Jacques Garrigue, 2019 年 5 月 22 日

単一化と数論

1　単一化と自動化

```
Lemma test x : 1 + x = x + 1.
  Check [eta addnC].
     : ∀ 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 etrans.
     : ∀ (A : Type) (x y z : A), x = y -> y = z -> x = z
  apply etrans.
Error: Unable to find an instance for the variable y.
  apply: etrans.                    (* y が結論に現れないので，apply: に変える *)
   x + y + z = ?Goal
  apply: addnC.
  apply: etrans.
  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: addnC.
  apply: addnA.
Restart.                                                (* 証明を元に戻す *)
  rewrite addnC.                                 (* rewrite も単一化を使う *)
  rewrite (addnC x).
  apply: addnA.
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．

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

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．

2 最大公約数の計算

ユークリッドが発明した互除法による最大公約数の計算は多分世界最古のアルゴリズムの一つである．その正しさを証明する．

```
let rec gcd m n =
  if m = 0 then n else gcd (n mod m) m
```

最大公約数の厳密な定義は

$$
q \text{ は } m \text{ と } n \text{ の最大公約数である} \Leftrightarrow \left\{ \begin{array}{l} q \mid m \ \wedge \ q \mid n \\ \forall q', (q' \mid m \ \wedge \ q' \mid n) \Rightarrow q' \mid q \end{array} \right.
$$

[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

二つ目に関して，本来は $q' \leq q$ のはずだが，$q' \mid q$ の方が証明しやすい．証明は m に関する簡単な帰納法である．

上の定義を Coq に与えると問題が生じる．

```
Fixpoint gcd (m n : nat) {struct m} : nat :=
  if m is 0 then n else gcd (n %% m) m.
Error:
Recursive definition of gcd is ill-formed.
Recursive call to gcd has principal argument equal to
"n %% m" instead of "n0".
```

どうも，Coq が n %% m が m より小さいことを理解していないようだ．解決法は 2 つある．

ダミーの引数　常に m より大きいダミーの引数を追加して，その引数に対する帰納法を使う．

```
Fixpoint gcd (h m n : nat) {struct h} : nat :=
  if h is h.+1 then
    if m is 0 then n else gcd h (n %% m) m
  else 0.
```

h に関する場合分けが常に成功する（h が 0 になることはない）ことを証明しなければならないが，難しくはない．しかし，このやり方を使うと，Extraction の後でも h がコードの中に残り，本来のアルゴリズムと少し違ってしまう．

整礎帰納法　整礎な順序とは，無限な減少列を持たない順序のことを言う．自然数の上では $<$ は整礎である．特定の引数が全ての再帰呼び出しで整礎な順序において減少しているならば，関数の計算が無限に続くことはないので，Coq が定義を認める．(実際には減少の証明の構造に関する構造的帰納法が使われている)

Fixpoint の代わりに Function を使い，struct（構造）を wf（整礎）に変える．この方法では，定義と同時に引数が小さくなることを証明しなければならない．

```
Require Import Wf_nat Recdef.
Check lt_wf.
     : well_founded lt
Check lt_wf_ind.
     : ∀n (P : nat → Prop), (∀n', (∀m, m < n' → P m) → P n') → P n

Function gcd (m n : nat) {wf lt m} : nat :=
  if m is 0 then n else gcd (modn n m) m.
Proof.
- move=> m n m0 _. apply/ltP.
  by rewrite ltn_mod.
- exact: lt_wf.
Qed.
gcd_ind is defined
...
gcd is defined
gcd_equation is defined
Check gcd_equation.
Check gcd_ind.
Print gcd_terminate.

Require Import Extraction.
Extraction gcd.                                    (* wf が消える *)
```

```
let rec gcd m n =
  match m with
  | O -> n
  | S n0 -> gcd (modn n (S n0)) (S n0)
```

では，これから正しさを証明する．

```
Search (_ %| _) "dvdn".                    (* 割り切ることに関する補題を表示 *)
Check divn_eq.
     : ∀ m d : nat, m = m %/ d * d + m %% d

Theorem gcd_divides m n : (gcd m n %| m) && (gcd m n %| n).
Proof.
  functional induction (gcd m n).
    by rewrite dvdn0 dvdnn.
Admitted.

Check addKn.
     : ∀ x y : nat, x + y - x = y

Theorem gcd_max g m n : g %| m -> g %| n -> g %| gcd m n.
Admitted.
```

3 $\sqrt{2}$ が無理数

まずは自然数で以下の定理を証明する．

定理 2 任意の自然数 n と p について，

$$n \cdot n = 2(p \cdot p) \text{ ならば } p = 0$$

証明は n の関する整礎帰納法を使う．

- $n = 0$ のとき，$p = 0$
- $n \neq 0$ のとき，
  - n と p が偶数でなければならないので，$n = 2n', p = 2p'$ とおける
  - 再び，$n' \cdot n' = 2(p' \cdot p')$ が得られ，$n' < n$
  - 帰納法の仮定より $p' = 0$
  - すなわち，$p = 0$

その定理を使って，$\sqrt{2}$ が無理数であることを証明する．もしも $\sqrt{2}$ が有理数なら，ある n と p が存在し，$\sqrt{2} = n/p$，すなわち $n^2 = 2p^2$．しかし上の定理から $p = 0$ となるので矛盾．

実際に証明する．

```
odd_mul    : ∀ m n : nat, odd (m * n) = odd m && odd n
odd_double : ∀ n : nat, odd n.*2 = false
odd_double_half : ∀ n : nat, odd n + (n./2).*2 = n
andbb      : ∀ x : bool, x && x = x
negbTE     : ∀ b : bool, ~~ b -> b = false
double_inj : ∀ x x2 : nat, x.*2 = x2.*2 -> x = x2
divn2      : ∀ m : nat, m %/ 2 = m./2
ltn_Pdiv   : ∀ m d : nat, 1 < d -> 0 < m -> m %/ d < m
```

```
muln2      : ∀ m : nat, m * 2 = m.*2
esym       : ∀ (A : Type) (x y : A), x = y -> y = x

Lemma odd_square n : odd n = odd (n*n). Admitted.
Lemma even_double_half n : ~~odd n -> n./2.*2 = n. Admitted.


(* 本定理 *)
Theorem main_thm (n p : nat) : n * n = (p * p).*2 -> p = 0.
Proof.
  elim/lt_wf_ind: n p => n.                                    (* 清楚帰納法 *)
  case: (posnP n) => [-> _ [] // | Hn IH p Hnp].
Admitted.

(* 無理数 *)
Require Import Reals Field.                (* 実数とそのための field タクティク *)

Definition irrational (x : R) : Prop :=
   forall (p q : nat), q <> 0 -> x <> (INR p / INR q)%R.

Theorem irrational_sqrt_2: irrational (sqrt (INR 2)).
Proof.
  move=> p q Hq Hrt.
  apply /Hq  /(main_thm p) /INR_eq.
  rewrite -mul2n !mult_INR -(sqrt_def (INR 2)) ?{}Hrt; last by auto with real.
  have Hqr : INR q <> 0%R by auto with real.
  by field.
Qed.
```

練習問題 3.1 Admitted を Qed に変え, 証明を完成せよ.