Presenting MetaCoq: A Safe Tactic Language for Coq

Beta Ziliani

FAMAF, UNC and CONICET

In colaboration with
Yann Régis-Gianas and Jan-Oliver Kaiser

Contribs by Batrice Carré, Jacques-Pascal Deplaix, and Thomas Refis.

January 13, 2017
False quotes from Coq’s power users

A tactic must succeed no matter what
— Adam Chlipala

A tactic must fail reliably
— Georges Gonthier
False quotes from Coq’s power users

A tactic must succeed no matter what
— Adam Chlipala

A tactic must fail reliably
— Georges Gonthier
A tactic must fail reliably

1. During the definition.
   ▶ A typechecker should catch as many errors as possible.
   ▶ But without getting on our way.

2. During the execution.
   ▶ Proper error handling.
   ▶ Sensible (and formal) semantics.
Today: The Ltac language (example)

\[
\textbf{Definition } \quad x_{\text{in}_z y x} : \forall x \ y \ z : \text{nat}, \ x \in [z; y; x]. \\
\textbf{Proof.} \\
\text{intros.} \\
\text{apply in\_cons.} \\
\text{apply in\_cons.} \\
\text{apply in\_eq.} \\
\text{Qed.}
\]
Today: The Ltac language (example)

Definition \texttt{x\_in\_zyx} : \( \forall x \ y \ z : \text{nat}, \ x \in [z; y; x] \).
Proof.
  intros.
  apply in\_cons.
  apply in\_cons.
  apply in\_cons.
  apply in\_eq.
Qed.

OK for a beginner...
Today: The Ltac language (automated example)

```
Ltac solve_in := repeat (apply in_eq || apply in_cons).

Definition x_in_zyx : \forall x y z : nat, x \in [z; y; x].
Proof.
  intros; solve_in.
Qed.
```
Today: The Ltac language (automated example)

```
Ltac solve_in := repeat (apply in_eq || apply in_cons).

Definition x_in_zyx : ∀ x y z : nat, x ∈ [z; y; x].
Proof.
    intros; solve_in.
Qed.
```

Better, but can we abstract solve_in for different domains?
Presenting MetaCoq

Introduction

Today: no fun writing tactics in Ltac

Today: The Ltac language (automated example 2)

```
Ltac apply_one l :=
  list_fold_left (λ a b ⇒ (b || apply (elem a))) l fail.

Ltac solve_in := repeat (apply_one [Dyn in_eq; Dyn in_cons]).

Definition x_in_zyx : ∀ x y z : nat, x ∈ [z; y; x].
Proof.
  intros; solve_in.
Qed.
```
Today: The Ltac language (automated example 2)

\texttt{Ltac apply\_one \textit{l} :=}
\begin{verbatim}
list\_fold\_left \texttt{Ltac}(\lambda \ a \ b \Rightarrow \ (b \ || \ \text{apply} \ (\text{elem} \ a))) \textit{l} \texttt{fail}.
\end{verbatim}

\texttt{Ltac solve\_in :=} \texttt{repeat} \ (\texttt{apply\_one \texttt{[Dyn in\_eq; Dyn in\_cons]}}).

\textbf{Definition} \texttt{x\_in\_zyx} : \ \forall \ x \ y \ z : \texttt{nat}, \ x \in \ [z; y; x].
\textbf{Proof}.
\begin{verbatim}
intros; solve\_in.
\end{verbatim}
\texttt{Qed.}
Presenting MetaCoq

| Introduction
| Today: no fun writing tactics in Ltac

Today: The Ltac language (automated example 2)

```
Ltac apply_one l :=
    list_fold_left ltac:(λ a b ⇒ (b || apply (elem a))) fail l.

Ltac solve_in := repeat (apply_one [Dyn in_eq; Dyn in_cons]).
```

```
Definition x_in_zyx : ∀ x y z : nat, x ∈ [z; y; x].
Proof.
    intros; solve_in.
Qed.
```
Presenting MetaCoq

Introduction

Today: no fun writing tactics in Ltac

Today: The Ltac language (automated example 2)

Ltac apply

one :=

list fold left ltac:

(λ a b ⇒ (b ||| apply (elem a))) .

