Pierre Corbineau, Basile Gros and Jean-François Monin

Proxy-based small inversions: a case study in MetaCoq programming

Pierre Corbineau[1], Basile Gros[1], and Jean-François Monin[1]

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

The Coq proof assistant is particularly powerful and relevant for defining and working on inductive types and properties. A common proof pattern is inversion reasoning on those inductive properties, where one seeks to deconstruct the last step of the proof of a given statement.

Many languages exist to automate reasoning in Coq. Among them, MetaCoq is built to inspect, manipulate and reason on internal Coq representations, and reinterpret them as new Coq objects. It allows to combine those operations into powerful meta-programs that are expressed in the Coq language itself.

In this article, we present how we used MetaCoq to automate the definition of new Coq objects as part of our work on *proxy-based small inversions*, an alternative to the `inversion` tactic. We discuss the relevance of using MetaCoq for this approach, and we sum up the lessons learned in the process, with the hope that it might be helpful to other users of MetaCoq.

To illustrate and demonstrate our work, we provide the source code of our MetaCoq program, together with a set of examples.

1 Introduction

Coq is an interactive proof assistant [**?**] used in particular to prove the correction of software and to verify mathematical proofs. Its implementation is based on the Calculus of Inductive Constructions (CIC). Coq allows the definition of algebraic and functional types in CIC, and properties on those types. The user can interact with the proof engine to write proofs of theorems, that the kernel checks automatically when completed. Coq has been used to complete several significant achievements in formalized mathematics and certified computer software, among which the proof of the Odd Order theorem [**?**], the CompCert certified C compiler [**?**], and more recently the certification of the number of steps of the fifth Busy Beaver [**?**].

MetaCoq is a tool built to manipulate the Coq internal representations of user-defined terms and definitions in Coq in the form of Abstract Syntax Trees (ASTs), and to define new Coq objects from those ASTs [**?**] [**?**]. MetaCoq allows the user to define meta-programs that combine these techniques in a way that is not possible with tactic scripting languages such as Ltac or Mtac.

Our current work uses MetaCoq to implement an alternative to the tactic `inversion`, called *proxy-based small inversions*, or simply *small inversions*[1]. Proxy-based small inversions

*Institute of Engineering Univ. Grenoble Alpes

[1]The name *small inversions* actually covers a family of more or less similar approaches in the folklore, but only one of them is considered in this paper.

are a generalization on previous works by Jean-François Monin (sketched in [**?**], itself a rework of [**?**] and [**?**]), aiming at providing smaller, more explainable proof terms than `inversion`.

This method defines intermediate inductive types and functions, that can be used with basic tactics such as `destruct`. So far, these proxy definitions were written by hand, and we wished to automate that process. We decided to use MetaCoq because of its crucial features: in MetaCoq programs, we can inspect the internal representation of Coq objects, especially inductive types, and make Coq validate new objects definitions from custom-made ASTs.

This article presents the MetaCoq tools we used in this project, as well as lessons we learned using them. It also considers the relevance of the use of MetaCoq in projects that are not directly linked to its original purpose of formalizing Coq in Coq. A theoretical description of small inversions can be found in another article currently under review. To make this paper self-contained, we however briefly present this technique on a simple example in the following section.

2 A quick presentation of proxy-based small inversions

Reasoning by inversion is a proof scheme used in Coq that deduces information about the premises of a proof from its conclusion [**?**]. It is used to simplify proofs by case analysis by removing situations that are impossible by construction.

In Coq, we can define inductive types, which are types whose elements are entirely described by a finite set of generating functions, called constructors. Those types can represent sets of objects like natural numbers or booleans, populating the sorts `Type`, or they can be logical propositions, populating the sort `Prop`.

Inversion reasoning in Coq consists in recovering the premises and constraints used to derive the proof of a given inductive relation.

Here is an example based on a type representing the three traffic light colors, `color`:

```
Inductive color : Type :=
    | green : color
    | orange : color
    | red : color.
```

Here is the succession relation for those colors:

```
Inductive next_color : ∀ c1 : color, ∀ c2 : color, Prop :=
    | NcG : next_color green orange
    | NcO : next_color orange red
    | NcR : next_color red green.
```

Inversion reasoning is typically used to prove subgoals such as:

```
c : color
green_c : next_color green c
————————————————————————
next_color c red.
```

A call to `inversion green_c; constructor` is enough to solve this subgoal. The only possible constructor that can derive `green_c` is `NcG`, which constraints `c` to be `orange`. To perform this, the `inversion` tactic infers additional equalities, in this case (`c` = `orange`) and performs case analysis to the proof term to eliminate impossible constructors [**?**]. It is very versatile and easy to call, but it tends to generate massive and unexplainable proof terms.

