

# 帰納的な定義

## 1 前回の課題

### 練習問題 1.1

Section Coq3.

```
Variable A : Set.
Variable R : A -> A -> Prop.
Variable P Q : A -> Prop.
```

```
Theorem exists_postpone :
  (exists x, forall y, R x y) -> (forall y, exists x, R x y).
```

Proof.

```
  intros H y.
  destruct H as [x r].
  exists x.
  apply r.
```

Qed.

```
Theorem or_exists : (ex P)  $\vee$  (ex Q) -> exists x, P x  $\vee$  Q x.
```

Proof.

```
  intros H.
  destruct H as [[x p]|[x q]].
  exists x; left; assumption.
  exists x; right; assumption.
```

Qed.

```
Hypothesis classic : forall P,  $\sim\sim$ P -> P.
```

```
Lemma remove_c : forall (a : A),
  (forall x y, Q x = Q y) ->
  (forall c, ((exists x, P x) -> P c) -> Q c) -> Q a.
```

Proof.

```
  intros a Eq H.
  apply classic.
  intros nq.
  apply nq.
  apply H.
  intros xp.
  destruct xp as [b pb].
  elimtype False.
  apply nq.
  rewrite (Eq a b).
  apply H.
  intros _.
  assumption.
```

Qed.

End Coq3.

## 練習問題 2.1

Lemma plus\_0 : forall n, plus n 0 = n.

Proof.

```
intros n.
induction n.
  reflexivity.
simpl.
rewrite IHn.
reflexivity.
```

Qed.

Lemma plus\_m\_Sn : forall m n, plus m (S n) = S (plus m n).

Proof.

```
intros m n.
induction m.
  reflexivity.
simpl.
rewrite IHm.
reflexivity.
```

Qed.

Lemma plus\_comm : forall m n, plus m n = plus n m.

Proof.

```
intros.
induction m.
  simpl.
  rewrite plus_0.
  reflexivity.
simpl.
rewrite IHm.
rewrite plus_m_Sn.
reflexivity.
```

Qed.

## 2 帰納的な定義

Coq の帰納的データ型 前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.<sup>1</sup>

```
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.
```

---

<sup>1</sup>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,  $\text{prod } A \ B$  は  $A * B$  として表示され,  $\text{pair } a \ b$  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。

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

マッチング

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

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) :=
  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 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、 $\text{apply } (t\_ind \ a_1 \dots a_n)$  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 False : Prop := .
```

and や or 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.

**練習問題 2.1** 以下の定理を証明せよ.

```
Section Coq4.
```

```
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 Coq4.
```