Code Generation via Meta-programming in Dependently Typed Proof Assistants

Mathis Bouverot-Dupuis[1,2], Yannick Forster[1]

[1]Inria Paris
[2]ENS Paris

Abstract. Dependently typed proof assistants offer powerful meta-programming features, which allow users to implement proof automation or compile-time code generation. This paper surveys meta-programming frameworks in Rocq, Agda, and Lean, with seven implementations of a running example: deriving instances for the `Functor` typeclass. This example is fairly simple, but realistic enough to highlight recurring difficulties with meta-programming: conceptual limitations of frameworks such as term representation – and in particular binder representation –, meta-language expressiveness, and verifiability, as well as current limitations such as API completeness, learning curve, and prover state management, which could in principle be remedied. We conclude with insights regarding features an ideal meta-programming framework should provide.

1 Introduction

All proof assistants support user-extensible tactics and code generation through meta-programming frameworks. Meta-programs are programs that produce or manipulate other programs as data. They can in particular be used to generate boilerplate code, i.e. code that can be mechanically derived from definitions, thereby increasing the productivity of proof assistant users. Common examples are induction principles [81, 48], equality deciders [81, 38], finiteness proofs [25], countability proofs [25], or substitution functions for syntax [79]. Naturally, the default meta-programming language of a proof assistant is its implementation language, and several proof assistants even come with multiple independent meta-programming frameworks. However, we can observe that meta-programming is not widespread on the example of boilerplate generation tools which often fall into one of the following: Either proof assistants come with built-in boilerplate generation support (such as induction principles or typeclass instances) which is widely used. Or tools for generating boilerplate are developed, but not adopted by the community [81, 38, 15]. Lastly, many papers remark that automatic boilerplate generation would be feasible and interesting, but do not carry it out [85, 87, 34, 30]. Furthermore, subcommunities often seem to be split into silos regarding frameworks and we are not aware of scientific comparative work between different frameworks and proof assistants. The notable exception is Dubois de Prisque's PhD thesis [27], using several meta-programming frameworks in Rocq, but not coming with one central example implemented in different frameworks and focusing solely on Rocq.

An additional barrier to adoption is that most frameworks are organically grown and documentation is not accessible to non-experts: pros and cons are often implicitly known by developers but not readily accessible. In fact, the situation is so chaotic that, at times, in order to generate boilerplate code authors create ad hoc meta-programming facilities from scratch [79, 51, 80, 39] instead of taking advantage of existing meta-programming facilities.

On the other hand, the vast choice of meta-programming frameworks also hints that we are at a point where enough evidence is available to evaluate the state of the art and suggest future developments. In this paper, we focus on three major dependently typed proof assistants based on the Calculus of Inductive Constructions (CIC) [21, 66]: Rocq [82], Agda [63, 64], and Lean 4 [57]. We survey their respective meta-programming frameworks: Rocq OCaml plugins (§ 3), MetaRocq [5, 76] (§ 5), Agda's Reflection API [86] (§ 6), Lean 4's meta-programming API (§ 8)[1], Ltac2 [72] (§ 7), and Elpi [31, 81] (§ 9). In the appendix (§ 4), we furthermore explain a novel approach to use Rocq's OCaml API with locally nameless syntax. This means that we focus on systems which are designed as proof assistants with consistent meta-theory, rather than dependently typed programming languages, and focus on those with conceptual similarity and shared foundations. In particular, we do not consider Idris [12, 13], HOL-based systems such as Isabelle, HOL4, or HOL light, or LF-based systems such as Beluga, but discuss them in § 10.

We evaluate the different meta-programming frameworks on a simple yet realistic example: automatically deriving instances of the `Functor` typeclass for a simple family of inductives, covering, amongst many other types, options, lists, and trees. For Rocq we e.g. want to generate the following for the list type:

```
Fixpoint map {A B : Type} (f : A -> B) (l : list A) : list B :=
  match l with [] => [] | x :: l => f x :: map f l end.
```

Our implementations support non-mutual, non-indexed, possibly nested inductives with a single parameter. The only exception is the Lean implementation which does not support recursive (and thus nested) inductives, the reasons of which are explained in section § 8.

Many tasks involving automatic boilerplate generation follow the same model as this example: take an inductive as input and produce a term as output. We choose this example because it is simple enough for code to be readable and explainable, yet complex enough to expose issues that arise in more realistic meta-programs, and makes use of common meta-programming features such as typeclass search or the ability to extend the global environment.

Our evaluation criteria are split into *conceptual* criteria, which are inherent to the approach used by the meta-programming framework, and *current* criteria, which are incidental characteristics of the framework and could be changed in the future. *Conceptual* criteria include the expressiveness of the meta-language (especially access to printing, exceptions, non-termination, and mutable state), term representation used (especially of binders), and verifiability of

[1] There is no publication on meta-programming in Lean 4 yet, just a collaborative book draft [67]. Lean 3's meta-programming was surveyed by Ebner et al. [32].

meta-programs. Since this is an experience report, we also comment on learning curve: the author(s) of examples in this paper had no previous contact to most meta-programming frameworks, excluding Ltac2 and OCaml Rocq plugins. *Current* criteria include API completeness, management of the prover state (such as the global environment or unification state), and the presence of term quotation.

We do not consider performance issues in this paper: most inductive to term meta-programs run in an order of magnitude of seconds, and only have to be run once per inductive definition. In practice, developments rarely contain many inductive type definitions or very large inductive type definitions. Thus, we deem performance less critical than the aspects discussed here. Performance issues are critical when implementing proof automation (such as tactics) or more complex meta-programs such as unification algorithms or type checkers (for instance Rocq's verified kernel in MetaRocq [77] or Lean's kernel in Lean4Lean [14]), which are out of scope for this paper. We also do not consider actual verification of meta-programs, as it is not achievable in most of the frameworks we consider as of today, but still comment on verifiability when relevant.

This paper is the first evaluation of the state of the art in meta-programming and lays the foundations for future projects regarding meta-programming. Our perspective is of course subjective, and decidedly the perspective of someone who is *not* an expert in any of the discussed frameworks, which we deem representative of average users. One of the authors is a developer of MetaRocq, and the other author had some amount of experience with meta-programming in Lean, Ltac2, and OCaml plugins prior to this survey.

Consequently, insights in this paper might be well-known or even folklore for experts in the field and developers of frameworks. However, as far as we are aware, none of these insights have ever been written down transparently, and they are thus inaccessible for users of meta-programming frameworks.

Of all the aspects we discuss, variable binding techniques are certainly the one which has been discussed the most densely in related work. However, these techniques have mainly been discussed from the perspective of doing meta-theoretic proofs [9, 1], and not from the perspective of meta-programming. We thus think that our report complements these insights. We would not go as far as saying that we have identified a meta-programming challenge akin to the POPLmark challenge for formalisation, but our report can certainly be seen as a first step towards such a challenge.

Contributions.

1. A comprehensive survey of the six meta-programming frameworks in Rocq, Agda, and Lean, from the point of view of users rather than experts
2. An overview of their pros and cons.
3. A simple but realistic tool with different implementations for automatically deriving instances of the `Functor` typeclass.
4. Suggestions for the development of future frameworks. Our paper can also be seen as a first step towards a suggested Rosetta Stone project for meta-programming in Rocq, which is stalled [link anonymised].

Outline of the paper. § 2 introduces the example and relevant notions. §§ 3 to 9 discuss differents aspects of each implementation (including pros and cons). § 10 discusses related work. Finally § 11 puts everything together, and § 12 discusses future work.

2 Preliminaries

Meta-programs crucially rely on the features provided by the *elaborator* of the proof assistant, such as unification, type checking or inference, and typeclass resolution. They also need to manipulate the *state* of a proof assistant, consisting of the *global environment* (stores global definitions and inductives), *local environment* (local variables, and in the case of Rocq also section variables and hypotheses), and *evar-map* (unification variables). This manipulation can be implicit, or explicit by threading the three components through programs.

How binders are represented is a key question for meta-programs. De Bruijn indices and the locally nameless approach were both introduced by de Bruijn in his seminal paper [26] and are used in the implementations of Rocq, Agda, and Lean and consequently in the associated meta-programming frameworks. The notable exception is Coq-Elpi, which uses higher-order abstract syntax (HOAS) [68]. We give examples of the representations in the respective sections.

Our example is the generation of instances of the following Functor typeclass:

```
Class Functor (F : Type -> Type) : Type :=
  { fmap {A B} : (A -> B) -> F A -> F B }.
```

We have already shown the instance for `list` in the introduction. Our implementation also handles more complex types, for instance using nesting:

```
Inductive tree (A : Type) : Type :=
| Leaf : tree A
| Node : A -> list (tree A) -> tree A.

Fixpoint fmap {A B : Type} (f : A -> B) (t : tree A) : tree B :=
  match t with Leaf => Leaf
             | Node x ts => Node (f x) (List.map (fmap f) ts) end.
```

In general, inductives can be non-recursive (e.g. `bool`, `option`), recursive (`nat`, `list`), have parameters (`option`, `list`), or have indices (`vector`). We support as input inductive types with a single (uniform) parameter and no indices. We do not support mutual recursion for simplicity. For simplicity, the code samples shown in the paper do not handle recursive inductives: the complete implementations do handle the general case, apart from the one in Lean, see § 8.

In the case of `Node` in the example above, we applied `f` to the first argument `x`, and `List.map (fmap f)` to the second argument `ts`. In general, there are various ways to disambiguate what function to apply. The canonical way is to do type-based disambiguation. An alternative is to use typeclass inference of the proof assistant, which is what we do in our implementation.

For instance, we define the second branch as `Node (fmap f x) (fmap f ts)` in the example above, and let typeclass resolution determine which functor we

are mapping over. To be able to do so, we need to use the identity functor for `x` and for `ts` the composition of the `list` and `tree` functors.

Consequently, we globally declare:

```
Instance fid : Functor (fun T => T).
Instance fcomp (F G : Type -> Type) `(Functor F) `(Functor G) :
  Functor (fun T => G (F T)).
```

Naively using the typeclass-based approach may fail in the case of recursive inductives such as `tree` or `list`. For instance in the `tree` example, typeclass resolution will fail to find a `Functor` instance for `fmap f ts` (because there is no instance of `Functor tree` in scope). We can solve this issue by using a local typeclass instance. In the case of `tree`:

```
Fixpoint fmap {A B : Type} (f : A -> B) (t : tree A) : tree B :=
  let _ := Build_Functor tree fmap in
  match t with Leaf => Leaf
             | Node x ts => Node (fmap f x) (fmap f ts) end.
```

Typeclass resolution will now consider the local instance `_ : Functor tree` when elaborating `fmap f ts`. Note that this stretches the limits of what the termination checker is capable of accepting: we had to disable Agda's termination checker, and in the case of Rocq we had to help the guard checker by normalizing `fmap` before adding it to the global environment.

