# Elpi: rule-based meta-languge for Rocq

Enrico Tassi
Université Côte d'Azur, Inria
France
enrico.tassi@inria.fr

## Abstract

Elpi is a high-level programming language designed to implement new commands and tactics for the Rocq prover. It provides native support for syntax trees with binders and holes, relieving programmers of the complexities associated with De Bruijn indices and unification variables.

In recent years, Elpi has been used to develop a variety of Rocq extensions, some of which have become widely adopted. In this talk, we will provide a gentle introduction to the Elpi programming language and its extensive API for interacting with Rocq. We will also survey notable applications written in Elpi and conclude by comparing Elpi to other meta-languages, highlighting both its strengths and weaknesses.

**ACM Reference Format:**
Enrico Tassi. 2025. Elpi: rule-based meta-languge for Rocq. In *Proceedings of The Eleventh International Workshop on Coq for Programming Languages (CoqPL25).* ACM, New York, NY, USA, 3 pages.

## 1 Elpi = $\lambda$Prolog + C.H.R.

From a programming language perspective, Elpi is a dialect of $\lambda$Prolog enriched with constraints and constraint handling rules [6, 9]. From a more practical standpoint, Elpi is as an extension language: an interpreter that can be easily embedded within a larger application, providing users with a high-level language to extend the application's functionality.

Elpi and its implementation provide a unique combination of features that make it particularly well-suited for extending an interactive theorem prover like Rocq [17].

***Rule based.*** Elpi source code is organized into rules, which can be added both statically and dynamically, either by the programmer or by the program itself. This feature aligns naturally with proof scripts, which similarly construct an ever growing environment of types and proofs. New rules can use new concepts and hence extend the capabilities of existing Elpi programs.

***Syntax trees with binders.*** Elpi data types can contain binders and the programming language offers primitives to cross then, following the Higher Order Abstract Syntax tradition [13]. In particular the programmer can postulate fresh constants and substitute bound variables with them, a technique called binder mobility [12].

***Context management.*** When a binder is crossed, it is often necessary to associate some data with the bound variable. For example, typing algorithms typically manage a typing context. In Elpi, these algorithms do not need to explicitly handle a context; instead, they can leverage the runtime of the programming language by dynamically adding new rules.

***Unification variables.*** As a logic programming language, Elpi provides unification variables, automatically manages substitutions for the programmer, and performs scope checking to prevent variable capture.

***Constraints.*** A constraint in Elpi represents a computation that remains suspended until its input, a unification variable, becomes instantiated. Constraint Handling Rules (CHRs) treat these suspended computations as first-class values, enabling them to deduce new information dynamically. For example, if `X` is an unknown natural number on which two incompatible computations, such as tests for being even and odd, are suspended, a CHR can detect the conflict and halt the program.

***Syntax trees with holes.*** Unification variables in Elpi can be used to represent incomplete syntax trees. Algorithms that manipulate these trees must attach data to holes, similar to how typing algorithms attach data to bound variables. For instance, if the syntax tree being constructed represents a Rocq term, these holes will certainly carry typing constraints.

Whenever a hole gets instantiated, its typing constraint must be validated. Additionally, since a hole cannot have two conflicting types, a CHR can enforce the unification of any typing constraints associated with the same hole, ensuring consistency throughout the process.

***API.*** The Elpi interpreter includes a highly flexible Foreign Function Interface. For example, it allows invoking the Rocq type checker deep under binders or scripting the vernacular language by programmatically declaring inductive types, modules, type class instances, `Arguments` directives, and more.

## 2 Applications

We briefly describe some applications written in Elpi.

***Hierarchy Builder.*** HB [4] is a high level language to describe hierarchies of interfaces and is adopted by many Rocq libraries including the Mathematical Components one [1]. HB synthesizes all the boilerplate in order to make these interfaces work in practice, like modules, records, canonical structure instances, implicit arguments declarations, notations. HB leverages the extensive API provided by Elpi, as well as its ability to incrementally build a database of known interfaces and their inheritance relations.

***TnT.*** Trakt and its successor Trocq [3] are frameworks designed to transport types (Rocq goals) over type (iso) morphisms, with or without univalence. The former is used by the Sniper tactic in Coq-smt [2]. Both tools leverage Elpi's ability to manipulate syntax with binders. Trocq, in particular, uses constraints to accumulate knowledge during term processing and ultimately computes an optimal solution.

