情報システム科学実習II第9回
ファンクター

担当: 山口 和紀・五十嵐 淳

2001年12月12日

OCaml, Standard ML のモジュールシステムを強力 (かつ複雑) たらしめているのが，**ファンクター**(*functor*) というモジュール上の関数を許している点である．ファンクターは下に見るように，抽象データ型に構造を導入する場合に便利なのだが，同時にモジュール間でのコンパイルに必要な型情報の伝播をするための機能が必要になってくる．

1 モジュール関数: ファンクター

例として，集合を表すデータ構造とそのモジュールを考えてみよう．モジュールには，ある要素が集合に属するかを判定する mem 関数，集合の共通部分を取る inter 関数，和を取る union 関数などを備えることにしよう．また，集合のデータ表現としてはリストを，また，集合演算を簡単にするために，要素は (なんらかの大小関係で) 昇順に並べられているものとする．大小関係を表現する型を order，また compare を〈要素型〉 -> 〈要素型〉-> order の関数として定義を行う．

たとえば整数の集合は，

```
# type order = LessThan | Equal | MoreThan
# module IntSet =
#   struct
#     type elm = int
#     type t = elm list
#
#     let compare i j =
#       if i = j then Equal else if i > j then MoreThan else LessThan
#
#     let rec mem elm = function
#         [] -> false
#       | x :: rest ->
#           match compare elm x with Equal -> true | _ -> mem elm rest
#
#     let rec inter s1 s2 =
```

```
#       match (s1, s2) with
#         (s1, []) -> []
#       | ([], s2) -> []
#       | ((e1::rest1 as s1), (e2::rest2 as s2)) ->
#           match compare e1 e2 with
#             Equal -> e1 :: inter rest1 rest2
#           | LessThan -> inter rest1 s2
#           | MoreThan -> inter s1 rest2
#     (*
#     let rec union s1 s2 = ...
#     *)
#   end;;
type order = LessThan | Equal | MoreThan
module IntSet :
  sig
    type elm = int
    and t = elm list
    val compare : 'a -> 'a -> order
    val mem : 'a -> 'a list -> bool
    val inter : 'a list -> 'a list -> 'a list
  end
```

と定義できる．同様にして，適当な compare 関数さえ定義すれば，どんな要素に対する集合モジュールなども定義できることがわかるだろう．(この例では compare 関数は自明なものであるが，複雑な型を要素とする場合には，compare も = や < で置換えることはできないかもしれない．)

しかし，それらのモジュールで違ってくるのは，(実装をリストとして固定している限り) 比較関数 compare と型 elm の定義だけである．このような，ほんの一部分だけが違ったモジュールをまとめて記述するために,「共通部分はパラメータ化」の原則をモジュールにも適用したも「モジュール関数」がファンクターである．では，何に関してパラメータ化されるのかというと，それは，本質的には型 elm と，その型上の関数 compare : elm -> elm -> order の組，すなわち，モジュールである．

さて，実際にファンクターを定義してみよう．ファンクターの引数は明示的にその型，つまりシグネチャを指定しなければいけない．まずは，引数となるモジュールのシグネチャは,

```
# module type OrderedType =
#   sig
#     type t
#     val compare : t -> t -> order
#   end;;
module type OrderedType = sig type t val compare : t -> t -> order end
```

と定義できる．t は要素型となるものである．すると，ファンクターの定義は，以下のようにできる．

```
# module MakeSet (Ord : OrderedType) =
#   struct
```

```
#     type elm = Ord.t
#     type t = elm list
#
#     let empty = []
#
#     let rec mem elm = function
#         [] -> false
#       | x :: rest ->
#           match Ord.compare elm x with Equal -> true | _ -> mem elm rest
#
#     (* 以下省略 *)
#   end;;
module MakeSet :
  functor (Ord : OrderedType) ->
    sig
      type elm = Ord.t
      and t = elm list
      val empty : 'a list
      val mem : Ord.t -> Ord.t list -> bool
    end
```

MakeSet がファンクターの名前，Ord は仮引数であるモジュール名，OrderedType がそのシグネチャである．sturct ... end の中では Ord が普通のモジュールであるかのごとく，compare や t にアクセスすることができる．また，コンパイラからの応答を見ると，MakeSet が生成するモジュールのシグネチャがわかる．この全体 (functor ... ->) は一種の関数型のようなものと考えることもできるだろう． ただし，通常の int -> int などの関数型と決定的に違う点がひとつある．それは生成されるモジュールのシグネチャ sig ... end 部分に仮引数の名前 Ord が出現しているところである．(通常の関数では引数の型でなく引数自体が型に現れることはない．) これは，生成されるモジュールのシグネチャが引数となるモジュール Ord の**値**に依存するからである．

