Certified Programming with Dependent Types Made Simple with Proxy-based Small Inversions

Pierre Corbineau, Basile Gros, and Jean-François Monin

Univ. Grenoble Alpes, CNRS, Grenoble INP*, VERIMAG, 38000 Grenoble, France
pierre.corbineau@univ-grenoble-alpes.fr
basile.gros@univ-grenoble-alpes.fr
jean-francois.monin@univ-grenoble-alpes.fr

In the Rocq prover, many proofs are based on exploiting information present in hypotheses with indexed inductive types. To obtain this information, reasoning by inversion consists in looking at the patterns present in the hypothesis and restraining the possible cases to those that are actually inhabited by those patterns. This reasoning is also very useful in dependent programming when using inductive types with informative sorts. In the Rocq prover, automation of the inversion reasoning is present in the form of the `inversion` tactic [3], Equations' `dependent elimination` tactic [10], or the elaboration of the `match` return types, e.g., when using the `refine` tactic or in toplevel definitions. Another tactic formerly available in Coq is `BasicElim` [4], which requires Uniqueness of Identity Proofs.

Small inversions were initially developed by Monin and Shi [8, 7] as a lightweight and axiom-free alternative to `inversion` that was more transparent, predictable, controllable and sometimes more powerful. More recently, Monin described a similar method [6] using auxiliary inductives rather than auxiliary functions for even more clarity of use. Here we advocate the use of the latter version as an aid to programming using dependent types [5, 1], taking advantage of the transparency and lightweight nature of this approach.

Small inversions work by building a syntactic partition of an indexed inductive type into auxiliary inductive types — called *partial inductive types* — depending on the syntactically decidable patterns of one or more of its inductively typed indices. Those partial inductive types can be either already existing types or custom-built for this purpose.

We define the proxy type as a function mapping *targetted* index patterns to the corresponding partial inductive types. The proxy function maps each constructor of the original type to the corresponding constructor of the respective partial inductive type. It is by performing case analysis on the proxy result that we reason by inversion on the *proxied* object.

Here is an example using the `Fin.t` bounded integers of Rocq. On the left are the definition of `Fin.t`, and two partial inductive types corresponding to the possible constructors of its only index, and on the right are the corresponding proxy type and proxy. The next page starts with the definition of a dependently typed function making use of the small inversion. Note that as the return type of the function takes the object to invert as an index, we need a *dependent inversion*, where the partial inductive types also carry the information of the original constructor in their type.

```
Inductive Fin.t : nat → Set :=
| F1 : ∀ n : nat, Fin.t (S n)
| FS : ∀ n : nat, Fin.t n → Fin.t (S n).

Definition Fin_0 := fun _ : Fin.t 0 ⇒
Empty_set.

Inductive Fin_S (n: nat) :
    Fin.t (S n) → Type :=
| isF1 :   Fin_S n F1
| isFS x : Fin_S n (FS x).
```

```
Definition Fin_proxy_type (n:nat) :
    Fin.t n → Type :=
  match n with
  | 0 ⇒  Fin_0
  | S m ⇒ Fin_S m
  end.

Definition Fin_inv (n:nat) (x : Fin.t n) :
    Fin_proxy_type n x :=
  match x as x' in Fin.t n'
    return Fin_proxy_type n' x' with
  | @F1 n' ⇒  isF1 n'
  | @FS n' x ⇒ isFS n' x
  end.
```

[1]Institute of Engineering Univ. Grenoble Alpes

```
Definition Fin_3_rect
  (P : Fin.t 3 → Type) (p1 : P F1) (p2 :  P (FS F1)) (p3 : P (FS (FS F1)))
   (x : Fin.t 3) : P x :=
  match Fin_inv 3 x with
  | isF1 _ ⇒ p1
  | isFS _ x ⇒ match Fin_inv 2 x with
              | isF1 _ ⇒  p2
              | isFS _ x ⇒ match Fin_inv 1 x with
                          | isF1 _ ⇒ p3
                          | isFS _ x ⇒ match Fin_inv 0 x with end
    end end end.
```

The partial inductive type `Fin_S` restricts the possible values of the bounding index of `Fin.t` to a successor `S n`; the `x` component in its second constructor has then the type `Fin.t n`. On the other hand, `Fin_0` states that there are no inhabitants when the bounding index is zero. As said earlier, we can have both custom inductive types (`Fin_S`) and reuse Rocq objects (`Fin_0`). The type of the proxy `Fin_inv` is *computed* in `Fin_proxy_type` by case analysis on `n`.

