Secure development of dependently typed programs using proxy-based small inversions

Pierre Corbineau
Univ. Grenoble Alpes, CNRS, Grenoble INP[1], VERIMAG, 38000 Grenoble, France

Basile Gros
Univ. Grenoble Alpes, CNRS, Grenoble INP[1], VERIMAG, 38000 Grenoble, France

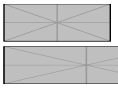
Jean-François Monin
Univ. Grenoble Alpes, CNRS, Grenoble INP[1], VERIMAG, 38000 Grenoble, France

Extended abstract

Coq [20] (nowadays renamed as Rocq) is the first major proof assistant to which Gilles Dowek has contributed at the beginning of his research carreer. Years later, he designed a new logical framework named Dedukti [1]. These two systems share several important features, such as a foundation based on lambda-calculus with dependent types, where the conversion rule plays a key role, in the tradition dating back to N.G. de Bruijn, as implemented in Automath. Incidentally, we are indebted to Gilles Dowek for having highlighted very early on the prominent role of conversion and calculation [9]. Consistently and especially relevant to this presentation, all those systems satisfy an important criterion precisely coined the *de Bruijn criterion* by Barendregt and Geuvers [2]: they *generate 'proof objects' (of some form) that can be checked by an 'easy' algorithm.* In the case of Rocq, we have a proof checker commonly called the *kernel*. This criterion is of paramount importance for the development of software systems subject to very high security requirements. The big advantage is that we get full freedom in the process of creation of proof objects: anything goes provided that, at the end, the proof object you get is accepted by the kernel. In particular, such proof assistants offer various means allowing the user to incrementally and interactively build a proof, for instance commands called *tactics* in Rocq, that can be combined using a scripting language, in combination with automated mechanisms such as decision procedures. By design, there is then no need to worry about providing guarantees on these additional tools, only their result matters. Tactics and scripting languages do not require the precise semantics required for the kernel-verified terms language. This makes it possible to state and prove theorems with a very high degree of confidence, even though all sorts of workarounds or heuristics are used in the process of developing the proofs. All this has been well-known for decades[1]. In the Curry-Howard-de Bruijn setting, where proofs are programs and their statements are types, the proof checker is essentially a type checker: checking that a proof object p is indeed a proof for a statement S is the same as checking that p, as a program, has the type S. With this in mind, we get for the same price an environment that helps us to define typed functional programs in an interactive way. Its usefulness is primarily educational in the case of simply typed programs – it helps illustrate the Curry-Howard isomorphism for propositional connectives. This approach becomes particularly powerful and convenient – even indispensable – when developing programs that process data with complex dependent types. However, one must be very careful, because in the case of programs, it is their content

[1] Institute of Engineering Univ. Grenoble Alpes

[1] We apologize for having to revisit such basic concepts but, to our great surprise, we recently discovered that advanced users of Rocq were improperly aware of this issue and misunderstood the point to follow.



© Pierre Corbineau, Basile Gros and Jean-François Monin;
licensed under Creative Commons License CC-BY 4.0

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

itself that matters, and not just their type.[2] Although it is perfectly legitimate to use any means necessary to write such functions, the thing to be exposed at the end is a piece of program expressed in a principled language that should be as short, clear, and readable as possible, so that we can be certain it will be interpreted in the same way by human readers and by computers.

As such, a script written in a tactic language of Rocq is certainly a useful helper in early design steps of complex functional programs. However, as soon as a calculation is performed with such a program, the link with the script is lost: the conversion rule is expressed on terms, not on source scripts. Therefore, can the latter be accepted as a final formulation for the former? Additionally, even if the type theory checked by the kernel of Rocq is clearly a good language when parsed and displayed consistently [17], we will not be satisfied with automatically generated terms if they are huge.

The situation is somewhat complicated by another typical feature of Rocq: program extraction, that allows us to interactively develop programs specified by a dependent type like $\forall x : X, \{y : Y \mid R\, x\, y\}$. This encourages mixing steps specifying the calculation to be performed with steps dealing with logical assertions. The latter involve proof terms that can be very long and that we neither want nor need to see.

