



Small inversions with MetaCoq

Petites inversions en MetaCoq

Journées Francophones des Langages Applicatifs

Lieu-dit Saint Hilaire, Roiffé

29 Janvier 2025

Pierre Corbineau

Basile Gros

Jean-François Monin

Univ. Grenoble Alpes, CNRS, Grenoble INP¹, VERIMAG, 38000
Grenoble, France

¹Institute of Engineering Univ. Grenoble Alpes

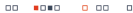


Inversion reasoning

Algebraic types

- Algebraic types: families of objects or proofs.
- Produced by constructor functions (introduction).
- Inspected by pattern-matching (elimination).
-

```
Inductive color : Set :=  
| cyan :      color  
| magenta :   color  
| yellow :    color  
| mix :       color → color → color .
```



Inversion reasoning

Algebraic types

- Algebraic types: families of objects or proofs.
- Produced by constructor functions (introduction).
- Inspected by pattern-matching (elimination).
- CIC types can be decorated by **indices** (dependent types, relations).

Inductive color : **bool** $\rightarrow$ **Set** :=

cyan :	color true
magenta :	color true
yellow :	color true
mix : $\forall$ b1 b2 , color b1 $\rightarrow$ color b2 $\rightarrow$	color false .





Inversion reasoning

Case analysis

Let us define `first : color false` $\rightarrow \{y:\text{bool} \ \& \ \text{color } y\}$
such that `first (mix b1 b2 c1 c2) = (b1; c1)`

```
Fail Definition first (c : color false) : {y:bool & color y} :=  
  match c return {b:bool & color b} with  
  | mix b1 b2 c1 c2 => (b1; c1)  
end.
```

The command has indeed failed with message:
Non exhaustive pattern-matching:
no clause found for pattern cyan





Inversion reasoning

Principle of inversion reasoning

A more accurate case analysis is needed: *inversion*.

- Inversion uses algebraic indices to:
 1. Partition the type depending on values of indices.
 2. Select the subset of relevant constructors possibly used to create a given object.
 3. Perform case analysis restricted on this subset.
- Mostly and frequently used for logical hypotheses, considered as dependent types in Coq TT.



Inversion reasoning

Changing the return type of the case analysis

- Track equality constraints on indices.

`inversion c`

C.Cornes, D.Terrasse, 1996

`match c in color b`

`return b = false → {y:bool & color y} with`

- Or return dummy values for unreachable branches
 - ▶ First versions of small inversions (JF. Monin 2010, 2013)
 - ▶ `Definition/refine (match c with ...)`
article to come out by P.Boutillier, H.Herbelin,
M.Lennon-Bertrand, T.Martinez

`match c in color b`

`return if b then IDProp else {y:bool & color y} with`



Inversion reasoning

Changing the type of the matched object

- Map relevant values to a tighter, more precise datatype, selected based on the construction of at least one **pilot index**

- Perform case analysis in this (proxy) datatype.
`destruct (color_inversion _ c)`
J-F. Monin, 2022

`match color_inversion false c return {y:bool & color y} with`

Our goal: automation of this approach





Inversion reasoning

Result with inversion

```
Definition first (c : color false) : {y:bool & color y} :=  
  let H :=  
    match c in (color b) return b = false → {y:bool, color y} with  
    | mix b1 _ x _ ⇒ fun _ : false = false ⇒ (b1; x)  
    | _ ⇒  
      fun H : true = false ⇒  
        let H0 : False :=  
          eq_ind true  
            (fun e : bool ⇒ if e then True else False)  
            I false H in  
          False_rec {y:bool & color y} H0  
        end in  
      H eq_refl
```





Inversion reasoning

Result with Definition/refine (match c with ...)

```
Definition first (c : color false) : {y:bool & color y} :=  
  match c in color b  
  |> return if b then IDProp else {y:bool & color y} with  
  | mix b1 _ c1 _ => (b1; c1)  
  | _ => idProp  
end
```





Inversion reasoning

Result with `destruct` (`color_inversion _ c`)

```
Definition first (c : color false) : {y:bool & color y} :=  
  match color_inversion false c return {y:bool & color y} with  
  | mix' b1 _ c1 _ => (b1; c1)  
end.
```

Advantages:

- Simple enough to do by hand.
- More readable, and writable, terms.
- Easily upgradable to dependent inversion.
 - ▶ Does not create equalities between dependent pairs, which happens with the `inversion` tactic.

