Krótkie omówienie systemu modułów

Objective Caml, podobnie jak niektóre inne języki funkcyjne, jest wyposażony w bardzo elastyczny system modułów, pozwalający ponadto na rozdzielną kompilację. Podstawowymi pojęciami tego systemu są *sygnatury*, *struktury* i *funktory*. Sygnatury stanowią interfejsy modułów, struktury są implementacjami interfejsów, natomiast funktory są modułami parametryzowanymi.

Pojęcia te wyjaśnimy na przykładach.

Główną motywacją do wprowadzenia pojęcia *struktury* jest powiązanie ze sobą w jedną całość związanych logicznie definicji (typów danych i operacji na wartościach tych typów) i uporządkowanie sposobu nazywania takich definicji. Umożliwia to uniknięcie przypadkowych konfliktów nazw i ułatwia wybmyślanie nowych nazw dla struktur danych. Struktury deklaruje się za pomocą konstrukcji `struct ... end`, wewnątrz której może występować dowolny ciąg definicji. Strukturę można nazwać za pomocą deklaracji `module`. Oto przykład struktury dla kolejek priorytetowych:

```
# module PrioQueue =
  struct
    type priority = int
    type 'a queue =
      Empty |
      Node of priority * 'a
              * 'a queue * 'a queue
    let empty = Empty
    let rec insert queue prio elt =
      match queue with
        Empty -> Node(prio, elt, Empty, Empty)
      | Node(p, e, left, right) ->
          if prio <= p
          then Node(prio, elt,
                    insert right p e, left)
          else Node(p, e,
                    insert right prio elt, left)
    exception Queue_is_empty
```

```
  let rec remove_top = function
      Empty -> raise Queue_is_empty
    | Node(prio, elt, left, Empty) -> left
    | Node(prio, elt, Empty, right) -> right
    | Node(prio, elt,
           (Node(lprio, lelt, _, _) as left),
           (Node(rprio, relt, _, _) as right))
       ->
        if lprio <= rprio
        then Node(lprio, lelt,
                  remove_top left, right)
        else Node(rprio, relt, left,
                  remove_top right)
  let extract = function
      Empty -> raise Queue_is_empty
    | Node(prio, elt, _, _) as queue
        -> (prio, elt, remove_top queue)
end;;
```

```
module PrioQueue :
  sig
    type priority = int
    type 'a queue =
       | Empty
       | Node of priority * 'a
                 * 'a queue * 'a queue
    val empty : 'a queue
    val insert : 'a queue -> priority
                 -> 'a -> 'a queue
    exception Queue_is_empty
    val remove_top : 'a queue -> 'a queue
    val extract : 'a queue -> priority
                   * 'a * 'a queue
  end
```

Na zewnątrz struktury jej składowe są dostępne przez tradycyjny zapis kropkowy. Na przykład `PrioQueue.insert` jest funkcją wstawiania do kolejki priorytetowej, a `PrioQueue.queue` jest konstruktorem typu kolejek priorytetowych.

```
# PrioQueue.insert PrioQueue.empty 1 "hello";;
- : string PrioQueue.queue =
  PrioQueue.Node
   (1, "hello", PrioQueue.Empty, PrioQueue.Empty)
```

Sygnatury opisują interfejsy struktur. Sygnatura określa, które składowe struktury są dostępne na zewnątrz jej definicji oraz z jakimi typami i może służyć do ukrywania składników struktur oraz do eksportowania składowych z ograniczonymi typami.

```
# module type PRIOQUEUE =
  sig
   type priority = int     (* still concrete *)
   type 'a queue           (* now abstract *)
   val empty : 'a queue
   val insert : 'a queue -> int -> 'a -> 'a queue
   val extract : 'a queue -> int * 'a * 'a queue
   exception Queue_is_empty
  end;;
module type PRIOQUEUE =
 sig
  type priority = int
  type 'a queue
  val empty : 'a queue
  val insert : 'a queue -> int -> 'a -> 'a queue
  val extract : 'a queue -> int * 'a * 'a queue
  exception Queue_is_empty
 end
```

Możemy teraz zadeklarować, że struktura `PrioQueue` ma być widoczna z perspektywy sygnatury `PRIOQUEUE`, ograniczając tym samym dostęp do jej niektórych składników.

```
# module AbstractPrioQueue =
           (PrioQueue : PRIOQUEUE);;
module AbstractPrioQueue : PRIOQUEUE

# AbstractPrioQueue.remove_top;;
Characters 0-28:
Unbound value AbstractPrioQueue.remove_top

# AbstractPrioQueue.insert
      AbstractPrioQueue.empty 1 "hello";;
- : string AbstractPrioQueue.queue = <abstr>
```

Możemy narzucić to ograniczenie już w czasie definiowania struktury:

```
module PrioQueue = (struct ... end : PRIOQUEUE);;
```

lub, w alternatywnej składni:

```
module PrioQueue : PRIOQUEUE = struct ... end;;
```

Funktor jest po prostu parametryzowanym modułem, albo „funkcją” ze struktur w struktury (wyliczaną w czasie kompilacji programu!). Struktura A parametryzowana strukturą B to po prostu funktor F z parametrem formalnym B (sygnatury odpowiedniej dla B), który daje jako wynik strukturę A. Funktor taki można zastosować do wielu implementacji struktury B, na przykład $B_1, \ldots, B_n$, uzyskując w wyniku struktury $A_1, \ldots, A_n$.

Oto przykład struktury implementującej zbiory jako listy uporządkowane, która jest sparametryzowana strukturą określającą typ elementów zbioru oraz funkcję porządku na tym typie:

```
# type comparison = Less | Equal | Greater;;
type comparison = | Less | Equal | Greater

# module type ORDERED_TYPE =
    sig type t
        val cmp: t -> t -> comparison
    end;;
module type ORDERED_TYPE =
  sig type t
      val cmp : t -> t -> comparison
  end
```

```
# module Set =
    functor (Elt: ORDERED_TYPE) ->
      struct
        type element = Elt.t
        type set = element list
        let empty = []
        let rec add x s =
          match s with
            [] -> [x]
          | hd::tl ->
             match Elt.cmp x hd with
               Equal   -> s
             | Less    -> x :: s
             | Greater -> hd :: add x tl
        let rec member x s =
          match s with
            [] -> false
          | hd::tl ->
              match Elt.cmp x hd with
                Equal   -> true
              | Less    -> false
              | Greater -> member x tl
      end;;
```

```
module Set :
  functor(Elt : ORDERED_TYPE) ->
    sig
      type element = Elt.t
      type set = element list
      val empty : 'a list
      val add : Elt.t -> Elt.t list -> Elt.t list
      val member : Elt.t -> Elt.t list -> bool
    end
```

Stosując funktor `Set` do struktury implementującej typ z porządkiem, otrzymujemy strukturę z operacjami na zbiorach elementów tego typu:

```
# module OrderedString =
    struct
      type t = string
      let cmp x y =
         if x = y then Equal
                  else if x < y
                       then Less else Greater
    end;;
```

```
module OrderedString :
  sig
    type t =      string
    val cmp : 'a -> 'a -> comparison
  end

# module StringSet = Set(OrderedString);;
module StringSet :
  sig
    type element = OrderedString.t
    type set = element list
    val empty : 'a list
    val add : OrderedString.t
              -> OrderedString.t list
                      -> OrderedString.t list
    val member : OrderedString.t
              -> OrderedString.t list -> bool
  end

# StringSet.member
    "bar" (StringSet.add "foo" StringSet.empty);;
- : bool = false
```