4 ファンクターと抽象型

ファンクターによって，値に依存する型が作れる．

```
module type NVect = sig                                  (* 長さ固定のベクトルの型 *)
  type 'a tp
  val length : int
  val init : (int -> 'a) -> 'a tp
  val get : 'a tp -> int -> 'a
  val to_list : 'a tp -> 'a list
end

module NVect(V : VectP)(N : sig val n : int end) : NVect =
  struct
    include V
    let length = N.n                                     (* 長さを N から取る *)
    let init f = V.init length f
  end
module NVect :
  functor (V : VectP) -> functor (N : sig val n : int end) -> NVect
```

ファンクターの結果に現われる抽象型は，ファンクターが適用される度に区別される．

```
# module NVect3 = NVect(ArrayVectP)(struct let n = 3 end) ;;
module NVect3 : NVect
# module NVect2 = NVect(ArrayVectP)(struct let n = 2 end) ;;
module NVect3 : NVect
# NVect2.to_list (NVect3.init (fun x -> x+2));;
This expression has type int NVect3.tp but is here used with type
  'a NVect2.tp
# NVect3.to_list (NVect3.init (fun x -> x+2));;
- : int list = [2; 3; 4]
```

全ての引数が名前の付いたモジュールであれば，その名前から構成された抽象型になる．

```
# module N3 = struct let n = 3 end;;
module N3 : sig val n : int end
# module NVect3' = NVect(ArrayVectP)(N3);;
module NVect3' :
  sig type 'a tp = 'a NVect(ArrayVectP)(N3).tp ... end
```

この場合，同じ名前のモジュールから合成された抽象型が互換性を持つ．

```
# module NVect3'' = NVect(ArrayVectP)(N3);;
module NVect3'' :
  sig type 'a tp = 'a NVect(ArrayVectP)(N3).tp ... end
# NVect3''.to_list (NVect3'.init (fun x -> x+2));;
- : int list = [2; 3; 4]
```

5 オブジェクトによる抽象化

拡張レコード

OCaml のオブジェクトは拡張レコードのように使えるので，レコードでうまく解決できなかった問題が解ける．

オブジェクトの型は定義なしでも使える．

```
let rec add_poly ~group p1 p2 =
  match p1, p2 with
  | [], p | p, [] -> p
  | h1::t1, h2::t2 -> group#add h1 h2 :: add_poly ~group t1 t2
val add_poly :
  group:< add : 'a -> 'a -> 'a; .. > -> 'a list -> 'a list -> 'a list
let rec evalm1 ~group = function
  | []   -> group#zero
  | h::t -> group#add h (group#neg (evalm1 ~group t))
val evalm1 :
  group:< add : 'a -> 'b -> 'c; neg : 'c -> 'b; zero : 'c; .. > -> 'a list -> 'c
let int_group =
  object method zero = 0 method neg x = -x method add = (+) end
val int_group : < add : int -> int -> int; neg : int -> int; zero : int > = <obj>
let add_poly_int = add_poly ~group:int_group
val add_poly_int : int list -> int list -> int list
```

型の読み方は「group はメソッド add, neg, zero をもっている」，さらに「..」は「それ以外のメソッドがあってもいい」という意味になる．オブジェクトの定義は object で始まり，method で let のようにメソッドが定義でき，end で終わる．

プログラムを読みやすくするために，型を定義した方がいい．

```
class type ['a] group = object                    (* パラメーターに [ ] を使う *)
  method zero : 'a
  method neg : 'a -> 'a
  method add : 'a -> 'a -> 'a
end
let int_group : 'a group = int_group              (* group 型が定義されている *)
val int_group : int group = <obj>
let add_poly : group:'a #group -> _ = add_poly   (* # は .. と同じ意味を持つ *)
val add_poly : group:'a #group -> 'a list -> 'a list -> 'a list
let rec evalm1 : group:'a #group -> _ = evalm1
val evalm1 : group:'a #group -> 'a list -> 'a
```