We do not expect alternative approaches to this problem to alter the conclusions of this experience report.

3 OCaml Plugin

As Rocq is implemented in OCaml, writing an OCaml plugin is historically the most common way of meta-programming [65, 73, 29, 24], see Fig. 1 for the code.

Pros – Conceptual	P1 - Plugins have access to full implementation.
Pros – Current	P2 - OCaml is a mature programming language.
Cons – Conceptual	C1 - De Bruijn index arithmetic is difficult.
	C2 - No term quotations.
Cons – Current	C3 - OCaml plugins are hard to set up.
	C4 - Cluttered meta-programming API.
	C5 - Explicit state management.

P1 - Conceptual OCaml plugins allow users to directly access all of Rocq's implementation. Meta-programs manipulate the kernel representation of terms:

```
type EConstr.t =
| tRel (idx : int)                          (** Local variable. *)
| tApp (f : EConstr.t) (l : EConstr.t list)  (** Application. *)
| ...
```

For instance the function `build_fmap` in our code simply returns a term corresponding to the mapping function over the given inductive:

```
let build_fmap env sigma ind : evar_map * EConstr.t = ...
```

The evar-map `sigma` is updated and returned alongside the resulting term. Meta-programs have access to all the functionality provided by Rocq, including term manipulation functions, unification, and the tactic engine. Such direct access guarantees that the API is complete: users can leverage every customisable aspect of the proof assistant, including features not commonly found in other meta-programming languages, such as the ability to extend the parser. Moreover, it ensures the API stays up to date with the latest Rocq features: when new features (e.g. universe polymorphism) are added to Rocq, one typically has to wait some time before the various Rocq meta-languages add support for them.

P2 - Current OCaml is a general-purpose programming language used in many applications besides Rocq meta-programming and thus enjoys a large ecosystem of packages, as well as robust and well-maintained tools (such as a language server, a code formatter, an optimising compiler, a package manager, etc), which is not the case for Rocq's other meta-languages.

C1 - Conceptual OCaml plugins directly manipulate the kernel representation of terms, which uses de Bruijn indices for variables. For instance the term $\lambda f.\lambda x.\lambda y.f\ x\ y$ is represented as $\lambda.\lambda.\lambda.3\ 2\ 1$. De Bruijn indices require a significant amount of experience to manipulate correctly: writing explicit indices and lifting terms was a major source of errors when getting started, e.g.

```
let sigma, arg' = build_arg env sigma
  (lift_inputs (i+1) inp) (mkRel 1)
  (Vars.lift 1 @@ EConstr.of_constr @@ get_type decl)
in loop env sigma (i+1) (lift (ca.cs_nargs-i-1) arg' :: acc) decls
```

C2 - Conceptual Most meta-languages provide a high-level method to build terms using *term quotations*, which is a lightweight mechanism allowing one to turn user syntax terms into the internal representation used by the meta-language. Plugins do not provide any quotation mechanism: building terms is thus rather verbose and tedious. In the absence of term quotations, one has to provide fully qualified kernel names, pass all implicit arguments to functions, provide typeclass instances by hand (or manually create unification variables to stand in for unknown instances), and explicitly instantiate all universe polymorphic constants and inductives.

C3 - Current Integrating a plugin into a build system currently requires significant overhead, even when using the modern `dune` build system for Rocq: one has to include plugin-specific dune stanzas as well as several other build-specific files. This is in stark contrast with most other meta-languages.

C4 - Current The plugin API is cluttered, thus hard to use for non-experts. It provides code to accomplish common meta-programming tasks but finding the right function often requires reading their *implementation* (i.e. reading **.ml* files in addition to **.mli* files), or asking the Rocq developers for help. Fortunately the developers are easy to reach (via online forums) and eager to provide help.

C5 - Current Finally, plugins provide no good solution for managing the prover state: the environment and evar-map are explicitly passed as arguments to and

returned from most functions. For instance, here is the code which builds the outer lambda abstractions of `fmap`:

```
let build_fmap env sigma ind : evar_map * EConstr.t =
  lambda env sigma "a" ta @@ fun env ->
  lambda env sigma "b" tb @@ fun env ->
  lambda env sigma "f" (arr (mkRel 2) (mkRel 1)) @@ fun env ->
  lambda env sigma "x" (apply_ind env ind @@ mkRel 3) @@ fun env ->
    (sigma, ...)
```

Here `lambda env sigma "x" T k` builds a lambda abstraction with a binder named `x` and of type `T`; the continuation `k` takes in the new environment (updated with a binding for `x`) and returns the body of the lambda abstraction alongside the updated evar-map. The resulting code is verbose and obfuscates the core logic.

4 OCaml Plugin - Locally Nameless Version

De Bruijn indices are arguably the most common binder representation in implementations of dependently-typed proof assistants (Rocq, Agda, and Idris all use de Bruijn indices internally), however an alternative is *locally nameless*, in which bound variables are represented using de Bruijn indices, but free variables are named. For instance the term $\lambda x.\lambda y.f\ x\ y$ (which has one free variable f) is represented in locally nameless as $\lambda.\lambda.f\ 1\ 0$. Maintaining the locally nameless invariant as terms are traversed requires some amount of bookkeeping from the user, but crucially all index arithmetic and lifting is performed by the API. We refer to McBride and McKinna [56] for an introduction to programming with locally nameless, and to the work of Charguèraud [16] for a more formal perspective.

Rocq already has most of the infrastructure needed for locally nameless: indeed, section variables and local hypotheses are represented using named variables instead of indices. We can reuse these named variables to represent free variables. Named variables have good support in the OCaml implementation: they have their own named local context, and most functions which take as input the global environment and local context – such as unification or typeclass search – also support the named local context. Using named variables, we can implement a small library of term manipulation functions in the locally nameless style, and use it to improve the plugin of the previous section by removing the friction points related to index manipulation.

Figure 2 shows the corresponding code. The `lambda` function has been replaced by `namLambda`:

```
namLambda (env : Environ.env) (sigma : Evd.evar_map) (name : string)
  (ty : EConstr.t) (mk_body : Environ.env -> Evd.evar_map ->
  Names.variable -> Evd.evar_map * EConstr.t) :
  Evd.evar_map * EConstr.t
```

The most notable difference is that `namLambda` passes a fresh named variable `x` to the continuation `mk_body`. The continuation builds a term in which `x` is free, and `namLambda` subsequently replaces all occurrences of `x` in the body by de Bruijn

index 1 (de Bruijn indices start at 1 in plugins). Additionally, when referring to variable `x` we use `mkVar x` instead of writing down the exact de Bruijn index.

The switch to locally nameless is the only difference between the plugin of § 3 and this version; we concentrate on the pros and cons that come with locally nameless.

Pros – Conceptual	P1 - Locally nameless removes the frictions of de Bruijn indices.
Pros – Current	P2 - Rocq is easy to extend with a locally nameless API.
Cons – Conceptual	C1 - Possible performance issues associated to locally nameless.
Cons – Current	

P1 - Conceptual We found programming using locally nameless much easier than using de Bruijn indices. Using locally nameless, one never has to write exact de Bruijn indices, perform index arithmetic, or lift terms. For instance, compare the first lines of `build_fmap` using de Bruijn indices (in Figure 1):

```
lambda env sigma "a" ta @@ fun env ->
lambda env sigma "b" tb @@ fun env ->
lambda env sigma "f" (arr (mkRel 2) (mkRel 1)) @@ fun env ->
lambda env sigma "x" (apply_ind env ind @@ mkRel 3) @@ fun env ->
```

And using the locally nameless style (in Figure 2):

```
namLambda env sigma "a" ta @@ fun env sigma a ->
namLambda env sigma "b" tb @@ fun env sigma b ->
namLambda env sigma "f" (mkArrowR (mkVar a) (mkVar b)) @@ fun env sigma f ->
namLambda env sigma "x" (apply_ind env ind @@ mkVar a) @@ fun env sigma x ->
```

P2 - Current Locally nameless integrates well with Rocq's internal API: even intermediate terms which contain named (free) variables are compatible with Rocq's infrastructure, and can be passed for instance to unification or type checking. The fact that Rocq has such good support for named variables out of the box made it very easy to use a locally nameless style. Without such support we would need to implement a separate representation for terms, and convert back and forth between these two representations, causing both performance and usability issues.

C1 - Conceptual The locally nameless binder representation requires very frequent substitution of named variables with de Bruijn indices, and vice versa; although this is hidden from the user, it does have an impact on performance. Measuring the precise cost of these substitutions in practice is outside the scope of this study, but we did not notice a significant performance degradation in our case. If it becomes an issue, one can use a pure de Bruijn representation in performance-critical sections, breaking the locally nameless invariant locally, while still enjoying locally nameless in the rest of the code. This is exactly what the implementation of Lean 4 does, in addition to some simple caching optimisations to improve the performance of substitutions in common cases. Implementing these optimisations in Rocq is not as straightforward as one would hope, and has not been done in this study.

Locally nameless in other frameworks One could also imagine using MetaRocq's implementation to work with locally nameless. Unfortunately most of the API does not work well with named variables because they do not occur in closed terms checked by the kernel, and thus MetaRocq does not even have a notion of named local context. Following a locally nameless discipline would require one to either convert terms to a pure de Bruijn representation on the fly when calling MetaRocq functions, or add support for named variables throughout the entirety of MetaRocq, requiring significant engineering effort.

5 MetaRocq

The MetaRocq project [5, 76] is a fully verified re-implementation of Rocq's kernel in Rocq, and also includes a meta-programming API in one of its subprojects. The bare-bones meta-programming framework has been used in [23, 48, 35, 77, 36, 7, 27, 4, 6]. Figure 3 shows the Rocq code for the MetaRocq plugin.

Pros – Conceptual	P1 - Users already know Rocq.
	P2 - Meta-programs can be formally verified.
Pros – Current	P3 - Significant parts are formally verified.
Cons – Conceptual	C1 - De Bruijn index arithmetic is difficult.
	C2 - Lack of abstractions to handle effects.
Cons – Current	C3 - Explicit state management.
	C4 - Missing high-level meta-programming features.
	C5 - Performance issues in some cases.

