

1 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.

```
Lemma exists_pred x : x > 0 -> exists y, x = S y.
Proof.
  case x => // n _ . (* case: x と同じだが項が読みやすい *)
  by exists n.
Qed.
Print exists_pred.
exists_pred =
fun x : nat =>
match x as n return (0 < n -> exists y : nat, n = y.+1) with
| 0 =>
  fun H : 0 < 0 =>
  let HO : False :=
    eq_ind (0 < 0) (fun e : bool => if e then False else True) I true H in
  False_ind (exists y : nat, 0 = y.+1) HO (* 0 はありえない *)
| n.+1 =>
  fun _ : 0 < n.+1 => ex_intro (fun y : nat => n.+1 = y.+1) n (erefl n.+1)
end (* n.+1 のとき n を返す *)
: forall x : nat, 0 < x -> exists y : nat, x = y.+1
Require Extraction.
Extraction exists_pred. (* 何も抽出されない *)
```

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 $\{x:T \mid Px\}$ とも書く. `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}.
  case x => // n _ . (* exists_pred と同じ *)
  by exists n. (* こちらも exists を使う *)
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.

```
Require Extraction.
```

```

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

```

2 Hint と auto

証明が冗長になることが多い。auto は簡単な規則で証明を補完しようとする。具体的には、auto は 仮定や Hint lem1 lem2 ... で登録した定理を apply で適用しようとする。これらを組み合わせ、深さ 5 の項まで作れる (auto n で深さ n にできる)。info_auto で使われたヒントを表示させる事もできる。

Hint Constructors で帰納型を登録すると、各構成子が定理として登録される。また、auto using lem1, lem2, ... で一回だけヒントを追加することもできる。

auto で定理が適用されるために、全ての変数が定理の結論に現れる必要がある。 eauto を使うと simple apply が eapply に変わるので、決まらない変数が変数のまま残せる。その代わり、可能な導出木が増えるので、探索が中々終わらない場合もある。

3 整列の証明

Section Sort.

Variables (A:Set) (le:A->A->bool). (* データ型 A とのその順序 le *)

(* 既に整列されたリスト l の中に a を挿入する *)

```

Fixpoint insert a (l: list A) :=
  match l with
  | nil => (a :: nil)
  | b :: l' => if le a b then a :: l else b :: insert a l'
end.

```

(* 繰り返しの挿入でリスト l を整列する *)

```

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

```

(* le は推移律と完全性をみたす *)

```

Hypothesis le_trans: forall x y z, le x y -> le y z -> le x z.
Hypothesis le_total: forall x y, ~~ le x y -> le y x.

```

(* le_list x l : x はあるリスト l の全ての要素以下である *)

```

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).

```

(* sorted l : リスト l は整列されている *)

```

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. (* auto の候補にする *)

Lemma le_list_insert a b l :
  le a b -> le_list a l -> le_list a (insert b l).
Proof.
  move=> leab; elim => {l} [|c l] /=. info_auto.
  case: ifPn. info_auto. info_auto.
Qed.

Lemma le_list_trans a b l :
  le a b -> le_list b l -> le_list a l.
Proof.
  move=> leab; elim. info_auto.
  info_eauto using le_trans. (* 推移律は eauto が必要 *)
Qed.

Hint Resolve le_list_insert le_list_trans. (* 補題も候補に加える *)

Theorem insert_ok a l : sorted l -> sorted (insert a l). Admitted.
Theorem isort_ok l : sorted (isort l). Admitted.

(* Permutation l1 l2 : リスト l2 は l1 の置換である *)
Inductive Permutation : list A -> list A -> Prop :=
| perm_nil: Permutation nil nil
| 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''.

Hint Constructors Permutation.

Theorem Permutation_refl l : Permutation l l. Admitted.
Theorem insert_perm l a : Permutation (a :: l) (insert a l). Admitted.
Theorem isort_perm l : Permutation l (isort l). Admitted.

(* 証明付き整列関数 *)
Definition safe_isort l : {l'|sorted l' /\ Permutation l l'}.
  exists (isort l).
  auto using isort_ok, isort_perm.
Defined.
Print safe_isort.
End Sort.

Check safe_isort. (* le と必要な補題を与えなければならない *)
Extraction leq. (* mathcomp の eqType の抽出が汚ない *)

Definition leq' m n := if m - n is 0 then true else false.
Extraction leq'. (* こちらはすっきりする *)

Lemma leq'E m n : leq' m n = (m <= n).
Proof. rewrite /leq' /leq. by case: (m-n). Qed.

```

```
Lemma leq'_trans m n p : leq' m n -> leq' n p -> leq' m p.  
Proof. rewrite !leq'E; apply leq_trans. Qed.
```

```
Lemma leq'_total m n : ~~ leq' m n -> leq' n m. Admitted.
```

```
Definition isort_leq := safe_isort nat leq' leq'_trans leq'_total.
```

```
Eval compute in proj1_sig (isort_leq (3 :: 1 :: 2 :: 0 :: nil)).  
= [:: 0; 1; 2; 3] : seq nat
```

```
Extraction "isort.ml" isort_leq.
```

練習問題 3.1 1. Admitted を Proof に変え, 証明を完成させよ.

2. le_list を以下のように一般化できる.

```
Inductive All (P : A -> Prop) : list A -> Prop :=  
  | All_nil : All P nil  
  | All_cons : forall y l, P y -> All P l -> All P (y::l).
```

このとき, All (le a) l が le_list a l と同じ意味になる.

こちらを使うように証明を修正せよ.