

Lean for the Curious Mathematician 2024

SSreflect Tactics in the Rocq/Coq Proof Assistant

Marie Kerjean

CNRS, LIPN, Université Sorbonne Paris Nord

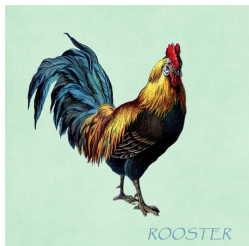
`https://github.com/CohenCyril/LFCM2024`

and then **Code** and **Codespace**.

The *tutorial.v* file will be used now, and the *practice.v* file after the break.

The Coq Proof Assistant

Soon to be renamed Rocq, and that is what I will use during this talk.



- ▶ Based on Dependent Type Theory, as Lean is.
- ▶ Known for both applications to certification (e.g. CompCert) and formalization in Maths (e.g. Math-Comp).
- ▶ Older than Lean: first release in 1984 (but it has changed considerably since that time).

Other Proof Assistants

There's a whole line of research leading to the development of proof assistants.

- ▶ ACL (1975 - ..): proofs of correctness of hardware
- ▶ LCF (1972): tactics
- ▶ Mizar (73-now): human readable proofs, library
- ▶ Automath (67): Formalisation of "grundlagen der analysis (76)"
- ▶ Isabelle/HOL (1986 - ...): Archive of Mathematical Proofs
- ▶ Agda, Rocq/Coq, Lean ..

Mathcomp and Ssreflect

- ▶ The Mathematical Components Library it was a *constructive* library for advanced algebra (The Odd Order Theorem), based on the small-scale reflection proof technique.
- ▶ While proofs in Rocq use Type Classes as Lean, Math-Comp uses a bundled approach. It traditionally used Canonical Structures, and now uses the Hierarchy Builder tool.
- ▶ Now it is an ecosystem of libraries:
 - ▶ MathComp-Analysis
 - ▶ Hierarchy-Builder
- ▶ The library is supported by the *ssreflect* tactic language which comes hand in hand with a formalization methodology.

Let's clear the air

Rocq is not intrinsically about constructive mathematics.

- ▶ The Mathematical Components was, because it could, and hence relied heavily on reflection between booleans and proposition.
- ▶ MathComp-Analysis is not:

```
Definition lim_in {U : Type} (T : filteredType U) :=  
  fun F : set_system U => get (fun l : T => F --> l).
```

Why customize your tactic language?

Small-scale reflection

- ▶ Reflection is a proof technique allowing to play between a proof oriented definition of an object, and a computation oriented definition of an object.
- ▶ This is used at large in Math-Comp, and in particular by making use of a small-scale reflection between `Prop` and `bool`.
- ▶ Small-scale reflection is facilitated by the `ssreflect` language.

Maintenance

- ▶ Rocq was 20 years, now 40
- ▶ Math-Comp has been maintained for 20 years with minimal effort
- ▶ By less than 10 people over 20 years.

Interesting research issue, not discussed here.

Ssreflect tactics

Punctuation

In Ssreflect

. ; ? ! [-] /(-) // =

all have a meaning and act on proofs. And that's were most of the fun is.

Show me your moves

The `move` tactic allows to move hypothesis back and forth from the top of your goal and the context.

► `move=> H`

Changes the goal from $H \rightarrow P$ to P , while putting H in the context.

► `move: H`

If H is an hypothesis, changes the goal from P to $H \rightarrow P$.

Also working:

► `move=> x Hx y P`
► `move=> x Hx + P`

`move=> [] .`
► `move=> [a b] .` destructs the hypotheses before introducing it
`move=> [a | b] .`

Easy proof, easy go

► `move=> //`.

Eliminates all goals that correspond to hypotheses in the context

► `move=> /`.

make computation run.

► `move=> //`.

do both.

► `by []`.

do all that and even more.

Rewrites

The Ssreflect language is extremely modular concerning rewrite.

- `rewrite H; rewrite -H`

Transforms a goal $P(a)$ into $P(b)$ with $H : a = b$ or $H : b = a$.

- `rewrite !H; rewrite ?H`

Rewrite H everywhere, or only where you can in the subgoals.

- `rewrite [X in A(X)]H`

Rewrites H , but only on subterms that could replace X in $A(X)$, where $A(X)$ is found in the goal.

- `rewrite -[A]/B`

Changes A into B as long as A can be computed into B .

- `rewrite /def; rewrite -/def` Folds and unfolds a definition

An apply a day

- ▶ `apply`: `H`.
uses `H : A -> B` to transform a goal `A` into a goal `B`.
- ▶ `apply/H`.
uses `H : A -> B` to transform a goal `A` into a goal `B` or `B` into `A`.
- ▶ `move=> /lem H`.
moves the hypothesis `H` from the top of the goal to the context, but first applies `lem` on it.
- ▶ `move=> /(_ a) H`.
moves the hypothesis `H` from the top of the goal to the context, but first feeds the argument `a` to it.

Forward reasoning

- ▶ Forward reasoning introduces intermediate statements with have:

```
have lem : H.  
  (* proof of my lemma *)  
  (* rest of the proof that needs to use H *)  
have -> : H
```

- ▶ Instead of naming lem, one can also describe how it is going to be used.

```
have -> : H by [].  
have /lemma2 [A B] : H by [].
```

Case

- ▶ `case=> H.`
`case: ab => [a | b].`

destructs an hypothesis while putting it in the context.

- ▶ `case: H.`

destructs inductive object (e.g. `bool`, `nat`) while taking it from the context.

And more:

- ▶ There some other tactics that have been introduced: `suff`, `wlog`, ...
But the idea is generally to keep a small number of tactics to make maintenance easier.
- ▶ Some tactics have been developed for Math-Comp Analysis allow neighborhoods reasoning in metric spaces (Affeldt, Cohen, Rouhling 2018), to avoid explicit ε handling.

- ▶ `forall x \nearrow y, P x`

A proposition stating that the property P holds in the filter of neighborhoods of y .

- ▶ `near=> x.`

Suppose that c is close enough to y , and continue with your proof.