Jacques Garrigue, 2018 年 12 月 12 日

# MathComp/SSReflect

## 1 MathComp/SSReflect のインストール

講義のホームページから ssreflect-1.7.0-media.tgz をダウンロードし、それをホームディレクトリに展開すると使えるようになる。確認する方法は以下のとおり。

```
% cd ~
% tar zxvf Downloads/ssreflect-1.7.0-media.tgz
...
% coqtop
Welcome to Coq 8.8.1

Coq < From mathcomp Require Import all_ssreflect.
[Loading ML file ssrmatching_plugin.cmxs ... done]
...
```

## 2 SSReflect のコマンド

SSReflect は 2 つの基本的な部分からできている。1 つは新しいコマンド。もう 1 つは新しい標準ライブラリ。

Coq に比べて、SSReflect の特徴は基本的に 7 つのコマンドのみで操作で証明が作れることだろう。

`move` 前提をゴールと仮定の間に移動させる。Coq の `intros`, `revert` や `generalize` に対応する。

`rewrite` ゴールや仮定の書き換えを行う。Coq の rewrite より強力になっていて、構文も少し違う。

`case` 場合分けを行う。Coq の `case` や `destruct` に対応する。

`have` 事実をおく。Coq の `assert` や `pose`, `set` に対応する。

`apply`, `elim` Coq の `apply` や `elim` と同じだが、SSReflect の修飾子が使える。

`by` ゴールの解決を強制する。

### 修飾子

これだけ少ないコマンドで済む理由は、各コマンドに修飾子を付けてその意味を強化できるからである。`move` がその代表的な例である。`move` は本来何もしないが、修飾子を自由に組み合わせることで様々な操作が可能である。まず、移動修飾子を見る。

`move => x y H.` ゴールの最初の 3 つの前提を仮定 `x y H` としておく。Coq の `intros x y H`。

`move: x H.` 仮定 `x` と `H` をゴールに戻す。`revert x H` と同じ。

`move: (H).` 仮定 `H` をゴールにコピーする (仮定として残す)。`generalize H` と同じ。

`move: H => HP.` 仮定 `H` 一端ゴールに戻して、`HP` という名前でもう一度おく。結果的に名前を `H` から `HP` に変えている。

`move H: e => p.` 前と同じだが、名前 H で等式 `e = p` を残す。

上に書かれている「前提」という概念が SSReflect で重要な役割を果す。ゴールと区別され、名前のついている仮定に対して、前提はゴールに含まれ、名前があっても基本的には順番で参照される。

```
x : T
H1 : prop1
H2 : prop2
============================
forall y : T', prop3 -> prop4
```

ここでは二重線の上が仮定、下はゴールである。そして、`T'` が 1 つ目の前提、`prop3` が 2 つ目の前提である。この前提は SSReflect にとってスタックの働きをしている。

分解パターンは `destruct` のような働きをする。書き換えパターンは前提でゴールを書き換える。適用パターンは前提を置き換える (`apply` の場合はゴール全体)。評価修飾子は `simpl` や `auto` のような単純化や枝刈りを行う。どれも、SSReflect のコマンドの直後か`=>` の右側にしか書けない。

`[x y] [Ha | Hb]` これらは `intros` や `destruct as` で使えるものと同じ。最初の前提は対として分解し、中身を `x` と `y` にする。次の前提について場合分けを行う。

`-> <-` 最初の前提で残りのゴールを書き換える。前提が消える。`intros H; rewrite H` か`<- H` と同じ。

`/H` 最初の前提に対して定理や仮定 H を適用し、結果を置き換える。

`/(_ e)` 最初の前提を `e` に適用し、結果を置き換える。

`/=` ゴールを評価する。`simpl` と同じ。

`//` 自明な場合を省く。後で見る `try done` と同じ。

`//=` 両方を同時に行う。

`case, apply, elim` で上記の全ての修飾子が使える。`rewrite` や Coq 本来のコマンドでは、`=>` 以降のものしか使えない。例えば `n : nat` で、`Hn` が `n` に依存したとき、

```
elim: n Hn => /= [|n IHn] // ->.
```

