SSReflect in Coq 8.10

New intro patterns and support for rewriting under binders

Érik Martin-Dorel[1] Enrico Tassi[2]

[1]IRIT, Université Toulouse 3, France

[2]Inria, Université Côte d'Azur, France

September 8th, 2019
The 10th Coq Workshop
Portland State University, OR, USA

Outline

SSReflect in a nutshell

SSR is a *proof language* (a bit more than a list of tactics)

- Way past break-in period: 4C Thm, Odd Order Thm, . . .
- Backward compatible (e.g. MathComp 1.9 works on Coq 8.7 → 8.10)
- Integrated in Coq since version 8.7 (`Require Import ssreflect.`)
- Enables SSR formalization style, but does not force it

Small Scale Reflection formalization style

The name: reflecting decideable propositions to bool. . . But it is more than that, too much for one slide.

Focus: easy to repair scripts = scripts that break early and locally

- basic bricks are dumb, predictable and do fail
- explicit naming of context items (bookkeeping discipline)

Example:

- `rewrite [in RHS]leq_ab` vs. `rewrite {35}H16`

In this talk we focus on intro patterns and rewriting

Intro patterns by examples: working the goal stack

```
Lemma test : ∀a b, a <= b -> G. Proof. move=> a ? leq_ab.

 a, _b_ : nat
 leq_ab : a <= _b_
===================
 G

Lemma test : ∀a b, a <= b -> G. Proof. move=> a [|b] leq_ab.

 a : nat                 a, b : nat
 leq_ab : a <= 0         leq_ab : a <= b.+1
=================      ====================
 G                       G

Lemma test : ∀a b, a <= b -> G. Proof. move=> a b /leqW; move: a b.

======================
 ∀a b, a <= b.+1 -> G
```

Outline

Block introduction: case

Destructuring an inductive type using standard names.

```
Inductive i :=
  | K1 (a : T)
  | K2 (_ : U) (b : T). (* these names are kept by Coq *)

Lemma test (x : i) : G.
Proof.
case: x => [^ y_ ].

                _y_?_ : U
 y_a : T        y_b : T
==========   ============
 G              G
```

Names are predictable (derived by simple concatenation) and unique (you choose a prefix/suffix that must not generate clashes).

Block introduction: elim

Destructuring also happens as a result of an induction.

```
Lemma my_ind P :
  P 0 ->
  (∀a (IHa : P a), P a.+1) ->
    ∀x, P x.
Proof.
...
Qed.
```

you can always put a name on a product

```
Lemma test (n : nat) : G n.
Proof.
elim/my_ind: n => [^~ 1 ].
```

```
             a1 : nat
             IHa1 : G a1
==========   =============
 G 0         G a1.+1
```

Fast and temporary introduction

Skip to the fist assumption with the > intro pattern

```
Lemma test : ∀a b, a <= b -> G.
Proof.
move=> >leq_ab

 _a_, _b_ : nat
 leq_ab : _a_ <= _b_
=====================
 G
```

Introduce now and revert at the end of the intro pattern

```
Lemma test: ∀a b, a <= b -> G.
move=> + + /leqW.

======================
 ∀a b, a <= b.+1 -> G
```

Ltac views

When the developer replies DIY...

```
Notation "'dupP'" := ltac:(code to duplicate an hypothesis) : ssripat_scope.

Lemma test x : x = 3 -> G x.
move=> /dupP def_x ->.

 x : nat
 def_x : x = 3
================
 G 3
```

Bonus: dupP could take arguments!

Outline

Big operators in a nutshell

- Formalization of $\sum_{\substack{i \in A \\ P(i)}} F(i), \prod_{\substack{i \in A \\ P(i)}} F(i), \bigcap_{\substack{i \in A \\ P(i)}} F(i), \bigcup_{\substack{i \in A \\ P(i)}} F(i), \max_{\substack{i \in A \\ P(i)}} F(i) \ldots$
- Implem: higher-order iterator applied to some lambda for P and F

Big operators in a nutshell

- Formalization of $\sum_{\substack{i \in A \\ P(i)}} F(i), \prod_{\substack{i \in A \\ P(i)}} F(i), \bigcap_{\substack{i \in A \\ P(i)}} F(i), \bigcup_{\substack{i \in A \\ P(i)}} F(i), \max_{\substack{i \in A \\ P(i)}} F(i) \ldots$
- Implem: higher-order iterator applied to some lambda for P and F

Example

$\sum_{\substack{i=1 \\ i \text{ odd}}}^{4} i^2$ can be formally written as: `\sum_(1 <= i < 5 | odd i) i^2`,

that is to say: `\big[addn/0]_(1 <= i < 5 | odd i) i^2`,

which expands to: `bigop _ _ 0 (index_iota 1 5) (fun i : nat => BigBody _ _ i addn (odd i) (i ^ 2))`

Higher-order iterators? Need for rewriting under binders. . .

