Jacques Garrigue, 2019 年 5 月 15 日

帰納的な定義と自己反映

1　帰納的な定義

Coq の帰納的データ型

前回は自然数の定義を見た.

```
Inductive nat : Set := O : nat | S : nat -> nat.
```

実は，Coq の全てのデータは帰納的データ型として定義される.[1]

```
Inductive prod (A B : Set) : Set := pair : A -> B -> prod A B.

Inductive sum (A B : Set) : Set :=  inl : A -> sum A B | inr : B -> sum A B.
```

帰納的データ型の値を作るのは構成を適用するだけでいい．しかし，分解するのに OCaml と同様にパターンマッチングを使わなければならない．その型付け規則が複雑になる．以下のようなデータ型を考える.

$$
\begin{array}{ll}
\texttt{Inductive} & t(a_1 \dots a_n : \texttt{Set}) : \texttt{Set} := \\
& \quad | \quad c_1 : \tau_{11} \to \dots \to \tau_{1k_1} \to t\ a_1 \dots a_n \\
& \quad \dots \\
& \quad | \quad c_m : \tau_{m1} \to \dots \to \tau_{mk_m} \to t\ a_1 \dots a_n.
\end{array}
$$

マッチング

$$
\frac{\Gamma \vdash M : t\ b_1 \dots b_n \qquad \Gamma, x_{i1} : \tau_{i1}[b_1/a_1, \dots, b_n/a_n], \dots, x_{ik_i} : \tau_{ik_i}[\dots] \vdash M_i : \tau[c_i\, x_{i1} \dots x_{ik_i}/x] \quad (1 \le i \le m)}{\Gamma \vdash \mathsf{match}\ M\ \mathsf{as}\ x\ \mathsf{return}\ \tau\ \mathsf{with}\ c_1\, x_{11} \dots x_{1k_1} \Rightarrow M_1 \mid \dots \mid c_m\, x_{m1} \dots x_{mk_m} \Rightarrow M_m\ \mathsf{end} : \tau[M/x]}
$$

as と return によって，返り値の型の中に入力を含めることができ，場合によって型が違うような関数が作れる．それを手でやるのは難しいが，作戦 case はこのパターンマッチングを構築してくれる.

帰納的データ型を定義すると，帰納法のための補題が自動的に定義されるが，定義は match を使う．定義が再帰的でないとき，パターンマッチングだけで済む．再帰的なデータ型について Fixpoint が使われる.

```
Definition prod_ind (A B:Set) (P:prod A B -> Prop) :=
  fun (f : forall a b, P (pair a b)) =>
  fun p => match p as x return P x with pair a b => f a b end.
Check prod_ind.
  : forall (A B : Set) (P : A * B -> Prop),
      (forall (a : A) (b : B), P (a, b)) -> forall p : A * B, P p

Definition sum_ind (A B:Set) (P:sum A B -> Prop) :=
```

[1]実際の定義を見ると，Set ではなく Type になっている．Type は Set より一般的なので，Set として使うことができる．さらに，prod A B は　A*B として表示され，pair a b は (a,b) として表示される．Coq の Notation という機能によって，機能的データ型の表示方法を変えることができる.

```
  fun (fl : forall a, P (inl _ a)) (fr : forall b, P (inr _ b)) =>
  fun p => match p as x return P x
           with inl a => fl a | inr b => fr b end.
Check sum_ind.
  :  forall (A B : Set) (P : A + B -> Prop),
       (forall a : A, P (inl B a)) -> (forall b : B, P (inr A b)) ->
       forall p : A + B, P p

Fixpoint nat_ind (P:nat -> Prop) (f0:P O) (fn:forall n, P n -> P (S n))
  (n : nat) {struct n} :=
  match n as x return P x
  with O => f0 | S m => fn m (nat_ind P f0 fn m) end.
Check nat_ind.
  : forall P : nat -> Prop, P 0 -> (forall n : nat, P n -> P (S n)) ->
      forall n : nat, P n
```

case と elim はよく似ているが，前者が単なる場合分けを行うのに対して，後者が生成された補題を利用しているので，効果が違ったりする．

```
Lemma plusn0 n : n + 0 = n.
Proof.
  case: n.
  + done.
forall n : nat, S n + 0 = S n
Restart.
  move: n.
  apply: nat_ind.                                          (* elim の意味 *)
  + done.
forall n : nat, n + 0 = n -> S n + 0 = S n
  + move=> n /= -> //.
Qed.
```