Let us then consider a cleaner situation, where we just want to use dependent types to take precise conditions on data types into account. As a simple example, consider the `map2` function, that is similar to `map` but takes *two* lists of the *same* length as input instead of just one. Given three types A, B, C, a function f of type $A \to B \to C$, `map2` takes two lists $u : \texttt{list}\ A$ and $v : \texttt{list}\ B$ as input and returns a list of type $\texttt{list}\ C$ such that the following recursive equations are satisfied:

$$\texttt{map2}\ []\ []\ =\ [] \tag{1a}$$

$$\texttt{map2}\ (x :: u)\ (y :: v)\ =\ f\, x\, y\ ::\ \texttt{map2}\ u\ v \tag{1b}$$

It is then natural to use the dependent data types $\texttt{vect}\ A\ n$ and $\texttt{vect}\ B\ n$ instead of $\texttt{list}\ A$ and $\texttt{list}\ B$. The family of types `vect` is similar to `list` but forces its inhabitants to have a given length. Its Rocq definition is:

```
Inductive vect (A : Type) : nat → Type :=
| nil : vect A 0
| cons n : A → vect A n → vect A (S n).
```

More elaborate examples include improvements of Wadler's *views* [21] to dependently typed data structures, as proposed by McBride and McKinna in [13].

A pleasant advantage of dependent data structures is that irrelevant situations can be discarded, e.g., $\texttt{map2}\ []\ (y :: v)$. To this effect, Agda implements a rather subtle pattern matching mechanism based on first-order unification, with non trivial issues related to axiom K and univalence [5, 12, 3, 4].

On the other hand, the pattern-matching mechanism implemented in Rocq's kernel is more primitive. It is based on a very useful distinction between recursion and pattern matching introduced by Gimenez [10, 11]. In the case of dependent types, it requires the addition of a `return` clause that specifies how the type of the result depends on the constructor used. The `return` clause is universally quantified on the indices, e.g. n in the case of `vect`. As remarked

[2] Moreover, it is common for logical specifications to use functions defined by programs. For a basic example, when we express that the length of the concatenation of two lists is the sum of their respective lengths, we are referring to a sum that is itself defined by a program.

by Cockx [3], unification is then no longer needed and issues about K and univalence are no longer raised as well, at least not immediately. This is good news because such issues seem to be relevant with higher types that occur for instance in homotopy type theory, but not necessarily in programs on more usual data types, including vectors, matrices (vectors of vectors), and the examples on *views* mentioned above. Note that in simple practical situations, the `return` clause can be automatically inferred, so that terms can be expressed in a reasonably light manner.

Unfortunately, even for `map2`, a suitable `return` clause is not trivial. The first relevant tool offered by Rocq to deal with this issue is a famous tactic called `inversion`, to be used in interactive proofs. Reasoning by inversion [7] is a proof scheme that deduces information about the premises of a hypothesis H from its conclusion. The `inversion` tactic is able to perform an accurate case analysis on H: only the cases that are compatible with the arguments in the type of H are selected and moreover, constraints on those arguments are propagated. However, it is quite complex, more so than what its only published description in [7] suggests. Its proof term generation is unpredictable: it produces subgoals cluttered with a seemingly random number of equality hypotheses and the generated proof-terms have a large size, with reduced readability. This can also be a problem in dependent programming, where inversion is needed to define some functions. The body of those functions cannot be easily read and understood by a user, especially when importing definitions written by someone else. In addition, common cases in dependent programming lead to unsolvable issues, such as equality between dependent pairs that necessitates K. The unpredictability in terms generation also makes proof scripts sensitive to Rocq updates and small changes in the data structures involved in the proof.

The Equations package [19] can also be used. The algorithm behind it is an implementation of the unification algorithm for dependent types described in Jesper Cocx's PhD thesis [3], itself a refinement of the algorithm proposed by Goguen, McBride and McKinna [12] for dependent pattern matching. However, the terms still tend to be of large size, and the criticisms mentioned above regarding source scripts could apply again to some extent.

In this presentation, we advocate that a lightweight approach to inversion, called *small inversions* in [16], later improved in [15, 14, 6], avoid the aforementioned issues by performing inversion without cluttering the elimination predicate with equality constraints on the indices. For example, we can handle all examples given in [13]. Of note, Jesper Cockx's PhD thesis makes an analysis between the earliest version of small inversions and his unification algorithm: it is more general, but small inversions produce smaller and more manageable terms. We also show that typical programs on vectors or bounded natural numbers, e.g., matrix transposition and multiplication, become rather straightforward to write and to reason about formally. In contrast with the matrix library developed by Rand for quantum computations [18, 8], neither functional extensionality nor any other axiom is necessary in our proofs.