これはまず `n` と `Hn` をゴールに戻し、最初の前提となる `n` に対して帰納法を使う。得られたゴールを`/=`で単純化し、`n` が 0 でないとき、新しい前提ができるので、それらを `n` と `IHn` として仮定に加える。次に自明な場合を省き、最後に `Hn` で残りを書き換える。

### rewrite

`rewrite` は独自の構文を使う。基本的には、定理の名前や適用を並べる。

```
rewrite lem1 lem2 (lem3 n 1)
```

各定理に対して、繰り返しや適用箇所の指定もできる。

`!lem` 定理 `lem` による書き換えを可能な限り繰り返す。

`2!lem` 定理 `lem` による書き換えを 2 回繰り返す。

`?lem` 定理 `lem` を 0 または一回使う。

`-lem` 定理 `lem` を左向きに使う。

`{2}lem` 2 番目の出現を書き換える

`[_ + n]lem` パターン $(\_ + n)$ にマッチする最初の出現を書き換える。

/def 定義 def を展開する。unfold と同じ。
-/def 定義 def を畳み込む。fold と同じ。

上記の書き換え修飾子を組合せることができるが、順番に気を付けなければならない。

```
rewrite -{2}[_ + n]lem
```

また、定理や定義の間に評価修飾子を (/=, //, //=) を挿入してもいい。

### have, suff, case, by, done

have には 2 つの形がある。

have H : prop. 命題 prop の証明を始める。証明が終ったらら仮定 H として加える。assert と同じ

have H := lem arg1 arg2. 定理 lem を適用したものを仮定 H として加える。pose と同じ。

どちらの場合でも、H が省略されると、命題が前提として置かれる。また、H の代わりに分解パターンや書き換えパターンを使ってもいい。

```
have [x Hx]: exists x, x * x = 9.
```

suff は have の一つ目の形と同じだが、生成されるゴールの順番が逆になる。先に仮定する命題を利用して元のゴールを証明し、後でその命題を証明する。
case にも特殊な構文がある。

case H: x. 場合分けに関する等式を仮定 H としておく。case_eq に似ている。

最後に by と done を見る。

by cmds. コマンド列 cmds の後に auto の弱い形でゴールを解決する。解決できない場合、エラーになる。

done. by [] と同じ。この時点で解決できなければエラーになる。

### Search

Ssreflect は Search コマンドも改良している。

```
Search "plus".                              (* 名前に plus を含む定理を検索する *)
Search (_ + S _).                          (* 結論がパターンを含む定理を検索する *)
Search _ (_ + S _).                (* 前提または結論がパターンを含む定理を検索する *)
Search (_ + _) (_ * _) "distr".                  (* 左を全てみたすものを検索する *)
```

### 例

```
Require Import ssreflect.

Section Koushin.
Variables P Q : Prop.

Theorem modus_ponens : P -> (P -> Q) -> Q.
Proof.
  by move=> p; apply.
```

```
Qed.

Theorem DeMorgan : ~ (P \/ Q) -> ~ P /\ ~ Q.
Proof.
  move=> npq.
  by split=> [p|q]; apply npq; [left | right].
Qed.

Theorem and_comm : P /\ Q -> Q /\ P.
Proof.
  by move=> [p q]; split.
Qed.
End Koushin.

Module Plus.
Lemma plus_assoc m n p : m + (n + p) = (m + n) + p.
Proof.
  elim: m => [|m IHm] //=.
  by rewrite IHm.
Qed.
End Plus.
```

# 3 MathCompのライブラリ

先週はssreflectのコマンドを見たが、MathCompの本当の強さはそのライブラリにある。その大きな特徴は書き換えを証明の基本手法とすること。

ライブラリは`ssreflect`, `fingroup`, `algebra`等、いくつかのの部分からできている。前者は一般的なデータ構造で、後者は代数系の証明に使う。

## 基本データ

まず、`ssreflect`を読み込む。それほど多くはない。

```
From mathcomp Require Import
  ssreflect ssrbool ssrnat ssrfun seq eqtype choice fintype.
```

