Meta-programming

Mathieu Sozeau  Yannick Forster  Théo Winterhalter
meta-program

a program manipulating programs
meta-program

a program manipulating programs

**Tactics**: analyse the goal and produce programs (proofs)

Ltac   ssreflect   Mtac2   Ltac2
meta-program
a program manipulating programs

Tactics: analyse the goal and produce programs (proofs)

Ltac ssreflect Mtac2 Ltac2

More tools: MetaCoq, Coq-Elpi, Ocaml plugins...

Also let you manipulate the global environment (definitions, inductive types...)
meta-program

a program manipulating programs

**Tactics**: analyse the goal and produce programs (proofs)

Ltac  ssreflect  Mtac2  Ltac2

**More tools**: MetaCoq, Coq-Elpi, Ocaml plugins...

Also let you manipulate the global environment (definitions, inductive types...)

meta-program

a program manipulating programs

Tactics: analyse the goal and produce programs (proofs)

Ltac ssreflect Mtac2 Ltac2

You used tactics but did you define any?

More tools: MetaCoq, Coq-Elpi, Ocaml plugins...

Also let you manipulate the global environment (definitions, inductive types...)
Tactics (Ltac)

You’ve seen a few:

intros, apply, exact, destruct, induction, split, rewrite, simpl, ...
Tactics (Ltac)

You’ve seen a few:

intros, apply, exact, destruct, induction, split, rewrite, simpl, ...

$idtac$ the tactic that does nothing??
Tactics (Ltac)

You’ve seen a few:

intros, apply, exact, destruct, induction, split, rewrite, simpl, ...

idtac the tactic that does nothing??

Maybe you saw tacticals (tactics taking tactics as arguments):

\[ t_0 ; t_1 \] not quite sequential composition

repeat t repeats the tactic until it stops making \textit{progress}

\textit{progress} t succeeds only if the tactic changes the goal
More advanced tactics

match goal, lazymatch goal let you manipulate the goal

match goal with
| |- ?A \?B => split
| ...
end
More advanced tactics

match goal, lazymatch goal let you manipulate the goal

match goal with
  |  |- $?A /\ $?B => split
  |  ...
end

clauses tried successively

if not using the lazy version, it will backtrack when a branch fails
More advanced tactics

match goal, lazymatch goal let you manipulate the goal

match goal with
|  | - ?A \ ?B => split
| ...
end

clauses tried successively
if not using the lazy version, it will backtrack when a branch fails

patterns matched syntactically
(not up to reduction, mostly...)

More advanced tactics

**match goal**, **lazymatch goal** let you manipulate the goal

```
match goal with
  |  |- ?A /\ ?B => split
  |  ...
end
```

- clauses tried successively
- if not using the lazy version, it will backtrack when a branch fails
- patterns matched *syntactically* (not up to reduction, mostly...)

**list of hypotheses** |- shape of the goal
(separated by commas)
More advanced tactics

match goal, lazymatch goal let you manipulate the goal

match goal with
| |– ?A /
| ?B => split
| ...
end

clauses tried successively
if not using the lazy version, it will backtrack when a branch fails

patterns matched syntactically (not up to reduction, mostly…)

list of hypotheses |– shape of the goal (separated by commas)

lazymatch goal is recommended unless you really want backtracking
(Type) classes

Another form of meta-programming

example: monoids

```
Class Monoid (A : Type) (neu : A) (op : A → A → A) := {
  m_left_neutral : ∀ x, op neu x = x ;
  m_right_neutral : ∀ x, op x neu = x ;
  m_assoc : ∀ x y z, op (op x y) z = op x (op y z)
}.

Instance MonoidNat : Monoid nat 0 plus. (requires a proof)
```

when using monoid laws on natural numbers, the instance is inferred automatically
Another form of meta-programming

example: monoids

Class Monoid (A : Type) (neu : A) (op : A → A → A) := {
  m_left_neutral : ∀ x, op neu x = x ;
  m_right_neutral : ∀ x, op x neu = x ;
  m_assoc : ∀ x y z, op (op x y) z = op x (op y z)
}.

Instance MonoidNat : Monoid nat 0 plus.  (requires a proof)

when using monoid laws on natural numbers, the instance is inferred automatically

also...  Instance MonoidList A : Monoid (list A) [] app.
Instance MonoidNat : Monoid nat.

Instance MonoidList : Monoid (list A).

Instance MonoidProd : Monoid A \rightarrow Monoid B \rightarrow Monoid (A \times B).

(simplified so it fits)
(Type) classes

Inference

**Instance** MonoidNat : Monoid nat.

**Instance** MonoidList : Monoid (list A).

**Instance** MonoidProd : Monoid A → Monoid B → Monoid (A * B).

(simplified so it fits)

**Definition** foo : Monoid (list bool * (nat * nat)) := _.
(Type) classes

Inference

Instance MonoidNat : Monoid nat.

Instance MonoidList : Monoid (list A).


(simplified so it fits)

Definition foo : Monoid (list bool * (nat * nat)) := _.

first looks up the instances of the monoid class that match
(Type) classes

Inference

Instance MonoidNat : Monoid nat.

Instance MonoidList : Monoid (list A).


(simplified so it fits)

Definition foo : Monoid (list bool * (nat * nat)) := _.

first looks up the instances of the monoid class that match
(Type) classes

Inference

Instance MonoidNat : Monoid nat.

Instance MonoidList : Monoid (list A).


(simplified so it fits)

Definition foo : Monoid (list bool * (nat * nat)) := _.

uses MonoidProd then looks recursively at
Monoid (list bool) and Monoid (nat * nat)
(Type) classes
From functional programming

Class `Eq A` := {
    eqb : A → A → bool ;
    eqb_iff : ∀ x y, eqb x y = true ↔ x = y
}.

enables polymorphism about types that have a decidable equality

Fixpoint `memb {A} `{Eq A} (x : A) (l : list A) : bool :=
    match l with
    | [] => false
    | y :: l => eqb x y || memb x l
end.
MetaCoq

Meta-programming with Coq
(and more...)

Inductive term :=
| tRel : nat → term
| tApp : term → list term → term
| ...

Representation of Coq terms in Coq
MetaCoq

Meta-programming with Coq
(and more...)

Coq terms

\((\lambda x. x) 0\)

quote

tApp (tLambda _ _ _) [tConstruct _ _ _]

unquote

Inductive term :=
| tRel : nat → term
| tApp : term → list term → term
| ...

Representation of Coq terms in Coq
MetaCoq

Meta-programming with Coq
(and more...)

Inductive term :=
| tRel : nat → term
| tApp : term → list term → term
| ...

Coq terms

\((\lambda x. x) \ 0\)

quote

unquote

tApp (tLambda _ _ _) [tConstruct _ _ _]

Representation of Coq terms in Coq

Representation of commands

tmDefinition "foo" 24
MetaCoq

Meta-programming with Coq
(and more...)

Coq terms

(\lambda x. x) 0 →
unquote

quote

Definition foo := 24

Representations of Coq terms in Coq

Inductive term :=
| tRel : nat → term
| tApp : term → list term → term
| ...

tApp (tLambda _ _ _) [tConstruct _ _ _]

Representation of commands

tmDefinition “foo” 24
Let’s practice now!

[link to Coq version]

if you don’t want unicode you can use this version instead

[link to ASCII version]

Also the slides:

[link to MPRI-2-7-2-2022 slides]