Jacques Garrigue, 2006 年 12 月 6 日

IV　抽象化

抽象化は二つの役割を持っている.

- 使われる機能を明確にして，隠れて依存関係を防ぐ
- 再利用を可能にする

1　汎関数による抽象化

```
let rec add_poly p1 p2 =
  match p1, p2 with
  | [], p | p, [] -> p
  | h1::t1, h2::t2 -> (h1 + h2) :: add_poly t1 t2
val add_poly : int list -> int list -> int list

(* 足算を抽象化 *)
let rec add_poly ~add p1 p2 =
  match p1, p2 with
  | [], p | p, [] -> p
  | h1::t1, h2::t2 -> add h1 h2 :: add_poly ~add t1 t2
val add_poly : add:('a -> 'a -> 'a) -> 'a list -> 'a list -> 'a list

let add_poly_int = add_poly ~add:(+)
val add_poly_int : int list -> int list -> int list
```

複数の演算に対する抽象化

```
let rec evalm1 = function                                    (* -1 での値 *)
  | []   -> 0
  | h::t -> h - evalm1 t
val evalm1 : int list -> int

(* 群の抽象化 *)
let rec evalm1 ~add ~zero ~neg = function
  | []   -> zero
  | h::t -> add h (neg (evalm1 ~add ~zero ~neg t))
val evalm1 :
  add:('a -> 'b -> 'c) -> zero:'c -> neg:('c -> 'b) -> 'a list -> 'c

(* レコード型を定義して演算をまとめる *)
type 'a group = { zero: 'a; neg: 'a -> 'a; add: 'a -> 'a -> 'a }

let rec evalm1 ~group = function
  | []   -> group.zero
  | h::t -> group.add h (group.neg (evalm1 ~group t))
```

```
val evalm1 : group:'a group -> 'a list -> 'a

let int_group = {zero = 0; neg = (fun x -> -x); add = (+)}
val int_group : int group = {zero = 0; neg = <fun>; add = <fun>}
let evalm1_int = evalm1 ~group:int_group
val evalm1_int : int list -> int
```

さらなる拡張

```
type 'a ring =
  { zero: 'a; neg: 'a -> 'a; add: 'a -> 'a -> 'a;
    one: 'a; mult: 'a -> 'a -> 'a }
let rec eval_poly ~ring p x =
  match p with
  | []   -> ring.zero
  | h::t -> ring.add h (ring.mult x (eval_poly ~ring t x))
val eval_poly : ring:'a ring -> 'a list -> 'a -> 'a

let rec eval_poly ~ring p x =
  match p with
  | []   -> ring.zero
  | h::t -> ring.add h (ring.mult x (eval_poly ~ring t x))
val eval_poly : ring:'a ring -> 'a list -> 'a -> 'a

let int_ring =
  zero = 0; neg = (fun x -> -x); add = (+); one = 1; mult = ( * )
val int_ring : int ring =
  {zero = 0; neg = <fun>; add = <fun>; one = 1; mult = <fun>}
let eval_poly_int = eval_poly ~ring:int_ring
val eval_poly_int : int list -> int -> int

(* このやりかたの問題 *)
let evalm1_int = evalm1 ~group:int_ring
This expression has type int ring but is here used with type 'a group
let group_add (g : 'a group) = g.add
This expression has type 'a group but is here used with type 'b ring
```

O'Caml のレコードは必要なラベルがそろっていても，型が違うとエラーになる．さら，同じラベルを複数の型で使うと，最後に定義されたものに所属するようになる．

練習問題 1 同じ抽象化を使い，定数との積 `times_poly` と多項式同士の積 `mult_poly` を定義せよ．

2 モジュールによる抽象化

モジュールの定義

```
module IntGroup = struct
  type t = int
  let zero = 0
  let neg x = -x
  let add = (+)
end
```

```
module IntGroup :
  sig
    type t = int
    val zero : int
    val neg : int -> int
    val add : int -> int -> int
  end

let n : IntGroup.t = IntGroup.add 3 4
val n : IntGroup.t = 7
```

モジュールのインターフェースと抽象型

```
module type Group = sig
  type t
  val zero: t
  val neg: t -> t
  val add: t -> t -> t
end
module AbsGroup : Group = IntGroup
module AbsGroup : Group

let z = AbsGroup.zero
val z : AbsGroup.t = <abstr>
let n = AbsGroup.add 3 4
This expression has type int but is here used with type AbsGroup.t
```

インターフェースの中の型が抽象型だと元の型として使えない.

モジュール間の関数：ファンクター

```
module Poly(G : Group) = struct
  let rec add p1 p2 =
    match p1, p2 with
    | [], p | p, [] -> p
    | h1::t1, h2::t2 -> G.add h1 h2 :: add t1 t2
  let rec evalm1 = function
    | []   -> G.zero
    | h::t -> G.add h (G.neg (evalm1 t))
end
module Poly :
  functor (G : Group) -> sig val add : G.t list -> G.t list -> G.t list end
module IntPoly = Poly(IntGroup)
module IntPoly :
  sig
    val add : IntGroup.t list -> IntGroup.t list -> IntGroup.t list
    val evalm1 : IntGroup.t list -> IntGroup.t
  end
```

インターフェースとモジュールの拡張

```
module type Ring = sig
  include Group
```

```
  val one: t
  val mult: t -> t -> t
end
module IntRing : (Ring with type t = int) = struct
  include IntGroup
  let one = 1
  let mult = ( * )
end
module IntRing :
  sig
    type t = int                    (* 上の with typeがあるので，抽象型ではない *)
    val zero : t
    val neg : t -> t
    val add : t -> t -> t
    val one : t
    val mult : t -> t -> t
  end

module Poly2(R : Ring) = struct
  include Poly(R)               (* Group に必要なフィールドが含まれるので大丈夫 *)
  let rec eval p x =
    match p with
    | []   -> R.zero
    | h::t -> R.add h (R.mult x (eval t x))
end
module Poly2 : functor (R : Ring) ->
  sig
    val add : R.t list -> R.t list -> R.t list
    val evalm1 : R.t list -> R.t
    val eval : R.t list -> R.t -> R.t
  end
```

練習問題 2　*1.* Poly2 の中に times と mult を定義せよ.

2. Poly2 を拡張して，結果も Ring になるようにすることで，*2* 変数の多項式を定義せよ.