`ssrbool`は論理式と述語の扱い。`ssrnat`は自然数。`ssrfun`は関数(写像)の様々な性質。`seq`はリスト。`eqtype`, `choice`, `fintype`はそれぞれ等価性、選択、有限性が使えるデータ構造のための枠組みを提供している。例えば、自然数の等価性は判定できるので、排中律を仮定しなくても場合分けができる。

中身について、ファイルを参照するしかないが、まず`ssrnat`の例をみよう。(ちなみに、ソースファイルはは`~/.local/share/coq/mathcomp/ssreflect`の下にある)

```
Module Test_ssrnat.
Fixpoint sum n :=
  if n is m.+1 then n + sum m else 0.

Theorem sum_square n : 2 * sum n = n * n.+1.
Proof.
  elim: n => [|n IHn] /=.
    done.
  rewrite mulnDr.
```

```
  rewrite -[n.+2]addn2 mulnDr.
  rewrite [n.+1*n]mulnC -IHn.
  by rewrite addnC (mulnC _ 2).
Qed.
End Test_ssrnat.
```

## 自己反映

論理式も書き換えで処理したい。そのために、ssrbool では論理演算子を型 bool の上の演算子として定義している。例えば、/\ は&&, \/ は||になる。二つの定義の間に行き来するために、reflect という自己反映を表した宣言を使う。それが SSReflect の名前の由来である。

```
Print reflect.
Inductive reflect (P : Prop) : bool -> Set :=
    ReflectT : P -> reflect P true | ReflectF : ~ P -> reflect P false
Check orP.
orP : forall b1 b2 : bool, reflect (b1  b2) (b1 || b2)
```

表現の切り替えはビュー機構によって行われる。前に見た適用パターンを使う。move, case, apply などの直後に/view を付けると、対処が可能な方向に変換される。=>の右でも使える。なお、ビューとしては上の reflect 型 だででなく、同値関係 (P <-> Q) や普通の定理 (P -> Q) も使える。

```
Module Test_ssrbool.
Variables a b c : bool.

Print andb.

Lemma andb_intro : a -> b -> a && b.
Proof.
  move=> a b.
  rewrite a.
  move=> /=.
  done.
Restart.
  by move ->.
Qed.

Lemma andbC : a && b -> b && a.
Proof.
  case: a => /=.
    by rewrite andbT.
  done.
Restart.
  by case: a => //= ->.
Restart.
  by case: a; case: b.
Qed.

Lemma orbC : a || b -> b || a.
Proof.
  case: a => /=.
    by rewrite orbT.
```

```
  by rewrite orbF.
Restart.
  move/orP => H.
  apply/orP.
  move: H => [Ha|Hb].
    by right.
  by left.
Restart.
  by case: a; case: b.
Qed.


Lemma test_if x : if x == 3 then x*x == 9 else x !=3.
Proof.
  case Hx: (x == 3).
    by rewrite (eqP Hx).
  done.
Restart.
  case: ifP.
    by move/eqP ->.
  move/negbT. done.
Qed.

End Test_ssrbool.
```

自己反映があると自然数の証明もスムーズになる。

```
Theorem avg_prod2 m n p : m+n = p+p -> (p - n) * (p - m) = 0.
Proof.
  move=> Hmn.
  have Hp0 q: p <= q -> p-q = 0.
    by rewrite -subn_eq0 => /eqP.
  suff /orP[Hpm|Hpn]: (p <= m) || (p <= n).
    + by rewrite (Hp0 m).
    + by rewrite (Hp0 n).
  case: (leqP p m) => Hpm //=.
  case: (leqP p n) => Hpn //=.
  suff: m + n < p + p.
    by rewrite Hmn ltnn.
  by rewrite -addnS leq_add // ltnW.
Qed.
```

### 数学関係の定理

こちらはモジュールが多過ぎて、簡単に紹介できる。よく使うもおとして、finset(fintype に基いた有限集合、基本的な線形代数は matrix、perm や vector、多項式は poly、素数は prime。

**練習問題 3.1** 今までの課題を *SSReflect* の構文を使って書き換えてみる。