帰納的述語

Coq では帰納的な定義は Set だけでなく Prop でもできる．この場合，パラメータは場合によって変わることが多い．

$$\begin{array}{ll} \texttt{Inductive} & t : \tau_1 \to \ldots \to \tau_n \to \texttt{Prop} := \\ & \mid \ c_1 : \forall (x_1 : \tau_{11}) \ldots (x_{k_1} : \tau_{1k_1}), t\ \theta_{11} \ldots \theta_{1n} \\ & \ldots \\ & \mid \ c_m : \forall (x_1 : \tau_{m1}) \ldots (x_{k_1} : \tau_{mk_m}), t\ \theta_{m1} \ldots \theta_{mn} \end{array}$$

マッチング

$$\frac{\Gamma \vdash M : t\ b_1 \ldots b_n \qquad \Gamma, x_{i1} : \tau_{i1}, \ldots, x_{ik_i} : \tau_{ik_i} \vdash M_i : \tau[\theta_{i1} \ldots \theta_{in}/x_1 \ldots x_n] \quad (1 \le i \le m)}{\Gamma \vdash \texttt{match}\ M\ \texttt{in}\ t\ x_1 \ldots x_n\ \texttt{return}\ \tau\ \texttt{with}\ c_1\, x_{11} \ldots x_{1k_1} \Rightarrow M_1 \mid \ldots \mid c_m\, x_{m1} \ldots x_{mk_m} \Rightarrow M_m\ \texttt{end} : \tau[b_1, \ldots, b_n/x_1 \ldots x_n]}$$

```
(* 偶数の定義 *)
Inductive even : nat -> Prop :=
  | even_0 : even 0
  | even_SS : forall n, even n -> even (S (S n)).

(* 帰納的述語を証明する定理 *)
```

```
Theorem even_double n : even (n + n).
Proof.
  elim: n => /= [|n IH].
  + apply: even_0.
  + rewrite -plus_n_Sm.
    by apply: even_SS.
Qed.

(* 帰納的述語に対する帰納法もできる *)
Theorem even_plus m n : even m -> even n -> even (m + n).
Proof.
  elim: m => //=.
Restart.
  move=> Hm Hn.
  elim: Hm => //= m m IH.
  apply: even_SS.
Qed.

(* 矛盾を導き出す *)
Theorem one_not_even : ~ even 1.
Proof.
  case.
Restart.
  move H: 1 => one He.  (* move H: exp => pat は H: exp = pat を作る *)
  case: He H => //.
Restart.
  move=> He.
  inversion He.
  Show Proof.  (* 証明が複雑で、SSReflect では様々な理由で避ける *)
Qed.

(* 等式を導き出す *)
Theorem eq_pred m n : S m = S n -> m = n.
Proof.
  move=> [].  (* 等式を分解する *)
  done.
Qed.
```

実は Coq の論理結合子のほとんどが帰納的述語として定義されている．

```
Inductive and (A B : Prop) : Prop :=  conj : A -> B -> A /\ B.

Inductive or (A B : Prop) : Prop :=
    or_introl : A -> A \/ B | or_intror : B -> A \/ B.

Inductive ex (A : Type) (P : A -> Prop) : Prop :=
    ex_intro : forall x : A, P x -> exists x, P x.

Inductive False : Prop := .
```

and，or や ex について case が使えた理由がこの定義方法である．

しかも，False は最初からあるものではなく，構成子のない述語として定義されている．生成される帰納法の補題をみると面白い．

```
Print False_ind.
```

```
fun (P : Prop) (f : False) => match f return P with end
     : forall P : Prop, False -> P
```

ちょうど，矛盾の規則に対応している．作戦　elim でそれが使える．

```
Theorem contradict (P Q : Prop) : P -> ~P -> Q.
Proof.
  move=> p.
  elim.
  exact: p.
Qed.
```

練習問題 1.1 以下の定理を証明しなさい．

```
Module Odd.
Inductive odd : nat -> Prop :=
  | odd_1 : odd 1
  | odd_SS : forall n, odd n -> odd (S (S n)).

Theorem even_odd n : even n -> odd (S n). Abort.
Theorem odd_even n : odd n -> even (S n). Abort.
Theorem even_not_odd n : even n -> ~odd n. Abort.
End Odd.
```

2　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．それほど多くはない．

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

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 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.
    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.
```

練習問題 2.1 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.
```