In contrast, using small inversions, we focus on the first argument of `next_color`, using the fact that it has an inductive type. We call it the *pilot index*.

We first define a distinct *partial inductive* for each possible constructor of the pilot index. Together, they form a partition of the inductive relation we want to invert according to the different values of this index.

```
Inductive next_color_green : ∀ c2 : color, Prop :=
    | NcG' : next_color_green orange.

Inductive next_color_orange : ∀ c2 : color, Prop :=
    | NcO' : next_color_orange red.

Inductive next_color_red : ∀ c2 : color, Prop :=
    | NcR' : next_color_red green.
```

Next, we define a *dispatch* function, that maps the constructors of the pilot index to the corresponding partial inductive. Finally, we define an *inverter* function that injects the inductive into its partition. When performing case analysis on the result of the inverter, only relevant cases are selected as specified by the corresponding partial inductive, and the desired substitutions are obtained without additional equalities, hence the name *proxy-based small inversions*.

```
Definition next_color_dispatch (c1 c2 : color) : Prop :=
match c1 with
    | green ⇒ next_color_green c2
    | orange ⇒ next_color_orange c2
    | red ⇒ next_color_red c2
end.

Definition next_color_inversion
    {c1 c2 : color} (c : next_color c1 c2) :
    next_color_dispatch c1 c2 :=
match c in next_color c1 c2
    return next_color_dispatch c1 c2 with
    | NcG ⇒ NcG'
    | NcO ⇒ NcO'
    | NcR ⇒ NcR'
end.
```

For the same subgoal as with `inversion`, we can simply use case analysis:

```
  destruct (next_color_inversion green_c).
```

In general, small inversions focus inversion reasoning on a single argument of the inductive relation (more accurately an index), called pilot index, which must have an inductive type. As is the case with the historic `inversion` tactic, the pilot index must have an informative inductive type — that lives in the sort `Type` — and its choice is up to the user.

3 MetaCoq

MetaCoq is a Coq library and plugin. It extends the Coq interpreter with reification and reflection mechanisms (i.e., quoting and unquoting) and can be used to manipulate the Coq kernel's internal representation of Coq objects using the Coq type `term`, as well as to create monadic programs manipulating those objects using the Coq type `TemplateMonad`.

This section provides an overview of those two types and their usage.

3.1 Manipulating Coq syntax trees as Coq objects

Coq ASTs are represented by objects in the inductive type `term`. The most frequently used ones are:

- The `tCons` constructor for constants like functions or theorems (e.g., `plus`)
- The `tInd` constructor for inductive types (e.g., `nat`)
- The `tConstruct` constructor for constructors of inductive types (e.g., `true`)
- The `tCase` constructor for case elimination (`match` ... `with` ... `end`)
- The `tLetIn` constructor for local definitions (`let` ... := ... `in` ...)
- The `tProd` constructor for dependent products ($\forall(\ldots:\ldots),\ldots$)
- The `tLambda` constructor for λ-abstractions (`fun` $(\ldots:\ldots) \Rightarrow \ldots$)
- Bound variables, in the form of de Bruijn indices, are represented by the `tRel` constructor.

The more complex objects in CIC such as inductive types and pattern matching use intermediate record types to gather relevant information in a composite structure. Here we explain the representation of inductive types.

Each Coq inductive type is represented as belonging to a mutual definition of one type. The `mutual_inductive_body` type describes the common parameters of a family of mutually-defined inductive types and a definition of each inductive type as an object of the record type `one_inductive_body`. For each inductive in the mutual definition, the `one_inductive_body` type contains its type arity and list of constructors. For each constructor of an inductive, `constructor_body` records its name and arity.

