Jacques Garrigue, 2012 年 2 月 1 日

依存型プログラミング

1 前回の課題

```
Lemma divides_plus : forall m n p,
  divides m n -> divides m p -> divides m (n+p).
Proof.
  induction 1. induction 1. apply (divide (a+a0)). ring.
Qed.

Lemma divides_1 : forall n, divides 1 n.
Proof. intros. apply (divide n). ring. Qed.

Lemma divides_0 : forall n, divides n 0.
Proof. intros. apply (divide 0). reflexivity. Qed.

Lemma divides_n : forall n, divides n n.
Proof. intros. apply (divide 1). ring. Qed.

Lemma plus_inj : forall m n p, m + n = m + p -> n = p.
Proof.
  intros.
  induction m. assumption.
  simpl in H.
  inversion H.
  auto.
Qed.

Theorem gcd_max : forall g m n,
  divides g m -> divides g n -> divides g (gcd m n).
Proof.
  intros.
  functional induction (gcd m n). assumption.
  destruct (mod' n m'). simpl in *.
  destruct e as [q [Hn Hm']].
  apply IHn0.
    apply (divides_plus' g (q * S m')).
      induction H.
      apply (divide (q * a)). ring.
    rewrite <- Hn. assumption.
  assumption.
Qed.
```

2　依存型プログラミング

長さ付きのリスト

通常のリストは長さが分からないので、２つのリストの長さが同じという様な性質を型で表現できない。

Coq の依存型を使えば、それを型の中で表現し、普通の関数でも使える。

```
Section Vector.
Variable A : Set.

Inductive vector : nat -> Set :=
  | Vnil : vector 0
  | Vcons : forall n, A -> vector n -> vector (S n).

Definition head n (v : vector (S n)) :=
  match v with
  | Vcons _ a _ => a
  end.

Definition tail n (v : vector (S n)) :=
  match v with
  | Vcons _ _ t => t
  end.

Fixpoint append n1 n2 (v1 : vector n1) (v2 : vector n2)
  : vector (n1 + n2) :=
  match v1 with
  | Vnil => v2
  | Vcons _ a v1' => Vcons _ a (append _ _ v1' v2)
  end.
End Vector.
```

ここで定義した vector 型は長さがついている。head と tail では引数の長さを 1 以上にしたので、空リストの場合を扱わなくてよい。append では関数の型に入れただけで、再帰呼び出しでは自動的に推論される。

計算も問題なくできる。

```
Implicit Arguments Vnil [A].
Implicit Arguments Vcons [A n].
Implicit Arguments append [A n1 n2].

Eval compute in append (Vcons 1 (Vcons 2 Vnil)) (Vcons 3 Vnil).
     = Vcons 1 (Vcons 2 (Vcons 3 Vnil))
     : vector nat (2 + 1)
```

Implicit Arguments で長さを省略できるようにした。

さて、これでベクトルの足し算が定義できる。しかし、この場合では Fixpoint による定義がうまく行かないので、Definition を使う。

```
Definition add_vector n (v1 v2 : vector nat n) : vector nat n.
  refine (fix addv n (v1 v2 : vector nat n) {struct v1} := _).
```

```
  destruct v1 as [|n1 a1 v1']. exact Vnil.
  inversion v2 as [|n2 a2 v2'].
  exact (Vcons (a1+a2) (addv n1 v1' v2')).
Defined.

Eval compute in add_vector (Vcons 1 (Vcons 2 Vnil)) (Vcons 3 (Vcons 4 Vnil)).
     = Vcons 4 (Vcons 6 Vnil)
     : vector nat 2
```

{n}は b が最初から省略可能という意味になる。

ここでは作戦を使ってプログラムを作る。

exact 作戦は項を直接に与える。証明ではあまり使わないが、プログラムを定義するときには曖昧性を避ける。

refine 作戦は exact と同じだが、項の中に穴を残していい。穴を _ で指定する。穴を埋めるゴールが生成されるので、直後に与えなければならない。プログラムモードと証明モードが行き来できるので便利。

型付インタープリタ

同じ方法を使って、型付の構文木が作れる。

```
Inductive exp : Type -> Type :=
  | Nat : nat -> exp nat
  | Pair : forall t1 t2, exp t1 -> exp t2 -> exp (t1 * t2)
  | App : forall t1 t2, exp (t1 -> t2) -> exp t1 -> exp t2
  | Plus : exp (nat -> nat -> nat).
```

ここでは Set の制限を避けるために Type を使っている。各構成子が型付け規則に対応している。

この構文木に対してインタープリタを定義する。

```
Fixpoint eval t (e : exp t) : t :=
  match e with
  | Nat n => n
  | Pair t1 t2 a b => (eval t1 a, eval t2 b)
  | App t1 t2 f g => (eval (t1 -> t2) f) (eval t1 g)
  | Plus => plus
  end.
```

型はなんと「$\forall$t, exp t $\to$ t」である。結果の型が完全に入力に依存している。

インタープリタを使ってみる。

```
Implicit Arguments eval [t].
Implicit Arguments Pair [t1 t2].
Implicit Arguments App [t1 t2].

Eval compute in eval (Pair (Nat 0) (App (App Plus (Nat 1)) (Nat 1))).
     = (0, 2)
     : nat * nat
```

条件付き型

前回の modulo のように、値が満たす述語をそのまま型に入れると言うやり方もある。前回は値の作り方と述語の証明を別々にしていたが、両方とも証明モードで作ることもできる。

```
Open Scope Z_scope.
Require Import ZArith.

Inductive interval : Set := I : forall m n : Z, m <= n -> interval.

Definition iplus (i1 i2 : interval) : interval.
  intros [m1 n1 l1] [m2 n2 l2].
  apply (I (m1+m2) (n1+n2)).
  auto with zarith.
Defined.
Print iplus.
```

証明の入った値の等価性は特に扱いにくい。２つの問題がある。まず、値が証明の中に現れると、rewrite が使えない。証明を抽象化して、結論い戻せば、値と証明の型を同時に書き換えることで解決。もう一つの問題は、同じ性質の異なる証明は等しくならないこと。それに関して、proof_irrelevance という公理を使えばいい。Coq の論理は proof_irrelevance が矛盾なく使えるように設計してある。

```
Require Import ProofIrrelevance.
Check proof_irrelevance.
proof_irrelevance : forall (P : Prop) (p1 p2 : P), p1 = p2

Lemma iplus_comm : forall i1 i2, iplus i1 i2 = iplus i2 i1.
Proof.
  intros [m1 n1 l1] [m2 n2 l2].
  simpl.
  set (la := Zplus_le_compat m1 n1 m2 n2 l1 l2).
  clearbody la.                                    (* 証明を忘れる *)
  set (lb := Zplus_le_compat m2 n2 m1 n1 l2 l1).
  clearbody lb.
  revert la lb.
  rewrite (Zplus_comm m2).
  rewrite (Zplus_comm n2).
  intros.
  apply f_equal.
  apply proof_irrelevance.
Qed.
```

練習問題 2.1　*1.* vector の最後の要素を返す関数を定義せよ。

2. exp に真偽値 bool と If 文を追加せよ。

```
If : forall t, exp bool -> exp t -> exp t -> exp t
```

3. iminus : interval -> interval -> interval を定義せよ。