIs brings stability, while `ppx_rocq` offers a lightweight syntax for conveniently writing terms. We are looking for early testers to experiment with porting plugins and tactics.

## Acknowledgments

We thank Arthur Charguéraud for initial discussions on the topic, Gaëtan Gilbert for early feedback, Pierre-Marie Pédrot for his help navigating APIs and designing the MLtac2 interface between Camltac and Rocq, and all participants at Rocq'n'share 2026 and the anonymous reviewers for their valuable feedback.

## References

[1] Robbert Krebbers. Efficient, Extensional, and Generic Finite Maps in Coq-std++. In *The Coq Workshop 2023*, Białystok, Poland, July 2023.

[2] Mathis Bouverot-Dupuis and Yannick Forster. Code Generation via Meta-programming in Dependently Typed Proof Assistants. In *Programming Languages and Systems - 35th European Symposium on Programming, ESOP 2026, Turin, Italy, April 11-16, 2026, Proceedings, Part I*, pages 166–189, 2026.

[3] Pierre-Marie Pédrot. Ltac2: Tactical Warfare. In *The Fifth International Workshop on Coq for Programming Languages (CoqPL 2019)*, Lisbon, Portugal, January 2019.

[4] Jan-Oliver Kaiser, Beta Ziliani, Robbert Krebbers, Yann Régis-Gianas, and Derek Dreyer. Mtac2: Typed Tactics for Backward Reasoning in Coq. *Proc. ACM Program. Lang.*, 2(ICFP):78:1–78:31, 2018.

[5] Enrico Tassi. Elpi: Rule-Based Meta-Language for Rocq. In *CoqPL 2025 - The Eleventh International Workshop on Coq for Programming Languages*, Denver, United States, January 2025.

[6] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The MetaCoq Project. *J. Autom. Reason.*, 64(5):947–999, 2020.

[7] Cyril Cohen, Kazuhiko Sakaguchi, and Enrico Tassi. Hierarchy Builder: Algebraic Hierarchies Made Easy in Coq with Elpi (System Description). In *5th International Conference on Formal Structures for Computation and Deduction, FSCD 2020, Paris, France (Virtual Conference), June 29 - July 6, 2020*, pages 34:1–34:21, 2020.

[8] Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 689–700, 2015.

[9] Simon Guilloud and Clément Pit-Claudel. Verified and Optimized Implementation of Orthologic Proof Search. In *Computer Aided Verification - 37th International Conference, CAV 2025, Zagreb, Croatia, July 23-25, 2025, Proceedings, Part III*, pages 130–152, 2025.

[10] Łukasz Czajka and Cezary Kaliszyk. Hammer for Coq: Automation for Dependent Type Theory. *J. Autom. Reason.*, 61(1-4):423–453, 2018.

[11] Jason Gross, Andres Erbsen, and Adam Chlipala. Reification by Parametricity - Fast Setup for Proof by Reflection, in Two Lines of Ltac. In *Interactive Theorem Proving - 9th International Conference, ITP 2018, Oxford, UK, July 9-12, 2018, Proceedings*, pages 289–305, 2018.

[12] Rocq's `tuto0` plugin tutorial in Rocq's repository. https://github.com/rocq-prover/rocq/tree/4a3c7952a6a063182185379590bc4fdf0f66d3c4/doc/plugin_tutorial/tuto0.

[13] Arthur Charguéraud. MLtac: raising Ltac to the power of ML. Coq meeting on tactics, 2009. https://github.com/rocq-prover/coq.github.io/blob/5544aec6562bb1cd467605c9acf1271596cd838f/files/adt-30jun09-arthur-mltac.pdf.

[14] Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer, 1979.

[15] Guido Martínez, Danel Ahman, Victor Dumitrescu, Nick Giannarakis, Chris Hawblitzel, Cătălin Hriţcu, Monal Narasimhamurthy, Zoe Paraskevopoulou, Clément Pit-Claudel, Jonathan Protzenko, Tahina Ramananandro, Aseem Rastogi, and Nikhil Swamy. ML as a Tactic Language, Again. In *Proceedings of the ML Family Workshop*. ACM, 2018. Workshop co-located with ICFP 2018.

[16] Lwt: OCaml promises and concurrent I/O. https://github.com/ocsigen/lwt.

[17] Eio — Effects-based Parallel IO for OCaml. https://github.com/ocaml-multicore/eio.

[18] Dario Halilovic. MLtac2: Ltac2 APIs for OCaml. https://github.com/epfl-systemf/mltac2, 2026.

[19] ppx_minidebug — Debug logging for OCaml with interactive trace exploration. https://github.com/lukstafi/ppx_minidebug.