Introduction à Coq and math-comp

François Thiré

January 24, 2019

Organisation du cours

Organisation:

- 8 séances sur Coq + SSreflect
- 4 séances sur l'implémentation de CDCL (un squelette sera fourni en OCaml)

Évaluation:

- Coq: projet à rendre
- CDCL: projet à rendre

Coéfficient à déterminer.

Pourquoi faire de la preuve formelle

Pourquoi Coq ?

Deux *success stories*:

- CompCert (X. Leroy)
- Feit Thompson (G. Gonthier)

Point stage

- Nantes - Galinette
- Nice (Sophia) - Marelle
- Strasbourg (J. Narboux)

Venez m'en parler !

Coq en quelques mots

- Un langage de spécification : *Gallina*
- Un langage pour écrire ses preuves : *les tactiques*
- Un langage de commande pour donner des ordres à Coq : *Vernacular*

Coq en quelques mots

- Un langage de spécification : *Gallina*
- Un langage pour écrire ses preuves : *les tactiques*
- Un langage de commande pour donner des ordres à Coq : *Vernacular*

Coq en quelques mots

- Un langage de spécification : *Gallina*
- Un langage pour écrire ses preuves : *les tactiques*
- Un langage de commande pour donner des ordres à Coq : *Vernacular*

Coq utilise *l'isomorphisme de Curry-Howard*

(grande victoire de l'informatique !!!)

Coq is hard!

- Gallina
- Vernacular
- Tactics
- Goal
- Hypothesis
- Coercions
- Canonical Structures
- Modules

- Types
- Terms
- Unification
- Matching
- Implicit Parameters
- Proof by Reflection
- Propositions
- Meta-variables
- Inductions

SSreflect & math-comp

- Un autre langage de tactique
- S'interface bien avec math-comp (énorme bibliothèque de maths prouvée en Coq)
- Utilise un paradigme : la réflection! (on verra ce paradigme à l'oeuvre plus tard)

Pointers

- The mathematical component book (A. Mahboubi & E. Tassi)
- The SSreflect manual (G. Gonthier, A. Mahboubi, E. Tassi)
- (Advanced) X. Leroy (Collège de France lectures)

Introduction to natural numbers

First thing first

```
From mathcomp Require Import all_ssreflect.
Set Implicit Arguments.
Unset Strict Implicit.
Unset Printing Implicit Defensive.
```

Figure 1: Coq

Natural numbers (Peano numbers)

```
type nat = Z | S of nat
```

Figure 2: OCaml

```
Inductive nat : Set :=
| O : nat
| S : nat -> nat.
```

Figure 3: Coq

Recursive Functions (1/2)

```
let rec add n m =
  match n with
  | Z -> m
  | S n -> S (add n m)
```

Figure 4: OCaml

```
Fixpoint add (n m : nat) :
  match n with
  | 0 => m
  | S n => S (add n m)
  end.
```

Figure 5: Coq

Terminaison

```
Fixpoint loop (n : nat) : nat := loop n.
```

Figure 6: Coq

Terminaison

```
Fixpoint loop (n : nat) : nat := loop n.
```

Figure 6: Coq

```
Error:
Recursive definition of loop is ill-formed.
In environment
loop : nat -> nat
n : nat
Recursive call to loop has principal argument equal
 ↪  to "n" instead of
a subterm of "n".
Recursive definition is: "fun n : nat => loop n".
```

Figure 7: Error message

Recursive function (2/2)

```
Fixpoint add (n m : nat) : nat :=
  match n with
  | 0 => m
  | S n => S (add n m)
  end.
```

Figure 8: Coq

```
add is defined
add is recursively defined (decreasing on 1st argument)
```

Figure 9: Message

Our first proof

```
Lemma addn0 : forall n, plus n 0 = n.
Proof.
  by elim=> // n IH.
Qed.
```

Figure 10: Coq

Our first proof

```
Lemma addn0 : forall n, plus n 0 = n.
Proof.
  by elim=> // n IH.
Qed.
```

Figure 10: Coq

What The Hell?!?