P1 - Conceptual An appealing aspect of MetaRocq is the ability to perform meta-programming directly using the host language Rocq, flattening the learning curve significantly. To this end, the AST of terms is *reified* in Rocq:

```
Inductive term :=
| tRel : nat -> term
| tApp : term -> list term -> term
| ...
```

Rocq's kernel is re-implemented in Rocq, thus standard functions such as reduction, conversion, and type checking are readily available:

```
(** Conversion checking, implemented in Rocq. *)
Definition eq_term : term -> term -> bool.
```

For higher-level APIs (e.g. extending the global environment), MetaRocq comes with a monad `TemplateMonad` with bindings to Rocq's actual OCaml implementation:

```
(** Declare a new constant. Simply a wrapper around OCaml code. *)
Axiom tmMkDefinition : constant_entry -> TemplateMonad unit.
```

Such monadic programs can be run using the `MetaRocq Run` command.

P2 - Conceptual In addition to useful meta-programming features, MetaRocq includes an extensive formalisation of Rocq's type theory, with proofs of key results of theoretical interest such as subject reduction. It is possible to formally verify meta-programs by leveraging Rocq's theorem proving features in combination with the numerous lemmas already present in MetaRocq. Note however that there is no specification for the operations which use the template monad: verifying meta-programs which use high-level features such as unification or type inference is still an active research area.

P3 - Current Many of the functions provided by MetaRocq are formally verified with respect to Rocq's type theory, providing strong correctness guarantees. This is used to build a certified extraction procedure by Forster, Sozeau, and Tabareau [35]. Formal verification is very appealing considering the complexity inherent to proof assistants; note however that verified implementations often do not benefit from the same optimisations and clever heuristics (for instance reduction using explicit substitutions or abstract machines) as the equivalent code in Rocq's OCaml implementation.

C1 - Conceptual MetaRocq implements binders using de Bruijn indices, which have the same drawbacks we explained in § 3.

C2 - Conceptual Rocq does not provide abstractions to handle effects such as printing, raising exceptions, or writing non-terminating functions, so writing non-trivial programs quickly becomes tedious. Herbelin's `reduction-effects` plugin [41] allows to print values, but is currently only suitable for debugging. Moreover, writing partial or possibly non-terminating programs is impractical: Common solutions include disabling the guard checker or using step-indexing.

MetaRocq relies crucially on *monads* to handle effects, most notably the template monad, which is in our experience a notable friction point: monadic programs in Rocq are difficult to debug, as programming errors often cause implicit argument resolution to fail, leading to obscure error messages. There does not seem to be a consensus on how to implement monads in Rocq: a promising attempt is the Monae library [74] which formalises monads in the style of Rocq's Mathematical-Components library using the Hierarchy Builder tool [19]. MetaRocq packages its own monad library following a simpler design called *semi-bundled* typeclasses. The latter approach is used successfully in languages such as Haskell and in Lean's mathematical library [55], but its current implementation in MetaRocq is unsatisfactory.

C3 - Current State management in MetaRocq is explicit: the global environment and local context have to be threaded manually, and this issue will only worsen as more state (such as the evar-map) is added. Rocq being a pure language, the obvious solution is to use monads to hide the state.

C4 - Current MetaRocq's high-level API lacks many crucial features. Indeed, MetaRocq only re-implements Rocq's kernel: higher-level features such as unification or typeclass resolution have to be exposed via bindings to Rocq's actual OCaml implementation. Many of these bindings are either missing or incomplete. For instance, unification is entirely missing, and defining new constants does not currently support universe polymorphism.

C5 - Current Performance of MetaRocq programs which use the template monad can be quite poor: we noticed slowdowns of up to two orders of magnitude. Obtaining reasonable performance required writing our program in two layers. The inner layer `build_fmap` does not make use of the template monad. All effects (e.g. declaring the new typeclass instance) are pushed to the outer layer `derive_functor`:

```
(** Inner layer: build the mapping function for a given inductive. *)
Definition build_fmap : inductive -> term.
(** Outer layer: wrap build_fmap with pre- and post-processing. *)
Definition derive_functor {A} : A -> TemplateMonad unit.
```

This approach does not scale: in a more realistic meta-program, `build_fmap` might need to perform unification or typeclass resolution, which requires using the template monad.

6 Agda

Meta-programming in Agda is similar to MetaRocq: Meta-programs are directly written in Agda, using the *Reflection* API [86], and interact with the elaborator and kernel via the `TC` monad. Figure 4 shows the code for the Agda plugin.

Pros – Conceptual	P1 - Users already know Agda.
Pros – Current	P2 - Implicit state management using monads.
Cons – Conceptual	C1 - De Bruijn index arithmetic is difficult.
	C2 - Restrictive term representation.
Cons – Current	C3 - Type-class search is hard to control.
	C4 - Performance issues in some cases.

P1 - Conceptual Agda meta-programs are simply Agda programs with a return type in the type checking monad (`TC`): this has the benefit that users do not need to learn a new programming language. MetaRocq uses the same approach based around a monad – the MetaRocq equivalent of the type checking monad is the template monad – however in Agda's case the type checking monad also contains the prover state, whereas in MetaRocq the state is threaded explicitly.

P2 - Current The prover state is contained in the type checking monad. It is accessed through primitives provided by the type checking monad, e.g.:

```
-- Get the definition of a constant.
getDefinition : Name -> TC Definition
-- Extend the current context with a variable.
extendContext : {a} {A : Set a} -> String -> Arg Type -> TC A -> TC A
```

We found monadic programming in Agda to be lightweight and enjoyable, thanks to the extensive library agda-stdlib-classes [20] based on typeclasses.

C1 - Conceptual Agda implements binders in terms using de Bruijn indices, which have the same drawbacks we explained in § 3. Additionally, we note that the API to manipulate the local (de Bruijn) context is currently awkward to use: for instance some functions expect the context in reverse order (last to first), and some expect it in normal order (first to last).

C2 - Conceptual The internal representation of terms is quite restrictive. First, `let` bindings (as well as `where` clauses) are not represented in the abstract syntax, but are instead inlined during type checking: meta-programs cannot make use of such features when building terms. Second, terms are represented in spine form (the head of an application cannot be a lambda abstraction), making it difficult to build some terms. Implementing substitution on this representation of terms is quite delicate, and in fact there is no substitution function available in the meta-programming API (some third-party libraries [62] implement substitution).

C3 - Current Type-class resolution is difficult to use when meta-programming. First, although Agda has the concept of local typeclass instances, using local instances via the meta-programming API is not directly supported and requires awkward workarounds. Second, Agda's implementation of typeclass search is quite weak when compared to Rocq and Lean: support for overlapping instances and backtracking search is relatively recent, and recursive instances can still cause typeclass resolution to loop. This is because instance search is implemented using a depth-first search. In our case a crucial typeclass instance (for the composition of functors) is recursive: we had to cap instance search at a very low depth to get acceptable type checking times. This workaround would not scale to real-world applications, and in our case means that our Agda example does not support deeply nested inductives.

C4 - Current Performance of meta-programs in the type checking monad can be poor: we noticed significant slowdowns compared to equivalent pure code on small examples, but providing larger benchmarks is outside our scope.

7 Ltac2

In the above sections, we have built `fmap` by directly manipulating the kernel representation of terms. An alternate approach, following the Curry-Howard correspondence, is to use tactics to produce a proof term:

```
fmap : forall A B, (A -> B) -> F A -> F B.
```

For example, using Ltac2 tactics one can define `fmap` on `option` as:

```
Definition fmap : forall A B, (A -> B) -> option A -> option B.
  intros A B f x. destruct x.
  - (* Some *) intros y. constructor 0. exact (f y).
  - (* None *) constructor 1.
Defined.
```

Here low-level details such as binder representation are taken care of by dedicated tactics such as `intros`: there is no need to handle the internal representation of terms. This observation remains true when generalising to arbitrary inductives. Consider the code corresponding to the tactics `intros` and `destruct` for an arbitrary inductive:

```
(* Expects a goal of the form [forall A B, (A -> B) -> F A -> F B]. *)
Ltac2 build_fmap F : unit :=
  (* intros A B f x *)
  intro @A ; intro @B ; intro @f ; intro @x ;
  (* destruct x *)
  Std.case false (Control.hyp @x, NoBindings) ;
  (* Build each branch. *)
  let n_ctors := ... in
  Control.dispatch (List.init n_ctors (build_branch F @A @B @f)).
```

Using tactics for meta-programming is not a novel idea: the elaborator of Idris 1 [12] uses tactics to transform concrete syntax into abstract syntax. We are however not aware of other meta-programming uses.

Pros – Conceptual	P1 - No need to manipulate the low-level term representation.
Pros – Current	P2 - Implicit state management.
Cons – Conceptual	C1 - Implicit backtracking.
Cons – Current	C2 - Tactics are hard to reason about. C3 - Ltac2 is missing many basic language features. C4 - Incomplete meta-programming API.

P1 - Conceptual Tactics remove the need to manipulate the low-level kernel representation of terms. For instance, one never has to deal with de Bruijn indices when building terms. In fact, tactics follow the locally nameless discipline: free variables are treated as local hypotheses, and are represented using names. The tactic `intro x` removes the outermost binder in the goal and replaces all occurrences of de Bruijn index 0 with named variable `x`; conversely, the tactic `revert x` replaces all occurrences of named variable `x` in the goal with de Bruijn index 0 and adds a binder. Tactics generally maintain the same invariant as locally nameless: free variables are represented using names, while bound variables use de Bruijn indices. Most importantly, users do not have to care about such considerations at all: while Ltac2 does provide ways to access the underlying representation of terms (using de Bruijn indices), at no point is this needed in our code.

P2 - Current Ltac2 does not explicitly expose the prover state. Information about global constants, local variables and unification variables can be queried when needed. However the current API to interact with the prover state is in our opinion difficult to use and could be improved, as discussed in C4.

C1 - Conceptual A unique characteristic of tactics is *implicit backtracking*, as described by Spiwack[78]. Tactics produce a lazy stream of successes, optionally followed by a single failure. For instance the `constructor` tactic tries to apply the first constructor of an inductive; if the following tactic fails, the proof engine backtracks up to `constructor`, which is asked for a second success, until all constructors have been tried. Such lightweight backtracking is useful for proof construction, but not for meta-programming, and we even argue that implicit backtracking could be harmful in the context of generating terms with computational content, as it decreases predictability.

C2 - Conceptual Reasoning about tactic programs, both informally and formally, is very difficult. The main reason – in addition to implicit backtracking which was discussed above – is that the tactics of most proof assistants pile heuristic on heuristic, such that statically specifying the effects of a tactic is practically impossible. We are not aware of existing work on formal reasoning about tactics, but since tactics are effectful programs using a tailored program logic is conceivable.

