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)

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

Today: The Ltac language (example)

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

Better, but can we abstract solve_in for different domains?

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)

Ltac apply_one l :=
  list_fold_left ltac:(λ a b $\Rightarrow$ (b || apply (elem a))) l fail.

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

Too… …e 2)

L…

L…).

D…
F…

C…

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?

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) : ∀ 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) : ∀ 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) : ∀ 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) : ∀ 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) : ∀ 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.
```

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 l 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.

  $$\text{goal} \to \text{M (list goal)}$$

  (But what is a goal?)

- Write basic tactics (intros, assumption, ...) in Mtac.

Adding tactics to Mtac: IMPOSSIBLE!

- Add a type for tactics.

  $$\text{goal} \to \text{M (list goal)}$$

  (But what is a goal?)

- Write basic tactics (intros, assumption, ...) 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.

MetaCoq

- 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:
  - $\&\rangle$, $|1\rangle$, $|l\rangle$
  - *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 $\forall$ 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.
```

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)

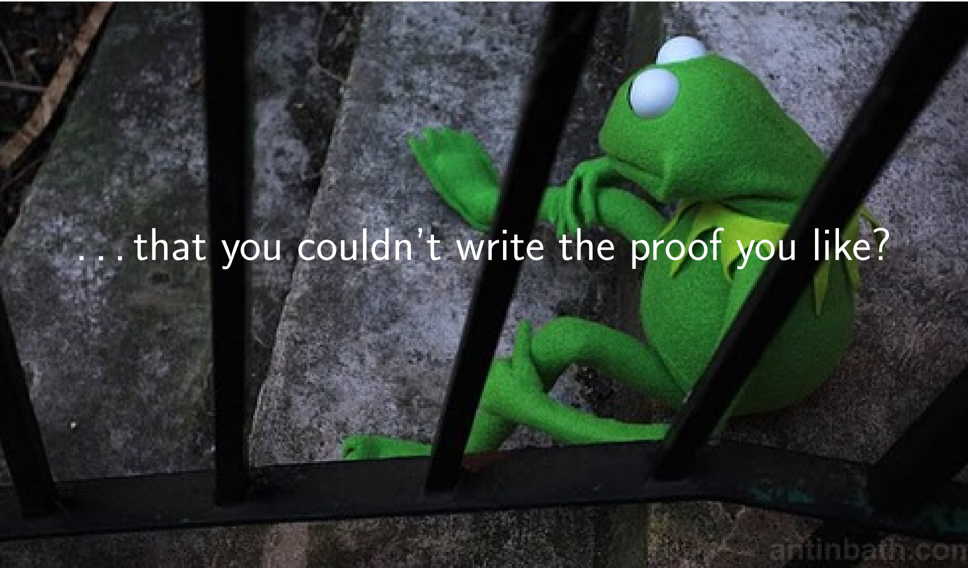
```
Definition apply_one l : tactic :=
  fold_left (λ a b⇒a or (apply (elem b))) l (fail CantApply).

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 : $\forall\ n,\ n + 0 = n$.
Proof.
  elim; first reflexivity.
  move$\Rightarrow$ n $/=$ $\rightarrow$; reflexivity.
Qed.

An example using Ssreflect

Definition add0 : $\forall$ n, $n + 0 = n$.
Proof.
  elim; first reflexivity.
  move$\Rightarrow$ n $/\!= \rightarrow$; reflexivity.
Qed.

An example using Ssreflect

Definition add0 : $\forall$ n, $n + 0 = n$.
Proof.
  elim; first reflexivity.
  move⇒ n /= $\rightarrow$; reflexivity.
Qed.

In MetaCoq

Definition add0 : $\forall$ 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.
```

In MetaCoq

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

Page intentionally left blank

case in MetaCoq (2)

```
Definition get_constrs :=
  mfix1 fill (T : Type) : M (list dyn) :=
    mmatch T with
    | [? A B] A → B ⇒ fill B
    | _ ⇒ l ← constrs T; let (_, l') := l in ret l'
    end.

Definition index {A} (c: A) :=
  l ← get_constrs A;
  (mfix2 f (i : nat) (l : list dyn) : M nat :=
    mmatch l with
    | [? l'] (Dyn c :: l') ⇒ ret i
    | [? d' l'] (d' :: l') ⇒ f (S i) l'
    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
 "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.

Being honest

Current issues with MetaCoq:

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