Jacques Garrigue, 2024 年 5 月 1 日

帰納的な定義と型付け

1 前回の課題

```
Lemma plusn0 n : n + 0 = n.
Proof. elim: n => /=. done. move=> n ->. done. Qed.

Lemma plusC m n : m + n = n + m.
Proof. elim: m => /= [|m ->]; by rewrite (plusn0,plusnS). Qed.

Lemma plusA m n p : m + (n + p) = (m + n) + p.
Proof. by elim: m => /= [|m ->]. Qed.

Lemma multn0 n : n * 0 = 0.
Proof. elim: n => //. Qed.

Lemma multC m n : m * n = n * m.
Proof. elim: m => /= [|m ->]; by rewrite (multn0,multnS). Qed.

Lemma multnDr m n p : (m + n) * p = m * p + n * p.
Proof. elim: m => /= [|m ->] //. by rewrite plusA. Qed.

Lemma multA m n p : m * (n * p) = (m * n) * p.
Proof. elim: m => /= [|m ->] //. by rewrite multnDr. Qed.

Lemma double_sum n : 2 * sum n = n * (n + 1).
Proof.
  elim: n => /= [|n IH] //.
  rewrite multnS -IH /=.
  rewrite !plusnS !plusA /=.
  rewrite ![_+0]plusC [_+n]plusC /=.
  by rewrite !plusA.
Qed.

Lemma square_eq a b : (a + b) * (a + b) = a * a + 2 * a * b + b * b.
Proof.
  rewrite multnDr.
  rewrite (multC a) (multC b) !multnDr.
  rewrite (multC b) /=.
  rewrite [_+0]plusC /=.
  by rewrite !plusA.
Qed.

Require Import ArithRing.                          (* nat における半環構造 *)
Lemma square_eq a b : (a + b) * (a + b) = a * a + 2 * a * b + b * b.
Proof. ring. Qed.
```

2 プログラムの型付け

型 $\tau, \theta ::= \mathsf{nat} \mid \mathsf{Z} \mid \ldots \mid \theta \to \tau \mid \tau \times \theta \mid \forall x{:}\theta, \tau$ データ型, 関数型, 直積, 依存積

型判定 $\Gamma \vdash M : \tau$ $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$ という仮定のもとで，M が型 τ をもつ．

型付け規則 Coq の式は以下の型付け規則によって型付けされる．

変数 $\Gamma \vdash x : \tau$ $(x : \tau$はΓに含まれる$)$

定義 $$\frac{\Gamma \vdash M : \theta \quad \Gamma, x : \theta \vdash N : \tau}{\Gamma \vdash (\mathtt{let}\ x\ \mathtt{:=}\ M\ \mathtt{in}\ N) : \tau}$$

抽象 $$\frac{\Gamma, x : \theta \vdash M : \tau}{\Gamma \vdash (\mathtt{fun}\ x{:}\theta \Rightarrow M) : \theta \to \tau}$$

不動点 $$\frac{\Gamma, f : \theta \to \tau, x : \theta \vdash M : \tau}{\Gamma \vdash (\mathtt{fix}\ f\ (x{:}\theta)\ \mathtt{:=}\ M) : \theta \to \tau}$$

適用 $$\frac{\Gamma \vdash M : \theta \to \tau \quad \Gamma \vdash N : \theta}{\Gamma \vdash M\ N : \tau}$$

直積 $$\frac{\Gamma \vdash M : \tau \quad \Gamma \vdash N : \theta}{\Gamma \vdash (M, N) : \tau \times \theta}$$

射影 $\Gamma \vdash \mathsf{fst} : \tau \times \theta \to \tau$ $\Gamma \vdash \mathsf{snd} : \tau \times \theta \to \theta$

抽象* $$\frac{\Gamma, x : \theta \vdash M : \tau}{\Gamma \vdash (\mathtt{fun}\ x{:}\theta \Rightarrow M) : \forall x{:}\theta, \tau}$$

適用* $$\frac{\Gamma \vdash M : \forall x{:}\theta, \tau \quad \Gamma \vdash N : \theta}{\Gamma \vdash M\ N : [N/x]\tau}$$

$x \notin fv(\tau)$ のときに $\theta \to \tau = \forall x : \theta, \tau$ とおくと，**抽象**と**適用**が**抽象***と**適用*** の特殊な場合になる．

型付けの例

$$\dfrac{\dfrac{\dfrac{\Gamma, x : \mathsf{nat} \vdash \mathsf{S} : \mathsf{nat} \to \mathsf{nat} \quad \Gamma, x : \mathsf{nat} \vdash x : \mathsf{nat}}{\Gamma, x : \mathsf{nat} \vdash \mathsf{S}\ x : \mathsf{nat}}\ \text{適用}}{\Gamma \vdash (\mathtt{fun}\ x{:}\mathsf{nat} \Rightarrow \mathsf{S}\ x) : \mathsf{nat} \to \mathsf{nat}}\ \text{抽象} \qquad \Gamma \vdash \mathsf{O} : \mathsf{nat}}{\Gamma \vdash (\mathtt{fun}\ x{:}\mathsf{nat} \Rightarrow \mathsf{S}\ x)\ \mathsf{O} : \mathsf{nat}}\ \text{適用}$$

