Jacques Garrigue, 2024年5月8日

単一化・自動化とプログラムの証明

1 前回の課題

```
Lemma eval_polyS n l x : eval_poly (S n) l x = x * eval_poly n l x.
Proof.
  elim: l n => //= a l IH n.
  by rewrite IH !(multC x) multnDr multA.
Qed.

Lemma horner_ok l x : horner l x = eval_poly 0 l x.
Proof.
  suff : forall n, exp x n * horner l x = eval_poly n l x.
    move <-. by rewrite /= plusn0.
  elim: l => //= a l IH n.
  by rewrite eval_polyS multC multnDr -IH (multC (exp x n)) multA.
Qed.

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

Theorem even_odd n : even n -> odd (S n).
Proof.
  elim: n/. apply: odd_1.
  move=> n Hn. apply: odd_SS.
Restart.
  by elim: n/ => [|n Hn]; constructor.
Qed.

Theorem odd_even n : odd n -> even (S n).
Proof.
  elim: n/. apply/even_SS/even_0.
  move=> n Hn. apply/even_SS.
Restart.
  by elim: n/ => [|n Hn]; do! constructor.
Qed.

Theorem even_not_odd n : even n -> ~odd n.
Proof.
  elim: n/.
  - move H0: 0 => z Ho. by case: Ho H0.
```

```
  - move=> n _ Ho.
    move HSS: (S (S n)) => m Hm.
    case: m/ Hm HSS Ho => // m Hm [] -> //.
Restart.
  by elim: n/ => [|n _ IH] Ho; inversion Ho.
Qed.
End Odd.
```

2 単一化

```
Lemma test x : 1 + x = x + 1.
  Check plusC.
     : ∀ x y : nat, x + y = y + x
  apply: addnC.
Abort.                                  (* 定理を登録せずに証明を終わらせる *)
```

ゴールと定理 addnC の字面が異なっているのに，ここでなぜ apply が使えるのか．
実は apply は複数のことをしている．

1. 定理の中の∀で量化されている変数を「単一化用の変数」に置き換える
2. 置き換えられた定理の結論をゴールと「単一化」する
3. 定理に前提があれば,「単一化」の結果を代入した前提を新しいゴールにする．

ここでいう単一化とは,（単一化用の）変数を含んだ項同士をその変数の値を定めることで同じものにする．例えば，1 + x = x + 1 と $?m+?n = ?n+?m$ を単一化するには，$?m = 1, ?n =$ x と定めれば良い．
Coq 本来の apply で変数が定まらないと，エラーになる．しかし，SSReflect の apply: や apply/ を使えば，変数が残せる．

```
Lemma test x y z : x + y + z = z + y + x.
  Check eq_trans.
     : ∀ (A : Type) (x y z : A), x = y -> y = z -> x = z
  apply eq_trans.
Error: Unable to find an instance for the variable y.
  apply: eq_trans.            (* y が結論に現れないので，apply: に変える *)
   x + y + z = ?Goal
  apply: plusC.
  apply: eq_trans.
  Check f_equal.
     : ∀ (A B : Type) (f : A -> B) (x y : A), x = y -> f x = f y
  apply: f_equal.                                  (*   x + y = ?Goal0 *)
  apply: plusC.
  apply: plusA.
Restart.                                              (* 証明を元に戻す *)
  rewrite plusC.                                (* rewrite も単一化を使う *)
```

```
  rewrite (plusC x).
  apply: plusA.
Abort.
```

一階の項に関して，単一化は最適な代入を見つけてくれるという結果が知られている．

定義 1 ある代入 σ_1 が代入 σ_2 **より一般的**であるとは，σ_{12} が存在し，$\sigma_2 = \sigma_{12} \circ \sigma_1$ であることをいう．

定義 2 単一化問題 $\{t_1 = t'_1, \ldots, t_n = t'_n\}$ に対して，$\sigma(t_i) = \sigma(t'_i)$ であるとき，σ はその単一化問題の**単一子**だという．

