

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

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帰ることができる.

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 : ti1[b1/a1, ..., bn/an], ..., xiki : tiki[...] ⊢ 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

```

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を作れる。それを手でやるのは無塚私意が、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 {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 a₁...a_n ならば、apply (t_ind a₁...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.
- `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.

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.
End Coq4.

```

練習問題 2.2 帰納的述語を定義した上で以下の定理を証明せよ.

Require Import List.

Section Sort.

Variables (A:Set)(le:A->A->Prop).

Variable le_refl: forall x, le x x.

Variable le_trans: forall x y z, le x y -> le y z -> le x z.

Variable le_total: forall x y, {le x y}+{le y x}.

Inductive le_list x : list A -> Prop :=

| le_nil : le_list x nil

| le_cons : forall y l, le x y -> le_list x l -> le_list x (y::l).

Inductive sorted : list A -> Prop :=

| sorted_nil : sorted nil

| sorted_cons : forall a l, le_list a l -> sorted l -> sorted (a::l).

Hint Constructors le_list sorted.

Fixpoint insert a (l: list A) :=

match l with

| nil => (a :: nil)

| b :: l' => if le_total a b then a :: l else b :: insert a l'

end.

Lemma le_list_insert : forall a b l,

le a b -> le_list a l -> le_list a (insert b l).

Lemma le_list_trans : forall a b l,

le a b -> le_list b l -> le_list a l.

Theorem insert_ok : forall a l, sorted l -> sorted (insert a l).

End Sort.

注: {le x y}+{le y x} は sumbool 型の *Notation* である. ゴールの中に le_total a b があるとき, destruct (le_total a b) で場合分けができる.