When working with MetaCoq ASTs, especially with inductive definitions and pattern matching definitions, one frequently has to manipulate terms starting with series of products or of lambda abstractions. Processing those terms requires a separate treatment of this binding prefix (called *telescope*) and of the inner term that is bound by this telescope. The notion of telescope was introduced by de Bruijn for Automath in the early 1970s [**?**, **?**] and is regularly used for representing mathematical structures (e.g., [**?**]). It describes a sequence of declarations and local definitions where later definition bodies and types can depend on earlier ones. In MetaCoq, a telescope is constructed with `tProd`, `tLetIn` and `tLambda`, and is used to interpret unbound variables that are represented by de Bruijn indices.

MetaCoq ASTs also use de Bruijn indices inside the definition of inductive types to refer to the parameters, indices, and constructor arguments of this inductive type. Similarly, recursive occurrences of an inductive type within its own definition use de Bruijn indices. In Section **??** below, we explain how substitutions in a bound term can be performed with nothing more than a manipulation of its telescope.

3.2 Template meta-programming: the `MetaCoq Run` command

The monad used for expressing MetaCoq programs is an error and state monad called `TemplateMonad`. This monad contains extra constructs that represent calls to Coq's API. Errors can happen when executing those calls, or be raised by user code. The state accounts for addition of new objects in the Coq global context. Note that when an error happens, state changes are unrolled, i.e., the whole program is aborted.

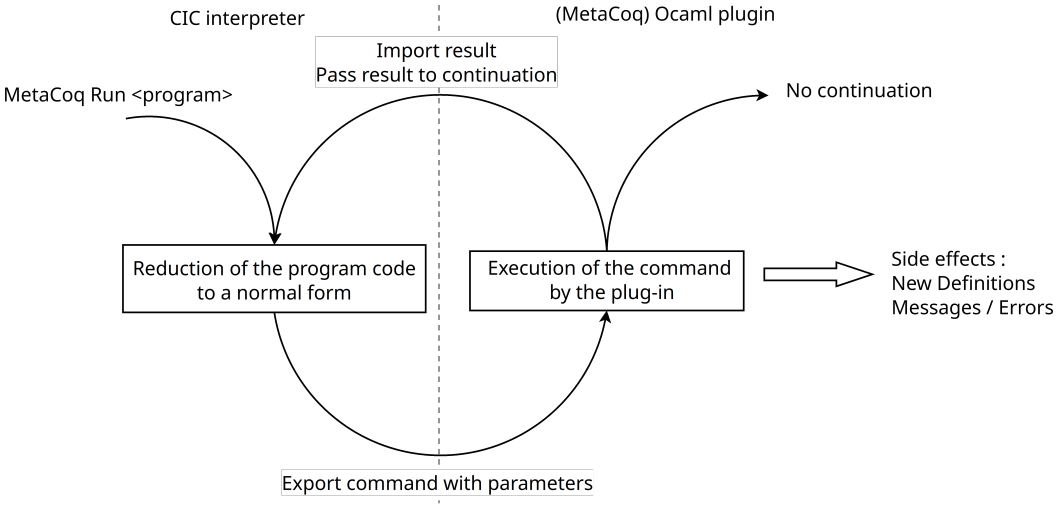

Figure 1. Sequential execution of a MetaCoq Run.

Two special operations in the `TemplateMonad` are the quote and unquote operations. The quote operation takes a Coq object and returns its syntax tree in type `term` (see below). The unquote operation takes a syntax tree and returns the corresponding Coq object.

Once a MetaCoq program is defined (with type `TemplateMonad unit`), it can be executed with the top level command `MetaCoq Run`. What follows reflects our understanding of what happens next, according to our experience and a partial reading of the source code.

We represent in Figure **??** the internal steps in the execution of a MetaCoq program. First the program is normalized: it yields a constructor of `TemplateMonad` applied to some parameters, themselves in normal form. If this constructor is the bind operation `tmBind`, its continuation is suspended waiting for the result of the bound body, and the same normalization is executed on that body. Otherwise, the command and parameters are given to the MetaCoq plugin, which interprets them. If there is a suspended continuation, it is applied to the value returned by the command, after it has been translated back into a Coq term. Otherwise, the `MetaCoq Run` terminates successfully.

4 Automating small inversions using MetaCoq

To automate the definition of the objects needed for proxy-based small inversions, we derive modified versions of the (AST of the) target inductive type Relation, using the different constructors of the type of the pilot index, Pilot. Then, case analyses are performed on Relation and the Pilot to create the *dispatch* and *inverter* functions. The modified versions of Relation are called partial inductives, and the dispatch function returns a partial inductive for each possible construction of the pilot.

