

帰納的な定義と多相性

1 帰納的な定義

Coqの帰納的なデータ型

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な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.
```

帰納的な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

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

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できる。それを手でやるのは難しいが、作戦 destruct はこのパターンマッチングを構築してくれる。

帰納的な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 より一般的なもので、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 (f1 : forall a, P (inl _ a)) (fr : forall b, P (inr _ b)) =>
fun p => match p as x return P x
  with inl a => f1 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 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.
: forall P : nat -> Prop, P 0 -> (forall n : nat, P n -> P (S n)) ->
  forall n : nat, P n

```

前回ならった induction n という作戦はこの補題を利用するが、作業がかなり複雑である。

1. n を含む全ての仮定をゴールに戻す. (作戦 `revert H1 ... Hn` でも手動でできる)
2. n の型を見て、型が $t a_1 \dots a_n$ ならば、`apply (t_ind a_1 ... a_n)` を行う. (このステップは作戦 `elim n` でもできる)
厳密には、ゴールのソートによって `t_rec`, `t_ind`, `t_rect` のいずれかが使われる.
3. 新しくできた各ゴールに対して、仮定をおく. (`intros` にあたる)

`destruct` と `induction/elim` はよく似ているが、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 plus_0 : forall n, plus n 0 = n.
Proof.
  apply nat_ind. reflexivity.
  intros n IHn.
  simpl.
  rewrite IHn.
  reflexivity.
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。

(* 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 : forall n, even (n + n).
Proof.
  induction n.
  apply even_0.
  simpl.
  rewrite <- plus_n_Sm.
  apply even_SS.

```

```
assumption.  
Qed.
```

(* 帰納的述語に対する帰納法もできる *)

```
Theorem even_plus : forall m n, even m -> even n -> even (m + n).
```

```
Proof.
```

```
  intros m n Hm Hn.  
  induction Hm.  
  apply Hn.  
  simpl.  
  apply even_SS.  
  assumption.
```

```
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 destruct 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_rect.  
fun (P : Type) (f : False) => match f return P with end  
  : forall P : Type, 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 : forall (P Q : Prop),  
  P -> ~P -> Q.
```

```
Proof.
```

```
  intros P Q p np.  
  elim np.  
  assumption.
```

```
Qed.
```

便利な作戦

役に立つ作戦を紹介する。

- assert ($H : prop$). 命題 $prop$ を新しいゴールとして定め、証明が完成したら名前 H で仮定に加える。
- auto. Hint で与えられた定理を使ってゴールを解こうとする。apply を 5 回までやってくれるので、証明がとても短くなることもある。
Hint Resolve lemma. lemma を自動的に使う。
Hint Constructors inductive. inductive の全ての構成子を自動的に呼ぶ。

Hint Constructors even.

```
Theorem even_plus' : forall m n,
  even m -> even n -> even (m + n).
Proof.
  intros m n Hm Hn.
  induction Hm; simpl; info auto.
Qed.
```

- `constructor`. ゴールの先頭の構成子を `apply` する.

```
Goal even 0.
  constructor.
```

- `discriminate`. 仮定の中で構成子の異なる値同士の方方程式を見つけ、枝狩りをする.

```
Goal 1 <> 2.
  intros eq.
  discriminate.
```

- `inversion H`. 帰納的述語で表現される仮定 H に対して場合分けを行う. それによって引数が定まったり, ゴールが解決されたりする.

```
Goal ~even 1.
  intros ne.
  inversion ne.
```

```
Goal forall n, even (S (S n)) -> even n.
  intros n H.
  inversion H.
  assumption.
```

- `repeat` 作戦. 同じ作戦を可能な限り繰り返す. 例えば, `repeat split` で連言をばらばらにできる.
- `pattern expr`. 式 $expr$ を含む現在のゴールを述語として見なす. さらに `at pos` を付けると, 特定の位置の出現のみが対象となる.

```
Goal (1 + 0 = 1).
  intros.
  pattern 1.
  (fun n : nat => n + 0 = n) 1
  apply plus_0.
```

練習問題 1.1 以下の定理を証明せよ.

```
Module Ex4_1.
  Inductive odd : nat -> Prop :=
    | odd_1 : odd 1
    | odd_SS : forall n, odd n -> odd (S (S n)).

  Theorem even_odd : forall n, even n -> odd (S n).
  Theorem odd_even : forall n, odd n -> even (S n).
  Theorem even_not_odd : forall n, even n -> ~odd n.
  Theorem even_or_odd : forall m, even m \/ odd m.
```