***Derive.*** Derive synthesizes code from type declarations, such as deep induction principles [16], equality tests [8], parametricity relations, lenses for record updates, and more. Derive is an extensible framework, where each derivation is defined by a set of rules that can depend on the results of other derivations. The formal methods team at BlueRock Security has extended this framework to cover the synthesis of concepts from the Std++ library.

***NES.*** NES emulates name spaces on top of Rocq module system. Unlike modules a namespace is never closed and new items can be added to in, even in different files. NES takes advantage of the capability of Elpi to script the Rocq vernacular language.

***Algebra-tactics.*** AT is a frontend to the ring, field, linear real arithmetic (lra), nonlinear real arithmetic (nra), and psatz tactics, designed to support the Mathematical Components algebraic hierarchy [14]. These tactics leverage Elpi's seamless integration with Rocq, repeatedly invoking Rocq's unification mechanisms deep inside terms to reify expressions up to a controlled form of conversion.

## 3 Related work

Elpi is in good company.

***$\mathcal{L}$tac.*** $\mathcal{L}$tac (version 1) is the legacy extension language for Rocq [5]. While its use in new projects is expected to decline, a substantial codebase still relies on this language. Despite having a syntax resembling that of functional programming languages, its semantics is closer to those of a logic programming language, as the runtime implements backtracking. Rocq terms can be manipulated using their natural syntax—without the need for De Bruijn indices—though the language suffers from an unclear binding discipline. This sometimes leads to confusion between the variables of the meta-language and those of the object language. The dynamic semantics of $\mathcal{L}$tac can also be surprising; for example, `t` and `idtac; t` are very different, with the latter being a thunk. Furthermore the language is untyped and lacks even basic data types such as lists. Despite these shortcomings $\mathcal{L}$tac has been instrumental in the success of Rocq.

***$\mathcal{L}$tac2.*** The second version of $\mathcal{L}$tac is an ML-like language that encapsulates the proof engine monad of Rocq [11]. Rocq terms are represented using an algebraic data type that closely mirrors the internal Rocq term data structure, including De Bruijn indices. The language introduces a notion of quotations and anti-quotations, which allows users to employ Rocq term syntax without ambiguity. Additionally, it exposes many APIs of the proof engine, making it well-suited for programming low-level tactic code and optimizing efficiency. Currently $\mathcal{L}$tac2 lacks APIs for declaring Rocq inductive types or scripting the vernacular language.

***Metacoq.*** The Metacoq project [15] encompasses several components, one of which is a description of Rocq's terms as an inductive type, faithfully reflecting the internal Rocq term data type, including De Bruijn indices. A meta-program in Metacoq is essentially a Rocq term running in the Template-Monad, which allows access to various APIs to read from and write to the logical environment. What sets Metacoq apart is its ability to reason about "meta" programs. For instance, one can prove within Rocq that a meta-program produces a term of a given type. However, Metacoq currently lacks APIs and formal definitions to handle unification variables, which limits the ability to prove properties about meta-programs that manipulate incomplete syntax trees.

***Mtac2.*** The Mtac2 tactic language [10] is no longer actively developed but has some interesting features worth highlighting. Unlike Metacoq and $\mathcal{L}$tac2, Mtac2 is a functional language that successfully hides De Bruijn indices from the programmer. In particular the `nu` operator, used for crossing binders, gives the comfort of HOAS and indeed resembles $\lambda$Prolog's `pi` operator and MLTS's `nab` operator [7].

Mtac2 offers a configurable type discipline that extends to the object language, ranging from dynamically typed to strongly typed. Additionally, it exposes the (higher-order) unification of the object language through the language's pattern matching construct.

***OCaml.*** While it is indeed possible to extend Rocq by writing an OCaml plugin, this approach is considerably more challenging compared to using higher-level languages like the ones mentioned above. Writing plugins in OCaml requires a deep understanding of the internal workings of Rocq, and the development process is more time-consuming.

However, if execution speed is a critical concern and the performance of the system is a higher priority than development time, using OCaml plugins might be a suitable choice.

## References

[1] Reynald Affeldt et al. "Porting the Mathematical Components library to Hierarchy Builder". In: *the COQ Workshop 2021*. virtuel- Rome, Italy, July 2021. URL: https://hal.science/hal-03463762.

[2] Valentin Blot et al. "Compositional Pre-processing for Automated Reasoning in Dependent Type Theory". In: *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2023, Boston, MA, USA, January 16-17, 2023*. Ed. by Robbert Krebbers et al. ACM, 2023, pp. 63–77. DOI: 10.1145/3573105.3575676. URL: https://doi.org/10.1145/3573105.3575676.