We now comment the data flow of our Metacoq implementation for small inversions displayed in Figure **??**.

Extracting data. We start by extracting all the data we need from the MetaCoq structure of the source inductive types Relation and Pilot. This mainly consists in gathering all needed fields from the MetaCoq records into a custom record type for simple access. Moreover, some processing is performed in order to reduce `let in` constructs that possibly occur in both Pilot and Relation, and manipulation techniques on de Bruijn indices described in Section **??** are used to instantiate the parameters of Pilot to their actual value in Relation. We also introduce a bidirectional mapping between each constructor D_j of Pilot and the

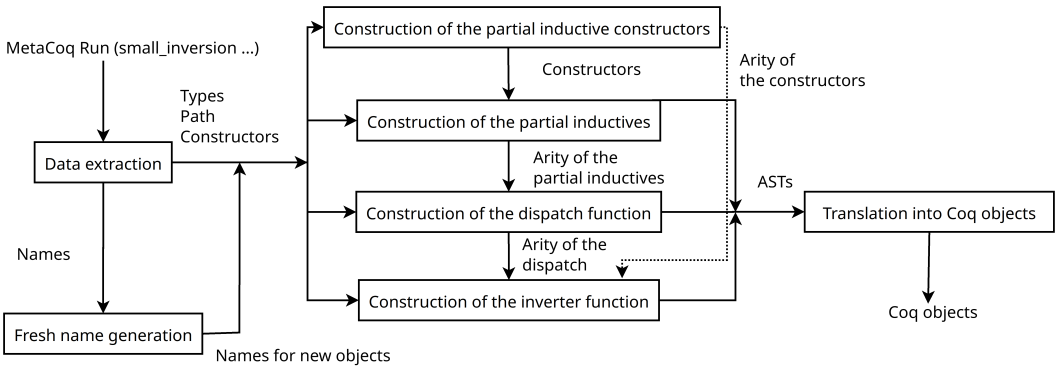

Figure 2. Data flow of MetaCoq code of small inversions

constructors of Relation where D_j can occur.

Generation of new names. We need new names for the partial inductives, their constructors, the dispatch and the inverter functions. They are created by simple string concatenations, using a predictable and (hopefully user-friendly) systematic scheme. The user can provide a suffix that is appended to all new names in order to avoid collisions.

Fresh variables are not given a name in our MetaCoq data structures. We use instead an anonymous name, and we let Coq's internal naming system find a suitable name to the variable at definition time.

Partial inductives. A partial inductive of Relation is defined for each constructor D_j of Pilot. Each partial inductive Relation_D_j is created by selecting the constructors C_i whose instance of the pilot index is compatible with D_j (it is either that constructor, or a variable that can take the constructor as value). From those selected constructors, we derive the constructors C_i_D_j of the partial inductive Relation_D_j by removing the pilot index from the arity, and adding the arguments of D_j to the parameters of this new inductive.

Here note that partial inductives are non-recursive by construction: constructor arguments of type Relation stay in type Relation. This is the place where one de Bruijn index gets a special treatment as mentioned below in Section **??**.

The dispatch function. Our inverter function needs a suitable return type which is the expected correct partial inductive, depending on the actual value of the pilot. To this effect we create a dispatch function that matches each constructor D_j of the pilot index to the corresponding partial inductive Relation_D_j.

At this step, the AST for the dispatch and inverter function has to be created from scratch, instead of being derived from an existing one. This leads to creating ranges of de Bruijn indices rather than manipulating existing ones.

The inverter function. This function takes an object in type Relation and, depending on its constructor C_i and the form of the pilot (either a variable or a constructor D_j), returns the corresponding constructor of the correct partial inductive C_i_D_j.

This is the most complicated part of our implementation for three reasons: First, if the pilot is not in the form of a constructor, but a variable, an additional embedded pattern matching is needed on that variable as well. In MetaCoq, the latter is translated using complex de Bruijn offset computing. Second, as the type or value of other indices may depend on the pilot's value, this embedded pattern matching needs lambda-abstractions for managing dependent types in its return branches. Finally, an additional preprocessing

computation is performed at data extraction stage in order to prepare the construction of terms (`C_i_D_j ...`) corresponding to patterns (`C_i ...`).