また，匿名関数 fun と同様に匿名のファンクターを考えることもでき，文法は

functor (⟨モジュール変数⟩ : ⟨モジュール型⟩) ->
  ⟨モジュール式⟩

となる．よって，上の宣言は

```
# module MakeSet = functor (Ord : OrderedType) ->
#   struct
#     type elm = Ord.t
#     type t = elm list
#
#     let empty = []
#
#     let rec mem elm = function
#         [] -> false
#       | x :: rest ->
```

```
#           match Ord.compare elm x with Equal -> true | _ -> mem elm rest
#
#     (* 以下同様 *)
#   end;;
module MakeSet :
  functor (Ord : OrderedType) ->
    sig
      type elm = Ord.t
      and t = elm list
      val empty : 'a list
      val mem : Ord.t -> Ord.t list -> bool
    end
```

と宣言することもできる.

関数を式に適用するのと同様，ファンクターはモジュールに「適用」することでパラメータと実体を結び付けることができる．整数用の比較関数からなるモジュール OrderedInt から，整数の集合モジュールを生成してみる.

```
# module OrderedInt =
#   struct
#     type t = int
#     let compare (x : int) y =
#       if x < y then LessThan else if x = y then Equal else MoreThan
#   end;;
module OrderedInt : sig type t = int val compare : int -> int -> order end
# module IntSet = MakeSet (OrderedInt);;
module IntSet :
  sig
    type elm = OrderedInt.t
    and t = elm list
    val empty : 'a list
    val mem : OrderedInt.t -> OrderedInt.t list -> bool
  end
```

ファンクター適用の文法は

⟨ファンクター名⟩ (⟨モジュール式⟩)

であり，引数となる⟨モジュール式⟩の前後に括弧が必要である.

Exercise 9.1 以下のように，OrderedInt のシグネチャを OrderedType として明示的に

```
module OrderedInt : OrderedType = ...
```

のように指定した場合，MakeSet を適用した結果

```
module IntSet' = MakeSet (OrderedInt);;
```

のモジュール IntSet' は，IntSet とどういった点で異っているだろうか.

2 ファンクターと情報隠蔽

さて，IntSet は整数を要素とした集合として使えるわけだが，このままでは集合の実装が整数リストであることが丸見えである．前回と同様に実装に関する情報を隠蔽することを考える．ひとつの方法は，IntSet を宣言する際に，適当なシグネチャをつけてやることである．

```
# module IntSet :
#   sig
#     type elm = int
#     type t
#     val empty : t
#     val mem : elm -> t -> bool
#   end
# = MakeSet (OrderedInt);;
module IntSet :
  sig type elm = int and t val empty : t val mem : elm -> t -> bool end
```

ここでポイントとなるのは，elm の定義は = int として見せつつ，t は隠すようなシグネチャを書かなければいけないことである．そのせいで，集合の要素型が変われば，シグネチャ，特に type elm = ... の ... 部分が変ってしまう．つまり，このままでは必要な集合の種類だけ似たようなシグネチャを書かなければならなくなってしまう．

シグネチャにパッチをあてる　これを解決するために，シグネチャに「パッチ」をあてて，型宣言を補うことができる．まず，どの集合にも共通な部分だけを書いたシグネチャを用意する．

```
# module type S =
#   sig
#     type elm
#     type t
#     val empty : t
#     val mem : elm -> t -> bool
#   end;;
module type S =
  sig type elm and t val empty : t val mem : elm -> t -> bool end
```

この S では elm の型も隠蔽されたようなシグネチャになっている．さて，このシグネチャに elm の定義部分 “= int” を加えるには

〈シグネチャ〉 with type 〈型名〉 = 〈型〉

という形で加えることができる．例えば，

```
# module type IntS = S with type elm = int;;
module type IntS =
  sig type elm = int and t val empty : t val mem : elm -> t -> bool end
```

という具合いである．これで，

```
# module IntSet : S with type elm = int = MakeSet (OrderedInt);;
module IntSet :
  sig type elm = int and t val empty : t val mem : elm -> t -> bool end
# module StringSet : S with type elm = string = MakeSet (OrderedString)
```