定理 1 任意の単一化問題に対して，解が存在するときには**最も一般的な単一子**を返し，存在しないときにはそれを報告するアルゴリズムが存在する．

具体的には，アルゴリズム $\mathcal{U}$ を以下の買い替え規則で定義できる．ここでは定数記号を 0 引数の関数記号と同一視する．

$$
\begin{array}{ll}
E \cup \{f(t_1, \ldots, t_n) = f(t'_1, \ldots, t'_n)\} \to E \cup \{t_1 = t'_1, \ldots, t_n = t'_n\} & \\
E \cup \{f(t_1, \ldots, t_n) = g(t'_1, \ldots, t'_m)\} \to \bot & f \neq g \\
E \cup \{x = t\} \to [t/x]E \cup \{x = t\} \quad x \in vars(E) \wedge (t = y \Rightarrow y \in vars(E)) \wedge x \notin vars(t) & \\
E \cup \{x = x\} \to E & \\
E \cup \{x = t\} \to \bot & x \in vars(t)
\end{array}
$$

E が書き換えを繰り返し，書き換えられない E' になれば，その E' が $\{x_1 = t''_1, \ldots, x_m = t''_m\}$ という形であり，それを代入 $\sigma = [t''_1/x_1, \ldots, t''_m/x]$ と見なし，$\mathcal{U}(E) = \sigma$．このときに，任意の $t = t' \in E$ について，$\sigma(t) = \sigma(t')$，かつ E の単一子になる任意の σ' について，σ が σ' より一般的である．また，$E' = \bot$ のとき，E には解がない．

上記の $\mathcal{U}$ は一階の項のためのものであるが，Coq はそれより強い**高階単一化**[1]を行っている．しかし，この場合には最も一般的な解が存在するとは限らない．

```
Goal
  (∀ P : nat -> Prop, P 0 -> (∀n, P n -> P (S n)) -> ∀n, P n) ->
     ∀ n m, n + m = m + n.
  move=> H n m.                                  (* 全ての変数を仮定に *)
  apply: H.                                            (* n + m = 0 *)
Restart.
  move=> H n m.
  pattern n.                              (* pattern で正しい述語を構成する *)
  apply: H.                                          (* 0 + m = m + 0 *)
Restart.
  move=> H n.                               (* forall n を残すとうまくいく *)
  apply: H.                                          (* n + 0 = 0 + n *)
Abort.
```

単一化は様々な作戦で使われている．apply 以外に elim，rewrite や set が挙げられる．

[1] 1970 年代に高階単一化の可能性を証明した Gérard Huet は Coq の最初のプロジェクトリーダーだった

3 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 Resolve 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て，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 (apply:) 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．

4 依存和

存在 (∃) は帰納型の特殊な例である．

```
Print ex.
Inductive ex (A : Type) (P : A -> Prop) : Prop :=
    ex_intro : ∀ 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．

```
Theorem 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 (n <> 0 -> exists y : nat, n = S y) with
| 0 => fun H : 0 <> 0 =>                          (* 0はありえない *)
    False_ind (exists y : nat, 0 = S y) (H eq_refl)
| S n => fun _ : S n <> 0 =>                  (* S n のとき n を返す *)
    ex_intro (fun y : nat => S n = S y) n eq_refl
end
     : ∀ x : nat, x <> 0 -> exists y : nat, x = S y
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 : ∀ x : A, P x -> sig P.
```

sig (fun x:T => Px) は {x:T | 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．

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

5 整列の証明

リストにおける整列ということで，ライブラリから ssreflect に加えて List と ssrbool をインポートします．前者はリストの記法のためで，後者はブール値の操作のためである．特に以下の機能を使う．

ブール演算 否定は ~~, 論理積は &&, 論理和は || と書ける．

ブール値を命題として扱う 命題が期待されている場所に bool 型の値 b を書くと，自動的に b = true として解釈される．たとえば，b_1 b_2 b_3 : bool の元で $b_1 \to b_2 \to b_3$ は $b_1 = \mathtt{true} \to b_2 = \mathtt{true} \to b_3 = \mathtt{true}$ という意味になる．

if 文の条件に対する場合分け case: ifPn という形でゴールの中の if 文の条件に対する場合分けができる．

証明のあらまし

```
Require Import ssreflect ssrbool List.

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 は推移律をみたす完全性順序である *)
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: l/. 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 *)

Definition leq m n := if m - n is 0 then true else false.
Extraction leq.                                        (* 効率がいい *)

Lemma leqSS m n : leq (S m) (S n) = leq m n. Admitted.

Lemma leq_trans m n p : leq m n -> leq n p -> leq m p.
Proof.
  elim: m n p => //= m IH.
Admitted.

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 :: nil : list nat

Extraction "isort.ml" isort_leq.
```

練習問題 5.1　*1.* Admitted を Qed 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．