But:

- We make no claims about structural decreasing.





Inversion reasoning

Auxiliary constructions for small inversions

Inductive `cmv` : `Set` :=

| `cyan'` : `cmv` | `magenta'` : `cmv` | `yellow'` : `cmv` .

Inductive `mixed` : `Set` :=

| `mix'` : $\forall b1\ b2\ (c1 : \text{color } b1)\ (c2 : \text{color } b2), \text{mixed}$.

Definition `color'` (`b:bool`) : `Set` :=

`fun b  $\Rightarrow$  if b`
`then cmv else mixed.`

Definition `color_inversion` (`b:bool`) (`c:color b`) : `color' b` :=

`match c with`
| `cyan` $\Rightarrow$ `cyan'`
| `magenta` $\Rightarrow$ `magenta'`
| `yellow` $\Rightarrow$ `yellow'`
| `mix b1 b2 c1 c2` $\Rightarrow$ `mix' b1 b2 c1 c2`
`end.`





Inversion reasoning

Auxiliary constructions for dependent small inversions

```
Inductive cmv : color true -> Set :=  
  | cyan' : cmv cyan | magenta' : cmv magenta | yellow' : cmv yellow.
```

```
Inductive mixed : color false -> Set :=  
  | mix' :  $\forall$  b1 b2 (c1 : color b1) (c2 : color b2), mixed (mix b1 b2 c1 c2).
```

```
Definition color' (b:bool) : color b -> Set :=  
  fun b => if b return (color b -> Set)  
    then cmv else mixed.
```

```
Definition color_inversiond (b:bool) (c:color b) : color' b c :=  
  match c with  
  | cyan => cyan'  
  | magenta => magenta'  
  | yellow => yellow'  
  | mix b1 b2 c1 c2 => mix' b1 b2 c1 c2  
  end.
```





Inversion reasoning

Motivation for dependent Inversion

Needed when the goal depends on the object inverted.

```
Lemma supair (c : color false) :  
  (let (b1, c1) := first c in  
   let (b2, c2) := second c in  
   mix b1 b2 c1 c2) = c.
```

Proof.

```
destruct (color_inversiond c).      cbn. reflexivity.
```

Qed.





MetaCoq

Presentation

- A Coq metaprogramming plugin.
- Created most notably to formalize Coq's kernel in Coq.
- Developed by "The Metacoq Development Team"².
- Allows obtaining, manipulating, redefining and reasoning on internal representation of Coq objects in Coq.
- First prototype for small inversions in MetaCoq developed by C. Thomazo under the supervision of J-F.Monin and P.Corbineau.

²A. Anand, D. Annenkov, S. Boulrier, C. Cohen, Y. Forster, J. Gross, M. Lennon-Bertrand, K. Maillard, G. Malecha, J. Botsch Nielsen, M. Sozeau, N. Tabareau and T. Winterhalter





MetaCoq

Example of MetaCoq function

```
Definition dup_constant{X}(const:X)(new_name : string) :=  
'(quoted_env, const_term) ← tmQuoteRec const ;;  
match const_term with  
| tConst const_kername _ ⇒  
    const_def <-o lookup_constant quoted_env const_kername ;;  
    const_body <-o cst_body const_def ;;  
    tmMkDefinition new_name const_body  
| _ ⇒ tmFail "The object is not a constant."  
end.
```

MetaCoq Run (dup_constant first "alt_first").





MetaCoq

Usage summary

In summary, small inversions are created with the following simple algorithm:

1. Extract data with `tmQuote`-like functions.
2. ??? (mess with De Bruijn indices)
3. Define the newly created objects in the Coq environment.
4. Profit! (prove it!)