Recap

- We use Coq as an interactive tool
- It is more restrictive than OCaml (don't upset him)
- Error messages are ruthless
- The tactic language is **hard**
- Proofs cannot be read, they have to be executed.

Let’s see Pai Mei

Basic rules 1

Well-defined[1] objects have a type.

```
Check 3.
Check (3 + 3).
Check true.
Check (2 + 2 = 5).
Check (2 + 2 = False).
Check add.
About add.
```

Well-typed propositions might not be provable.

When the object is a definition, an inductive or a recursive function, we can use the command `About` instead to have more information.

[1]The meaning of *well-typed* won't be defined here.

Basic rules 2

One can defined new definitions with the commands: `Definition`

```
Definition foo : nat := 4.
Definition bar : bool := true.
Definition foo_type := nat.
Definition bar2 : foobar := true.
Definition awesome_theorem := 2 + 2 = 4.
```

A type can be omitted while defining a new proposition.

One can define new types with the command: `Inductive`.

```
Inductive nat : Set :=
| O : nat
| S : nat -> nat.

Inductive bool : Set :=
| true  : bool
| false : bool.

Inductive seq (A : Set) : Type :=
| Nil  : seq A
| Cons : A -> seq A -> seq A.
```

One can define recursive functions with the command `Fixpoint`.

The function has to have a decreasing argument!

```
Fixpoint add (n m : nat) {struct n} : nat :=
  match n with
  | 0 => m
  | S n => S (add n m)
  end.
```

Coq is gentle enough to guess the decreasing argument for you.

One may use the command `Notation` to pretty-print objects.

```
Notation "n + m" := (add n m).
```

Standard definitions such as add,times, less than, . . . comes with a notation.

One may use the command Locate to search a notation.

```
Locate "+".
```

```
Notation
"{ A } + { B }" := sumbool A B : type_scope (default
↪ interpretation)
"A + { B }" := sumor A B : type_scope (default
↪ interpretation)
"m + n" := Nat.add m n : coq_nat_scope"m + n" :=
↪ addn_rec m n : nat_rec_scope
"m + n" := addn m n : nat_scope (default
↪ interpretation)"x + y" := sum x y : type_scope
```

Figure 11: Message

Parameters which are *guessable* by Coq can be omitted.

```
Inductive seq {A : Set} : Type :=
| Nil  : seq
| Cons : A -> seq -> seq.

Check (Cons 3 Nil).
```

Exercices

- What is the type of `2+2`?
- What is the type of `nat`?
- What is the type of `2+2=4`?
- Define the function `odd : nat -> bool`
- Define the function `even : nat -> bool`
- Define a notation `.+1` for the successor of a natural number
- Define the concatenation on `seq` with the notation `++`
- (With the manual) How to define in a mutual way `even` and `odd`?

Book exercices

At home, do the exercises of the (chapter 1) mathcomp Book!

Computation

Computation in Coq

One may ask Coq to compute using the command

`Eval compute in t.`

```
Eval compute in (2 + 2).
Eval compute in nat.
Eval compute in odd 3.
```

Computation plays a main role in proof assistants. It influences a lot the definitions and the proofs!

Proving stuff

Theorems, Lemmas, ...

A theorem is introduced by either the keyword `Theorem` or `Lemma`.
It does not matter, except for the reader.

```
Lemma foo : 2 + 2 = 4.
Theorem bar (x : nat) : x + x = 2 * x.
Theorem foobar : forall (x : nat), even x -> odd x.+1.
```

Proof

A proof is introduced by the command `Proof.` and finishes with the command `Qed.`.

Inside a proof, only some commands and tactics are allowed.

