

プログラムの証明1

1 前回の課題

```

Module Ex4_2.
...
Fixpoint eval_poly (p : list Z) (x : Z) :=
  match p with
  | nil => 0
  | a :: p' => a + (eval_poly p' x) * x
  end.
End Ex4_2.

Section Coq5.
...
Lemma reduce_fold : forall l, reduce l = fold_right op e l.
Proof. induction l; simpl; auto. Qed.

Lemma reduce_app : forall l1 l2, reduce (l1 ++ l2) = op (reduce l1) (reduce l2).
Proof.
  intros l1 l2.
  induction l1; simpl; auto.
  rewrite IHl1; auto.
Qed.
End Coq5.

Section F.
...
Theorem or'_ok : forall P Q, or' P Q <-> P Q.
Proof.
  split; intros H.
  apply H; auto.
  intros X HP HQ.
  destruct H; auto.
Qed.

Theorem False'_ok : False' <-> False.
Proof.
  split; intros H. apply H. elim H.
Qed.

Theorem Equal'_ok : forall T x y, Equal' T x y <-> x = y.
Proof.
  split; intros H.
  destruct (H (fun x => x = y)). apply H. reflexivity.
  split; intros HP.
  rewrite <- H. assumption.
  rewrite H. assumption.
Qed.
End F.

```

2 依存和

存在 (\exists) は帰納型の特殊な例である .

```
Print ex.
```

```
Inductive ex (A : Type) (P : A -> Prop) : Prop :=
  ex_intro : forall x : A, P x -> ex P.
```

`ex (fun x:A => P(x))` を `exists x:A, P(x)` と書いてもいい .

この定義を見ると, `ex P = $\exists x, P(x)$` は x と $P(x)$ の対でしかない . 対の第 2 要素に第 1 要素が現れているので, この積を「依存和」という (元々依存のある関数型を定義域を添字とした依存積と見なすなら, こちらは A を添字とする直和集合になる)

既に見ているように, 証明の中で依存和を構築する時に, `exists` という作戦を使う .

```
Require Import Arith.
```

```
Print le.
```

```
Inductive le (n : nat) : nat -> Prop :=
  le_n : n <= n / le_S : forall m : nat, n <= m -> n <= S m.
```

```
Lemma exists_pred : forall x, x > 0 -> exists y, x = S y.
```

```
Proof.
```

```
  intros x Hx.
```

```
  destruct Hx.
```

```
    exists 0. reflexivity.
```

```
  exists m. reflexivity.
```

```
Qed.
```

上記の `ex` は `Prop` に住むものなので, 論理式の中でしか使えない . しかし, プログラムの中で依存和を使いたい時もある . この時には `sig` を使う .

```
Print sig.
```

```
Inductive sig (A : Type) (P : A -> Prop) : Type :=
  exist : forall x : A, P x -> sig P.
```

`sig (fun x:T => Px)` は $\{x:T \mid Px\}$ と書く . `ex` と同様に, 具体的な値は `exists` で指定する . こういう条件付きな値を扱う安全な関数が書ける .

```
Definition safe_pred x : x > 0 -> {y | x = S y}.
```

```
  intros x Hx.
```

```
  destruct Hx.
```

```
Error: Case analysis on sort Set is not allowed for inductive definition le.
```

値を作ろうとしている時, `Prop` の証明に対する場合分けが行えない . 普通の値に対する場合分けを行わなければならない .

```
Definition safe_pred x : x > 0 -> {y | x = S y}.
```

```
  intros x Hx.
```

```
  destruct x as [|x'].
```

```
    elim (le_Sn_0 0).
```

(* この場合が不要であることの証明 *)

```
    exact Hx.
```

```
  exists x'.
```

```
  reflexivity.
```

(* 条件の証明 *)

```
Defined.
```

(* 定義を透明にし, 計算に使えるようにする *)

証明された関数を OCaml の関数として輸出できる . その場合, `Prop` の部分が消される .

```

Extraction safe_pred.
f(** val safe_pred : nat -> nat **)
let safe_pred = function
  | 0 -> assert false (* absurd case *)
  | S x' -> x'

```

二択の依存和

通常の実偽値に対して、書く場合に条件を付ける依存和も定義されている。

```

Print sumbool.
Inductive sumbool (A B : Prop) : Set :=
  left : A -> {A} + {B} | right : B -> {A} + {B}.

```

上にもあるように、`sumbool A B` は `{A}+{B}` と書ける。

これを使うと、条件を判定するような関数の型が簡単に書ける。

```

Check le_lt_dec.
      : forall n m : nat, {n <= m} + {m < n}

```

3 整列の証明

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.

```

```

Fixpoint isort (l : list A) : list A :=
  match l with
  | nil => nil
  | a :: l' => insert a (isort l')
  end.

```

```

Lemma le_list_insert : forall a b l,
  le a b -> le_list a l -> le_list a (insert b l).
Proof.

```

```

intros.
induction H0.
  simpl. info auto.
simpl.
destruct (le_total b y). (* le_total の結果の場合分け *)
  info auto.
info auto.
Qed.

Lemma le_list_trans : forall a b l,
  le a b -> le_list b l -> le_list a l.
Proof.
  intros.
  induction H0. constructor.
  info eauto using le_trans. (* le_trans をヒントに加えて自動証明 *)
Qed.

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

Parameter isort_ok : forall l, sorted (isort l).

Inductive Permutation : list A -> list A -> Prop :=
| perm_nil: Permutation nil nil (* リストの組み替え *)
| perm_skip: forall x l l',
  Permutation l l' -> Permutation (x::l) (x::l')
| perm_swap: forall x y l, Permutation (y::x::l) (x::y::l)
| perm_trans: forall l l' l'',
  Permutation l l' -> Permutation l' l'' -> Permutation l l''.

Parameter Permutation_refl : forall l, Permutation l l.
Parameter insert_perm : forall l a, Permutation (a :: l) (insert a l).
Parameter isort_perm : forall l, Permutation l (isort l).

Definition safe_isort : forall l, {l'|sorted l' /\ Permutation l l'}.
  intros. exists (isort l).
  split. apply isort_ok. apply isort_perm.
Defined.
End Sort.

Check safe_isort.

Definition le_total : forall m n, {m <= n} + {n <= m}.
  intros. destruct (le_lt_dec m n). auto. auto with arith.
Defined.

Definition isort_le := safe_isort nat le le_total.
(* 全てを証明すると, le_refl と le_trans も必要になる *)

Eval compute in proj1_sig (isort_le (3 :: 1 :: 2 :: 0 :: nil)).
= 0 :: 1 :: 2 :: 3 :: nil

Extraction "isort.ml" isort_le. (* コードをファイル isort.ml に書き込む *)

```

練習問題 3.1 Parameter を Theorem に変え, 証明を完成させよ.