Ltac solve

in := repeat (apply one [Dyn in eq; Dyn in cons]).

Definition x in zyx :

∀ x y z : nat,

x ∈ [z; y; x].

Proof.

intros; solve in.

Qed.
Summary: Ltac

1. During the definition.
   - The typechecker does not catch many errors.

2. During the execution.
   - Improper error handling.
   - Insensible semantics.
The Mtac language

- Gallina is a pure dependently-typed language.
The Mtac language

- Gallina is a pure dependently-typed language.
- Can we add typed tactic programming to Gallina?
Presenting MetaCoq

The Mtac language

- Gallina is a **pure** dependently-typed language.
- Can we add **typed** tactic programming to Gallina?
- Use a monad!
The Mtac language

- Gallina is a **pure** dependently-typed language.
- Can we add **typed** tactic programming to Gallina?
- Use a monad!
  - Provide **meta-programming** primitives a Gallina type.
The Mtac language

- Gallina is a pure dependently-typed language.
- Can we add typed tactic programming to Gallina?
- Use a monad!
  - Provide meta-programming primitives a Gallina type.
  - Provide an interpreter to execute them.
The Mtac language

Definition solve_in \{A\} (x:A) : \forall l, M (x \in l) :=
  \text{mfix1} f (l : \text{list} A) : M (x \in l) :=
  \text{mmatch} l \text{ with}
  | [? l'] x :: l' ⇒ \text{ret} (\text{in_eq} _ _)
  | [? y l'] y :: l' ⇒ r ← f l';
  \quad \text{ret} (\text{in_cons} _ _ _ r)
  | _ ⇒ \text{failwith} "Not found"
end.

Lemma x_in_zyx : \forall x y z : \text{nat}, x \in [z; y; x].
Proof.
  intros; mrun (solve_in _ _).
Qed.
The Mtac language

Definition solve_in {A} (x:A) : ∀ l, M (x ∈ l) :=
  mfix1 f (l : list A) : M (x ∈ l) :=
  mmatch l with
  | [? l’] x :: l’ ⇒ ret (in_eq _ _)
  | [? y l’] y :: l’ ⇒ r ← f l’;
    ret (in_cons _ _ r)
  | _ ⇒ failwith "Not found"
end.

Lemma x_in_zyx : ∀ x y z:nat, x ∈ [z; y; x].
Proof.
  intros; mrun (solve_in _ _).
Qed.
The Mtac language

Definition solve_in \{A\} (x:A) : \forall l, M (x \in l) :=
  mfix1 f (l : list A) : M (x \in l) :=
    mmatch l with
    | [? l'] x :: l' \Rightarrow \text{ret (in_eq _ _)}
    | [? y l'] y :: l' \Rightarrow r \leftarrow f l';
      \text{ret (in_cons _ _ _ r)}
    | _ \Rightarrow \text{failwith ”Not found”}
  end.

Lemma x_in_zyx : \forall x y z:nat, x \in [z; y; x].
Proof.
  intros; mrun (solve_in _ _).
Qed.
The Mtac language

Definition solve_in \{A\} (x:A) : \forall l, M (x \in l) := 
mfix1 f (l : list A) : M (x \in l) := 
  mmatch l with 
  | [? l'] x :: l' \Rightarrow ret (in_eq _ _) 
  | [? y l'] y :: l' \Rightarrow r \leftarrow f l'; 
  | _ \Rightarrow failwith "Not found" 
end.

Lemma x_in_zyx : \forall x y z:nat, x \in [z; y; x].
Proof.
  intros; mrun (solve_in _ _).
Qed.
The Mtac language

**Definition** `solve_in` \( \{A\} (x:A) : \forall l, M (x \in l) := \)

`mfix1 f (l : list A) : M (x \in l) :=`

`mmatch l with`

\[| [? l'] x :: l' \Rightarrow \text{ret } (\text{in_eq } \_ \_ )|\]

\[| [? y l'] y :: l' \Rightarrow r \leftarrow f l'; \]

\[\quad \text{ret } (\text{in_cons } \_ \_ \_ r)\]

\[| \_ \Rightarrow \text{failwith } "\text{Not found}" \]

end.

