Jacques Garrigue, 2013 年 1 月 30 日

プログラムの証明2

1 前回の課題

```
Section Sort.
  Theorem insert_ok : forall a l, sorted l -> sorted (insert a l).
  Proof.
    intros a l HS. induction HS; simpl; auto.
    destruct (le_total a a0); info eauto using le_list_trans, le_list_insert.
  Qed.
  Hint Constructors Permutation.
  Lemma Permutation_refl : forall l, Permutation l l.
  Proof.
    induction l; auto.
  Qed.
  Lemma insert_perm : forall l a, Permutation (a :: l) (insert a l).
  Proof.
    intros. induction l; simpl; auto.
    destruct (le_total a a0); eauto.
  Qed.
  Lemma isort_perm : forall l, Permutation l (isort l).
  Proof.
    induction l; simpl; auto.
    eauto using insert_perm.
  Qed.
End Sort.
...
Definition isort_le := safe_isort nat le le_refl le_trans le_total.
```

プログラムの抽出

前回に既に見ているが，Extraction というコマンドで関数を OCaml のプログラムとしてファイルに抽出することができる．

```
Extraction "isort.ml" isort_le.
```

そのファイルの中身を見ると，指定した関数だけではなく，必要な定義も一緒に抽出されている．ただし，Prop 型の引数や結果は自動的に削除されるので，Prop による依存関係は無視される．
証明付のプログラムを抽出することで，絶対に安全なコードが得られる．

2 整数のための作戦

auto with arith
前回見た auto 作戦を自然数に対して使うとき，arith という定理のデータベースを使わなければならない．

```
Require Import Arith.

Goal forall m n, m + n = n + m.
  auto with arith.
Qed.
```

ring

auto with arith は定理を順番に適用するだけなので，等式が複雑になると解けないことが多い．多項式に変換できる時には，ring を使えば解ける．

```
Require Import Ring.

Goal forall m n p, m * n + p = p + n * m.
  intros.
  auto with arith. (* 何も起きない *)
  ring.
Qed.
```

omega

不等式に関して，omega がとても便利である．不等式を証明するだけでなく，仮定の中の矛盾も見つけてくれる．

```
Require Import Omega.

Goal forall m n, m <= n -> m < n + 1.
  intros.
  omega.
Qed.

Goal forall m n, m < n -> n < m -> False.
  intros.
  omega.
Qed.
```

3 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
```

このアルゴリズムを Coq で実装するのに，2つの問題がある．まず，mod という演算子が定義されていない．しかし，Euclid というライブラリの中で modulo という関数が定義されている．

```
Require Import Arith Euclid Ring Omega.

Check modulo.
     : forall n : nat, n > 0 ->
       forall m : nat, {r : nat | exists q : nat, m = q * n + r /\ n > r}
```

引数が 0 でないという条件があり，結果は依存型になっている．プログラムで使うには，まず引数の条件を削らなければならない．引数に後者関数 S をかけることで条件が満たせる．引数の順番も普通に戻す．

```
Definition mod' n m := modulo (S m) (lt_O_Sn m) n.
```

これで gcd が定義できるはず．結果は依存積なので，proj1_sig で中身を取り出せばいい．

```
Fixpoint gcd (m n : nat) {struct m} : nat :=
  match m with
  | O    => n
  | S m' => gcd (proj1_sig (mod' n m')) m
```

```
  end.
Error:
Recursive definition of gcd is ill-formed.
Recursive call to gcd has principal argument equal to
"proj1_sig (mod' n m')" instead of m'.
```

どうも，Coq が n mod 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:nat) m n {struct h} :=
  match m with
  | 0 => n
  | S m' =>
    match h with
    | 0 => 1
    | S h' => gcd h' (proj1_sig (mod' n m')) m
    end
  end.
```

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．

清楚帰納法　清楚な順序とは，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 < は清楚である．特定の引数が全ての再帰呼び出しで清楚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（清楚）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 Recdef.

Function gcd (m n : nat) {wf lt m} : nat :=
  match m with
  | 0    => n
  | S m' => gcd (proj1_sig (mod' n m')) m
  end.
  (* 減少の証明 *)
  intros.
  destruct (mod' n m'). simpl.
  destruct e as [q [Hn Hm]].
  apply Hm.
  (* 清楚性の証明 *)
  Search well_founded.
  exact lt_wf.
Defined.
gcd_ind is defined
...
gcd is defined
gcd_equation is defined
```

関数と一緒に，様々な補題が定義される．特に，gcd_ind という帰納法の原理が functional induction (gcd m n) という作戦で役に立つ．

```
Extraction "gcd.ml" gcd.
Check gcd_ind.
```

では，これから正しさを証明する．

```
Inductive divides (m : nat) : nat -> Prop :=                (* m が n を割る *)
  divi : forall a, divides m (a * m).

(* 上の定義を使いやすくするための補題 *)
Lemma divide : forall a m n, n = a * m -> divides m n.
Proof.
  intros. rewrite H. constructor.
Qed.

Lemma divides_mult : forall m q n, divides m n -> divides m (q * n).
Proof.
  induction 1. apply (divide (q*a)). ring.
Qed.

Parameter divides_plus :
  forall m n p, divides m n -> divides m p -> divides m (n+p).
Parameter divides_1 : forall n, divides 1 n.
Parameter divides_0 : forall n, divides n 0.
Parameter divides_n : forall n, divides n n.

Hint Resolve divides_plus divides_mult divides_1 divides_0 divides_n.

Theorem gcd_divides : forall m n,
  divides (gcd m n) m /\ divides (gcd m n) n.
Proof.
  intros.
  functional induction (gcd m n).                (* 関数の定義に対する帰納法 *)
    auto.
  destruct (mod' n m').
  simpl in *.                                          (* 仮定も単純化する *)
  destruct e as [q [Hn Hm]].
  destruct IHn0.
  split; auto.
  rewrite Hn.
  info auto.
Qed.

Parameter plus_inj : forall m n p, m + n = m + p -> n = p.

Lemma divides_plus' : forall m n p,
  divides m n -> divides m (n+p) -> divides m p.
Proof.
  induction 1.
  intro.
  induction a. assumption.
  inversion H.
  destruct a0.
    destruct p. auto.
    elimtype False.
    destruct m; destruct a; try discriminate; omega.
  simpl in H1.
  apply IHa.
  rewrite <- plus_assoc in H1.
  rewrite <- (plus_inj _ _ _ H1).
  constructor.
Qed.

Theorem gcd_max : forall g m n,
  divides g m -> divides g n -> divides g (gcd m n).
```

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