MetaCoq

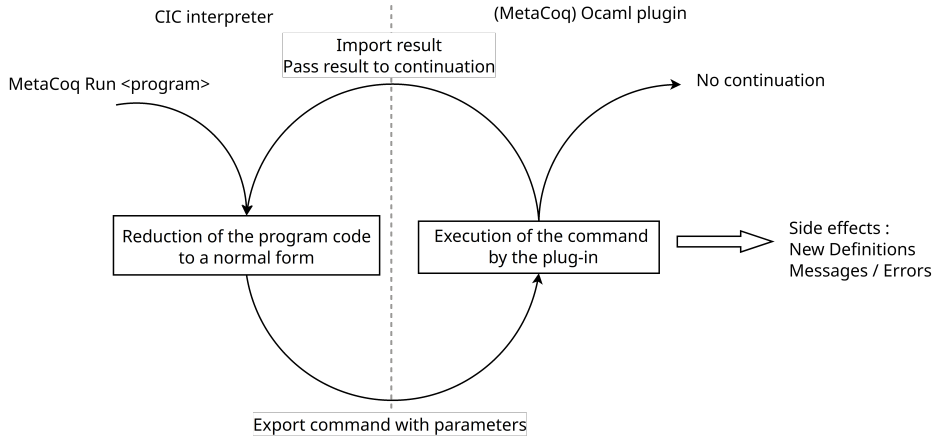
Lessons learned : advantages of MetaCoq

- Allows easy conversion from Coq terms to MetaCoq terms and vice-versa.
- We already know how to program in Gallina.
- Possibility of proving meta-programs.
- Good and fast community support on Zulip.



MetaCoq

Lessons learned: monad usage





MetaCoq

Lessons learned: monad usage

To improve performance, calls to the `TemplateMonad` should be limited or avoided.

- Use another monad like `option` wherever possible.
- Avoid binding recursive calls.
 - ▶ Do not use `tmBind` on a non tail-recursive call.
- Use `tmQuoteRec` in lieu of multiple API calls.
 - ▶ The standard method is to use `tmQuote` to get the anture and qualified name of the object, and then `tmQuoteInductive` or `tmQuoteConstant` to get its AST.



MetaCoq

Lessons learned: manipulating terms

Manipulating MetaCoq terms is complex because we manipulate terms after their elaboration.

- Bound variable are encoded as De Bruijn indices.

$$\lambda x y. x + y \implies \lambda x y. (1) + (0)$$

- They should be manipulated indirectly via **let in** local definitions.

Switch y for x :

$$\lambda x. \text{let } y = x \text{ in } x + y \implies \lambda x. x + x$$

$$\lambda x. \text{let } y = (0) \text{ in } (1) + (0) \implies \lambda x. (0) + (0)$$

- The internal representation of **match ... with** has complex scopes.





MetaCoq

Example of MetaCoq term : nat.add

```
tFix
[{|
  dname := name("add");
  dtype :=
    tProd name("n") N (tProd name("m") N N);
  dbody :=
    tLambda name("n") N (tLambda name("m") N
      (tCase cnat
        {|
          puinst := [];
          pparams := [];
          pcontext := [name("n")];
          preturn := N
        |} (tRel 1)
      [| bcontext := []; bbody := tRel 0 |}];
        {|
          bcontext := [name("p")];
          bbody := tApp tS [tApp (tRel 3) [tRel 0; tRel 1]]
        |})));
  rarg := 0
|}] 0
```

where

```
N = tInd {| inductive_mind := (MPfile ["Datatypes"; "Init"; "Coq"], "nat"); inductive_ind := 0 |}
cnat = {| ci_ind := tnat; ci_npar := 0; ci_relevance := Relevant |}
and name(x) = {| binder_name := nNamed x; binder_relevance := Relevant |}
```





Conclusion

Our work

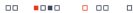
- Functional prototype of automatic generation of structures for proxy-based small inversions, non-dependent and dependent.
- Limited to simple calls.
 - Inversion on only one non-dependent index, and no nested patterns in the type to invert.
- We are exploring ways to generalize AST transformations to allow for complex cases.
- Also exploring ways to generically call our automation with a typeclass.





Questions?





What about ELPI ?

Main source : PhD thesis of Louise Dubois De Prisque.

Pros:

- Avoid using De Bruijn indices (uses Higher Order Abstract Syntax).
- Ease in traversing and manipulating ASTs.
- More readable code.

Cons:

- Great difficulty in writing terms from scratch.
- No integrated proof system of the transformations.
- Greater difficulty to translate into an Ocaml plug-in.