**Lemma** `x_in_zyx` : \( \forall x y z : \text{nat}, x \in [z; y; x] \).

**Proof.**

`intros; mrun (solve_in _ _)`.  
**Qed.**
Problem with the Mtac language

Compare

**Ltac** `solve_in := repeat (apply in_eq || apply in_cons).` 

with

**Definition** `solve_in {A} (x:A) : ∀ l, M (x ∈ l) := mfix1 f (l : list A) : M (x ∈ l) :=` 

```
  mmatch / with
  | [? l'] x :: l' ⇒ ret (in_eq _ _)
  | [? y l'] y :: l' ⇒ r ← f l'; ret (in_cons _ _ r)
  | _ ⇒ failwith "Not found"
```

end.
Adding tactics to Mtac

- Add a type for tactics.

\[
\text{goal} \to \mathbb{M} \left(\text{list goal}\right)
\]

(But what is a goal?)

- Write basic tactics (intros, assumption, \ldots) in Mtac.
Adding tactics to Mtac: IMPOSSIBLE!

- Add a type for tactics.
  
  \[ \text{goal} \rightarrow M(\text{list goal}) \]

  (But what is a goal?)

- Write basic tactics (intros, assumption, \ldots) in Mtac.
  - Insufficient primitives!
  - Inconvenient semantics!
Mtac2: improving Mtac

- Several new primitives.
  - hypotheses, abs_prod, abs_let, abs_fix, unify, ...

- Revised semantics.
  - Backtracking of meta-context.

- `mmatch` in Gallina.
Builds on top of Mtac2.
• Adds a type for tactics and goals.
• Adds a proof environment MProof.
• Several basic tactics:
  ▶ intros, apply, assumption, reflexivity, generalize, clear, constructor, pose, assert, simpl, cbv, fix, repeat, . . .
• Several tactic combinators:
  ▶ &⟩, |1⟩, |l⟩
  ▶ Insert your combinator here.
MetaCoq (example)

Definition apply_one l : tactic :=
  fold_left (λ a b ⇒ a or (apply (elem b))) l (fail CantApply).

Definition solve_in := repeat (apply_one [Dyn in_eq; Dyn in_cons]).

Goal ∀ x y z : nat, x ∈ [z; y; x].
MProof.
  intros &⟩ solve_in.
Qed.
MetaCoq (example)

Definition apply_one l : tactic :=
    fold_left (λ a b ⇒ a or (apply (elem b))) l (fail CantApply).

Definition solve_in := repeat (apply_one [Dyn in_eq; Dyn in_cons]).

Goal ∀ x y z : nat, x ∈ [z; y; x].
MProof.
   intros &⟩ solve_in.
Qed.
MetaCoq (example)

Definition apply_one l : tactic :=
    fold_left (\ a b \to a or (apply (elem b))) l (fail CantApply).

Definition solve_in := repeat (apply_one [Dyn in_eq; Dyn in_cons]).

Goal \forall x y z : \text{nat}, x \in [z; y; x].
MProof.
    intros &\ solve_in.
Qed.
MetaCoq (example)

Definition apply_one l : tactic :=
  fold_left (\ a b \rightarrow a or (apply (elem b))) l (fail CantApply).

Definition solve_in := repeat (apply_one [Dyn in_eq; Dyn in_cons]).

Goal \forall x y z : nat, x \in [z; y; x].
MProof.
  intros & solve_in.
Qed.
MetaCoq (example)

Definition apply_one l : tactic :=
    fold_left (λ a b => a or (apply (elem b))) l (fail CantApply).

Definition solve_in := repeat (apply_one [Dyn in_eq; Dyn in_cons]).

Goal ∀ x y z : nat, x ∈ [z; y; x].
MProof.
    intros &.solve_in.
Qed.
MetaCoq (example)

Definition apply_one l : tactic :=
    fold_left (λ a b ⇒ a or (apply (elem b))) l (fail CantApply).