環へ拡張する．

```
class type ['a] ring = object
  inherit ['a] group                                (* group の定義を挿入 *)
  method one : 'a
  method mult : 'a -> 'a -> 'a
end
class type ['a] ring =
  object
    method add : 'a -> 'a -> 'a
    method mult : 'a -> 'a -> 'a
    method neg : 'a -> 'a
```

```
    method one : 'a
    method zero : 'a
  end
let rec eval_poly ~(ring:'a #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

class int_group : [int] group =
  object method zero = 0 method neg x = -x method add = (+) end
class int_group : [int] group                    (* class type はクラスの型 *)
class int_ring : [int] ring = object
  inherit int_group                                          (* 実装の継承 *)
  method one = 1
  method mult = ( * )
end
class int_ring : [int] ring
let eval_poly_int = eval_poly ~ring:(new int_ring)        (* 使うときに new *)
val eval_poly_int : int list -> int -> int
```

ファンクターとクラス

オブジェクトを使ってファンクターとよく似たことができる.

```
class type ['a,'v] vector = object
  method length : 'v -> int
  method get : 'v -> int -> 'a
  method init : int -> (int -> 'a) -> 'v
end
class ['v] poly (g : 'a #group) (v : ('a,'v) #vector) = object
  method add p1 p2 =
    let l1 = v#length p1 and l2 = v#length p2 in
    v#init (max l1 l2)
      (fun i -> if i < l1 then if i < l2 then g#add (v#get p1 i) (v#get p2 i)
                else v#get p1 i else v#get p2 i)
end
class ['a] poly :
  'b #group -> ('b, 'a) #vector -> object method add : 'a -> 'a -> 'a end
class ['a] array_vector : ['a,'a array] vector = object
  method length = Array.length
  method get = Array.get
  method init = Array.init
end
class ['a] array_vector : ['a, 'a array] vector
let poly_int_array = new poly (new int_ring) (new array_vector)
val poly_int_array : int array poly = <obj>
let v = poly_int_array#add [|3;4;5|] [|6;7|]
val v : int array = [|9; 11; 5|]

class type ['a,'v] polyvec = object
  inherit ['a,'v] vector
  method map : ('a -> 'a) -> 'v -> 'v
```

```
  method append : 'v -> 'v -> 'v
end
class ['v] poly2 (r : 'a #ring) (v : ('a,'v) #polyvec) = object (self)
  inherit ['v] poly r v
  method private times x p = v#map (r#mult x) p
  method mult p1 p2 =
    let zero = v#init 1 (fun _ -> r#zero) in
    let rec mult i =
      if v#length p1 <= i then zero else
      self#add (self#times (v#get p1 i) p2) (v#append zero (mult (i+1)))
    in mult 0
end
class ['a] poly2 :
  'b #ring -> ('b, 'a) #polyvec ->
  object
    method add : 'a -> 'a -> 'a
    method mult : 'a -> 'a -> 'a
    method private times : 'b -> 'a -> 'a
  end
```

同じオブジェクトの他のメソッドを呼ぶとき，object (self) と宣言し，その self を通じて呼ばなければならない．method private は中からしか見えないので，パラメーターにない型変数を使ってもいい．