5 Lessons learned while using MetaCoq

5.1 Performance issues related to the use of TemplateMonad

As seen in figure **??**, each intermediate call to the bind constructor of the `TemplateMonad` creates another cycle of normalization and MetaCoq interpretation of the plug-in. This is accompanied by a translation of the term for each passage from one side to the other. This process is therefore time-consuming, and is better limited to the parts of a program where the `TemplateMonad` is really necessary.

We use a custom error monad for the cases where a monad is needed (e.g., getting an element at a given index in a list), with a function to translate a value of this error monad into an element of `TemplateMonad`. The switch from `TemplateMonad` to this custom-made monad improved execution times by a significant margin.

Lesson 1. Usage of `TemplateMonad` should be restricted to where API calls are necessary, and a custom monad should be used for other monadic needs.

5.2 Quoting and unquoting ASTs

Getting the AST of a Coq object is a two-step process. First, quoting an object returns the structure to call the quoted object by its qualified name. Then, using the qualified name, the AST can be quoted from the environment.

First, using `tmQuote`, the term representing the call to the object is obtained[2]. For example when quoting the term `nat` (the type of natural numbers), the result is

```
tInd {| inductive_mind := (MPfile ["Datatypes"; "Init"; "Coq"], "nat"); inductive_ind := 0 |} []
```

and when quoting the term for commutativity of addition (`Nat.add_comm`) we get:

```
tConst (MPdot (MPfile ["PeanoNat"; "Arith"; "Coq"]) "Nat", "add_comm") [].
```

However, a user quoting an object might be interested in the AST of its definition rather than the AST of a global reference to it. From such a qualified name, the AST corresponding to the body of its definition is retrieved using suitable API calls, respectively `tmQuoteInductive` and `tmQuoteConstant`.

To avoid two calls to the MetaCoq API, it is possible to use `tmQuoteRec` instead, which will return a global environment containing only the dependencies of the quoted object. Then, the call to `tmQuoteInductive` can be replaced by the pure Coq function `lookup_mind_decl` and similarly `tmQuoteConstant` can be replaced by `lookup_constant`. This tip, given to us by Yannick Forster, leads to better performance.

Lesson 2. Using `tmQuoteRec` to get a restricted environment, and then the lookup functions `lookup_mind_decl` and `lookup_constant` reduces the number of API calls and the scope to search the wanted AST in.

When producing a Coq object from an AST, the API call differs according to the nature of the object to be created. The API call to define a constant is `tmDefinition`. The definition body of type `term` has to be provided as a parameter. The type of the newly defined constant is computed during the API call.

The API call to define a possibly mutual inductive type is `tmMkInductive`'. It takes the AST of type `mutual_inductive_body`. As the structure `mutual_inductive_body` has redundant information to simplify access, some fields are not used in this definition of a Coq object. As such, they can be left empty or used to store debugging information that will

[2]The trailing empty lists [] are placeholders for universe polymorphism annotations.

disappear once the AST is converted into a Coq object. The fields we found to be unused in the redefinition are the list of parameters and indices of an inductive `ind_params` and `ind_indices`, as well as the lists of arguments and indices of a constructor `cstr_args` and `cstr_indices`.

Lesson 3. To define constants in the Coq environment, use `tmDefinition` applied to the AST of its body. To define inductive types, use `tmMkInductive'` applied to the `mutual_inductive_body`.

5.3 Manipulating de Bruijn indices

The manipulation of de Bruijn indices is tricky and error-prone, especially when performing recursive substitutions inside a term. Fortunately it is possible to work around such manipulations by placing ad-hoc local `let in` definitions in the telescope of the term to be substituted. We have two main usages for this technique: either replacing a variable by a definition body, or redirecting a de Bruijn index to a binding further up the AST, which results in replacing a variable by another. Then we use the let-reduction function `expand_lets` to perform substitution in place. This reduction is called zeta-reduction in the specification of CIC as documented in the Coq manual [**?**].

