SSReflect in Coq 8.10

Érik Martin-Dorel[1] and Enrico Tassi[2]

[1]IRIT, Université Toulouse 3 (erik.martin-dorel@irit.fr)
[2]Inria, Université Côte d'Azur (enrico.tassi@inria.fr)

July 22, 2019

SSReflect is a proof language for the Coq system. It was created by Georges Gonthier as part of the proof of the Four Color Theorem [1] that was completed in 2005. The proof language lived as an add-on for Coq since 2017 when it was integrated in the Coq code repository. Today, any Coq user can activate that proof language by just writing `Require Import ssreflect`, and can mix SSReflect's proof commands with the standard ones whenever it fits. With the release of Coq 8.10, the SSReflect proof language got support for rewriting under binders and managing in a concise way the large number of proof variables that is typical in programming languages' metatheory proofs. This talk focuses on these new features.

Rewriting under binders Many constructs in the Mathematical Components library [2] can be defined using a higher-order iterator applied to one or several lambdas. The most typical example is the notion of big operator, or `bigop`, for iterating sum, product, intersection, etc. In this context, rewriting under binders is often needed and prior to this work, SSReflect offered no support for it: the user was given congruence lemmas for `bigop`s, but its application had to be performed manually. The `under` tactic fills this gap.

In the following example, we use the `eq_big` congruence lemma, stating that two `bigop` expressions are equal whenever their higher-order arguments are.

```
eq_big : ∀R idx op I (r : list I) (P1 P2 : I -> bool) (F1 F2 : I -> R),
    (∀i, P1 i = P2 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
```

Here `op` is the operation being iterated, `idx` its unit, and `r` the range. While the lemma is generic, the Mathematical Component library provides specific notations for common data types and operations, e.g., `\sum` stands for `\big[plus/0]`.

The arguments `P1, P2, F1` and `F2` are higher order, and the `under` tactic gives a way to obtain them "interactively," instead of providing them by hand. For example:

```
Lemma test (n : nat) : \sum_(0 <= k < n | odd k && (k != 1)) (k - k) = 0.
Proof. 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 roughly behaves as SSReflect's `rewrite` (including support for contextual patterns or occurrence selectors) but it uses existential variables (evars) to delay the need to provide the higher-order terms. In this case `P1` and `F1` can be found by matching the goal against the left hand side of the lemma, while evars are introduced for the still missing `P2` and `F2` (see the third subgoal). The tactic runs the provided intro patterns in the subgoals corresponding to the side conditions and protects the evars from unwanted instantiation using the `'Under` constant, e.g., replacing `i - i = ?F2 i` with `'Under[ i - i ]` in the second subgoal. The user can then use the `rewrite` tactic to manipulate the expressions *under* the binder that was crossed and can signal the end of this process by using the `over` tactic or the `over` rewrite rule (i.e., `by rewrite over`).

The `under` tactic also provides a *"one-liner" or batch* mode where the tactics to be used in each subgoal are given immediately. In particular:

```
under eq_big =>[i_1|i_2] do [tac1|tac2].
```

is a shorter form for, roughly:

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

If the intro pattern is omitted, "`under eq_lem do [tac1|...|tacn]`" defaults to: "`under eq_lem => [* | ... | *] do [tac1|...|tacn]`", that is all variables are introduced before running tactics. Finally, if there is only one side-condition, then the square brackets can be omitted, as in "`under eq_bigr => i do rewrite addnC.`"

Intro patterns The SSReflect proof language enforces a very strict discipline with respect to proof variables names: the ones given by the user are accessible, the ones generated by the machine are not. The cost of spending time to actually give meaningful names to hypotheses is trade for easy-to-repair scripts, since it is easy to spot when the hypothesis names that are used in the proof do not correspond anymore to their types. Still, there are domains where this strict discipline is often overwhelming: meta theory of programming languages. In this domain, proofs typically branch in dozen of cases, each one introducing a substantial amount of variables.

The block intro pattern feature, suggested by Cyril Cohen around 2012, lets one declare "good names" for variables as part of an inductive type declaration and use these names, proviso a prefix/suffix that makes them unique, at introduction time.

```
Inductive syntax := K1 (a : T) | ... | Kn (a : T) (b : T).

Lemma test : ∀s t : syntax, P s t.
Proof. move => [^ _1] [^ _2].

 a_1, a_2 : T                a_1, b_1, a_2, b_2 : T
==========================  ...  ============================
P (K1 a_1) (K1 a_2)             P (Kn a_1 b_1) (Kn a_2 b_2)
```

Here, both `s` and `t` are inspected by case analysis. All variables from `s` are named by appending `_1` to the names found in the constructors (`_2` for them ones coming from `t`).

Another way of weakening the naming discipline while retaining its benefits was found by Arthur Charguéraud and implemented by the `introv` tactic of the TLC library. The idea is to name hypotheses rather than quantified variables, since there are used in the proof script way more often than the other. The SSReflect `>` intro pattern introduces all quantified variables letting one then name the hypothesis that follows them. For example:

```
Lemma test : ∀x y : nat, x < y -> P x y.  Proof. move=> >lt_xy.

 _x_, _y_ : nat
 lt_xy : _x_ < _y_
==========================
 P _x_ _y_
```

A few additional features were added to SSReflect in Coq version 8.10 and will be presented if time permits.

References

[1] Georges Gonthier. The Four Colour Theorem: Engineering of a Formal Proof. In *ASCM*, page 333, 2007. doi:10.1007/978-3-540-87827-8_28.

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