Code Generation via Meta-programming in Dependently Typed Proof Assistants

Mathis Bouverot-Dupuis & Yannick Forster

March 2025

Meta-programming

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

In the context of proof assistants:

- Boilerplate generation: mechanically generate terms/inductives.
- Tactics: build terms of a given type.
- Macros: elaborate raw syntax into abstract syntax.
- ...

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

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.

Example: Functor typeclass

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

```
Inductive option A :=
| Some : A -> option A
| None : option A.
```

```
Definition fmap_option {A B} f x :=
  match x with
  | Some a => Some (f a)
  | None => None
  end.
```

```
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.

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

Plugins are very low level and must deal with the quirks of Rocq's implementation (for instance de Bruijn indices).

Ocaml - Code (1/2)

```
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
      (Inductiveops.make_case_info env ind Constr.RegularStyle)
      (case_return, ERelevance.relevant)
      Constr.NoInvert (mkRel inp.x) branches )
```

OCaml - Code (2/2)

```
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 = function
    | [] -> (sigma, acc)
    | decl :: decls ->
        let env = Environ.push_rel decl env in
        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
  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
  (* Package everything together. *)
  let branch = ... in (sigma, branch)
```

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 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 (1/2)

```
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 - Code (2/2)

```
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
      loop ctx (i + 1) (lift0 (n - i - 1) mapped_arg :: acc) decls
    end
  in
  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.
```

MetaRocq - Pros and Cons

	Conceptual	Current
Pros	- Users already know Rocq. - Meta-programs can be formally verified.	- Significant parts are 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 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)).

(* 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).
```

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 meta-programming API.

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 (1/2)

```
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 - Code (2/2)

```
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.
```

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 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 (1/2)

```
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 - Code (2/2)

```
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
```

Lean - Pros and Cons

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

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 (1/2)

```
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
```

Agda - Code (2/2)

```
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. - Weak notation system and parser.

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, even for pure inductive to term transformations: 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

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.

Future work (my PhD)

Develop a meta-programming framework based on our insights, most likely by extending MetaRocq.

Concentrate on:

- Binder representation.
- Powerful term quotations.
- Effect handling.
- Verifiability.

Use dedicated program logics to verify effectful meta-programs, and in particular separation logic to handle the evar map.

Questions