References

1 Ali Assaf, Guillaume Burel, Raphaël Cauderlier, David Delahaye, Gilles Dowek, Catherine Dubois, Frédéric Gilbert, Pierre Halmagrand, Olivier Hermant, and Ronan Saillard. Dedukti: a Logical Framework based on the λΠ-Calculus Modulo Theory. *CoRR*, abs/2311.07185, 2023.

2 Henk Barendregt and Herman Geuvers. Proof-assistants using dependent type systems. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes)*, pages 1149–1238. Elsevier and MIT Press, 2001.

3 Jesper Cockx. *Dependent Pattern Matching and Proof-Relevant Unification*. PhD thesis, Katholieke Universiteit Leuven, Belgium, 2017. URL: `https://lirias.kuleuven.be/handle/123456789/583556`.

4 Jesper Cockx and Dominique Devriese. Proof-relevant unification: Dependent pattern matching with only the axioms of your type theory. *J. Funct. Program.*, 28:e12, 2018.

5 Thierry Coquand. Pattern matching with dependent types. In B. Nordstrom, K. Petersson, and G. Plotkin, editors, *Proceedings of the 1992 Workshop on Types for Proofs and Program*, pages 66–79. Chalmers University of Technology, 1992.

6 Pierre Corbineau, Basile Gros, and Jean-François Monin. Proxy-based small inversions and their automation. Preprint, January 2026. URL: `https://hal.science/hal-05469909`.

7 Cristina Cornes and Delphine Terrasse. Automating inversion of inductive predicates in Coq. In Stefano Berardi and Mario Coppo, editors, *Types for Proofs and Programs*, pages 85–104, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

8 The INQWIRE Developers. QuantumLib/Matrix.v, 2026. URL: `https://github.com/inQWIRE/QuantumLib/blob/main/Matrix.v`.

9 Gilles Dowek. *La part du calcul*. Habilitation à diriger des recherches, Université de Paris 7, June 1999. URL: `https://inria.hal.science/tel-04114581`.

10 Eduardo Giménez. Codifying guarded definitions with recursive schemes. In Peter Dybjer, Bengt Nordström, and Jan Smith, editors, *Types for Proofs and Programs*, pages 39–59, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

11 Eduardo Giménez. Structural recursive definitions in type theory. In Kim G. Larsen, Sven Skyum, and Glynn Winskel, editors, *Automata, Languages and Programming*, pages 397–408, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

12 Healfdene Goguen, Conor McBride, and James McKinna. Eliminating dependent pattern matching. In Kokichi Futatsugi, Jean-Pierre Jouannaud, and José Meseguer, editors, *Algebra, Meaning, and Computation: Essays dedicated to Joseph A. Goguen on the Occasion of His 65th Birthday*, pages 521–540, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

13 Conor McBride and James McKinna. The view from the left. *J. Funct. Program.*, 14(1):69–111, 2004.

14 Jean-François Monin. Small inversions for smaller inversions. In *TYPES 2022 Abstracts*, Nantes, June 2022.

15 Jean-François Monin and Xiaomu Shi. Handcrafted Inversions Made Operational on Operational Semantics. In S. Blazy, C. Paulin, and D. Pichardie, editors, *ITP 2013*, volume 7998 of *LNCS*, pages 338–353, Rennes, France, July 2013. Springer.

16 Jean-François Monin. Proof Trick: Small Inversions. In Yves Bertot, editor, *Second Coq Workshop*, Edinburgh, UK, July 2010. URL: `https://hal.inria.fr/inria-00489412/en/`.

17 Robert Pollack. How to believe a machine-checked proof. In *Twenty Five Years of Constructive Type Theory*. Oxford University Press, 10 1998.

18 Robert Rand, Jennifer Paykin, and Steve Zdancewic. QWIRE practice: Formal verification of quantum circuits in coq. In Bob Coecke and Aleks Kissinger, editors, *Proceedings 14th International Conference on Quantum Physics and Logic, QPL 2017, Nijmegen, The Netherlands, 3-7 July 2017*, EPTCS, pages 119–132, 2017.

19 Matthieu Sozeau and Cyprien Mangin. Equations reloaded: high-level dependently-typed functional programming and proving in Coq. *Proc. ACM Program. Lang.*, 3(ICFP), July 2019.

20 The Coq Development Team. Main website of the Rocq project, 2025. URL: `https://rocq-prover.org/`.

21 Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987*, pages 307–313. ACM Press, 1987.