

Un'introduzione a OCaml (seconda parte)

Fausto Spoto

19 ottobre 2006

Strutture

```
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)
```

Structure

```
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;;
```

Strutture

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

Signature

```
module type PrioQueue =
  sig
    type priority = int
    type 'a queue
    val empty : 'a queue
    val insert :
      'a queue -> priority -> 'a -> 'a queue
    val extract :
      'a queue -> priority * 'a * 'a queue
    exception Queue_is_empty
  end;;

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

```
# let q = AbstractPrioQueue.insert
    AbstractPrioQueue.empty 1 "hello";;
val q : string AbstractPrioQueue.queue = <abstr>
# AbstractPrioQueue.remove_top q;;
Unbound value AbstractPrioQueue.remove_top
#
```

E se levassimo il tipo concreto?

```
module type PRIOQUEUE =  
  sig  
    type priority = int  
    type 'a queue  
    val empty : 'a queue  
    ...  
  end;;  
module AbstractPrioQueue = (PrioQueue :PRIOQUEUE);;  
  
# let q = AbstractPrioQueue.insert  
    AbstractPrioQueue.empty 1 "hello";;
```

This expression has type `int`
but is here used with type
`AbstractPrioQueue.priority`

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

module type ORDERED_TYPE =
  sig
    type t
    val compare: 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.compare 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.compare x hd with  
  Equal -> true  
  | Less -> false  
  | Greater -> member x tl  
end;;
```

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

```
module StringSet = Set(OrderedString);;
```

```
# let s = StringSet.add "giorgio" StringSet.empty;;  
  val s : OrderedString.t list = ["giorgio"]  
# let ss = StringSet.add "giorgio" s;;  
val ss : OrderedString.t list = ["giorgio"]  
# StringSet.member "fausto" ss;;  
- : bool = false  
# s = ["giorgio"];;  
- : bool = true  
#
```

Mettere insieme funtori e segnature

```
module type SETFUNCTOR =  
  functor (Elt : ORDERED_TYPE) ->  
    sig  
      type element = Elt.t  
      type set  
      val empty : set  
      val add : element -> set -> set  
      val member : element -> set -> bool  
    end;;  
  
module AbstractSet = (Set : SETFUNCTOR);;  
  
module AbstractStringSet =  
  AbstractSet(OrderedString);;
```

Mettere insieme funtori e segnature

```
# let s = AbstractStringSet.add "giorgio"
    AbstractStringSet.empty;;
  val s : AbstractStringSet.set = <abstr>
# let ss = AbstractStringSet.add "giorgio" s;;
val ss : AbstractStringSet.set = <abstr>
# AbstractStringSet.member "fausto" ss;;
- : bool = false
# s = ["giorgio"];;
This expression has type 'a list
but is here used with type
AbstractStringSet.set
```

Implementazione e specifica in file separati

- 1 L'interno dello `struct ...end` si mette nel file `A.ml`
- 2 L'interno del `sig ...end` si mette nel file `A.mli`
- 3 È come se avessimo definito un file contenente

```
module A: sig A.mli end
= struct A.ml end;;
```

Compilazione e linking in tempi distinti

```
1 ocamlc -c Aux.mli
2 ocamlc -c Aux.ml
3 ocamlc -c Main.mli
4 ocamlc -c Main.ml
5 ocamlc -o Main Aux.cmo Main.cmo
```

```
1 genera Aux.cmi
2 genera Aux.cmo
3 genera Main.cmi
4 genera Main.cmo
5 genera Main
```

Main può fare riferimento alle definizioni in Aux ma non viceversa