```
Theorem odd_odd_even : forall m n, odd m -> odd n -> even (m+n).
End Ex4_1.
```

2 多相性

今回は全称 $\forall x:S, T$ には様々なパターンがあると説明した。
今回はそれを具体的に見る。もしも $\vdash v : \forall x:S, T$ ならば

- $S : \text{Set}$ かつ $T : \text{Prop}$ のとき
 v は命題の証明: S の任意の項について T が成り立つ
 例: $\forall n : \text{nat}, 2 * n = n + n$
- $S : \text{Set}$ かつ $T : \text{Prop}$ のとき
 v は命題文を返す関数
 例: $\text{fun } n : \text{nat} => 2 * n = n + n$
- $S = \text{Prop}$ かつ $T : \text{Prop}$ のとき
 v は任意の命題について成り立つ命題の証明
 例: $\forall P : \text{Prop}, P \rightarrow P$
- $S = \text{Prop}$ かつ $T = \text{Prop}$ のとき
 命題引数を取る述語の定義
 例: $\text{and} : \forall P Q : \text{Prop}, \text{Prop}$
- $S = \text{Set}$ かつ $T : \text{Set}$ のとき
 v は普通に関数
 例: $\text{fun } x => x + 1 : \text{nat} \rightarrow \text{nat}$
- $S = \text{Set}$ かつ $T : \text{Set}$ のとき
 v は多相的な値: 任意のデータ型 X について、 v は型 T をもつ
 例: $\text{fun } (X : \text{Set}) (x : X) => x : \forall X : \text{Set}, X \rightarrow X$
- $S = \text{Set}$ かつ $T = \text{Set}$ のとき
 型引数を取るデータ型の定義
 例: $\text{prod} : \forall X Y : \text{Set}, \text{Set}$
- $S : \text{Set}$ かつ $T = \text{Set}$ のとき
 値から型を作る関数
 例:

```
Fixpoint vec (n:nat) (T:Set) :=
  match n with
  | 0 => unit
  | S m => (vec m T * T)%type
  end.
Check ((tt, 1, 2, 3) : vec 3 nat).
```

この中で特に $S = \text{Set}$ または $S = \text{Prop}$ の場合を多相的という。関数や定理が任意のデータ型または任意の命題について定義される。

標準ライブラリのリスト 典型的な多相的なデータ構造として、リストが挙げられる。

```
Require Import List.
```

```
Module Lists.
```

```
Print list.
```

```
Inductive list (A : Type) : Type :=
  nil : list A | cons : A -> list A -> list A (* 型引数がある *)
```

```
Definition l1 := 1 :: 2 :: 3 :: 4 :: nil. (* cons は :: と書ける *)
```

```
Print l1.
```

```
l1 = 1 :: 2 :: 3 :: 4 :: nil
    : list nat
```

```
Definition hd {A:Set} (l:list A) :=
  match l with
  | cons a _ => a
  end. (* { } で A が省略可能になる *)
```

```
Error: Non exhaustive pattern-matching: no clause found for pattern nil
```

```
Definition hd {A:Set} (d:A) (l:list A) :=
  match l with
  | cons a _ => a
  | nil => d (* nil のときは d を返す *)
  end.
```

```
Eval compute in hd 0 l1.
```

```
= 1
  : nat
```

```
Fixpoint length {A:Set} (l : list A) :=
  match l with
  | nil => 0
  | cons _ l' => 1 + length l'
  end. (* 長さ *)
```

```
Eval compute in length l1.
```

```
= 4
  : nat
```

```
Fixpoint append {A:Set} (l1 l2:list A) struct l1 : list A :=
  match l1 with
  | nil => l2
  | a :: l' => a :: append l' l2
  end. (* 結合 *)
```

```
Eval compute in append (1 :: 2 :: nil) (3 :: 4 :: nil).
```

```
= 1 :: 2 :: 3 :: 4 :: nil
  : list nat
```

```
Fixpoint fold_right {A B:Set} (f : A -> B -> B) (z : B) (l : list A) :=
  match l with
  | nil => z
  | a :: l' => f a (fold_right f z l')
  end.
```

```
(* fold_right  $\oplus$  b (a1 :: ... :: an) = a1  $\oplus$  ...  $\oplus$  an  $\oplus$  b *)
```

```

Definition sum := fold_right plus 0.
Eval compute in sum l1.
  = 10
  : nat

```

```

Eval compute in fold_right mult 1 l1.
  = 24
  : nat
End Lists.

```

練習問題 2.1 多項式を係数のリストとして定義する。 \mathbf{Z} 上の多項式をある点 x で計算する関数を定義せよ。

```

Module Ex4_2.
  Require Import ZArith.
  Open Scope Z_scope.

  Definition eval_poly (p : list Z) (x : Z) := ...
  Eval compute in eval_poly (1 :: 2 :: 3 :: nil) 5. (* = 1 + 2*5 + 3*5*5 *)
End Ex4_2.

```

多相性と論理 Coq の論理演算子は Inductive で定義されているが、実は多相的な論理式として定義できる。

```

Definition and' (P Q : Prop) := forall (X : Prop), (P -> Q -> X) -> X.
Definition or' (P Q : Prop) := forall (X : Prop), (P -> X) -> (Q -> X) -> X.
Definition False' := forall (X : Prop), X.
Definition Equal' (T : Type) (x y : T) := forall (P : T -> Prop), P x <-> P y.

```

```

Theorem and'_ok : forall P Q, and' P Q <-> P /\ Q.

```

Proof.

```

  split.
  intros.
  apply H.
  split; assumption.
  intros pq X pqx.
  destruct pq as [p q].
  apply pqx; assumption.

```

Qed.

```

Theorem or'_ok : forall P Q, or' P Q <-> P \/ Q.

```

```

Theorem False'_ok : False' <-> False.

```

```

Theorem Equal'_ok : forall T x y, Equal' T x y <-> x = y.

```

練習問題 2.2 or'_ok、False'_ok および Equal'_ok を証明せよ。