3 命題と型の対応

カリー・ハワード同型により，命題論理と型理論 (型付 λ 計算) が対応している．
具体的には，以下のような対応が見られる．

命題 (論理式)	型
証明 (導出)	プログラム
仮定 Δ	型環境 Γ
$\supset$	$\to$
$\wedge$	$*$

導出規則と型付け規則も基本的には 1 対 1 で対応している．それぞれの体系を少し修正すると以下の定理がなりたつ．

定理 1 (Curry-Howard 同型) ある同型 $\langle \_ \rangle$: 命題 $\to$ 型 が存在し，任意の Δ と P について，導出 Π より $\Delta \vdash P$ が示せるならば，Π からプログラム M が作れ，$\langle \Delta \rangle \vdash M : \langle P \rangle$．また，任意の Γ, M, τ について型理論で $\Gamma \vdash M : \tau$ が導出できれば，命題論理において $\langle \Gamma \rangle^{-1} \vdash \langle \tau \rangle^{-1}$ が導出できる．

修正の内容は二種類ある．

まず，上の**不動点**の規則は矛盾を生んでしまう．具体的には，$\theta = True$ と $\tau = False$ にすると，以下の導出が可能になる．

$$\frac{\Gamma, f : True \to False, x : True \vdash f\ x : False}{\Gamma \vdash (\texttt{fix}\ f\ (x{:}\theta)\ \texttt{:=}\ f\ x) : True \to False}$$

しかし，Coq の本当の**不動点**の規則はさらに f が x より小さな引数に適用されることを求めているので，この矛盾が実際には起きない．本当の規則が複雑なのでここには書かない．

もう一つは，**背理法**に対する規則は Coq の型体系にはない．それは Coq は**直観主義論理**に基いているからである．メリットとして，全ての証明が計算的な意味を持つ—証明は関数である．

4 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．

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

マッチング

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

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．

[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，帰納的データ型の表示方法を変えることができる．

```
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.
  : ∀ (A B : Set) (P : A * B -> Prop),
      (∀ (a : A) (b : B), P (a, b)) -> ∀ p : A * B, P p

Definition sum_ind (A B:Set) (P:sum A B -> Prop) :=
  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.
  :  ∀ (A B : Set) (P : A + B -> Prop),
       (∀ a : A, P (inl B a)) -> (∀ b : B, P (inr A b)) ->
       ∀ 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.
  : ∀ P : nat -> Prop, P O -> (∀ n : nat, P n -> P (S n)) ->
      ∀ 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.
(* ∀ n : nat, S n + 0 = S n *)
Restart.
  move: n.
  apply: nat_ind.                                    (* elim の意味 *)
  - done.
(* ∀ n : nat, n + 0 = n -> S n + 0 = S n *)
  - move=> n /= -> //.
Qed.
```

リスト

リストは関数型プログラミングで最も使われているデータ構造である．

```
Require Import List. (* リストの関数と補題を使う *)
Print list.
Inductive list (A : Type) : Type :=
  | nil : list A
  | cons : A -> list A -> list A.

Check (cons 1 (cons 2 nil)).
1 :: 2 :: nil
```

```
     : list nat

Import ListNotations. (* リストの記法を使う *)
Check [1; 2; 3].
[1; 2; 3]
     : list nat
```

cons は二項演算子 :: として書ける．また，[a; b; c] という形でリスト cons a (cons b (cons c nil)) を書くこともできる．

リストの結合は app または++と書く．長さは length

```
Compute [1;2] ++ [3;4].
     = [1; 2; 3; 4]
     : list nat
Compute length [1;2;3].
     = 3
     : nat
```

それぞれ，Fixpoint と場合分けで定義されている．

```
Print app.
fix app (l m : list A) struct l : list A :=
  match l with
  | [] => m
  | a :: l1 => a :: app l1 m
  end
     : forall A : Type, list A -> list A -> list A
Print length.
fix length (l : list A) : nat :=
  match l with
  | [] => 0
  | _ :: l' => S (length l')
  end
     : forall A : Type, list A -> nat
```

自然数とほぼ同じ形で帰納法が使える．

```
Lemma length_app A (l1 l2 : list A) :
  length (l1 ++ l2) = length l1 + length l2.
Proof.
  elim: l1 => /=. done.
  move=> a l1 IH. (* 自然数と違い，リストは中身がある *)
  by rewrite IH.
Qed.
```

多項式の表現と計算　多項式をリストとして書く場合には，係数を字数の上る順に書く．x^3+3x+2 は [2; 3; 0; 1] になる．

計算は Horner の公式が最も簡単．

```
Fixpoint horner (l : list nat) x :=
  match l with
  | nil => 0
  | a :: l' => a + x * horner l' x
  end.
```

```
Compute horner [2; 3; 0; 1] 4.
     = 78
     : nat
```

練習問題 4.1 先週証明した補題を使って，*Horner* の公式が以下の多項式の定義と同値であることを示しなさい．ただし，関数 ***exp x n*** $= x^n$ は ***Fixpoint*** を使って定義しなさい．

```
Fixpoint exp (x n : nat) := 1. (* x の n 乗 *)

Fixpoint eval_poly n (l : list nat) x :=
  match l with
  | nil => 0
  | a :: l' => a * exp x n + eval_poly (S n) l' x
  end.

Lemma eval_polyS n l x : eval_poly (S n) l x = x * eval_poly n l x.
Proof. elim: l n. Admitted.

Lemma horner_ok l x : horner l x = eval_poly 0 l x.
Proof. suff : forall n, exp x n * horner l x = eval_poly n l x. Admitted.
```

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．

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

マッチング

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

```
(* 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: m / Hm => //= m IH.
  apply: even_SS.
Qed.

(* 矛盾を導き出す *)
Theorem one_not_even : ~ even 1.
Proof.
  case.
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 *)
Restart.
  move H: 1 => one He.  (* move H: exp => pat は H: exp = pat を作る *)
  by case: He H.
Qed.

(* 等式を導き出す *)
Theorem eq_pred m n : S m = S n -> m = n.
Proof.
  case.  (* 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
     : ∀ 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 4.2 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.
```