

帰納的な定義と型付け

1 プログラムの型付け

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

型判定 $\Gamma \vdash M : \tau$ $\Gamma = x_1 : \tau_1, \dots, x_n : \tau_n$ 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	$\frac{\Gamma \vdash M : \theta \quad \Gamma, x : \theta \vdash N : \tau}{\Gamma \vdash (\text{let } x := M \text{ in } N) : \tau}$
抽象	$\frac{\Gamma, x : \theta \vdash M : \tau}{\Gamma \vdash (\text{fun } x:\theta \Rightarrow M) : \theta \rightarrow \tau}$	不動点	$\frac{\Gamma, f : \theta \rightarrow \tau, x : \theta \vdash M : \tau}{\Gamma \vdash (\text{fix } f(x:\theta) := M) : \theta \rightarrow \tau}$
適用	$\frac{\Gamma \vdash M : \theta \rightarrow \tau \quad \Gamma \vdash N : \theta}{\Gamma \vdash M N : \tau}$	直積	$\frac{\Gamma \vdash M : \tau \quad \Gamma \vdash N : \theta}{\Gamma \vdash (M, N) : \tau \times \theta}$
		射影	$\Gamma \vdash \text{fst} : \tau \times \theta \rightarrow \tau \quad \Gamma \vdash \text{snd} : \tau \times \theta \rightarrow \theta$
抽象*	$\frac{\Gamma, x : S \vdash M : T}{\Gamma \vdash (\text{fun } x:S \Rightarrow M) : \forall x:S, T}$	適用*	$\frac{\Gamma \vdash M : \forall x:S, T \quad \Gamma \vdash N : S}{\Gamma \vdash M N : [N/x]T}$

$x \notin \text{fv}(\tau)$ のときに $\theta \rightarrow \tau = \forall x : \theta, \tau$ 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

$$\frac{\frac{\frac{\Gamma, x : \text{nat} \vdash S : \text{nat} \rightarrow \text{nat} \quad \Gamma, x : \text{nat} \vdash x : \text{nat}}{\Gamma, x : \text{nat} \vdash S x : \text{nat}} \text{適用}}{\Gamma \vdash (\text{fun } x:\text{nat} \Rightarrow S x) : \text{nat} \rightarrow \text{nat}} \text{抽象}}{\Gamma \vdash (\text{fun } x:\text{nat} \Rightarrow S x) O : \text{nat}} \text{適用}} \Gamma \vdash O : \text{nat}$$

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 (型付入計算) 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	\rightarrow
\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 $\langle _ \rangle : \text{命題} \rightarrow \text{型}$ 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、 $\theta = \text{True}$ と $\tau = \text{False}$ にすると、以下の導出が可能になる.

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

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

今回は自然数の定義を見た.

```
Inductive nat : Set := 0 : 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.

```
Inductive t(a1 ... an : Set) : Set :=
  | c1 : τ11 → ... → τ1k1 → t a1 ... an
  ...
  | cm : τm1 → ... → τmkm → t a1 ... an.
```

マッチング

```
Γ ⊢ M : t b1 ... bn
```

```
Γ, xi1 : τi1[b1/a1, ..., bn/an], ..., xiki : τiki[...] ⊢ Mi : τ[ci xi1 ... xiki/x] (1 ≤ i ≤ m)
```

```
Γ ⊢ match M as x return τ with c1 x11 ... x1k1 ⇒ M1 | ... | cm xm1 ... xmkm ⇒ Mm end : τ[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ような関数を作れる. 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.

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.

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`.

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

Check `nat_ind`.

```
: ∀ P : nat -> Prop, P 0 -> (∀ 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.

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。

```

Inductive t :  $\tau_1 \rightarrow \dots \rightarrow \tau_n \rightarrow \text{Prop} :=
| c_1 : \forall (x_1 : \tau_{11}) \dots (x_{k_1} : \tau_{1k_1}), t \theta_{11} \dots \theta_{1n}
\dots
| c_m : \forall (x_1 : \tau_{m1}) \dots (x_{k_m} : \tau_{mk_m}), t \theta_{m1} \dots \theta_{mn}$ 
```

マッチング

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

(* 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=> 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 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, \forall 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 \/. 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。

非可述性 (impredicativity)

Coq の抽象の推論規則をより詳しくみると、Set と Prop の違いが見えて来る。

$$\frac{\Gamma, X : S \vdash M : T \quad \Gamma \vdash \forall X : S, T : \text{Type}_i}{\Gamma \vdash (\text{fun } X : S \Rightarrow M) : \forall X : S, T} \quad \Gamma \vdash \text{Prop} : \text{Set} \quad \Gamma \vdash \text{Set} : \text{Type}_i \quad \Gamma \vdash \text{Type}_i : \text{Type}_{i+1}$$

$$\frac{\Gamma \vdash A : s_1 \quad \Gamma \vdash s_1 : s_2 \quad s_1 \in \{\text{Set}, \text{Prop}, \text{Type}_i\}}{\Gamma \vdash A : s_2} \quad \frac{\Gamma \vdash A : \text{Type}_i \quad \Gamma, X : A \vdash B : \text{Prop}}{\Gamma \vdash \forall X : A, B : \text{Prop}}$$

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

System F 的な表現方法は非可述性を必要としている。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 と同様に非可述的になる。Set の中の排中律 (forall A:Prop, {A}+{~A}) と矛盾するので要注意。

System F のデータ表現

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、自然数に関しては、型なし入計算で有名な 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).

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。CoqIde では、ファイル読み込み後に Tools/Coqtop arguments で -impredicative-set を入力。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。

```

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だが, 非可述的である *)
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.
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 0 = n. Abort.

End SystemF.

練習問題 4.1 *System F* での表現に関する定理を好きなだけ証明せよ.