Here is an example of this technique: We have an AST

$\forall$ x, $\forall$ y, x+y=y

Its representation with de Bruijn indices is

$\forall$ x, $\forall$ y, (1)+(0)=(0)

To swap `x` and `y` in this formula, we insert two `let in` at the end of the telescope, so that the de Bruijn indices (1) and (0) point to them. These `let in` will have interchanged values to exchange the place of `x` and `y`. Please note that the inner term (1)+(0)=(0) does not need any direct modification, and that the technique would still work if inner binders were present.

$\forall$ x, $\forall$ y, let x' := (0) in let y' := (1) in (1)+(0)=(0)

which would unquote into:

$\forall$ x, $\forall$ y, let x' := y in let y' := x in x'+y'=y'

Finally, a reduction of the `let in` will give

$\forall$ x, $\forall$ y, (0)+(1)=(1)
$\forall$ x, $\forall$ y, y+x=x

Lesson 4. When substitution of de Bruijn indices is needed, use `let in` structures inserted in the telescope and reduce them to perform the operation.

In other situations, we use the `lift` function that increases all de Bruijn indices above a certain binding level by a given value. This is mainly used when adding new elements to the type telescope so that the indices continue to point to the same term as before. In reverse situations where we need to decrement de Bruijn indices because we are removing a term from a telescope we replace that element with a dummy `let in` definition whose reduction updates for free all other de Bruijn indices correspondingly.

Lesson 5. When adding an element to a telescope, use `lift` to restore proper de Bruijn references. To remove an element from a telescope, use the `let in` technique with a dummy definition.

The only situation where a direct manipulation of a de Bruijn index could not be avoided was when we needed to have some occurrences of a given de Bruijn changed into something different from the rest.

5.4 Working with the `match ... with` construct

In general, ASTs expressions in MetaCoq are quite finicky, and if there is an error while defining them, the error message will not be helpful in figuring out why. This is even more relevant for ASTs representing the `match` construct, because case analysis is a complex CIC concept and, not surprisingly, this complexity is reflected in the internal Coq representation. For instance, the scopes of the branches of the case analysis are different from the scope of the return type of the case analysis. As a result, a difference between the `return` type and the type returned by the branches may cause anomalies instead of errors.

The complexity of the `match` construct includes other aspects. For the `match` construct (of constructor `tCase`), the `in` clause binds the indices of the inductive type and the object itself, but not the parameters. To reference the parameters in the `return` clause, their appearance prior to the pattern matching must be referenced, while the de Bruijns for the indices of the inductive refer to the ones redefined in the `in` clause.

Lesson 6. Be mindful of the different scopes present in a match construct.

6 Conclusion

6.1 Our experience using MetaCoq

We have chosen to use MetaCoq because the `add_constructor` example from [**?**] presented just the right usage of key features. Such features are the extraction and representation in the form of ASTs of Coq objects, the definition of new Coq objects, inductive types and constants, from ASTs, and the ability to use these actions as part of functional programs in the `TemplateMonad`.

MetaCoq fulfilled its promises, it offered features otherwise exclusive to OCaml plug-ins. We managed to implement the automation we wanted in a reasonable time frame. Good coding practices, especially regarding de Bruijn indices lead to a code that can be incrementally expanded upon, and the quoting mechanism allowed us to compare the ASTs produced by our automation to the ASTs of the target Coq objects we wanted to define.

However, those coding practices are not easy to acquire, the learning curve is steep, and the package's API is difficult to find. This issue is compounded by a sparse online documentation, which is not always up to date. The community is welcoming and helpful to new users, and the best way to get help is the Zulip MetaCoq channel, where other users and the developers quickly answer most questions. A good place to look at some MetaCoq examples is [**?**].

Overall, we have found that MetaCoq is a very powerful tool that is still rough around the edges. Some errors may still result in anomalies, and the error trace is rarely useful. We can see that its self-proclaimed goal is to allow for formalization of the meta-theory as it lacks user-oriented API, such as a named layer to abstract away de Bruijn indices, or a printer for unchecked ASTs.

6.2 Future work using MetaCoq

Our current implementation of proxy-based small inversions has limits that we are striving to overcome: we are studying the systematisation of partial inductive creation in the presence various complications, such as deep patterns (`S (S n)` for `even`), multiple pilots, dependently typed pilots (like `VectorDef.t`), and various cases with non-linear patterns.

We are also considering possible applications of on-demand Coq object definition in other situations where Coq definitions need to be derived from user-defined objects, such as in the Braga method [**?**].