と，差分だけを with で加えることですむ．(OrderedString は OrderedInt と同様にして宣言したモジュールである．)

ファンクター宣言での with　しかし，まだこれでは不十分である．シグネチャの共通部分を共有することができたとはいえ，集合モジュールひとつひとつに対して，シグネチャを付加しなければいけない．そもそも，t を隠したいというのは，ファンクターをつくった側の都合であり，ファンクターを使う側が明示的に隠す義務を負う必要はないはずである．

この例の最後として，ファンクター側で生成されるモジュールのシグネチャをコントロールする方法を見る．以下が，最終的な MakeSet の定義である．

```
# module MakeSet (Ord : OrderedType) : S with type elm = Ord.t =
#   struct
#     type elm = Ord.t
#     type t = elm list
#
#     let empty = []
#
#     let rec mem elm = function
#         [] -> false
#       | x :: rest ->
#           match Ord.compare elm x with Equal -> true | _ -> mem elm rest
#
#     (* 以下省略 *)
#   end;;
module StringSet :
  sig type elm = string and t val empty : t val mem : elm -> t -> bool end
module MakeSet :
  functor (Ord : OrderedType) ->
    sig type elm = Ord.t and t val empty : t val mem : elm -> t -> bool end
```

: S with type elm = Ord.t という部分が適用されたときに返るモジュールのシグネチャを表している．(関数宣言 let f x : int = ... が返り値の型を int として指定しているのと文法的には似ている．) ここでは，結果のモジュールの elm の内容は，引数である Ord の中に格納されているので，Ord.t を使って S に「パッチ」をあてている．このファンクターの型 functor ... と最初の宣言での型を比べてみるとまさに，t の中身が露になっているかいないかの違いになっている．これで，所定の目的であった t だけの隠蔽をファンクター宣言の側で達成することができる．

```
# module IntSet = MakeSet (OrderedInt);;
```

```
module IntSet :
  sig
    type elm = OrderedInt.t
    and t = MakeSet(OrderedInt).t
    val empty : t
    val mem : elm -> t -> bool
  end
```

つまり，IntSet の宣言にはシグネチャをつけることなく，先ほどと同じ t だけが隠蔽されたシグネチャが得られている．

Exercise 9.2 MakeSet を完成させよ．シグネチャ S の完全な定義は以下の通り．各関数の機能は，コメントの通りである．また，IntSet モジュールをファンクター適用によって生成して，各関数がうまく機能していることも *(*できる限り多くの例を用いて*)* 示せ．

```
module type S =
  sig
    type elm
    type t
    val empty : t
    val is_empty : t -> bool
    val mem : elm -> t -> bool
      (* returns true if a given element belongs to a set *)
    val add : elm -> t -> t
      (* add one element to a set *)
    val inter : t -> t -> t
      (* intersection of two sets *)
    val union : t -> t -> t
      (* union of two sets *)
    val diff : t -> t -> t
      (* difference of two sets *)
    val elements : t -> elm list
      (* returns a list of elements sorted in increasing order
         with respect to compare function for elm *)
  end;;
```

Exercise 9.3 MakeSet を次のように，返すモジュールのシグネチャに with をつけずに S だけとして，宣言するとどんな不都合が発生するだろう？

```
module MakeSet (Ord : OrderedType) : S =
  struct
    type elm = Ord.t
    type t = elm list

    let empty = []

    (* 以下省略 *)
  end;;
```

3　複数のモジュールを引数とするファンクター

さて，関数定義においてもそうであったように，複数のモジュールに関してパラメータ化されたモジュールを考えることができる．例えば，実際に集合演算を行うコードのモジュールは，集合モジュールと，その要素モジュールに関してパラメータ化されたものとして表現される．この場合，実現方法として，複数の引数を取る関数の実現方法と同様に

- モジュールの「組」を引数とするファンクター
- カリー化関数のように，ひとつめの引数をとると,「ふたつめの引数を取って結果のモジュールを返すファンクター」を返すようなファンクター, (**高階モジュール**(*higher-order module*) と呼ぶことが多い)

という二通りの実現方法があるが，ここでは前者の方法について触れる．高階モジュールは「ファンクターを引数にとるファンクター」なども理論的に可能にするが，あまり実用的な例を耳にしない．