C3 - Current Ltac2 is missing many convenience features at the language level, such as syntactic sugar or a proper printing mechanism. Its standard library is bare-bones, and documentation and learning resources are scarce. The language is however under active development.

C4 - Current Ltac2 is primarily a tactic language, and is missing many basic meta-programming features. Extending the global environment is impossible, declaring new unification variables or local variables is difficult, and manipulating the kernel representation of terms is impractical. In general the low-level API to manipulate the kernel terms is lacking, and most high-level facilities (including unification, term quotations, and most of the OCaml API re-exported by Ltac2) do not support terms containing free de Bruijn indices. Moreover, switching between tactic mode and direct term manipulation is hard to perform: in our case it was necessary (for technical reasons related to Rocq's guard checker) to normalise the term built by `build_fmap`, which required awkward workarounds.

8 Lean

The elaborator of Lean 4 (including parsing, unification, type inference, and typeclass resolution) is implemented in Lean itself [58], and self-hosting the kernel is subject to active research (see the Lean4Lean project [14]). Meta-programs are simply Lean programs which have access to the Lean implementation. Lean's meta-programming features are used in many projects [50, 10, 49, 60], notably its mathematical library [55], and most of the implementation of Lean's elaborator can be considered meta-programming. Fig. 5 shows the Lean code.

Pros – Conceptual	P1 - Users already know Lean.
	P2 - Access to complete Lean implementation.
	P3 - Locally nameless binder representation.
Pros – Current	P4 - Implicit state management using monads.
Cons – Conceptual	C1 - Restricted term representation.
Cons – Current	

P1 - Conceptual Meta-programming is done directly in Lean: this has the benefit of relieving the user from learning a new domain-specific language, but goes much further, as explained in the next paragraph.

P2 - Conceptual Meta-programs can access the entire API of the Lean implementation. This has benefits for the developers, which do not need to manually expose bindings to every useful API function, and allows users to seamlessly

access parts of the implementation which would typically not be part of a meta-programming language, e.g. related to the *concrete* syntax of terms, such as the parser. We also note that support for instrumenting the parser to implement various notations, macros, and embedded DSLs is particularly good. In fact Lean macros are so powerful that they allow some form of basic meta-programming, although we did not make use of such functionality during this study. We refer the reader to the work by Ullrich [83] for an overview of Lean macros.

P3 - Conceptual Local variables are internally represented using locally nameless: free variables use names, while bound variables use de Bruijn indices. For instance the term $\lambda x.\lambda y.f\ x\ y$, which has one free variable f and two bound variables x and y, is represented as $\lambda.\lambda.\lambda.f\ 2\ 1$. The user is responsible for maintaining the invariant that free variables are named and bound variables use indices. For instance in our code:

```
-- Build the function `fmap' as `fun A B f x => body'
def buildFmap ind : MetaM Expr := do
  -- The body contains free (i.e. named) variables.
  let body := ...
  -- Replace names with indices and add lambda abstractions.
  mkLambdaFVars #[A, B, f, x] body
```

Overall there is no need to directly manipulate de Bruijn indices, and we found locally nameless to be pleasant to use.

P4 - Current Lean is a pure language, and modifying prover state is done using monads: this provides the lightweight programming experience of implicit state, while keeping some level of control over which effects can be performed. A meta-program in `CoreM` can access the global environment but may not assign metavariables, while a meta-program in `MetaM` has access to the global environment, local context, and metavariables. We note that Lean has excellent support for monadic programming, as described by Ullrich and de Moura [84].

C1 - Conceptual Lean's abstract term syntax has no first-class fixpoints or case expressions (as for instance in Rocq): instead, concrete syntax fixpoints and case expressions are compiled to primitive recursors, which are represented as global constants with special reduction rules. For instance a case expression on an `option` is represented as an application of the ``` ``option.casesOn ``` primitive recursor.

Benefits include having a simpler meta theory and no need for a guard checker in the kernel. However, we perceived this as a severe limitation. In practice recursors for nested inductives quickly become unwieldy. Because the fixpoint and pattern matching compiler only accepts concrete syntax, it cannot be used by meta-programs which work on abstract syntax.

Generating concrete syntax (instead of abstract syntax) solves this particular issue with fixpoints and case expressions, but concrete syntax is very difficult to manipulate, and most functions in the Lean meta-programming API (such as unification or type inference) do not work with concrete syntax. Due to these difficulties, our Lean implementation does not support recursive inductives (e.g. lists or trees).

9 Elpi

Elpi is a logic programming language based on λProlog [59] which can be used as a meta-programming language for Rocq [75, 27, 22, 18, 81, 38, 54]. Elpi is notably used to implement Hierarchy Builder [19], a type checker and elaborator for the Calculus of Inductive Constructions [40], and a typeclass resolution algorithm for Rocq [33]. Figure 7 shows the Elpi code.

Pros – Conceptual	P1 - Higher-order abstract syntax.
Pros – Current	P2 - Powerful quoting and unquoting mechanism.
Cons – Conceptual	C1 - Paradigm shift (logic programming).
Cons – Current	C2 - Limited representations for structured data.

P1 - Conceptual A key feature of Elpi is the abstract syntax it uses for Rocq terms, and in particular for binders, called higher-order abstract syntax (HOAS) and due to Pfenning and Elliott [68]. The syntax of Rocq terms is encoded in a data-type `term`, of which we show a few constructors:

```
type app      list term -> term.
type fun      name -> term -> (term -> term) -> term.
```

Application nodes are represented using `app`, lambda abstractions using `fun`. There is no constructor for variables. The last argument of `fun` is the body of the lambda abstraction, an elpi function of type `term -> term`: Rocq variables correspond to Elpi variables. For instance the Rocq function `fun x : nat => x` is encoded in Elpi as `fun `x` (global (indt «nat»)) (x\ x)`, where `x\ x` is the Elpi identity function.

Meta-programming in Elpi does not require dealing with de Bruijn arithmetic, and the type checker helps catch scope issues when building terms. Overall we found HOAS easy to use.

We note that HOAS relies crucially on the logic programming aspects of Elpi. HOAS is incompatible with dependently typed proof assistants due to the strict positivity condition on inductives, and even in languages such as OCaml in which it is possible to define a HOAS term grammar, it is unclear how to define basic term manipulations (such as counting the number of variables) [17, Section. 2.1]. We did not investigate parametric higher-order abstract syntax [17].

P2 - Current Elpi offers a powerful quotation mechanism to build the AST of Rocq terms with Rocq user syntax. Quotations are inserted using braces:

```
pred build-fmap i:inductive, o:term.
build-fmap I {{fun A B (f : A -> B) (x : lp:(FI A)) => lp:(M A B f x)}
```

Anti-quotations `lp:(...)` insert elpi code inside quotations. Most importantly, quotations and anti-quotations allow open terms. This is not the case in most meta-languages which support quoting (Lean being a notable exception). Moreover, Rocq unification variables correspond almost one to one with Elpi unification variables, allowing meta-programs to trigger Rocq's unification simply by using Elpi's built-in unification.

C1 - Conceptual Elpi is a logic programming language, which is a paradigm shift compared to dependently-typed proof assistants. An Elpi program is composed of predicates, which relate input(s) to output(s):

```
pred build-fmap i:inductive, o:term.
build-fmap I F :- ...
```

The program above declares the predicate `build-fmap` with one input `I` (the inductive we are mapping over) and one output `F` (the term `fmap` we are building). The second line adds a rule with conclusion `build-fmap I F`, which describes how to build the term `F` given `I`. A consequence of this paradigm shift is that Elpi comes with a steep learning curve.

C2 - Current On the language level, Elpi provides limited options for representing structured data. There are no ML-style records; in fact it is common for Elpi functions to have more than half a dozen input and output parameters, e.g.:

```
pred build-branch i:inductive, i:term, i:term, i:term, i:term,
                  i:term, i:list term, i:list term, o:term.
```

Only open sums are available: constructors can be added at any point in the program. Open sums enable clever programming tricks, but the lack of closed sums prevents static checking of whether a function handles all input cases.

10 Related Work

Dubois de Prisque [27] compares Ltac1, Ltac2, MetaRocq, and Elpi as meta-programming languages in tutorial style through different examples. The methodology differs from ours in that the choice of frameworks is restricted to a single proof assistant (Rocq), and we focus on a single example which - while realistic enough to highlight many issues - yields implementations simple enough to be understood by non-experts. The conclusions of Dubois de Prisque are that Ltac1 lacks a clear semantics and static typing, and tactics not being allowed to have side effects and return a value leads to ubiquitous, hard-to-read CPS translations. Ltac2 solves many of the issues of Ltac1, but manipulating the low-level term representation is difficult (many things were even impossible at the time the thesis was written). MetaRocq allows to manipulate the low-level term representation, and might allow formally proving correctness of the meta-programs. However, de Bruijn arithmetic and the lack of proper abstractions for effects (in particular mutable state and non-terminating functions) is criticised. For Elpi, Dubois de Prisque lauds the benefits of HOAS and remarks that term quotations are beneficial. However, HOAS seems to be difficult for term-to-term transformations when the structure of the output is very different from the input. Writing tactics was reported as tedious, which however could be a problem related to how Elpi represents the proof context, i.e. might not be conceptual.

HOL-based proof assistants come with meta-programming support in their host language. HOL4 and HOL light might offer the most natural experience, since proving happens just in an OCaml session. Isabelle allows meta-programming in Standard ML. Beluga is a proof assistant for the mechanisation of meta-theory based on contextual modal type theory. There is a lot of ongoing work on how to make quotation native in contextual modal type theory and thus allow certifiable meta-programming [43, 69, 44, 46, 45, 42]. The concerns of this setting

are somewhat orthogonal to ours: they try to understand the foundations of meta-programming by extending type theory, whereas we focus on power and usability of frameworks that are built on top of type theory.

11 Conclusion

	OCaml	MetaRocq	Agda	Lean	Ltac2	Elpi
De Bruijn indices	✗	✗	✗		✗	
Restricted term AST			✗	✗		
No quasi-quotations	✗	✗	✗		✗	
Explicit prover state handling	✗	✗				
Cannot verify meta-programs	✗	✗	✗	✗	✗	✗

Conceptual issues with each meta-programming framework.

	OCaml	MetaRocq	Agda	Lean	Ltac2	Elpi
Need to learn a new language	✗				✗	✗
Incomplete API		✗	✗		✗	✗
Lack of learning resources	✗	✗				
Lack of documentation	✗	✗	✗	✗	✗	✗

Current issues with each meta-programming framework.