[3] Cyril Cohen, Enzo Crance, and Assia Mahboubi. "Trocq: Proof Transfer for Free, With or Without Univalence". In: *Programming Languages and Systems*. Ed. by Stephanie Weirich. Cham: Springer Nature Switzerland, 2024, pp. 239–268. ISBN: 978-3-031-57262-3.

[4] Cyril Cohen, Kazuhiko Sakaguchi, and Enrico Tassi. "Hierarchy Builder: Algebraic hierarchies Made Easy in Coq with Elpi". In: *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020)*. Ed. by Zena M. Ariola. Vol. 167. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020, 34:1–34:21. ISBN: 978-3-95977-155-9. DOI: 10.4230/LIPIcs.FSCD.2020.34. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.FSCD.2020.34.

[5] David Delahaye. "A tactic language for the system Coq". In: *Proceedings of the 7th International Conference on Logic for Programming and Automated Reasoning*. LPAR'00. Reunion Island, France: Springer-Verlag, 2000, pp. 85–95. ISBN: 3540412859.

[6] Cvetan Dunchev et al. "ELPI: Fast, Embeddable, λProlog Interpreter". In: *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*. Ed. by Martin Davis et al. Vol. 9450. 2015, pp. 460–468. DOI: 10.1007/978-3-662-48899-7_32. URL: https://inria.hal.science/hal-01176856v1.

[7] Ulysse Gérard, Dale Miller, and Gabriel Scherer. "Functional programming with λ-tree syntax". In: *Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming*. PPDP '19. Porto, Portugal: Association for Computing Machinery, 2019. ISBN: 9781450372497. DOI: 10.1145/3354166.3354177. URL: https://doi.org/10.1145/3354166.3354177.

[8] 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 '23: 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. Boston, MA, USA: ACM, Jan. 2023, pp. 167–181. DOI: 10.1145/3573105.3575683. URL: https://inria.hal.science/hal-03800154.

[9] Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. "Implementing type theory in higher order constraint logic programming". In: *Mathematical Structures in Computer Science* 29.8 (2019), pp. 1125–1150. DOI: 10.1017/S0960129518000427.

[10] Jan-Oliver Kaiser et al. "Mtac2: typed tactics for backward reasoning in Coq". In: *Proc. ACM Program. Lang.* 2.ICFP (July 2018). DOI: 10.1145/3236773. URL: https://doi.org/10.1145/3236773.

[11] Oleg Kiselyov et al. "Backtracking, interleaving, and terminating monad transformers: (functional pearl)". In: *SIGPLAN Not.* 40.9 (Sept. 2005), pp. 192–203. ISSN: 0362-1340. DOI: 10.1145/1090189.1086390. URL: https://doi.org/10.1145/1090189.1086390.

[12] Dale Miller. "Mechanized metatheory revisited". In: *Journal of Automated Reasoning* 63.3 (Oct. 2019), pp. 625–665. DOI: 10.1007/s10817-018-9483-3. URL: https://inria.hal.science/hal-01884210.

[13] F. Pfenning and C. Elliott. "Higher-order abstract syntax". In: *SIGPLAN Not.* 23.7 (June 1988), pp. 199–208. ISSN: 0362-1340. DOI: 10.1145/960116.54010. URL: https://doi.org/10.1145/960116.54010.

[14] Kazuhiko Sakaguchi. "Reflexive Tactics for Algebra, Revisited". In: *13th International Conference on Interactive Theorem Proving (ITP 2022)*. Ed. by June Andronick and Leonardo de Moura. Vol. 237. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022, 29:1–29:22. ISBN: 978-3-95977-252-5. DOI: 10.4230/LIPIcs.ITP.2022.29. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ITP.2022.29.

[15] Matthieu Sozeau et al. "The MetaCoq Project". In: *Journal of Automated Reasoning* (Feb. 2020). DOI: 10.1007/s10817-019-09540-0. URL: https://inria.hal.science/hal-02167423.

[16] Enrico Tassi. "Deriving proved equality tests in Coq-elpi: Stronger induction principles for containers in Coq". In: *ITP 2019 - 10th International Conference on Interactive Theorem Proving*. Portland, OR, United States, Sept. 2019. DOI: 10.4230/LIPIcs.CVIT.2016.23. URL: https://inria.hal.science/hal-01897468.

[17] 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, Jan. 2018. URL: https://inria.hal.science/hal-01637063.