Code Generation via Meta-programming in Dependently Typed Proof Assistants

Mathis Bouverot-Dupuis & Yannick Forster

April 2025

Meta-programming

Meta-programming

Meta-programs: programs which manipulate other programs as data.

Meta-programming

Meta-programs: programs which manipulate other programs as data.

In the context of proof assistants:

- Boilerplate generation: mechanically generate terms/inductives.
- Tactics.
- Macros.
- ...

Meta-programming

Meta-programs: programs which manipulate other programs as data.

In the context of proof assistants:

- Boilerplate generation: mechanically generate terms/inductives.
- Tactics.
- Macros.
- ...

Common boilerplate generation: inductive to term transformations, e.g. induction principles, equality deciders, printing functions, substitution functions.

Surveying meta-programming frameworks

Surveying meta-programming frameworks

Methodology:

- A tool to generate **Functor** instances for a simple class of inductives (including lists and trees).
- One implementation in each framework.

Surveying meta-programming frameworks

Methodology:

- A tool to generate **Functor** instances for a simple class of inductives (including lists and trees).
- One implementation in each framework.

Scope:

- Rocq: OCaml plugin, MetaRocq, Ltac2, Elpi
- Lean
- Agda

Surveying meta-programming frameworks

Methodology:

- A tool to generate **Functor** instances for a simple class of inductives (including lists and trees).
- One implementation in each framework.

Scope:

- Rocq: OCaml plugin, MetaRocq, Ltac2, Elpi
- Lean
- Agda

Goals:

- Assess the pros and cons of each framework.
- Focus on usability rather than performance.

Case study: Functor typeclass

Case study: Functor typeclass

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

Case study: Functor typeclass

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

```
Inductive tree A :=
| N : list (tree A) -> tree A.
| L : A -> tree A
```

```
Fixpoint fmap_tree {A B} f x :=
  match x with
  | N xs => N (List.map (fmap_tree f) xs)
  | L a => L (f a)
  end.
```

OCaml

OCaml plugins are historically the most common way to implement meta-programs.

OCaml

OCaml plugins are historically the most common way to implement meta-programs.

Rocq is implemented in OCaml: plugins extend Rocq's implementation (without modifying the kernel).

OCaml

OCaml plugins are historically the most common way to implement meta-programs.

Rocq is implemented in OCaml: plugins extend Rocq's implementation (without modifying the kernel).

Plugins are low level and must deal with the quirks of Rocq's implementation, e.g. de Bruijn indices.

Ocaml - 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 sigma, branches = ... in
  (* Finally construct the case expression. *)
  ( sigma, Inductiveops.simple_make_case_or_project env sigma ... )
```

OCaml - Pros and Cons

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

MetaRocq

MetaRocq

MetaRocq provides tools for meta-programming in Rocq.

MetaRocq

MetaRocq provides tools for meta-programming in Rocq.

MetaRocq includes a verified reimplementation of Rocq's kernel.

MetaRocq

MetaRocq provides tools for meta-programming in Rocq.

MetaRocq includes a verified reimplementation of Rocq's kernel.

The template monad gives access to Rocq's elaborator:

```
tmQuote : forall {A}, A  -> TemplateMonad term
```

```
tmMkInductive : mutual_inductive_entry -> TemplateMonad unit
```

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 return clause. *)
  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 ... pred (tRel inp.(x)) branches.
```

MetaRocq - Pros and Cons

	Conceptual	Current
Pros	- Users already know Rocq. - Meta-programs can be formally verified.	- Many functions are already formally verified.
Cons	- De Bruijn index arithmetic is difficult. - Lack of abstractions to handle effects.	- Explicit state management. - Missing high level meta-programming features. - Performance issues in some cases.

Ltac2

Ltac2

Ltac2 is a tactic language, which provides some facilities for meta-programming.

Ltac2

Ltac2 is a tactic language, which provides some facilities for meta-programming.

Still under active development.

Ltac2

Ltac2 is a tactic language, which provides some facilities for meta-programming.

Still under active development.

Tactics provide a nice API to build terms with computational content. For instance to define `fmap` on `option` types:

```
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.
```

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)).
```

Ltac2 - Pros and Cons

	Conceptual	Current
Pros	- Tactics provide a nice API to build terms.	- Implicit state management.
Cons	- Tactics are hard to reason about. - Implicit backtracking.	- Ltac2 is missing many basic language features. - Weak term manipulation API.

Elpi

Elpi is a logic programming language (derived from Lambda-Prolog), which provides facilities for meta-programming.

Elpi

Elpi is a logic programming language (derived from Lambda-Prolog), which provides facilities for meta-programming.

Programming is done using predicates rather than functions. For instance:

```
pred map i:list A, i:(pred i:A, o:B), o:list B.
map [] _ [].
map [X|XS] F [Y|YS] :- F X Y, map XS F YS.
```

Elpi

Elpi is a logic programming language (derived from Lambda-Prolog), which provides facilities for meta-programming.

Programming is done using predicates rather than functions. For instance:

```
pred map i:list A, i:(pred i:A, o:B), o:list B.
map [] _ [].
map [X|XS] F [Y|YS] :- F X Y, map XS F YS.
```

Rocq terms are represented using Higher-Order Abstract Syntax (HOAS).

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 case expression.
  coq.build-match x (FI a) (_\_\_\r\ r = FI b)
    (build-branch I a b f) (M a b f x).
```

Elpi - Pros and Cons

	Conceptual	Current
Pros	- Higher-order abstract syntax.	- Powerful quoting and unquoting mechanism.
Cons	- Paradigm shift (logic programming).	- Many trivial bugs are caught only at runtime. - Limited representations for structured data.

Lean

Lean

Lean 4's elaborator is bootstrapped (implemented in Lean). Includes type checking/inference, reduction, unification, type-class inference.

Lean

Lean 4's elaborator is bootstrapped (implemented in Lean). Includes type checking/inference, reduction, unification, type-class inference.

Meta-programs are Lean programs which use facilities from the elaborator.

Lean

Lean 4's elaborator is bootstrapped (implemented in Lean). Includes type checking/inference, reduction, unification, type-class inference.

Meta-programs are Lean programs which use facilities from the elaborator.

Meta-programs use a family of monads, most notably `MetaM`:

```
reduce (e : Expr) (explicitOnly skipTypes skipProofs := true) : MetaM Expr
```

```
isDefEq : Expr → Expr → MetaM Bool
```

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
```

Lean - Pros and Cons

	Conceptual	Current
Pros	- Users already know Lean. - Access to complete Lean implementation. - Locally-nameless API.	- Implicit state management with monads.
Cons	- No first-class fixpoints or pattern matching.	

Agda

Agda

Very similar to MetaRocq: meta-programming is done in Agda, using the *Reflection* API.

Agda

Very similar to MetaRocq: meta-programming is done in Agda, using the *Reflection* API.

The elaborator & kernel are accessed through the typechecking monad `TC`:

```
quoteTC : forall {A} → A → TC Term
```

```
inferType : Term → TC Type
```

Agda - Code

```
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
    -- Get the types of the constructor arguments.
    let (args-tele , n-args) = ...
    inContext (List.reverse @@ inp-tele ++ args-tele) @@ do
      let inp = lift-inputs n-args inp
      -- Transform each argument as needed.
      args' <- ...
     -- Build the clause.
      let body = con ctor (hArg (var (Inputs.b inp) []) :: hArg (var (Inputs.B inp) []) :
      Clause.clause (inp-tele ++ args-tele) ... body
```

Agda - Pros and Cons

	Conceptual	Current
Pros	- Users already know Agda.	- Implicit state management using monads.
Cons	- De Bruijn index arithmetic is difficult. - Term representation is difficult to manipulate.	- Type-class search is hard to control. - Performance issues in some cases.

Conceptual insights

Conceptual insights

Term representation (especially binders) is key:

- De Bruijn indices are difficult to use.
- Locally nameless and HOAS are better, but still have downsides.

Conceptual insights

Term representation (especially binders) is key:

- De Bruijn indices are difficult to use.
- Locally nameless and HOAS are better, but still have downsides.

Term quotations ease manipulating syntax. Quasi-quotations and quoting open terms are especially useful.

Conceptual insights

Term representation (especially binders) is key:

- De Bruijn indices are difficult to use.
- Locally nameless and HOAS are better, but still have downsides.

Term quotations ease manipulating syntax. Quasi-quotations and quoting open terms are especially useful.

State manipulation (local context, global environment, and unification state) is important: meta-programs are inherently stateful.

Conceptual insights

Term representation (especially binders) is key:

- De Bruijn indices are difficult to use.
- Locally nameless and HOAS are better, but still have downsides.

Term quotations ease manipulating syntax. Quasi-quotations and quoting open terms are especially useful.

State manipulation (local context, global environment, and unification state) is important: meta-programs are inherently stateful.

Verification of meta-programs is desirable: not for users (the output of meta-programs can be checked *a posteriori*) but for developers of meta-programs.

Practical insights

Practical insights

Learning curve is steeper when programming in a different language from the proof assistant. Learning Elpi was especially challenging.

Practical insights

Learning curve is steeper when programming in a different language from the proof assistant. Learning Elpi was especially challenging.

Tooling is important (compiler, language server, package manager):

- Comes for free when meta-programming in an established language.
- DSLs often have subpar tooling.

Practical insights

Learning curve is steeper when programming in a different language from the proof assistant. Learning Elpi was especially challenging.

Tooling is important (compiler, language server, package manager):

- Comes for free when meta-programming in an established language.
- DSLs often have subpar tooling.

Performance was not considered thoroughly. Choosing a good benchmark for meta-programming frameworks is not easy.

Questions