Jacques Garrigue, 2020 年 11 月 4 日

Mathcomp, 自己反映と単一化

非可述性 (impredicativity)

Coq の抽象の推論規則をより詳しくみつと，Set と Prop の違いが見えて来る．

$$\textbf{抽象}\quad \frac{\Gamma, X : S \vdash M : T \quad \Gamma \vdash \forall X{:}S, T : \mathrm{Type}}{\Gamma \vdash \mathtt{fun}\ X{:}S \Rightarrow M : \forall X{:}S, T} \qquad \Gamma \vdash \mathrm{Prop} : \mathrm{Set} \qquad \Gamma \vdash \mathrm{Set} : \mathrm{Type}$$

$$\frac{\Gamma \vdash A : \mathrm{Prop}}{\Gamma \vdash A : \mathrm{Set}} \qquad \frac{\Gamma \vdash A : \mathrm{Set}}{\Gamma \vdash A : \mathrm{Type}_i} \qquad \frac{\Gamma \vdash A : \mathrm{Type}_i}{\Gamma \vdash A : \mathrm{Type}_{i+1}} \qquad \frac{\Gamma \vdash A : \mathrm{Type}_i \quad \Gamma, X : A \vdash B : \mathrm{Prop}}{\Gamma \vdash \forall X{:}A, B : \mathrm{Prop}}$$

$$\frac{\Gamma \vdash A : \mathrm{Set} \quad \Gamma, X : A \vdash B : \mathrm{Set}}{\Gamma \vdash \forall X{:}A, B : \mathrm{Set}} \qquad \frac{\Gamma \vdash A : \mathrm{Type}_i \quad \Gamma, X : A \vdash B : \mathrm{Type}_i}{\Gamma \vdash \forall X{:}A, B : \mathrm{Type}_i}$$

Prop に対して抽象化しても結果が Prop になるが，Set に対して抽象化すると結果が Type にならざるを得ない．`-impredicative-set` を使うと，Set が Prop と同様に非可術的になる．Set の中の排中律 (`forall A:Prop, {A}+{~A}`) と矛盾するので要注意．System F 的な表現方法はその非可術性を必要としている．

練習問題 0.1 *Nat* に関する証明もやってみましょう．

1 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．

Search

Ssreflect の Search コマンドが強力で，ライブラリを探すのに便利．

```
Search "add".                                  (* 名前に add 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 (_ + _) (_ * _) "mul".                  (* 左を全てみたすものを検索する *)
```

基本データ

まず，`ssreflect` を読み込む．[1]

```
From mathcomp Require Import all_ssreflect.
```

[1] もしも mathcomp がまだインストールされていなければ，講義のホームページからダウンロードして展開する．

いくつかのモジュールが読み込まれます．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 と/opt/local/lib/coq/theories/ssr の下にある)

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

Theorem double_sum n : 2 * sum n = n * n.+1.
Proof.
  elim: n => [|n IHn] //=.
  rewrite -[n.+2]addn2 !mulnDr.
  rewrite addnC !(mulnC n.+1).
  by rewrite IHn.
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.
    rewrite -subn_eq0. by move/eqP.
  suff /orP[Hpm|Hpn]: (p <= m) || (p <= n).
    - by rewrite (Hp0 m) // muln0.
    - 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.
```

練習問題 1.1 以下の等式を証明しなさい．タクティクは rewrite のみでできる．
ssrnat_doc.v の補題でほぼ足りるが, leq_mul も便利.

```
Module Equalities.
  Theorem square_sum a b : (a + b)^2 = a^2 + 2 * a * b + b^2. Abort.
  Theorem diff_square m n : m >= n -> m^2 - n^2 = (m+n) * (m-n). Abort.
  Theorem square_diff m n : m >= n -> (m-n)^2 = m^2 + n^2 - 2 * m * n. Abort.
End Equalities.
```

2 単一化と自動化

```
Lemma test x : 1 + x = x + 1.
  Check [eta addnC].
     : ∀ x y : nat, x + y = y + x
  apply: addnC.
Abort.                                        (* 定理を登録せずに証明を終わらせる *)
```

ゴールと定理 addnC の字面が異なっているのに，ここでなぜ apply が使えるのか.
実は apply は複数のことをしている.

1. 定理の中の ∀ で量化されている変数を「単一化用の変数」に置き換える
2. 置き換えられた定理の結論をゴールと「単一化」する
3. 定理に前提があれば,「単一化」の結果を代入した前提を新しいゴールにする.

ここでいう単一化とは,（単一化用の）変数を含んだ項同士をその変数の値を定めることで同じものにする．例えば，1 + x = x + 1 と ?m+?n =?n+?m を単一化するには，?m = 1, ?n = x と定めれば良い.

Coq 本来の apply で変数が定まらないと，エラーになる．しかし，SSReflect の apply: や apply/ を使えば，変数が残せる.