Definition solve_in := repeat (apply_one [Dyn in_eq; Dyn in_cons]).

Goal ∀ x y z : nat, x ∈ [z; y; x].
MProof.
    intros & solve_in.
Qed.
MetaCoq (example)

Definition apply_one l : tactic :=
  fold_left (λ a b⇒a or (apply (elem b))) l (fail CantApply).

Definition solve_in := repeat (apply_one [Dyn in_eq; Dyn in_cons]).

Goal ∀ x y z : nat, x ∈ [z; y; x].
MProof.
  intros & solve_in.
Qed.

Tactics that fail reliably with MetaCoq
Bonus track: Ever happened to you... that you couldn’t write the proof you like?
An example using Ssreflect

Definition add0 : ∀ n, n + 0 = n.
Proof.
  elim; first reflexivity.
  move⇒ n /= →; reflexivity.
Qed.
An example using Ssreflect

Definition add0 : ∀ n, n + 0 = n.
Proof.
  elim; first reflexivity.
  move⇒ n /= →; reflexivity.
Qed.
An example using Ssreflect

**Definition** \( \text{add0} : \forall \ n, \ n + 0 = n. \)

**Proof.**

- elim; first reflexivity.
- move\( \Rightarrow \ n \neq \rightarrow \); reflexivity.

**Qed.**
In MetaCoq

Definition add0 : ∀ n, n + 0 = n.
MProof.
  elim &⟩ case 0 do reflexivity.
  intros &⟩ simpl. select ( = ) rrewrite &⟩ reflexivity.
Qed.
In MetaCoq

Definition add0 : ∀ n, n + 0 = n.
MProof.
  elim &⟩ case 0 do reflexivity.
  intros &⟩ simpl. select (≈) rrewrite &⟩ reflexivity.
Qed.
In MetaCoq

Definition add0 : ∀ n, n + 0 = n.
MProof.
  elim &⟩ case 0 do reflexivity.
  intros &⟩ simpl. select (_ = _) rrewrite &⟩ reflexivity.
Qed.
Definition add0 : \( \forall \, n, \, n + 0 = n \).
MProof.
   elim \& case 0 do reflexivity.
   intros \& simpl. select (\_ = \_) rrewrite \& reflexivity.
Qed.

More understandable and robust proofs with MetaCoq
Presenting MetaCoq

Appendix

case in MetaCoq (2)

01 Definition get_constrs :=
02    mfix1 fill (T : Type) : M (list dyn) :=
03        mmatch T with
04          | [? A B] A → B ⇒ fill B
05          | _ ⇒ l ← constrs T; let (_, l') := l in ret l'
06        end.
07
08 Definition index {A} (c : A) :=
09      l ← get_constrs A;
10    (mfix2 f (i : nat) (l : list dyn) : M nat :=
11       mmatch l with
12          | [? l'] (Dyn c :: l') ⇒ ret i
13          | [? d' l'] (d' :: l') ⇒ f (S i) l'
14       end) 0 l.
“Type” error in Coq 8.6

In nested Ltac calls to "apply_one_of" and "list_fold_left", last call failed.
Error:
Must evaluate to a closed term offending expression:
l
this is a closure with body fail
in environment
Type error in MetaCoq

Toplevel input, characters 85-99:
Error:
In environment
l : ?T
The term "fail exception" has type "tactic" while it is expected to have type "list dyn".
Being honest

Current issues with MetaCoq:

- Performance.
Being honest

Current issues with MetaCoq:

- Performance.
- Performance.
Being honest

Current issues with MetaCoq:

- Performance.
- Performance.
- Seriously, performance.
Being honest

Current issues with MetaCoq:

▶ Performance.
▶ Performance.
▶ Seriously, performance.
▶ Some coercions unavoidable.
Current issues with MetaCoq:

- Performance.
- Performance.
- Seriously, performance.
- Some coercions unavoidable.
- Some issues with universes (so far avoidable).