Conceptual. *Binder representation* was a recurring issue in this paper. Meta-programs involve manipulating terms as data, and as such it must be easy to construct and inspect the structure of terms, including binders. In particular, we note that our de Bruijn-based implementations use arithmetic on indices, which leads to frequent mistakes and bugs. The locally nameless representation simplifies writing correct code, but comes with minor efficiency considerations. Finding the best representation for meta-programming is still an open problem.

Term representation is crucial, which became especially apparent for the example in Lean 4, where the need to fall back on primitive recursors prevented us from implementing a plugin with the same features as in the other systems (our implementation does not support recursive types such as lists). Term representation was also an issue in Agda, because the abstract syntax does not contain let-bindings and terms can only be beta-normal. Thus, a meta-programming framework must either expose a sufficiently expressive term representation, or a high-level API to build terms if the representation of kernel terms is too low-level.

Term quotations allow one to use user syntax directly when constructing and pattern matching on terms, thereby removing the need to spell out low-level details such as fully qualified constant names, implicit arguments, typeclass instances, or universe levels. Additionally, term quotations allow some amount of type checking at compile time (such as scope analysis) which allows one to catch errors earlier. Quasiquotations (i.e. the ability to nest anti-quotations and quotations) and the ability to quote open terms are especially useful, but are currently only supported in Lean and Elpi.

State is inherent to meta-programs, which can read and modify the global environment, local environment, and unification state. A good meta-programming framework must more generally provide good abstractions to deal with various kinds of effects, such as printing, exceptions, non-termination, and (prover) state. Lean and Agda handle printing using an `IO` monad and generally provide good libraries, while Rocq only provides ad hoc printing using the `reduction-effects` plugin [41] and does not have a satisfactory monad library (see § 5).

Verifiability is a desirable property of frameworks. Ideally formal verification of meta-programs should be possible. Verification is not so attractive for users of meta-programs because properties such as well-typedness can be checked a posteriori by the kernel, but implementors of meta-programs might be interested in (partial) correctness guarantees. Indeed, formal specifications can partly replace documentation – which is lacking for all considered frameworks anyway – and can help in writing correct meta-programs, which is far from an easy task considering the complexity of the underlying systems.

Current. *The learning curve* of a meta-programming framework is crucial, and writing meta-programs in a different language than that of the underlying proof assistant leads to a steeper curve. Learning Elpi was especially challenging due to the paradigm shift to logical programming.

Many meta-programming frameworks provide an *incomplete meta-programming API*, missing crucial features such as the ability to define new constants dynamically (Ltac2, Agda), bindings to high-level algorithms such as unification and type inference (MetaRocq), or support for e.g. mutual inductives (Elpi).

A proper meta-programming language requires adequate tooling, such as a language server, a documentation generator, an optimising compiler or efficient interpreter, and a good integration with the proof assistant's build system. Most of these tools come for free when the meta-language is the proof assistant itself or an already established programming language (such as OCaml), but require significant engineering work in the case of a DSL (e.g. Elpi or Ltac2).

Finally, we note that documentation is lacking for all considered frameworks, and some frameworks even lack basic learning resources.

Precise performance considerations are outside the scope of this paper, although we did comment on performance when relevant. We argue that meta-programming should prioritise usability over performance when possible. Performance is however important when considering tactic programming or more complex meta-programs such as unification algorithms and type checkers.

Summary Conceptually, two promising meta-programming approaches emerge: directly in the proof assistant or using a domain-specific language (DSL).

The first option, while of course relieving users from learning a new programming language, also provides crucial benefits to the quality of the tooling and libraries available for meta-programming. Moreover, this option allows users to verify their meta-programs. Verifying meta-programs requires both a specification of the basic meta-programming operations provided by the framework, and adequate means to use these basic specifications in order to derive guarantees about complex meta-programs. We note that one cannot realistically expect

to prove that the meta-programming framework fulfills its specification, as this would amount to proving the correctness of the entire elaborator and kernel of the underlying proof assistant. A more realistic approach is to interface with two implementations of the elaborator and kernel: a naive but verified implementation à la MetaRocq [77], and an efficient but unverified implementation.

The second option does not allow certifying meta-programs, but enables using domain-specific programming language features. Elpi is an example of such a DSL: logic programming is a valuable tool for working with syntax and binders.

In both cases, one needs a feature-complete meta-programming API, which stays up to date with the evolution of the proof assistant. For implementors of a meta-programming framework, *bootstrapping* the proof assistant gives a feature-complete API for free, but requires significant work a priori (for instance Lean 4's elaborator is bootstrapped). An alternate approach, which MetaRocq and Agda follow, is to do meta-programming directly in the proof assistant, without bootstrapping. Interfacing with the elaborator is done using a meta-programming monad, which from a user's point of view is very similar to bootstrapping.

12 Future Work

A natural direction for future work is to extend this study to other systems. On the side of dependently-typed programming languages, Idris 2 seems to come with a built-in plugin for deriving functor instances [3], which is however more general than our plugin, since it covers all types that can be proved functorial. In this paper, we focused on proof assistants rather than programming languages. Regarding proof assistants, it would be interesting to extend the study to systems from the HOL family, as well as to Beluga [70], Abella [37], or Dedukti [8]. We believe that insights will be largely orthogonal though, since both the underlying theory and the implementation methods differ vastly.

We also want to study *term-to-term* transformations, where we expect most of our results to carry over, although Dubois de Prisque remarks that HOAS can cause issues for these transformations [27]. Surveying tactic frameworks might also provide valuable insights. There are however far fewer tactic- than meta-programming frameworks: a survey would amount to comparing proof assistants.

An orthogonal direction is to extend one of the implementations into a standalone tool, and derive `Functor` instances for more general types, following the ideas of Laurent, Lennon-Bertrand, and Maillard [47].

A central direction for future work is to develop a meta-programming framework based on our insights, potentially in parallel for different proof assistants. We believe that a key insight is that meta-programs are inherently effectful programs, which have access both to generic effects such as failure and non-termination, and to domain-specific effects such as the ability to read and modify the global environment, local context and evar-map. We also argue that, considering the complexity of the underlying systems, verifying meta-programs is of great interest for implementors of meta-programs. These ideas hint at the possibility of using powerful verification techniques to ease reasoning about meta-

programs: following the line of work on Dijkstra monads [2, 52, 53] one can use specialised program logics tailored to domain-specific effects, and in particular separation logic to handle the evar-map, building on ideas from Nigron and Dagand [61] and Vistrup, Sammler, and Jung [88]. The line of work on algebraic effects [71, 28] and the Andromeda proof assistant [11] is also very relevant.

Bibliography

[1] Andreas Abel, Guillaume Allais, Aliya Hameer, Brigitte Pientka, Alberto Momigliano, Steven Schäfer, and Kathrin Stark. POPLMark Reloaded: Mechanizing proofs by logical relations. *J. Funct. Program.*, 29, 2019.

[2] Danel Ahman, Cătălin Hriţcu, Kenji Maillard, Guido Martínez, Gordon Plotkin, Jonathan Protzenko, Aseem Rastogi, and Nikhil Swamy. Dijkstra monads for free. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '17, page 515–529, New York, NY, USA, 2017. Association for Computing Machinery.

[3] Guillaume Allais and André Videla. Deriving for functor instances in idris 2 (idris 2 standard library).

[4] Abhishek Anand, Andrew W. Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Bélanger, Matthieu Sozeau, and Matthew Z. Weaver. Certicoq : A verified compiler for coq. 2016.

[5] Abhishek Anand, Simon Boulier, Cyril Cohen, Matthieu Sozeau, and Nicolas Tabareau. Towards certified meta-programming with typed template-coq. In *ITP 2018*, pages 20–39, 2018.

[6] Abhishek Anand, Anvay Grover, John Li, Greg Morrisett, Randy Pollack, Olivier Savary Belanger, Matthew Weaver, Andrew Appel, Yannick Forster, Joomy Korkut, Zoe Paraskevopoulou, Kathrin Stark, and Matthieu Sozeau. Certicoq: A verified compiler for coq (github repository). accessed Feb 18th 2025, 2025.

[7] Danil Annenkov, Jakob Botsch Nielsen, and Bas Spitters. Concert: a smart contract certification framework in coq. In Jasmin Blanchette and Catalin Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pages 215–228. ACM, 2020.

[8] 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 $\lambda\Pi$-calculus modulo theory. *CoRR*, abs/2311.07185, 2023.

[9] Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized metatheory for the masses: The POPLMark challenge. In Joe Hurd and Tom Melham, editors, *Theorem Proving in Higher Order Logics*, pages 50–65, Berlin & Heidelberg, 2005. Springer.

[10] Anne Baanen. A lean tactic for normalising ring expressions with exponents (short paper). In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning*, pages 21–27, Cham, 2020. Springer International Publishing.

[11] Andrej Bauer, Gaëtan Gilbert, Philipp G. Haselwarter, Matija Pretnar, and Christopher A. Stone. Design and Implementation of the Andromeda

Proof Assistant. In Silvia Ghilezan, Herman Geuvers, and Jelena Ivetic, editors, *22nd International Conference on Types for Proofs and Programs (TYPES 2016)*, volume 97 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:31, Dagstuhl, Germany, 2018. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[12] Edwin C. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Program.*, 23(5):552–593, 2013.

[13] Edwin C. Brady. Idris 2: Quantitative type theory in practice. In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference)*, volume 194 of *LIPIcs*, pages 9:1–9:26. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[14] Mario Carneiro. Lean4lean: Towards a verified typechecker for lean, in lean, 2024.

[15] Tej Chajed. Record updates in coq. In *CoqPL 2021: The Seventh International Workshop on Coq for Programming Languages*, 2021. Extended Abstract.

[16] Arthur Charguéraud. The locally nameless representation. *Journal of Automated Reasoning - JAR*, 49:1–46, 10 2012.

[17] Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. *SIGPLAN Not.*, 43(9):143–156, September 2008.

[18] Cyril Cohen, Enzo Crance, and Assia Mahboubi. Trocq: Proof transfer for free, with or without univalence. *CoRR*, abs/2310.14022, 2023.

[19] Cyril Cohen, Kazuhiko Sakaguchi, and Enrico Tassi. Hierarchy builder: Algebraic hierarchies made easy in coq with elpi (system description). In Zena M. Ariola, editor, *5th International Conference on Formal Structures for Computation and Deduction, FSCD 2020, June 29-July 6, 2020, Paris, France (Virtual Conference)*, volume 167 of *LIPIcs*, pages 34:1–34:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[20] Agda Community. agda-stdlib-classes.

[21] Thierry Coquand and Gérard P Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, 1988.

[22] Enzo Crance. *Méta-programmation pour le transfert de preuve en théorie des types dépendants*. Theses, Nantes Université, December 2023.