練習問題 5 poly2 から継承して，メソッド eval を追加せよ．

オブジェクトにより隠蔽

オブジェクトの中の val は外から見えないが，継承した後でも使える．

```
class type ['a] vector = object                (* vector 自身をオブジェクトに *)
  method length : int
  method get : int -> 'a
end
class ['a] array_vector v0 = object
  val v : 'a array = v0                                  (* 状態を内部に持つ *)
  method length = Array.length v
  method get = Array.get v
  method to_list = Array.to_list v
end
class ['a] array_vector : 'a array ->
  object
    val v : 'a array
    method get : int -> 'a
    method length : int
    method to_list : 'a list
  end
class virtual ['g,'v] poly = object (self)
  method virtual ops : 'a #group as 'g          (* as の後の型変数は共有を表す *)
  method virtual init : int -> (int -> 'a) -> ('a #vector as 'v)
  method add (p1 : 'v) (p2 : 'v) =           (* 'v がないと p1 の型が分からない *)
    let l1 = p1#length and l2 = p2#length in
    self#init (max l1 l2)
```

```
      (fun i -> if i < l1 then if i < l2 then self#ops#add (p1#get i) (p2#get i)
                else p1#get i else p2#get i)
end
class virtual ['a, 'b] poly :
  object
    constraint 'a = 'c #group
    constraint 'b = 'c #vector
    method add : 'b -> 'b -> 'b
    method virtual init : int -> (int -> 'c) -> 'b
    method virtual ops : 'a
  end
let init wrap n f = wrap (Array.init n f)
val init : ('a array -> 'b) -> int -> (int -> 'a) -> 'b
let poly_int_array = object
  inherit [_,_] poly
  method ops = new int_group
  method init = init (new array_vector)
end
val poly_int_array : (int group, int array_vector) poly = <obj>
let v =
  poly_int_array#add (new array_vector [|3;4;5|]) (new array_vector [|6;7|]) ;;
val v : int array_vector = <obj>
v#to_list ;;
- : int list = [9; 11; 5]
```

ここでは vector を状態をもったオブジェクトに変えたが，group はそうしていない．二項演算子は状態をもったオブジェクトで表現しにくいからである．

```
class type ['a] polyvec = object ('v)              (* 'v は自分と同じ型を表す *)
  inherit ['a] vector
  method map : ('a -> 'a) -> 'v
  method cons : 'a -> 'v
end
class ['a] array_vec v = object (self : 'a #polyvec)
  inherit ['a] array_vector v
  method map f = {< v = Array.map f v >}          (* 状態を変えた自分のコピー *)
  method cons a = {< v = Array.append [|a|] v >}
end
class virtual ['r,'v] poly2 = object (self)
  inherit ['a #ring as 'r, 'a #polyvec as 'v] poly
  method times x (p : 'v) = p#map (self#ops#mult x)
  method mult (p1 : 'v) (p2 : 'v) =
    let zero = self#ops#zero in
    let rec mult i =
      if p1#length <= i then self#init 1 (fun _ -> zero) else
      self#add (self#times (p1#get i) p2) ((mult (i+1))#cons zero)
    in mult 0
end
class virtual ['a, 'b] poly2 :
  object
    constraint 'a = 'c #ring
    constraint 'b = 'c #polyvec
```

```
    method add : 'b -> 'b -> 'b
    method virtual init : int -> (int -> 'c) -> 'b
    method mult : 'b -> 'b -> 'b
    method virtual ops : 'a
    method times : 'c -> 'b -> 'b
  end
let poly_int_array = object
  inherit [_,_] poly2
  method ops = new int_ring
  method init = init (new array_vec)
end
val poly_int_array : (int ring, int array_vec) poly2 = <obj>
let v = poly_int_array#mult
    (new array_vec [|3;4;5|]) (new array_vec [|6;7|]) ;;
val v : int array_vec = <obj>
v#to_list ;;
- : int list = [18; 45; 58; 35]
```

ここで見たように，隠蔽を含めて，モジュールとファンクターでできる多くのことはオブジェクトでもできる．しかし，前章でみたような，値に依存するような型などは表現できないので，やはりファンクターの方が強い．

練習問題 6　*1.* 以下の型のクラスを定義せよ

```
class ['a] vector' :
  'a #vector ->
  object ('v)
    method length : int
    method get : int -> 'a
    method tl : 'v
  end
```

メソッド `length` と `get` は引数のベクトルと同じ動作をする．メソッド `tl` は 0 番目が消えた，長さ $n-1$ のベクトルを返す．元の長さが *0* のときは例外を起こす．ヒント：`tl` が使われた回数を内部状態として持てばいい．

2. 同様の方法でメソッド `cons : 'a -> 'v` を追加せよ．ヒント：内部に追加された要素のリストを持てばいい．

3. メソッド `cons` と `tl` を同時に追加せよ．

4. `poly2` から継承して，型が `ring` になりうるようなクラスを定義せよ．それによって多変数の多項式を定義せよ．