From mathcomp Require Import bigop. ⇝ provides congruence lemmas to be applied by hand

```
eq_big : (* main congruence lemma for bigops *)
  ∀ (R : Type) (idx : R) (op : R -> R -> R) (I : Type) (r : seq I),
  ∀ (P1 P2 : pred I) (F1 F2 : I -> R),
  (∀ i : I, P1 i = P2 i) -> (∀ i : I, P1 i -> F1 i = F2 i) ->
  \big[op/idx]_(i <- r | P1 i) F1 i = \big[op/idx]_(i <- r | P2 i) F2 i.
```

Higher-order iterators? Need for rewriting under binders. . .

From mathcomp **Require Import** bigop. ⇝ provides congruence lemmas to be applied by hand

```
eq_big : (* main congruence lemma for bigops *)
  ∀(R : Type) (idx : R) (op : R -> R -> R) (I : Type) (r : seq I),
  ∀(P1 P2 : pred I) (F1 F2 : I -> R),
  (∀i : I, P1 i = P2 i) -> (∀i : I, P1 i -> F1 i = F2 i) ->
  \big[op/idx]_(i <- r | P1 i) F1 i = \big[op/idx]_(i <- r | P2 i) F2 i.
```

Running example

```
  n : nat
  ============================================
  \sum_(0 <= k < n | odd k && (k != 1)) (k - k) = 0

rewrite subnn.  (* Error:  The LHS of subnn, (_ - _), does not match any subterm of the goal *)
rewrite eq_big. (* Error:  Unable to find an instance for the variables P2, F2.  *)
```

Higher-order iterators? Need for rewriting under binders. . .

`From mathcomp Require Import bigop.` ⇝ provides congruence lemmas to be applied by hand

```
eq_big : (* main congruence lemma for bigops *)
  ∀(R : Type) (idx : R) (op : R -> R -> R) (I : Type) (r : seq I),
  ∀(P1 P2 : pred I) (F1 F2 : I -> R),
  (∀i : I, P1 i = P2 i) -> (∀i : I, P1 i -> F1 i = F2 i) ->
  \big[op/idx]_(i <- r | P1 i) F1 i = \big[op/idx]_(i <- r | P2 i) F2 i.
```

Running example

```
  n : nat
  ============================================
  \sum_(0 <= k < n | odd k && (k != 1)) (k - k) = 0

rewrite subnn.  (* Error:  The LHS of subnn, (_ - _), does not match any subterm of the goal *)
rewrite eq_big. (* Error:  Unable to find an instance for the variables P2, F2.  *)
```

We need to provide P2 and F2 by hand (the lambda terms we want to obtain *after* the rewrite):

```
rewrite (eq_big (fun k => odd k && (k != 1)) (fun k => 0));
  [ | done | by move=> ? _; rewrite subnn].
```

The under tactic - I

One-liner (a.k.a. batch) mode

```
  n : nat
  ==================================================
  \sum_(0 <= k < n | odd k && (k != 1)) (k - k) = 0

under eq_big do [ | rewrite subnn].
```

The under tactic - I

One-liner (a.k.a. batch) mode

```
  n : nat
  ==================================================
  \sum_(0 <= k < n | odd k && (k != 1)) (k - k) = 0

under eq_big do [ | rewrite subnn].

  n : nat
  ============================================
  \sum_(0 <= i < n | odd i && (i != 1)) 0 = 0
```

The under tactic - II

Interactive mode (without do clause)

```
n : nat
===============================================
\sum_(0 <= k < n | odd k && (k != 1)) (k - k) = 0

under eq_big => [i | i /andP[i_odd i_neq1]].
```

The under tactic - II

Interactive mode (without do clause)

```
  n : nat
  ==================================================
  \sum_(0 <= k < n | odd k && (k != 1)) (k - k) = 0

under eq_big => [i | i /andP[i_odd i_neq1]].

                            n, i : nat
                            i_odd : odd i
n, i : nat                  i_neq1 : i != 1 n : nat
=========================== =============== ====================================
'Under[ odd i && (i != 1) ] 'Under[ i - i ] \sum_(0 <= i < n | ?P2 i) ?F2 i = 0
```