[23] Adrian Dapprich. Autosubst metacoq, 2021.

[24] Adrian Dapprich. Generating infrastructural code for terms with binders using metacoq, 2021. Bachelor's thesis, Saarland University.

[25] Arthur Azevedo de Amorim. Deriving instances with dependent types. In *Proceedings of the Sixth International Workshop on Coq for Programming Languages (CoqPL 2020)*, 2020.

[26] N.G de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, 1972.

[27] Louise Dubois de Prisque. *Prétraitement compositionnel en Coq. (Compositional preprocessing in Coq)*. PhD thesis, University of Paris-Saclay, France, 2024.

[28] Paulo Emílio de Vilhena and François Pottier. A separation logic for effect handlers. *Proc. ACM Program. Lang.*, 5(POPL), January 2021.

[29] Pablo Donato, Pierre-Yves Strub, and Benjamin Werner. A drag-and-drop proof tactic. In Andrei Popescu and Steve Zdancewic, editors, *CPP '22: 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, Philadelphia, PA, USA, January 17 - 18, 2022*, pages 197–209. ACM, 2022.

[30] Catherine Dubois, Nicolas Magaud, and Alain Giorgetti. Pragmatic isomorphism proofs between coq representations: Application to lambda-term families. In Delia Kesner and Pierre-Marie Pédrot, editors, *28th International Conference on Types for Proofs and Programs, TYPES 2022, June 20-25, 2022, LS2N, University of Nantes, France*, volume 269 of *LIPIcs*, pages 11:1–11:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[31] Cvetan Dunchev, Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. ELPI: fast, embeddable, λprolog interpreter. In Martin Davis, Ansgar Fehnker, Annabelle McIver, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*, volume 9450 of *Lecture Notes in Computer Science*, pages 460–468. Springer, 2015.

[32] Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. A metaprogramming framework for formal verification. *Proc. ACM Program. Lang.*, 1(ICFP), August 2017.

[33] Davide Fissore and Enrico Tassi. A new Type-Class solver for Coq in Elpi. In *The Coq Workshop 2023*, Bialystok, Poland, July 2023.

[34] João Paulo Pizani Flor, Wouter Swierstra, and Yorick Sijsling. Pi-ware: Hardware description and verification in agda. In Tarmo Uustalu, editor, *21st International Conference on Types for Proofs and Programs, TYPES 2015, May 18-21, 2015, Tallinn, Estonia*, volume 69 of *LIPIcs*, pages 9:1–9:27. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015.

[35] Yannick Forster, Matthieu Sozeau, and Nicolas Tabareau. Verified extraction from coq to ocaml. *Proc. ACM Program. Lang.*, 8(PLDI), June 2024.

[36] Yannick Forster and Kathrin Stark. Coq à la carte: a practical approach to modular syntax with binders. In Jasmin Blanchette and Catalin Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pages 186–200. ACM, 2020.

[37] Andrew Gacek. The abella interactive theorem prover (system description). In *Proceedings of the 4th International Joint Conference on Automated Reasoning (IJCAR)*, pages 154–161. Springer, 2008.

[38] 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 2023: Proceedings of the 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, pages 167–181, Boston MA USA, France, January 2023. ACM.

[39] Jason Gross, Théo Zimmermann, Miraya Poddar-Agrawal, and Adam Chlipala. Automatic test-case reduction in proof assistants: A case study in coq. In June Andronick and Leonardo de Moura, editors, *13th International Conference on Interactive Theorem Proving, ITP 2022, August 7-10, 2022, Haifa, Israel*, volume 237 of *LIPIcs*, pages 18:1–18:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[40] Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. Implementing Type Theory in Higher Order Constraint Logic Programming. *Mathematical Structures in Computer Science*, 29(8):1125–1150, March 2019.

[41] Hugo Herbelin. reduction-effects.

[42] Jason Z. S. Hu and Brigitte Pientka. A layered approach to intensional analysis in type theory. *ACM Trans. Program. Lang. Syst.*, 46(4):15:1–15:43, 2024.

[43] Jason Z. S. Hu and Brigitte Pientka. Layered modal type theory - where meta-programming meets intensional analysis. In Stephanie Weirich, editor, *Programming Languages and Systems - 33rd European Symposium on Programming, ESOP 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part I*, volume 14576 of *Lecture Notes in Computer Science*, pages 52–82. Springer, 2024.

[44] Jason Z. S. Hu and Brigitte Pientka. A dependent type theory for meta-programming with intensional analysis. *Proc. ACM Program. Lang.*, 9(POPL):416–445, 2025.

[45] Jason Z. S. Hu, Brigitte Pientka, and Ulrich Schöpp. A category theoretic view of contextual types: From simple types to dependent types. *ACM Trans. Comput. Log.*, 23(4):25:1–25:36, 2022.

[46] Junyoung Jang, Samuel Gélineau, Stefan Monnier, and Brigitte Pientka. Mœbius: metaprogramming using contextual types: the stage where system f can pattern match on itself. *Proc. ACM Program. Lang.*, 6(POPL):1–27, 2022.

[47] Théo Laurent, Meven Lennon-Bertrand, and Kenji Maillard. Definitional functoriality for dependent (Sub)Types. In *Lecture Notes in Computer Science*, pages 302–331. 2024.

[48] Bohdan Liesnikov, Marcel Ullrich, and Yannick Forster. Generating induction principles and subterm relations for inductive types using metacoq. *CoRR*, abs/2006.15135, 2020.

[49] Jannis Limperg. A novice-friendly induction tactic for lean. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2021, page 199–211, New York, NY, USA, 2021. Association for Computing Machinery.

[50] Jannis Limperg and Asta Halkjær From. Aesop: White-box best-first proof search for lean. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2023, page 253–266, New York, NY, USA, 2023. Association for Computing Machinery.

[51] Nicolas Magaud. Towards automatic transformations of coq proof scripts. In Pedro Quaresma and Zoltán Kovács, editors, *Proceedings 14th International Conference on Automated Deduction in Geometry, ADG 2023, Belgrade, Serbia, 20-22th September 2023*, volume 398 of *EPTCS*, pages 4–10, 2023.

[52] Kenji Maillard, Danel Ahman, Robert Atkey, Guido Martínez, Cătălin Hriţcu, Exequiel Rivas, and Éric Tanter. Dijkstra monads for all. *Proc. ACM Program. Lang.*, 3(ICFP), July 2019.

[53] Kenji Maillard, Cătălin Hriţcu, Exequiel Rivas, and Antoine Van Muylder. The next 700 relational program logics. *Proc. ACM Program. Lang.*, 4(POPL), December 2019.

[54] Matteo Manighetti, Dale Miller, and Alberto Momigliano. Two Applications of Logic Programming to Coq. In Ugo de'Liguoro, Stefano Berardi, and Thorsten Altenkirch, editors, *26th International Conference on Types for Proofs and Programs (TYPES 2020)*, volume 188 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:19, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[55] The mathlib Community. The lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, page 367–381, New York, NY, USA, 2020. Association for Computing Machinery.

[56] Conor McBride and James McKinna. Functional pearl: i am not a number–i am a free variable. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, Haskell '04, page 1–9, New York, NY, USA, 2004. Association for Computing Machinery.

[57] Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In *Lecture Notes in Computer Science*, pages 625–635. 2021.

[58] Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction – CADE 28*, pages 625–635, Cham, 2021. Springer International Publishing.

[59] Gopalan Nadathur and Dale Miller. An overview of lambda prolog. Technical report, USA, 1988.

[60] Wojciech Nawrocki, Edward W. Ayers, and Gabriel Ebner. An extensible user interface for lean 4. In Adam Naumowicz and René Thiemann, editors, *14th International Conference on Interactive Theorem Proving, ITP 2023, July 31 to August 4, 2023, Białystok, Poland*, volume 268 of *LIPIcs*, pages 24:1–24:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

[61] Pierre Nigron and Pierre-Évariste Dagand. Reaching for the Star: Tale of a Monad in Coq. In *Leibniz International Proceedings in Informatics (LIPIcs)*, volume 193 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 29:1–29:19, Rome, Italy, June 2021. Schloss Dagstuhl.

[62] Ulf Norell. agda-prelude: Programming library for agda.

[63] Ulf Norell. *Towards a practical programming language based on dependent type theory*, volume 32. Chalmers University of Technology, 2007.

[64] Ulf Norell, Nils Anders Danielsson, Jesper Cockx, and Andreas Abel. Agda wiki. `http://wiki.portal.chalmers.se/agda/pmwiki.php`.

[65] Zoe Paraskevopoulou, Aaron Eline, and Leonidas Lampropoulos. Computing correctly with inductive relations. In Ranjit Jhala and Isil Dillig, editors, *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, pages 966–980. ACM, 2022.

[66] Christine Paulin-Mohring. Inductive definitions in the system Coq rules and properties. In *International Conference on Typed Lambda Calculi and Applications*, pages 328–345. Springer, 1993.

[67] Arthur Paulino, D Testa, E Ayers, H Böving, J Limperg, S Gadgil, and S Bhat. Metaprogramming in lean 4. *Online Book. https://github.com/arthurpaulino/lean4-metaprogramming-book*, 2024.

[68] F. Pfenning and C. Elliott. Higher-order abstract syntax. *SIGPLAN Not.*, 23(7):199–208, June 1988.

[69] Brigitte Pientka. A type-theoretic framework for certified meta-programming (invited talk extended abstract). In Guillaume Allais and Yanhong Annie Liu, editors, *Proceedings of the 2025 ACM SIGPLAN International Workshop on Partial Evaluation and Program Manipulation, PEPM 2025, Denver, CO, USA, 21 January 2025*, pages 10–11. ACM, 2025.

[70] Brigitte Pientka and Joshua Dunfield. Beluga: A framework for programming and reasoning with contextual data. In *Proceedings of the 10th International Symposium on Functional and Logic Programming (FLOPS)*, pages 1–17. Springer, 2010.

[71] Gordon D. Plotkin and Matija Pretnar. Handling algebraic effects. *Log. Methods Comput. Sci.*, 9, 2013.

[72] Pierre-Marie Pédrot. Ltac2: Tactical warfare. In *The 5th International Workshop on Coq for Programming Languages (CoqPL 2019)*, 2019. Talk at CoqPL 2019, affiliated with POPL 2019.

[73] Talia Ringer. *Proof Repair*. PhD thesis, University of Washington, USA, 2021.

[74] Ayumu Saito and Reynald Affeldt. Towards a practical library for monadic equational reasoning in coq. In Ekaterina Komendantskaya, editor, *Mathematics of Program Construction*, pages 151–177, Cham, 2022. Springer International Publishing.