We successfully experimented the use of small inversions in various programs. We can, for instance, program without effort functions on vectors such as `map2`, `map3`, ... that apply respectively a function of arity 2, 3, ... to 2, 3, ... vectors of the *same size*. A somewhat more advanced example is the transposition of (m, n) matrices, seen as m-vectors of n-vectors, together with a proof that the transposition of the transposition returns the original matrix. We are then in position to address more difficult programming problems that looked manually intractable without this technique.

The code for the proxy can be more complicated than in `Fin_inv`, when several constructors of the type to be inverted share common values of the index to be analyzed: an additional case analysis on this index has then to be performed inside the proxy function. More advanced uses cases require *nested* small inversions to handle indices with dependent types.

The auxiliary inductive types and definitions are rather easy to write by hand, especially in simple use cases. It is of course interesting to automate this task, and there is ongoing work to automatically recognize situations where small inversions can be used and then creating those auxiliary definitions for a given inductive type [2]. Our current implementation in MetaRocq [9] is already able to handle examples such as `Fin.t`, vectors and many more. The goal of this implementation is to automate the most common cases, in particular those where the size of the inductive type definition would make writing the partial inductive types by hand tedious.

Defining `Fin_3_rect` in interactive mode with the `inversion` tactic fails in Rocq 9.0. In the code above the `return` clauses can be inferred by Rocq. In contrast, `dependent elimination` does allow to interactively define the function and subsequently to prove its properties. In that respect small inversions and Equations provide a similar service and are even interoperable as proof techniques.

We observe that `dependent elimination` produces more convoluted terms, while small inversions is more restricted in its scope. Attaining `dependent elimination`'s scope is not the goal of small inversions: this approach focuses on the rather common situations where one or more indices of the conclusion of constructors can be discriminated according to a linear pattern, in order to obtain a more transparent inversion in those cases.

In conclusion, while our method has a smaller scope than the already existing `dependent elimination`, we can use it to directly write definitions as terms, and to obtain more readable definitions. Moreover, the complexity of the inversion is contained in separate reusable definitions in the form of proxies, allowing for lighter code.

References

[1] Adam Chlipala. *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2013. URL: http://mitpress.mit.edu/books/certified-programming-dependent-types.

[2] Pierre Corbineau, Basile Gros, and Jean-François Monin. Proxy-based small inversions: a case study in metacoq programming. In Adrien Guatto and Marie Kerjean, editors, *JFLA 2025 - 36es Journées Francophones des Langages Applicatifs*, Jan 2025.

[3] 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.

[4] Conor McBride. Elimination with a Motive. In Paul Callaghan, Zhaohui Luo, James McKinna, and Robert Pollack, editors, *TYPES*, volume 2277 of *Lecture Notes in Computer Science*, pages 197–216. Springer, 2000.

[5] James McKinna. Why dependent types matter. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, page 1. ACM, 2006. doi:10.1145/1111037.1111038.

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

[7] 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.

[8] Jean-François Monin. Proof Trick: Small Inversions. In Yves Bertot, editor, *Second Coq Workshop*, Edinburgh, United Kingdom, July 2010. URL: https://hal.inria.fr/inria-00489412/en/.

[9] Matthieu Sozeau, Yannick Forster, Meven Lennon-Bertrand, Jakob Nielsen, Nicolas Tabareau, and Théo Winterhalter. Correct and Complete Type Checking and Certified Erasure for Coq, in Coq. *J. ACM*, 72(1), January 2025. doi:10.1145/3706056.

[10] 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. doi:10.1145/3341690.