Our first tactic

A *trivial* goal can be solved with the tactic `by [].`

```
Goal 2 + 2 = 4. by [].
Goal 2 + 3 = 4. Fail by []. (* Not provable *)
Goal 3 < 4. by [].
Goal forall x, x < 3 -> x < 4. Fail by [].(* Not trivial *)
Goal forall x, x < 3 -> x.+1 < 4. by [].(* trivial *)
Goal forall x y, x <= y -> x.+1 <= y.+1. by [].
```

Learning what is trivial or not for Coq is a long journey.

IMHO, the genius behind math-comp is specially to find definitions that makes things trivial.

Proving equalities

An equality can be proven trivially if the two sides are equal modulo computation.

```
About eq.
Goal forall (x : nat), x = x. by [].
Goal 2 + 2 = 4. by [].
Goal forall (n m : nat), n + m = n + m. by [].
```

Reasoning by cases

One can make a reasoning by case with the tactic `case`.

```
Goal forall (b:bool), b || ~~b.
  (* two cases. either b is true, either b is false *)
  case.
  (* case 1: b is true, the goal is true || false *)
  by [].
  (* case 2: b is false, the goal is false || true *)
  by [].
```

Moving things around

One can introduce things into the goal or put them out into the context using the tactic `move`

```
Goal forall (b:bool), b || ~~b.
  move=> b.
  move: b.
  move=> b.
  case: b.
  by [].
  by [].
```

The real things happen thanks to =>, move basically does nothing.

Moving things around and destruct

`=>` can be combined with other tactics such as `case`.

```
Lemma leqn0 n : (n <= 0) = (n == 0).
Proof.
  case: n => [| k].
```

`[...|...|...]` is an intro pattern that allows you to select a goal and name things when a tactic creates several subgoals.

Exercises

Prove the following lemmas:

- `Lemma leqn0 n : (n <= 0) = (n == 0).`
- `Lemma negbK b : ~~ (~~ b) = b.`
- `Lemma addSn m n : m.+1 + n = (m + n).+1.`

You can use /= to simplify a goal and // to close trivial goals.
They can be combined as //=.

```
Goal  forall b, true || b = (b && false) || true.
    case => /=.
```

You can use the tactic `rewrite` to use a lemma where the statement looks like `forall a ... b, t = u` where `t` occurs in your goal.

```
Lemma muln_eq0 m n : (m * n == 0) = (m == 0) || (n ==
↪ 0).
Proof.
case: m => [|m] //.
case: n => [|k] //.
rewrite muln0.
```

Figure 12: Coq

The rewrite tactics accepts that lemmas can be chained without having to repeat the tactic `rewrite`

Modifiers can be used with the rewrite tactic:

- `!` to use the lemma as many times as possible (can loop)
- `!` to use the lemma if possible
- `-` to rewrite from right to left

```
Lemma leq_mul2l m n1 n2 : (m * n1 <= m * n2) = (m ==
↪  0) || (n1 - n2) == 0.
Proof.
  rewrite !leqE -mulnBr muln_eq0.
```

You can use the tactic `apply:` to use a lemma where the statement looks like `forall a ... b, t -> u` when your goal is `u`.

```
Lemma leqnn n : n <= n. Proof. Admitted.

Lemma example a b : a + b <= a + b.
Proof. by apply: leqnn. Qed.
```

Figure 13: Coq

Proof by induction are handled thanks to the tactic `elim`:

```
Lemma addn0 m : m + 0 = m.
Proof.
  elim: m => [ // |m IHm].
```

Figure 14: Coq

Exercise

Asumme the following lemmas:

- Lemma contraLR (c b : bool) : (~~ c -> ~~ b) -> (b -> c).
- Lemma dvdn_addr m d n : d %| m -> (d %| m + n) = (d %| n).
- Lemma dvdn_fact m n : 0 < m <= n -> m %| n`!.
- Lemma prime_gt0 p : prime p -> 0 < p.
- Lemma gtnNdvd n d : 0 < n -> n < d -> (d %| n) = false.
- Lemma prime_gt1 p : prime p -> 1 < p.

Prove

- Lemma example m p : prime p -> p %| m `! + 1 -> m < p.