The under tactic - II

Interactive mode (without do clause)

```
  n : nat
  ==================================================
  \sum_(0 <= k < n | odd k && (k != 1)) (k - k) = 0

under eq_big => [i | i /andP[i_odd i_neq1]].

                              n, i : nat
                              i_odd : odd i
n, i : nat                    i_neq1 : i != 1 n : nat
============================ =============== ====================================
'Under[ odd i && (i != 1) ] 'Under[ i - i ] \sum_(0 <= i < n | ?P2 i) ?F2 i = 0
↓                             ↓
over.                         rewrite subnn.
                              over.
```

The under tactic - III

- Batch mode: can be viewed as a shortcut for interactive mode + dispatch:

```
under eq_big =>[i_1 | i_2] do [tac1 | tac2].
≡
(under eq_big)=>[i_1 | i_2 | ]; [tac1; over | tac2; over | ].
```

[1] see also https://github.com/math-comp/math-comp/blob/master/CONTRIBUTING.md#proof-style

The under tactic - III

- Batch mode: can be viewed as a shortcut for interactive mode + dispatch:

```
under eq_big =>[i_1 | i_2] do [tac1 | tac2].
≡
(under eq_big)=>[i_1 | i_2 | ]; [tac1; over | tac2; over | ].
```

- Some even shorter syntax is available (with automatic introduction):

```
under eq_big do [ | rewrite subnn].
   is the defective form for:
under eq_big =>[* | *] do [ | rewrite subnn].
```

[1] see also https://github.com/math-comp/math-comp/blob/master/CONTRIBUTING.md#proof-style

The under tactic - III

- Batch mode: can be viewed as a shortcut for interactive mode + dispatch:

```
under eq_big =>[i_1 | i_2] do [tac1 | tac2].
≡
(under eq_big)=>[i_1 | i_2 | ]; [tac1; over | tac2; over | ].
```

- Some even shorter syntax is available (with automatic introduction):

```
under eq_big do [ | rewrite subnn].
   is the defective form for:
under eq_big =>[* | *] do [ | rewrite subnn].
```

- Interactive mode: useful to debug/repair a broken proof script
- Choice between batch & interactive versions? mostly a matter of style[1]

[1] see also https://github.com/math-comp/math-comp/blob/master/CONTRIBUTING.md#proof-style

The under tactic - IV

- The tactic also supports occurrences switches and contextual patterns, which are both optional:

```
under {2}[in RHS]eq_lem.
```

The under tactic - IV

- The tactic also supports occurrences switches and contextual patterns, which are both optional:

```
under {2}[in RHS]eq_lem.
```

- Intro patterns are optional, but recommended:

```
under eq_big => [i|i ?].
under eq_bigl => i.
```

The under tactic - IV

- The tactic also supports occurrences switches and contextual patterns, which are both optional:

```
under {2}[in RHS]eq_lem.
```

- Intro patterns are optional, but recommended:

```
under eq_big => [i|i ?].
under eq_bigl => i.
```

(notably as under attempts to preserve the name of bound variables from the first branch, as we'll see in the demo)

The under tactic - V

Design decisions

- Implemented in OCaml to avoid Ltac1 limitations[a]
- Give a protected context `'Under[ _ ]` for evars
- Name all bound variables
- Compatibility with SSReflect's intro patterns
- Compatibility with precedence level of tacticals "`;`" and "`do`"

[a] a prototype was first coded in Ltac [mid-2016]: github.com/erikmd/ssr-under-tac

What about `setoid_rewrite`?

`setoid_rewrite`

+ automatic way to rewrite a bunch of occurrences
− not precise enough: doesn't allow to specify contextual patterns for the desired rewrite

`under`

+ more flexibility (one can choose the congruence lemma to follow and precisely select the redex to rewrite), can be nested
++ ability to perform conditional rewrites
+ compatible with registered Setoid equalities [$\rightsquigarrow$ Coq 8.11]

[Demo]

Outline

Concluding remarks & perspectives

SSReflect

- In Coq since version 8.7, documented [1, 2], stable proof language
- Since Coq 8.10:
  - Fast, temporary, block, DIY intro patterns:
    `[ssr] extended intro patterns`
    https://github.com/coq/coq/pull/6705
  - Rewriting under binders:
    `[ssr] Add tactics under and over`
    https://github.com/coq/coq/pull/9651
- In the pipeline for Coq 8.11:
  - Make under support equivalence relations other than "=":
    `[ssr] Generalize tactics under and over to any Setoid relation`
    https://github.com/coq/coq/pull/10022

References

[1] The Coq Development Team.
The Coq Proof Assistant, version 8.10.0, August 2019.
URL: https://coq.inria.fr/distrib/current/refman/proof-engine/ssreflect-proof-language.html.

[2] Assia Mahboubi and Enrico Tassi.
Mathematical Components.
draft, v1-183-gb37ad7, 2018.
URL: https://math-comp.github.io/mcb.