Jacques Garrigue, 2014 年 5 月 19 日

プログラムの証明1

1 単一化と自動化

```
Require Import Arith.
Goal forall x, 1 + x = x + 1.          (* Goal で名前のない定理を証明する *)
  intros.
   1 + x = x + 1
  Check plus_comm.
     : forall n m : nat, n + m = m + n
  apply plus_comm.
Abort.                                 (* 定理を登録せずに証明を終わらせる *)
```

ゴールと定理 plus_comm 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．

apply で変数が定まらないと，エラーになる．しかし，eapply を使えば，変数が残せる．

```
Goal forall x y z, x + y + z = z + y + x.
  intros.
  Check eq_trans.
     : forall (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.
  eapply eq_trans.            (* y が結論に現れないので，eapply に変える *)
   x + y + z = ?13
  apply plus_comm.
  eapply eq_trans.
  Check f_equal.
     : forall (A B : Type) (f : A -> B) (x y : A), x = y -> f x = f y
  apply f_equal.                                    (* ?f = plus z *)
  apply plus_comm.
  apply plus_assoc.
Restart.                                            (* 証明を元に戻す *)
  intros.
  rewrite plus_comm.                          (* rewrite も単一化を使う *)
  rewrite (plus_comm x).
```

```
  apply plus_assoc.
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
  (forall P : nat -> Prop,
    P 0 -> (forall n, P n -> P (S n)) -> forall n, P n)
  -> forall n, n + 1 = 1 + n.
  intros H n.                                  (* 全ての変数を intro する *)
  apply H.
Restart.
  intros H n.
  pattern n.                                (* pattern で正しい述語を構成する *)
  apply H.
Restart.
  intros H.                                   (* forall n を残すとうまくいく *)
  apply H.
Abort.
```

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

inversion でも単一化が使われるが，このときは特殊な単一化用の変数ではなく，通常の変数が使われる．具体的には，inversion H は仮定 H で使われる帰納型を調べ，その型の各場合について

[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

1. その場合に現れる変数を新しい変数として導入し，前提も仮定に加える
2. その場合の結論を H の型と単一化する
3. 単一化が失敗したら，この場合は否定される
4. 成功したら，得られた等式を仮定に加え，ゴールにその代入をかける

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 lem 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 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．

rewrite

rewrite でも単一化が使われる．

```
Section rewrite.
Variable x : nat.
Goal 5 + x + 0 = 5 + x.
  rewrite plus_comm.
Abort.
```

また, <- で等式を逆向きに使ったり, , で複数の等式を使ったりできる.

```
Check plus_assoc.
Goal 5 + x + 0 = 5 + x.
  rewrite <- plus_assoc, (plus_comm x).
Abort.
End rewrite.
```

2 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 : 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 | 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}.
  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_O 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.
(** val safe_pred : nat -> nat **)
let safe_pred = function
  | O -> assert false (* absurd case *)
  | S x' -> x'
```

二択の依存和

通常の真偽値に対して，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．