モジュールの「組」を表現するにはモジュールを入れ子にすればよい．たとえば, OrderedInt と IntSet の組は,

```
# module Pair =
#   struct
#     module Fst = OrderedInt
#     module Snd = IntSet
#   end;;
module Pair :
  sig
    module Fst : sig type t = int val compare : int -> int -> order end
    module Snd :
      sig
        type elm = OrderedInt.t
        and t = MakeSet(OrderedInt).t
        val empty : t
        val mem : elm -> t -> bool
      end
  end
```

というように定義することができる．Pair 内のモジュールは Pair.Fst, Pair.Snd といった記法でアクセスできる．さて，このような組のモジュールを受け取るファンクターを考えてみよう．

```
module Client (P : ...) =
  struct
    (* test whether the function elements correctly
       yields a list of elements sorted in increasing arder *)
    let test_elements set =
      let rec loop = function
        [] | [_] -> true
```

```
      | x::y::rest ->
         if P.Fst.compare x y = MoreThan then false
         else loop (y::rest)
    in loop (P.Snd.elements set)
  end;;
```

関数 test_elements は，P.Snd モジュールの elements 関数が正しく昇順に要素を並べた要素リストを返しているかを検査する関数である．ここで，... にあてはまるシグネチャを考えるわけであるが，もっとも素朴なものは，各サブモジュールのシグネチャを並べたものである．

```
# module type NaivePsig =
#   sig
#     module Fst : OrderedType
#     module Snd : S
#   end;;
module type NaivePsig = sig module Fst : OrderedType module Snd : S end
```

が，これを引数 P のシグネチャとすると型エラーが発生してしまう．

```
# module Client (P : NaivePsig) =
#   struct
#     (* test whether the function elements correctly
#        yields a list of elements sorted in increasing arder *)
#     let test_elements set =
#       let rec loop = function
#         [] | [_] -> true
#       | x::y::rest ->
#          if P.Fst.compare x y = MoreThan then false
#          else loop (y::rest)
#     in loop (P.Snd.elements set)
#   end;;
Characters 360-378:
This expression has type P.Snd.elm list but is here used with type
  P.Fst.t list
```

この原因は，良く考えてみると，集合モジュール P.Snd の要素をしめす型 P.Snd.elm とそれを比較する関数 P.Fst.compare の引数の型 P.Fst.t があっていないことに起因することがわかる．

本来ならば P に含まれる二つのサブモジュール Fst, Snd には,「Snd は MakeSet(Fst) して作られた集合モジュール」であるべきである．(そうでないと，テストプログラムの意味がわからない．) 言い換えると Snd の要素型 elm と Fst で compare 関数による比較の対象になる型 t は等しいようなモジュールの組で P は構成されているべきであるにもかかわらず．そのことが NaivePsig に明示されていないのである．

これを表現するためには，Snd サブモジュールの elm が Fst.t と等しいことを，with を用いて記述してやれば良い．

```
# module type Psig =
#   sig
#     module Fst : OrderedType
#     module Snd : S with type elm = Fst.t
#   end;;
module type Psig =
  sig
    module Fst : OrderedType
    module Snd :
      sig
        type elm = Fst.t
        and t
        val empty : t
        val is_empty : t -> bool
        val mem : elm -> t -> bool
        val add : elm -> t -> t
        val inter : t -> t -> t
        val union : t -> t -> t
        val diff : t -> t -> t
        val elements : t -> elm list
      end
  end
# module Client (P : Psig) =
#   struct
#     (* test whether the function elements correctly
#        yields a list of elements sorted in increasing arder *)
#     let test_elements set =
#       let rec loop = function
#         [] | [_] -> true
#       | x::y::rest ->
#          if P.Fst.compare x y = MoreThan then false
#          else loop (y::rest)
#     in loop (P.Snd.elements set)
#   end;;
module Client :
  functor (P : Psig) -> sig val test_elements : P.Snd.t -> bool end
```

MakeSet での with が，ファンクター引数と生成されるモジュールの間の依存関係を表現していたのに対し，Psig での with はファンクターに与えられる (複数の) 引数同士の依存関係を表現するために使われたもの，と考えることができる．

Exercise 9.4 Client を適当な引数に適用して，test_elements を起動することによって，集合モジュールの elements 関数が正しく実装されていることを確かめよ．

4　レポートその4: 締切1月9日

必修課題:　9.2, 9.3

オプション課題:　第9回の資料の残り全部の問題.