Jacques Garrigue, 2020 年 10 月 28 日

帰納的な定義と多相性

1　プログラムの型付け

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

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　$\Gamma \vdash x : \tau \quad (x : \tau \text{は} \Gamma \text{に含まれる})$

定義　$\dfrac{\Gamma \vdash M : \theta \quad \Gamma, x : \theta \vdash N : \tau}{\Gamma \vdash \texttt{let } x \texttt{ := } M \texttt{ in } N : \tau}$

抽象　$\dfrac{\Gamma, x : \theta \vdash M : \tau}{\Gamma \vdash \texttt{fun } x{:}\theta \Rightarrow M : \theta \to \tau}$

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

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

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

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

型付けの例

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

2　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，$\theta = \mathit{True}$ と $\tau = \mathit{False}$ にすると，以下の導出が可能になる．

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

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**直感主義論理**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．

3 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．

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

マッチング

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

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，機能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.
  : 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) :=
  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．

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

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_O : even O
  | 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_O.
  - 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.
  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
     : 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 3.1 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.
```

4 System F

Coq の *Prop* 型は Girard が考案した System F という型体系を実装しており，∀ と適用のみで二階述語論理を表現できる．

```
Section SystemF.
Definition Fand P Q := forall X : Prop, (P -> Q -> X) -> X.
Definition For P Q := forall X : Prop, (P -> X) -> (Q -> X) -> X.
Definition Ffalse := forall X : Prop, X.
Definition Ftrue := forall X : Prop, (X -> X).
Definition Feq T (x y : T) := forall P, Fand (P x -> P y) (P y -> P x).
Definition Fex T (P : T -> Prop) := forall X : Prop, (forall x, P x -> X) -> X.

Theorem Fand_ok (P Q : Prop) : Fand P Q <-> P /\ Q.
Proof.
  split => [pq | [p q] X].
  - split; by apply: pq.
  - by apply.
Qed.

Theorem For_ok (P Q : Prop) : For P Q <-> P \/ Q. Abort.
Theorem Ffalse_ok : Ffalse <-> False. Abort.
Theorem Ftrue_ok : Ftrue <-> True. Abort.
Theorem Feq_ok T (x y : T) : Feq x y <-> x = y. Abort.
Theorem Fex_ok T (P : T -> Prop) : Fex P <-> exists x, P x. Abort.
```

各論理演算子の System F での表現がその除去規則を模倣している．

特に等価性に関する定義 Feq は Leibniz equality と言い，古くから知られている．x と y が等しいとは，任意の述語 p について，$p(x)$ と $p(y)$ が同値であること，言い換えれば，x と y を区別する述語が存在しないこと．

System F では命題だけでなく，データも表現できるのが特徴である.[2] 例えば，自然数に関して，型なし λ 計算で有名な Church 符号が使えます．自然数を関数の繰り返しで表現し，後者関数は繰り返しを一回増やす.

```
Definition Nat := forall X : Prop, (X -> X) -> X -> X.
Definition Zero : Nat := fun X f x => x.
Definition Succ (N : Nat) : Nat := fun X f x => f (N X f x).
Definition Plus (M N : Nat) : Nat := fun X f x => M X f (N X f x).
Definition Mult (M N : Nat) : Nat := fun X f x => M X (N X f) x.
(* こちらの定義はより直感的 *)
Definition Plus' (M N : Nat) : Nat := M Nat Succ N.         (* 1 を M 回足す *)
Definition Mult' (M N : Nat) : Nat := M Nat (Plus' N) Zero. (* N を M 回足す *)

Fixpoint Nat_of_nat n : Nat :=                          (* 自然数を Nat に変換 *)
  match n with 0 => Zero | S m => Succ (Nat_of_nat m) end.

(* Nat の元の等価性は適用された物を比較するべき *)
Definition eq_Nat (M N : Nat) := forall X f x, M X f x = N X f x.
Definition eq_Nat_fun F f := forall n,
  eq_Nat (F (Nat_of_nat n)) (Nat_of_nat (f n)).
Definition eq_Nat_op Op op := forall m n,
  eq_Nat (Op (Nat_of_nat m) (Nat_of_nat n)) (Nat_of_nat (op m n)).

Theorem Succ_ok : eq_Nat_fun Succ S. Proof. by elim. Qed.      (* 実は自明 *)

Theorem Plus_ok : eq_Nat_op Plus plus.
Proof.
  move=> m n X f x.
  elim: m x => //= m IH x.
  by rewrite Succ_ok /= [in RHS]/Succ -IH.
Qed.

Theorem Mult_ok : eq_Nat_op Mult mult. Abort.

Definition Pow (M N : Nat) := fun X => N _ (M X).               (* M の N 乗 *)
Fixpoint pow m n := match n with 0 => 1 | S n => m * pow m n end.

Lemma Nat_of_nat_eq : forall n X f1 f2 x,
    (forall y, f1 y = f2 y) ->
    Nat_of_nat n X f1 x = Nat_of_nat n X f2 x.
Abort.
Theorem Pow_ok : eq_Nat_op Pow pow. Abort.

Section ProdSum.                               (* 値の対と直和も定義できます *)
  Variables X Y : Prop.
  Definition Prod := forall Z : Prop, (X -> Y -> Z) -> Z.
  Definition Pair (x : X) (y : Y) : Prod := fun Z f => f x y.
  Definition Fst (p : Prod) := p _ (fun x y => x).
  Definition Snd (p : Prod) := p _ (fun x y => y).
  Definition Sum := forall Z : Prop, (X -> Z) -> (Y -> Z) -> Z.
  Definition InL x : Sum := fun Z f g => f x.
  Definition InR x : Sum := fun Z f g => g x.
```

[2]データ構造を表現するとき，*Prop* より *Set* を使う方がいいが，Nat を Nat に適用したいので，その場合には Coq の起動時には-impredicative-set というオプションを指定しなければならない．Emacs では M-x set-variable<ret> coq-prog-args<ret> ("-emacs" "-impredicative-set")<ret>の後に C-cC-x で Coq を再起動.

```
End ProdSum.
Arguments Pair [X Y]. Arguments Fst [X Y]. Arguments Snd [X Y].
Arguments InL [X] Y. Arguments InR X [Y].     (* 型引数を省略できるようにする *)

Definition Pred (N : Nat) :=                      (* 前者関数の定義は工夫が必要 *)
  Fst (N _ (fun p : Prod Nat Nat => Pair (Snd p) (Succ (Snd p)))
           (Pair Zero Zero)).

Theorem Pred_ok : eq_Nat_fun Pred pred. Abort.

(* Nat が Set で定義されているときだけ証明可能 *)
Lemma Nat_of_nat_ok : forall n, Nat_of_nat n _ S O = n. Abort.
End SystemF.
```

練習問題 4.1 *System F* での符号化に関する定理を好きなだけ証明せよ.