[75] Kazuhiko Sakaguchi. Reflexive tactics for algebra, revisited. In June Andronick and Leonardo de Moura, editors, *13th International Conference on Interactive Theorem Proving, ITP 2022, August 7-10, 2022, Haifa, Israel*, volume 237 of *LIPIcs*, pages 29:1–29:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[76] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The MetaCoq Project. *Journal of Automated Reasoning*, February 2020.

[77] Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. Coq coq correct! verification of type checking and erasure for coq, in coq. *Proc. ACM Program. Lang.*, 4(POPL), December 2019.

[78] Arnaud Spiwack. An abstract type for constructing tactics in Coq. In *Proof Search in Type Theory*, Edinburgh, United Kingdom, July 2010.

[79] Kathrin Stark, Steven Schäfer, and Jonas Kaiser. Autosubst 2: reasoning with multi-sorted de bruijn terms and vector substitutions. In Assia Mahboubi and Magnus O. Myreen, editors, *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019, Cascais, Portugal, January 14-15, 2019*, pages 166–180. ACM, 2019.

[80] Carst Tankink, Herman Geuvers, James McKinna, and Freek Wiedijk. Proviola: A tool for proof re-animation. *CoRR*, abs/1005.2672, 2010.

[81] Enrico Tassi. Deriving Proved Equality Tests in Coq-Elpi: Stronger Induction Principles for Containers in Coq. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving (ITP 2019)*, volume 141 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 29:1–29:18, Dagstuhl, Germany, 2019. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[82] The Coq Development Team. The coq proof assistant, September 2024.

[83] Sebastian Ullrich and Leonardo de Moura. Beyond notations: Hygienic macro expansion for theorem proving languages. In *Automated Reasoning: 10th International Joint Conference, IJCAR 2020, Paris, France, July 1–4, 2020, Proceedings, Part II*, page 167–182, Berlin, Heidelberg, 2020. Springer-Verlag.

[84] Sebastian Ullrich and Leonardo de Moura. 'do' unchained: embracing local imperativity in a purely functional language (functional pearl). *Proc. ACM Program. Lang.*, 6(ICFP), August 2022.

[85] Cas van der Rest and Wouter Swierstra. A completely unique account of enumeration. *Proc. ACM Program. Lang.*, 6(ICFP):411–437, 2022.

[86] Paul van der Walt and Wouter Swierstra. Engineering proof by reflection in agda. In *International Symposium on Implementation and Application of Functional Languages*, 2012.

[87] Marcell van Geest and Wouter Swierstra. Generic packet descriptions: verified parsing and pretty printing of low-level data. In Sam Lindley and Brent A. Yorgey, editors, *Proceedings of the 2nd ACM SIGPLAN International Workshop on Type-Driven Development, TyDe@ICFP 2017, Oxford, UK, September 3, 2017*, pages 30–40. ACM, 2017.

[88] Max Vistrup, Michael Sammler, and Ralf Jung. Program logics à la carte. *Proc. ACM Program. Lang.*, 9(POPL), January 2025.

A Rocq Plugin - de Bruijn code

```
let build_fmap env sigma ind : Evd.evar_map * EConstr.t =
  (* Construct the lambda abstractions. *)
  lambda env sigma "a" ta @@ fun env ->
  lambda env sigma "b" tb @@ fun env ->
  lambda env sigma "f" (arr (mkRel 2) (mkRel 1)) @@ fun env ->
  lambda env sigma "x" (apply_ind env ind @@ mkRel 3) @@ fun env ->
  let inp = { a = 4; b = 3; f = 2; x = 1 } in
  (* Construct the case return clause. *)
  let sigma, case_return =
    lambda env sigma "_" (apply_ind env ind @@ mkRel inp.a) @@ fun env ->
    (sigma, apply_ind env ind @@ mkRel (1 + inp.b))
  in
  (* Construct the case branches. *)
  let rec loop sigma acc ctrs_a ctrs_b =
    match (ctrs_a, ctrs_b) with
    | [], [] -> (sigma, Array.of_list @@ List.rev acc)
    | ca :: ctrs_a, cb :: ctrs_b ->
        let sigma, branch = build_branch env sigma inp ca cb in
        loop sigma (branch :: acc) ctrs_a ctrs_b
    | _ -> Log.error "build_fmap : different lengths"
  in
  let sigma, branches =
    loop sigma [] (constructors env ind @@ mkRel inp.a) (constructors env ind @@ mkRel inp.b)
  in
  (* Finally construct the case expression. *)
  ( sigma
  , Inductiveops.simple_make_case_or_project env sigma
      (Inductiveops.make_case_info env ind Constr.RegularStyle)
      (case_return, ERelevance.relevant)
      Constr.NoInvert (mkRel inp.x) branches )

let build_branch env sigma inp ca cb : Evd.evar_map * EConstr.t =
  (* Arguments are processed from outermost to innermost. *)
  let rec loop env sigma i acc decls =
    match decls with
    | [] -> (sigma, acc)
    | decl :: decls ->
        let env = Environ.push_rel decl env in
        let sigma, arg' =
          (* Call build_arg at a depth which is consistent with the local context
             of the environment, and lift the result to bring it at depth [n]. *)
          build_arg env sigma
            (lift_inputs (i + 1) inp)
            (mkRel 1)
            (Vars.lift 1 @@ EConstr.of_constr @@ get_type decl)
        in
        loop env sigma (i + 1) (lift (ca.cs_nargs - i - 1) arg' :: acc) decls
  in
  let sigma, args' =
    loop env sigma 0 [] (List.rev @@ EConstr.to_rel_context sigma ca.cs_args)
  in
  (* Apply the constructor to the arguments. *)
  let body = mkApp (mkConstructU cb.cs_cstr,
                    Array.of_list (mkRel (ca.cs_nargs + inp.b) :: List.rev args'))
  in
  (* Bind the constructor arguments. *)
  let branch =
    it_mkLambda body @@ List.map
        (function LocalAssum (b, ty) | LocalDef (b, _, ty) -> (b, ty))
        ca.cs_args
  in
  (sigma, branch)
```

Fig. 1: OCaml code for `build_fmap` and `build_branch`.

Most functions in OCaml plugins take as input the (global and local) environment `env` and the evar-map `sigma`, and return the updated evar-map. For instance `lambda env sigma "x" T k` builds a lambda abstraction with a binder named `x` and of type `T`; the continuation `k` takes in the new environment (updated with a binding for `x`) and returns the body of the lambda abstraction alongside the new updated evar-map. In the code above, the continuation is a lambda abstraction `fun env -> ...`, and the double at symbol `@@` stands for right-associative function application.

B Rocq Plugin - locally nameless code

```
let build_fmap env sigma ind : Evd.evar_map * EConstr.t =
  (* Abstract over the input variables. *)
  namLambda env sigma "a" ta @@ fun env sigma a ->
  namLambda env sigma "b" tb @@ fun env sigma b ->
  namLambda env sigma "f" (mkArrowR (mkVar a) (mkVar b)) @@ fun env sigma f ->
  namLambda env sigma "x" (apply_ind env ind @@ mkVar a) @@ fun env sigma x ->
  (* Construct the case return clause. *)
  let sigma, case_return =
    namLambda env sigma "_" (apply_ind env ind @@ mkVar a) @@ fun env sigma _ ->
    (sigma, apply_ind env ind @@ mkVar b)
  in
  (* Construct the case branches. *)
  let rec loop sigma acc ctrs_a ctrs_b =
    match (ctrs_a, ctrs_b) with
    | [], [] -> (sigma, Array.of_list @@ List.rev acc)
    | ca :: ctrs_a, cb :: ctrs_b ->
        let sigma, branch =
          build_branch env sigma { fmap; a; b; f; _x = x } ca cb
        in loop sigma (branch :: acc) ctrs_a ctrs_b
    | _ -> Log.error "build_map : different lengths"
  in
  let sigma, branches =
    loop sigma [] (constructors env @@ mkVar a)
      (constructors env @@ mkVar b)
  in
  (* Finally construct the case expression. *)
  ( sigma
  , Inductiveops.simple_make_case_or_project env sigma
      (Inductiveops.make_case_info env ind Constr.RegularStyle)
      (case_return, ERelevance.relevant)
      Constr.NoInvert (mkVar x) branches )

let build_branch env sigma inp ca cb : Evd.evar_map * EConstr.t =
  (* Bind the arguments of the constructor. *)
  namLambdaContext env sigma ca.cs_args @@ fun env sigma args ->
  (* Map the correct function over each argument. *)
  let arg_tys = List.map Declaration.get_type ca.cs_args in
  let rec loop sigma acc args arg_tys =
    match (args, arg_tys) with
    | [], [] -> (sigma, acc)
    | arg :: args, ty :: arg_tys ->
        let sigma, arg' = build_arg env sigma inp arg ty in
        loop sigma (arg' :: acc) args arg_tys
    | _ -> Log.error "build_branch : length mismatch"
  in
  let sigma, args' = loop sigma [] (List.map mkVar args) arg_tys in
  (* Apply the constructor to the arguments. *)
  (sigma, mkApp (mkConstructU cb.cs_cstr, Array.of_list (mkVar inp.b :: args')))
```

Fig. 2: OCaml code for `build_fmap` and `build_branch`, locally nameless version.