We could look into other solutions which seem to fulfill the same requirements as MetaCoq, such as Coq-ELPI [**?**].

References

[CT96] Cristina Cornes et Delphine Terrasse : Automating inversion of inductive predicates in Coq. *In* Stefano Berardi et Mario Coppo, éditeurs : *Types for Proofs and Programs*, pages 85–104, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[dB91] N. G. de Bruijn : Telescopic mappings in typed lambda calculus. *Inf. Comput.*, 91(2):189–204, 1991.

[GAA+13] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi et Laurent Théry : A machine-checked proof of the odd order theorem. *In* Sandrine Blazy, Christine Paulin-Mohring et David Pichardie, éditeurs : *Interactive Theorem Proving*, pages 163–179, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[GGMR09] François Garillot, Georges Gonthier, Assia Mahboubi et Laurence Rideau : Packaging mathematical structures. *In* Stefan Berghofer, Tobias Nipkow, Christian Urban et Makarius Wenzel, éditeurs : *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 de *Lecture Notes in Computer Science*, pages 327–342. Springer, 2009.

[LWM21] Dominique Larchey-Wendling et Jean-François Monin : *The Braga Method: Extracting Certified Algorithms from Complex Recursive Schemes in Coq*, chapitre 8, pages 305–386. World Scientific Publishing Company, 2021.

[Mon10] Jean-François Monin : Proof Trick: Small Inversions. *In* Yves Bertot, éditeur : *Second Coq Workshop*, Edinburgh, United Kingdom, July 2010.

[Mon22] Jean-François Monin : Small inversions for smaller inversions. *In TYPES 2022 Abstracts*, Nantes, June 2022.

[MS13] Jean-François Monin et Xiaomu Shi : Handcrafted Inversions Made Operational on Operational Semantics. *In* S. Blazy, C. Paulin et D. Pichardie, éditeurs : *ITP 2013*, volume 7998 de *LNCS*, pages 338–353, Rennes, France, July 2013. Springer.

[msF+24] mxdys, savask, Nathan Fenner, Justin Blanchard, Mateusz Naściszewski, Konrad Deka, Iijil, mei, Shawn Ligocki, Jason Yuen, Shawn Ligocki, Pavel Kropitz, Chris Xu, Shawn Ligocki et Dan Briggs : [july 2nd 2024] we have proved "bb(5) = 47,176,870". `https://discuss.bbchallenge.org/t/july-2nd-2024-we-have-proved-bb-5-47-176-870/237`, July 2024.

[SAB+20] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau et Théo Winterhalter : The metacoq project. *J. Autom. Reason.*, 64(5):947–999, juin 2020.

[SWB+24] Matthieu Sozeau, Théo Winterhalter, Simon Boulier, Nicolas tabareau, Yannick Forster, Jason Gross, Abhishek Anand, Meven Lennon-Bertrand, Gregory Malecha, Pierre-Marie Pédrot, Jakob Botsch Nielsen, Kenji Maillard, Gaëtan Gilbert, Danil Annenkov, yannl35133, Hugo Herbelin, Marcel Ullrich, Enrico Tassi, Maxime Dénès, Gabriel Scherer, Pierre Roux, Andrej Dudenhefner, Fabian Kunze, Emilio Jesús Gallego Arias,

4EVER2, Jim FEHRLE, Julin S, Karl PALMSKOG et Pierre ROUSSELIN : Metacoq/metacoq: Metacoq 1.3.2 for coq 8.20, août 2024.

[Tas18] 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, janvier 2018.

[Tea24a] The CompCert Development TEAM : Main website of the CompCert project. `https://compcert.org/`, 2024.

[Tea24b] The Coq Development TEAM : The Coq reference manual, release 8.19.1. pages 327–332, 2024.

[Tea24c] The Coq Development TEAM : Main website of the Coq project. `https://coq.inria.fr/`, 2024.

[WFLB] Théo WINTERHALTER, Yannick FORSTER et Meven LENNON-BERTRAND : MetaCoq tutorial at POPL24. `https://github.com/MetaCoq/tutorials/tree/main/popl24`.

[Zuc75] J. ZUCKER : *Formalization of classical mathematics in AUTOMATH*, pages 135–145. Department of Mathematics, Eindhoven University of Technology, July 1975.