```
Lemma test x y z : x + y + z = z + y + x.
  Check etrans.
     : ∀ (A : Type) (x y z : A), x = y -> y = z -> x = z
  apply etrans.
Error: Unable to find an instance for the variable y.
  apply: etrans.                    (* y が結論に現れないので，apply: に変える *)
   x + y + z = ?Goal
  apply: addnC.
  apply: etrans.
  Check f_equal.
     : ∀ (A B : Type) (f : A -> B) (x y : A), x = y -> f x = f y
  apply: f_equal.                                    (*   x + y = ?Goal0 *)
  apply: addnC.
  apply: addnA.
Restart.                                                (* 証明を元に戻す *)
  rewrite addnC.                                  (* rewrite も単一化を使う *)
  rewrite (addnC x).
  apply: addnA.
Abort.
```

一階の項に関して，単一化は最適な代入を見つけてくれるという結果が知られている.

定義 1 ある代入 σ_1 が代入 σ_2 **より一般的**であるとは，σ_{12} が存在し，$\sigma_2 = \sigma_{12} \circ \sigma_1$ であることをいう．

定義 2 単一化問題 $\{t_1 = t'_1, \ldots, t_n = t'_n\}$ に対して，$\sigma(t_i) = \sigma(t'_i)$ であるとき，σ はその単一化問題の**単一子**だという．

定理 1 任意の単一化問題に対して，解が存在するときには**最も一般的な単一子**を返し，存在しないときにはそれを報告するアルゴリズムが存在する．

具体的には，アルゴリズム $\mathcal{U}$ を以下の買い替え規則で定義できる．ここでは定数記号を 0 引数の関数記号と同一視する．

$$
\begin{array}{ll}
E \cup \{f(t_1, \ldots, t_n) = f(t'_1, \ldots, t'_n)\} \to E \cup \{t_1 = t'_1, \ldots, t_n = t'_n\} & \\
E \cup \{f(t_1, \ldots, t_n) = g(t'_1, \ldots, t'_m)\} \to \bot & f \neq g \\
E \cup \{x = t\} \to [t/x]E \cup \{x = t\} & x \in vars(E) \wedge (t = y \Rightarrow y \in vars(E)) \wedge x \notin vars(t) \\
E \cup \{x = x\} \to E & \\
E \cup \{x = t\} \to \bot & x \in vars(t)
\end{array}
$$

E が書き換えを繰り返し，書き換えられない E' になれば，その E' が $\{x_1 = t''_1, \ldots, x_m = t''_m\}$ という形であり，それを代入 $\sigma = [t''_1/x_1, \ldots, t''_m/x]$ と見なし，$\mathcal{U}(E) = \sigma$．このときに，任意の $t = t' \in E$ について，$\sigma(t) = \sigma(t')$，かつ E の単一子になる任意の σ' について，σ が σ' より一般的である．また，$E' = \bot$ のとき，E には解がない．

上記の $\mathcal{U}$ は一階の項のためのものであるが，Coq はそれより強い**高階単一化**[2]を行っている．しかし，この場合には最も一般的な解が存在するとは限らない．

```
Goal
  (∀ P : nat -> Prop, P 0 -> (∀n, P n -> P (S n)) -> ∀n, P n) ->
     ∀ n m, n + m = m + n.
  move=> H n m.                                    (* 全ての変数を仮定に *)
  apply: H.                                              (* n + m = 0 *)
Restart.
  move=> H n m.
  pattern n.                                  (* pattern で正しい述語を構成する *)
  apply: H.                                          (* 0 + m = m + 0 *)
Restart.
  move=> H n.                                   (* forall n を残すとうまくいく *)
  apply: H.                                          (* n + 0 = 0 + n *)
Abort.
```

単一化は様々な作戦で使われている．`apply` 以外に `elim`，`rewrite` や `set` が挙げられる．

[2] 1970 年代に高階単一化の可能性を証明した Gérard Huet は Coq の最初のプロジェクトリーダーだった