C MetaRocq code

```
Definition build_fmap ctx ind ind_body : term :=
  (* Abstract over the input parameters. *)
  mk_lambda ctx "A" (tSort @@ sType fresh_universe) @@ fun ctx =>
  mk_lambda ctx "B" (tSort @@ sType fresh_universe) @@ fun ctx =>
  mk_lambda ctx "f" (mk_arrow (tRel 1) (tRel 0))   @@ fun ctx =>
  mk_lambda ctx "x" (tApp (tInd ind []) [tRel 2])  @@ fun ctx =>
  let inp := {| fmap := 4 ; A := 3 ; B := 2 ; f := 1 ; x := 0 |} in
  (* Construct the case information. *)
  let ci := {| ci_ind := ind ; ci_npar := 1 ; ci_relevance := Relevant |} in
  (* Construct the case predicate. *)
  let pred :=
    {| puinst := []
    ; pparams := [tRel inp.(A)]
    ; pcontext := [{| binder_name := nNamed "x" ; binder_relevance := Relevant |}]
    ; preturn := tApp (tInd ind []) [tRel (inp.(B) + 1) ] |}
  in
  (* Construct the branches. *)
  let branches := mapi (build_branch ctx ind inp) ind_body.(ind_ctors) in
  tCase ci pred (tRel inp.(x)) branches.

Definition build_branch ctx ind inp ctor_idx ctor : branch term :=
  (* Get the context of the constructor. *)
  let bcontext := List.map decl_name ctor.(cstr_args) in
  let n := List.length bcontext in
  (* Get the types of the arguments of the constructor at type [A]. *)
  let arg_tys := cstr_args_at ctor (tInd ind []) [tRel inp.(A)] in
  (* Process the arguments one by one, starting from the outermost one. *)
  let loop := fix loop ctx i acc decls :=
    match decls with
    | [] => List.rev acc
    | d :: decls =>
      let ctx := d :: ctx in
      (* We call build_arg at a depth which is consistent with the local contex,
         and we lift the result to bring it at depth [n]. *)
      let mapped_arg := build_arg ctx (lift_inputs (i + 1) inp) (tRel 0) (lift0 1 d.(decl_type)) in
      loop ctx (i + 1) (lift0 (n - i - 1) mapped_arg :: acc) decls
    end
  in
  (* The mapped arguments are at depth [n]. *)
  let mapped_args := loop ctx 0 [] (List.rev arg_tys) in
  (* Apply the constuctor to the mapped arguments. *)
  let bbody := tApp (tConstruct ind ctor_idx []) @@ tRel (inp.(B) + n) :: mapped_args in
  (* Assemble the branch's context and body. *)
  mk_branch bcontext bbody.
```

Fig. 3: MetaRocq code for `build_fmap` and `build_branch`

The data-structures and low-level APIs exposed by MetaRocq are very similar to those used in OCaml: the MetaRocq plugin is thus very similar to the OCaml plugin using de Bruijn indices (Figure 1). The function `mk_lambda` corresponds to the OCaml function `lambda`, and the representation of terms is almost identical. The code manages only a local (de Bruijn) context `ctx`: MetaRocq does not have a notion of an evar-map.

D Agda code

```
build-fmap : Name -> Name -> TC (List Clause)
build-fmap ind func = do
  ind-def <- getDefinition ind
  ctors <-
    case ind-def of \
    { (data-type npars ctors) -> return ctors
    ; _ -> typeError ... }
  mapM (build-clause ind func) ctors

build-clause : Name -> Name -> Name -> TC Clause
build-clause ind func ctor = do
  -- Bind the input arguments.
  let inp = record { ind = ind ; func = func ; a = 4 ; A = 3 ; b = 2 ; B = 1 ; f = 0 }
      inp-tele =
        ("a" , hArg (quoteTerm Level)) ::
        ("A" , hArg (agda-sort @@ Sort.set @@ var 0 [])) ::
        ("b" , hArg (quoteTerm Level)) ::
        ("B" , hArg (agda-sort @@ Sort.set @@ var 0 [])) ::
        ("f" , vArg (pi (vArg @@ var 2 []) @@ abs "_" @@ var 1 [])) :: []
  inContext (List.reverse inp-tele) @@ do
    -- Fetch the type of the constructor at parameter [A].
    ctor-ty <- inferType (con ctor (hArg (var (Inputs.a inp) []) :: hArg (var (Inputs.A inp) []) :: []))
    -- Get the types of the arguments of the constructor.
    let (args-tele , _) = pi-telescope ctor-ty
        n-args = List.length args-tele
    inContext (List.reverse @@ inp-tele ++ args-tele) @@ do
      let inp = lift-inputs n-args inp
      -- Transform each argument as needed.
      mapped-args <-
        mapM (\(i , (_ , ty)) -> build-arg inp i @@ Arg.map (weaken (i + 1)) ty)
          (List.zip (downFrom n-args) args-tele)
      -- Build the clause.
      let args-patt =
            List.zipWith
              (\{ (_ , arg info _) i -> arg info (Pattern.var i) })
              args-tele (downFrom n-args)
          patt =
            hArg (Pattern.var @@ Inputs.a inp) ::
            hArg (Pattern.var @@ Inputs.A inp) ::
            hArg (Pattern.var @@ Inputs.b inp) ::
            hArg (Pattern.var @@ Inputs.B inp) ::
            vArg (Pattern.var @@ Inputs.f inp) ::
            vArg (Pattern.con ctor args-patt) :: []
          body = con ctor (hArg (var (Inputs.b inp) []) :: hArg (var (Inputs.B inp) []) :: mapped-args)
      Clause.clause (inp-tele ++ args-tele) patt body
```

Fig. 4: Agda code for `build-fmap` and `build-clause`.

Agda does not have a notion of local fixpoint: instead, constants such as `fmap` are defined via a collection of (possible recursive) equations, represented as a collection of clauses. Agda's meta-programming API supports *nested* pattern matching: clauses are built using `Clause.clause vars patt body`, where `vars` is the list of variables bound by the clause, `patt` is the left-hand side of the clause (or pattern) which may contain nested constructors, and `body` is the right-hand side of the clause.

E Lean code

```
def buildFmap ind : MetaM Expr := do
  -- Declare the input parameters.
  withLocalDecl `A .implicit (.sort ...)         fun A => do
  withLocalDecl `B .implicit (.sort ...)         fun B => do
  withLocalDecl `f .default  (← mkArrow A B)     fun f => do
  withLocalDecl `x .default  (← apply_ind ind A) fun x => do
    -- Construct the case return type.
    let ret_type := Expr.lam `_ (← apply_ind ind A) (← apply_ind ind B) .default
    -- Construct the case branches.
    let branches ← ind.ctors.toArray.mapM fun ctr => do
      let info ← getConstInfoCtor ctr
      buildBranch A B f info
    -- Construct the case expression.
    let cases_func ← freshConstant (← getConstInfo @@ .str ind.name "casesOn")
    let body := mkAppN cases_func @@ Array.append #[A, ret_type, x] branches
    -- Bind the input parameters.
    mkLambdaFVars #[A, B, f, x] body

def buildBranch A B f ctor : MetaM Expr := do
  -- Get the arguments of the constructor applied to A.
  let ctr_ty ← instantiateTypeLevelParams (ConstantInfo.ctorInfo ctor) [...]
  forallTelescope (← instantiateForall ctr_ty #[A]) fun args _ => do
    -- Map over each argument of the constructor.
    let mapped_args ← args.mapM (buildArg A B f)
    -- Apply the constructor to the new arguments.
    let freshCtor ← freshConstant @@ .ctorInfo ctor
    let body := mkAppN freshCtor @@ Array.append #[B] mapped_args
    -- Abstract with respect to the arguments.
    instantiateMVars =<< mkLambdaFVars args body
```

Fig. 5: Lean code for `buildFmap` and `buildBranch`.

Notice the return type `MetaM Expr` of `build_fmap` and the use of `do` notation: the code above runs in the `MetaM` monad, which provides implicit access to the global environment, local context, and metavariable context. In Lean, binders are represented using the locally nameless style: bound variables use de Bruijn indices, and free variables are named. The function `withLocalDecl` extends the local context with a new named variable, and `mkLambdaFVars` replaces free (named) variables with bound (de Bruijn) variables in a term. Back-ticks, as in `` `A ``, provide syntax sugar to construct names of global constants or local variables. Because Lean is a pure language, assigning a metavariable does not update the terms in which it occurs, thus one has to remember to use `instantiateMVars` to expand defined metavariables in terms. For simplicity we assume that all arguments of `option` and its constructors are explicit, and do not show how to handle universe polymorphism (we use ellipses in the code above).

F Ltac2 code

```
(* Expects a goal of the form [forall A B, (A -> B) -> F A -> F B]. *)
Ltac2 build_fmap F : unit :=
  (* intro *)
  intro @A ; intro @B ; intro @f ; intro @x ;
  (* destruct *)
  Std.case false (Control.hyp @x, NoBindings) ;
  (* Build each branch. *)
  let n_ctors := ... in
  Control.dispatch (List.init n_ctors (build_branch F @A @B @f)).

(* Expects a goal of the form [forall arg_1 ... arg_n, F B]. *)
Ltac2 build_branch F a b f i () : unit :=
  (* Introduce the arguments with fresh names. *)
  let n_args := ... in
  let args := n_intro n_args in
  (* Apply constructor [i]. *)
  constructor_n false i NoBindings ;
  (* Process each argument separately. *)
  Control.dispatch (List.map (build_arg a b f) args).
```

Fig. 6: Ltac2 code for `build_fmap` and `build_branch`.

The at symbol, as in @x, is used to construct Ltac2 identifiers. Chaining tactics is done using `Control.dispatch [t_1 ; ... ; t_n]`, which applies tactic `t_i` to the `i`-th open goal. Many functions take explicit flags, e.g. `Std.case false (Control.hyp @x, NoBindings)` simply performs case analysis on the local variable `x`, and `constructor_n false i NoBindings` applies the i-th constructor when the goal is of inductive type.
A quirk of the Ltac2 API is that introducing variables with fresh identifiers is slightly awkward; `n_intro` is a custom introduction tactic which handles fresh name generation.

G Elpi code

```
pred build-fmap i:inductive, o:term.
build-fmap I {{ fun (A B : Type) (f : A -> B) (x : lp:(FI A)) => lp:(M A B f x) }}
:-
  % Declare FI
  (pi x\ coq.mk-app { coq.env.global (indt I) } [x] (FI x)),
  % Bind the parameters.
  @pi-decl `A` {{ Type }} a\
  @pi-decl `B` {{ Type }} b\
  @pi-decl `f` {{ lp:a -> lp:b }} f\
  @pi-decl `x` (FI a) x\
  % Build the inner match.
  coq.build-match x (FI a) (_\_\_\r\ r = FI b)
    (build-branch I a b f) (M a b f x).

pred build-branch i:inductive, i:term, i:term, i:term, i:term,
  i:term, i:list term, i:list term, o:term.
build-branch I A B F CA _ Args ArgsTy Branch :-
  % Process each argument.
  std.map2 Args ArgsTy (build-arg I A B F) MappedArgs,
  % Change A with B in the constructor.
  (copy A B => copy CA CB),
  % Apply the constructor to the new arguments.
  coq.mk-app CB MappedArgs Branch.
```

Fig. 7: Elpi code for `build-fmap` and `build-branch`.

Elpi code makes heavy use of term quotations: `{{ ... }}` quotes Rocq code into Elpi, and `lp:(...)` quotes Elpi code into Rocq. Back-ticks, as in `` `x` ``, provide syntactic sugar to build identifiers for local variables. These variables are tracked in a local context: `@pi-decl `x` T k` adds a variable with name `` `x` `` and type `T` to the local context, and runs continuation `k` in this extended context. In the code above, continuations are in fact simply lambda abstractions $\lambda x.t$, written `x\ t` in Elpi.