A library for the automated transformation of Rocq AST

Alexandre Jean*

Lab. ICube UMR 7357 CNRS Université de Strasbourg, France

The goal of an interactive theorem prover such as Rocq is to allow users to write proofs interactively, but this interaction often ends once the proof is completed. This is partly because interactive theorem provers are used to verify results that come from paper proofs. Once the verification is done, the proofs are are usually left untouched and only serve to confirm the correctness of the results. But this is also due to the inherent difficulty of modifying proofs, as changing earlier tactics can cause cascading errors later in the proof, and the process usually involves trial and error.

One solution to avoid manually modifying proofs is to write mechanized transformations, automating as much of the refactoring process as possible. Automatic transformation of Rocq code is an emerging topic, with multiple implementations in recent papers. One example of such transformation is turning a proof using a single tactic constructed by combining tactics with *tacticals* such as `;` and `[]` into a proof made from the elementary tactics in that constructed tactic. This approach has been implemented by both Magaud *et al.* and Shi *et al.* [4, 5].

Both implementations use different approaches, with Shi *et al.* using a custom grammar to parse *Ltac* and *Magaud et al.* using anonymous pipes to communicate with *Serapi* [2], a library and protocol for interacting with a Rocq process. The fact that these implementations use different methods motivated us to write a library capable of writing various kinds of transformations on Rocq abstract syntax trees (ASTs) without having to start from scratch each time.

We first looked at what libraries existed in Rocq and OCaml to see if we could use some of them to build on. We wanted a library capable of rewriting whole documents, which excluded simply writing Ltac. While there exist multiple meta-programming libraries in Rocq, such as *coq-elpi* [7] or *Meta-Coq* [6], our requirements included support for speculative execution, meaning being able to parse potentially invalid code, severely limiting our options. We decided to base our library, *Rocq-ditto* on *rocq-lsp* [3], a library that provides a way to parse a Rocq file and get a Rocq AST as well as a way to do speculative execution. *Rocq-lsp* also supports quoting and unquoting, the action of turning a string into its AST representation and vice versa.

We developed *Rocq-ditto* alongside a set of transformations to validate the utility of the concepts we implemented. One example of such transformation is one to replace each call to the tactic *intros* with *intros* $V_1\ V_2\ \ldots V_n$ where each V_i corresponds to a variable automatically introduced by *intros*. We use this transformation as a running example to illustrate our library throughout the remainder of this abstract.

The first step of any transformation is to transform the AST produced by *Rocq-lsp* into a document structure D. This structure contains a list of all the AST nodes of depth zero, tagged with their position in the file and a unique identifier.

From D, we extract a list of proofs, where a proof is a structure made of a proposition to prove and of a list of tactics and commands used in the proof of this proposition.

We formally define a proof transformation as a function $f : proof \rightarrow transformation\_step\ list$ which takes a proof as input and returns a list of transformation steps drawn from the set:

$$\{Remove(id),\ Replace(id, new\_node),\ Add(new\_node),$$
$$Attach(new\_node, attach\_position, anchor\_id)\}$$

*alexandre.jean@unistra.fr

where $Remove(id)$ removes the node with the identifier id, $Replace(id, new\_node)$ replaces it by new_node and $Add(new\_node)$ insert new_node at a fixed position the document.

$Attach(new\_node, attach\_position, anchor\_id)$ places new_node on a position relative to the node with the id $anchor\_id$. We added $Attach$ because Add uses a fixed position and is therefore neither commutative nor associative.

These transformation steps can then be applied to D with each associated function moving the other nodes in D to guarantee that the final document remains valid. The functions also try to minimize the number of empty lines and spaces that may be introduced by removing or replacing a node.

To compute the steps of our transformation, we first need to identify which steps in each proof are *intros* tactics. This is achieved by quoting each proof nodes and checking whether its string representation matches the text representation of '*intros*'.

The *intros* tactics will automatically name and introduce variables to the context after it's run. So, to get the names of the variables introduced, we run each node in the proof consecutively, associating with each the state of the proof after its execution. Then, we obtain V_{prev}, the set of variables in the state of the node before the *intros* node and V_{intros}, the set of variables in the state of the *intros* node. To obtain the new variables introduced by *intros*, we compute the difference $C_{intros} \setminus C_{prev}$ of the two sets. Once we have the name of the new variables, we construct the new *intros* step, by concatenating the name of each of the new variables to the string 'intros', we then use a quoting function to turn this string into an AST node and wrap it inside a $Replace$ step. The transformation then simply returns a list of $Replace$ steps where each step matches an *intros* tactic inside the proof.

In conclusion, while *Rocq-ditto* is still at an experimental stage, it already enables the implementation of simple transformations such as the one presented here, as well as more complex ones such as a transformation to replace instances of *auto* with the explicit steps found by the *auto* proof search. In future work, we aim to improve the usability of *Rocq-ditto* and develop new transformations with the objective of releasing it as an open-source tool. We are also interested in exploring alternative forms of proof representation, such as the structure of *hiproof* [1].

References

[1] Ewen Denney, John Power, and Konstantinos Tourlas. Hiproofs: A Hierarchical Notion of Proof Tree. *Electronic Notes in Theoretical Computer Science*, 155:341–359, 2006. Proceedings of the 21st Annual Conference on Mathematical Foundations of Programming Semantics (MFPS XXI).

[2] Emilio Jesús Gallego Arias. SerAPI: Machine-Friendly, Data-Centric Serialization for COQ. working paper or preprint, October 2016.

[3] Emilio Jesús Gallego Arias, Ali Caglayan, Shachar Itzhaky, Fréderic Blanqui, Rodolphe Lepigre, et al. rocq-lsp: a Language Server for the Rocq Prover, 2025.

[4] Titouan Lozac'h and Nicolas Magaud. Post-processing Coq Proof Scripts to Make Them More Robust. In *2nd Workshop on the development, maintenance, refactoring and search of large libraries of proofs , September 13-14, 2024, Tbilissi, Georgia*, 2024.

[5] Jessica Shi, Cassia Torczon, Harrison Goldstein, Andrew Head, and Benjamin Pierce. Designing Proof Deautomation in Rocq. In *Proceedings of the 15th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU)*, 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, June 2020.

[7] 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.