Code Generation via Meta-programming in Dependently Typed Proof Assistants

Mathis Bouverot-Dupuis (ENS Paris, Inria Paris) & Yannick Forster (Inria Paris)

September 2024 - Februrary 2025

Online pre-print

https://hal.science/hal-05024207v3

Code Generation via Meta-programming in Dependently Typed Proof Assistants

Mathis Bouverot-Dupuis (1, 2) , Yannick Forster (2)

Show details

1 ENS-PSL - École normale supérieure - Paris
2 CAMBIUM - Langages de programmation : systèmes de types, concurrence, preuve de programme

Abstract en

Dependently typed proof assistants offer powerful meta-programming features, which users can take advantage of 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 type class. This example is fairly simple, but sufficiently realistic 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.

Domains

Computer Science [cs]

Complete list of metadata

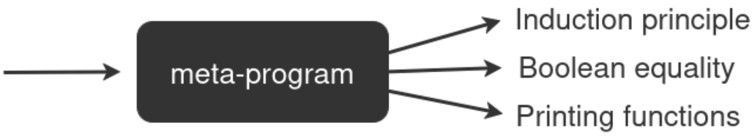
Meta-programming

Proof assistants provide powerful automation:

- Tactics: build proofs interactively.
- Macros: implement custom notations/EDSLs.
- Boilerplate generation: mechanically generate functions/lemmas.

Common archetype: generate terms (functions) based on an inductive.

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



Surveying meta-programming frameworks

Problem: no consensus on how to do meta-programming.

- Many different frameworks (Rocq has 4!)
- Users don't know the pros/cons of each framework.
- How do meta-programming frameworks compare?

Survey of meta-programming in Rocq (OCaml plugins, MetaRocq, Ltac2, Elpi), Agda, and Lean.

Case study: deriving instances for a Functor typeclass.

Our case study

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

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

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

One implementation in each meta-programming framework.

Pros and cons of each framework

OCaml - Pros

1. **Interface directly with Rocq's implementation.**
   - Define new commands/tactics.
   - Access to all elaborator features: unification, typeclass resolution, ...
   - And more: extend the parser, modify persistent state, ...

2. **OCaml is a battle-tested programming language.**
   - Many good libraries (including stdlib).
   - Performant.
   - High quality tools (LSP, package manager, ...)

3. **... and that's it.**

OCaml - Cons

1. Low-level API to build terms.

```
mkApp : constr -> constr array -> constr

mkCase : case_info * univ_instance * constr array * case_return *
  case_invert * constr * case_branch array -> constr
```

All low-level details have to be given:

- implicit arguments
- universe instances
- relevance of binders
- dependent pattern matching info

OCaml - Cons

2. De Bruijn indices.

```
mkRel : int -> constr
```

E.g. in the Rocq term

```
fun A B (f : A -> B) (x : A) => Some (f x)
```

The body `Some (f x)` is built as

```
mkApp tSome [| tRel 3 ; mkApp (mkRel 2) [| mkRel 1 |] |]
```

De Bruijn indices are very error prone.

MetaRocq - Pros

1. Meta-programs are Rocq programs.

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

Bindings to the kernel/elaborator are exposed through a monad:

```
tmEval : reductionStrategy -> term -> TemplateMonad term
tmMkDefinition : constant_entry -> TemplateMonad unit
```

MetaRocq - Pros

2. **Basic term quotations** i.e. a high-level API to build terms.

```
tmQuote {A} : A -> TemplateMonad term
tmUnquote : term -> TemplateMonad (A : Type, a : A)
```

3. **Formal verification** (in theory...).

```
typing : context → term → term → Type
red : context → term → term → Type
```

No specification for the **TemplateMonad** operations!

MetaRocq - Cons

1. **De Bruijn indices** (same as in OCaml).
2. **Term quotations are too basic.**
   - Can't quote terms with free variables.
   - No quasi-quotations (alternate *quote* and *unquote*).
3. **Programming in Rocq can be difficult.**
   - No input/output (e.g. printing).
   - Functions need to terminate.
   - Working with exceptions is difficult.
   - No good monad library.

Agda - Pros

Agda's Reflection API is very similar to MetaRocq.

1. **Meta-programs are Agda programs.**
   Bindings to the kernel/elaborator are exposed through a monad (similar to MetaRocq):

```
inferType : Term -> TC Term
defineFun : Name → List Clause → TC T
```

2. **Basic term quotations.**

```
quoteTC : forall {A} -> A -> TC Term
unquoteTC : forall {A} → Term → TC A
```

3. **Agda handles the prover state.**
   - The prover state is managed by the **TC** monad (contrary to MetaRocq).
   - Agda has a good monad library (exceptions, I/O, ...).

Agda - Cons

1. De Bruijn indices.

2. Term representation is constrained.
   - Terms are beta-normal by construction.
   - No let-bindings or local definitions.

Ltac2 - Pros

1. **Tactics provide a high-level API to build terms.**

```
Definition map_option : 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.
```

2. **Ltac2 manages the prover state**, which can be queried/modified imperatively:
   - global environment
   - local context
   - unification state

   No need to perform bookkeeping manually or work in a monad.

3. **Basic term quotations.**

Ltac2 - Cons

1. **Tactics are difficult to reason about.**
   Tactics work on an implicit goal. For instance the function:

```
Ltac2 build_map (I : inductive) : unit :=
  intros A B f x ; destruct x ...
```

   expects a goal of the form `forall A B, (A -> B) -> I A -> I B`.

2. **Weak low-level term manipulation API.**
   - De Bruijn indices.
   - Many standard functions are missing.

3. **Crucial meta-programming features are missing.**
   - Can't declare new constants or new commands.

Elpi - Pros

1. **Higher-order abstract syntax (HOAS).**
   Elpi does not use de Bruijn indices: binders instead re-use Elpi functions.

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

2. **Powerful term quotations.**
   Elpi has quotations `{{ ... }}` and anti-quotations `lp:(...)`, e.g.

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

3. **Elpi manages the prover state** (same as Ltac2).

Elpi - Cons

1. **Logic programming (paradigm shift).**
   Steep learning curve and standard tricks can be unintuitive.

2. **Unintuitive/missing language features.**
   - Implicit backtracking.
   - Type-checker is very permissive (e.g. no closed sums).
   - Lack of representation for structured data (e.g. records).

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

Lean - Pros

1. **Lean's elaborator is bootstrapped**
   - Meta-programs are simply Lean programs.
   - Meta-programs have access to the complete Lean implementation.

2. **Locally nameless binder representation.**
   - Bound variables use de Bruijn indices.
   - Free variables use names.

   For instance to build the term `fun A B (f : A -> B) (x : I A) => f x`

```
-- Declare the bound variables.
withLocalDecl `A _ (.sort _)              fun A => do
withLocalDecl `B _ (.sort _)              fun B => do
withLocalDecl `f _  (← mkArrow A B)       fun f => do
withLocalDecl `x _  (← apply_ind ind A) fun x => do
-- Bind the input parameters.
mkLambdaFVars #[A, B, f, x] (.app f x)
```

Lean - Pros

3. **Powerful term quotations.**

4. **Effect handling using monads.**
   - Excellent support for monads at the language level (notations, ...).
   - Meta-programs use monads, notably MetaM:

```
reduce : Expr -> MetaM Expr
isDefEq : Expr -> Expr -> MetaM Expr
```

Lean - Cons

1. **Building pattern matching/fixpoints is very difficult.**

    - Fixpoints are elaborated to recursors.
    - Recursors for nested inductives are very complex.
    - Our Lean implementation **does not support recursive inductives** e.g. lists.

Insights

Recurring issues

	OCaml	MetaRocq	Agda	Ltac2	Elpi	Lean
De Bruijn indices	✗	✗	✗	✗		
Restricted term AST			✗			✗
No quasi-quotations	✗	✗	✗	✗		
Meta-programming $\neq$ ITP language	✗			✗	✗	
Incomplete meta-programming API		✗	✗	✗	✗	
Explicit prover state handling	✗	✗				
Lack of learning resources	✗	✗		✗		
Lack of documentation	✗	✗	✗	✗	✗	✗
Can't verify meta-programs	✗	✗	✗	✗	✗	✗

Ideal meta-programming features

Choice of term AST is important, especially binder representation:

- (unscoped) de Bruijn indices are difficult to use.
- HOAS and locally nameless.

Term quotations (and anti-quotations).

Effect handling:

- generic effects (printings, non-termination, exceptions)
- domain-specific effects (manipulating the prover state)

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

Future work

Extend this study to other proof assistants/languages (Idris, HOL, Beluga, ...) or meta-programs.

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

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

Code is on github

MathisBD add license		db7c360 · now 61 Commits
Agda	cleanup	2 months ago
Elpi	cleanup	2 months ago
Lean	cleanup	2 months ago
Ltac2	cleanup	2 months ago
MetaRocq	cleanup	2 months ago
OCaml_de_Bruijn	cleanup	2 months ago
OCaml_locally_nameless	cleanup	2 months ago
BUILDING.md	cleanup	2 months ago
LICENSE	add license	now
README.md	Create README.md	now

https://github.com/MathisBD/metaprogramming-survey-code