The OCaml Module System

Abstract and concrete data types

Walter Cazzola

Dipartimento di Informatica
Università degli Studi di Milano
e-mail: cazzola@di.unimi.it
twitter: @w_cazzola

The OCaml Module System

Introduction

Modules are used to realize data type (ADT and implementation) and collecting functions.

Modules are composed of two parts:

- a (optional) public interface exposing the types and operations defined in the module (**sig** . . . **end**);
- the module implementation (**struct** . . . **end**).

Modules can abstract data and hide implementation details

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

Modules are useful for organizing large implementations in smaller self-contained pieces of code.

The OCaml Module System

Structure (Struct . . . End)

```
module PrioQueue =
   struct
     type priority = int
     type char_queue = Empty | Node of priority * char * char_queue * char_queue
     exception QueueIsEmpty

     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)

     let rec remove_top = function
         Empty -> raise QueueIsEmpty
       | 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 QueueIsEmpty
       | Node(prio, elt, _, _) as queue -> (prio, elt, remove_top queue)
   end;;
```


The OCaml Module System

Structure Evaluation

```
# #use "char_pqueue.ml" ;;
module PrioQueue :
  sig
    type priority = int
    type char_queue =
        Empty
      | Node of priority * char * char_queue * char_queue
    exception QueueIsEmpty
    val empty : char_queue
    val insert : char_queue -> priority -> char -> char_queue
    val remove_top : char_queue -> char_queue
    val extract : char_queue -> priority * char * char_queue
  end

# let pq = empty ;;
val pq : PrioQueue.char_queue = Empty

# let pq = insert pq 0 'a' ;;
val pq : PrioQueue.char_queue = Node (0, 'a', Empty, Empty)

# let pq = insert (insert pq 3 'c') (-7) 'w';;
val pq : PrioQueue.char_queue =
  Node (-7, 'w', Node (0, 'a', Empty, Empty), Node (3, 'c', Empty, Empty))

# let pq = extract pq;;
val pq : PrioQueue.priority * char * PrioQueue.char_queue =
  (-7, 'w', Node (0, 'a', Empty, Node (3, 'c', Empty, Empty)))
```

The OCaML Module System
Signature (Sig ... End)

```
module type CharPQueueAbs =
  sig
    type priority = int          (* still concrete *)
    type char_queue              (* now abstract *)
    val empty : char_queue
    val insert : char_queue -> int -> char -> char_queue
    val extract : char_queue -> int * char * char_queue
    exception QueueIsEmpty
  end;;
```

WRT the previous implementation this:

- opacifies the type char_pqueue and hides the remove_top operation.

```
# #use "CharPQueueAbs.mli" ;;
module type CharPQueueAbs =
  sig
    type priority = int
    type char_queue
    val empty : char_queue
    val insert : char_queue -> int -> char -> char_queue
    val extract : char_queue -> int * char * char_queue
    exception QueueIsEmpty
  end
# module AbstractPrioQueue = (PrioQueue: CharPQueueAbs);;
module AbstractPrioQueue : CharPQueueAbs
# AbstractPrioQueue.remove_top;;
Error: Unbound value AbstractPrioQueue.remove_top
# AbstractPrioQueue.insert AbstractPrioQueue.empty 1 'a' ;;
- : AbstractPrioQueue.char_queue = <abstr>
```


The OCaML Module System
Separate Compilation

Modules and their interface can be separately compiled.

```
[17:11]cazzola@surtur:~/lp/ml/mod-01>ls
CharPQueueAbs.mli  CharPQueue.ml      main.ml
[17:11]cazzola@surtur:~/lp/ml/mod-01>ocamlc -c CharPQueueAbs.mli
[17:12]cazzola@surtur:~/lp/ml/mod-01>ocamlc -c CharPQueue.ml
[17:16]cazzola@surtur:~/lp/ml/mod-01>ocamlc -o main CharPQueue.cmo main.ml
[17:19]cazzola@surtur:~/lp/ml/mod-01>ls
CharPQueueAbs.cmi  CharPQueueAbs.mli  CharPQueue.cmi  CharPQueue.cmo  CharPQueue.ml  mai
```

```
open CharPQueue.AbstractPrioQueue;;
let x = insert empty 1 'a' ;;
```

In this case the file names for the module implementation and interface must be different (and start with a capital letter).

The OCaML Module System
Separate Compilation (Cont'd).

The implementation and interface of the module can share the same file name (apart of the suffix)

- module, sig and struct keywords are dropped

The module name comes after the module file name.

```
[17:39]cazzola@surtur:~/lp/ml/mod-02>ls
CharPQueue.mli  CharPQueue.ml      main.ml
[17:39]cazzola@surtur:~/lp/ml/mod-02>ocamlc -c CharPQueue.mli
[17:39]cazzola@surtur:~/lp/ml/mod-02>ocamlc -c CharPQueue.ml
[17:39]cazzola@surtur:~/lp/ml/mod-02>ocamlc -o main CharPQueue.cmo main.ml
[17:39]cazzola@surtur:~/lp/ml/mod-02>ls
CharPQueue.cmi  CharPQueue.cmo  CharPQueue.ml  CharPQueue.mli  main*  main.cmi  main.cmo  mai
```

This is how the signature looks:

```
type priority = int          (* still concrete *)
type char_queue              (* now abstract *)
val empty : char_queue
val insert : char_queue -> int -> char -> char_queue
val extract : char_queue -> int * char * char_queue
exception QueueIsEmpty
```


The OCaML Module System
Functors.

Functors are «functions» from structures to structures.

This means

- fixed the signatures of the input and output structures; then
- the implementation details can change without affecting any of the modules that use it.

Functors allow to

- avoid duplication and
- increase orthogonality

in a type safe package.

`is_balanced()` checks that a string uses balanced parenthesis.

```
let is_balanced str =
  let s = Stack.empty in try
    String.iter
      (fun c -> match c with
          '(' -> Stack.push s c
        | ')' -> Stack.pop s
        | _ -> ()) str;
      Stack.is_empty s
  with Stack.EmptyStackException -> false
```

```
module type StackADT =
 sig
  type char_stack
  exception EmptyStackException
  val empty : char_stack
  val push : char_stack -> char ->  unit
  val top :  char_stack -> char
  val pop :  char_stack -> unit
  val is_empty :  char_stack -> bool
end
```

The idea is to iterate on the string and

- to push any open parenthesis on a stack; and
- to pop it when a close parenthesis is encountered

If the algorithm ends with an empty stack the string is balanced otherwise it is unbalanced.

`is_balanced()` checks that a string uses balanced parenthesis.

```
module Matcher (Stack : StackADT.StackADT) =
  struct
    let is_balanced str =
      let s = Stack.empty in try
        String.iter
          (fun c -> match c with
              '(' -> Stack.push s c
            | ')' -> Stack.pop s
            | _ -> ()) str;
          Stack.is_empty s
      with Stack.EmptyStackException -> false
end
```

```
module type StackADT =
 sig
  type char_stack
  exception EmptyStackException
  val empty : char_stack
  val push : char_stack -> char ->  unit
  val top :  char_stack -> char
  val pop :  char_stack -> unit
  val is_empty :  char_stack -> bool
end
```

Matcher is a functor that binds our algorithm to a **Stack** abstract data type.

```
#use "balanced.ml" ;;

module Matcher :
  functor (Stack : StackADT.StackADT) -> sig val is_balanced : string -> bool end
```

Instantiation make concrete the algorithm.

Functors **must** be instantiated.

```
module UnboundedStack = struct
  type char_stack = {
      mutable c : char list
  }
  exception EmptyStackException

  let empty = { c = [] }

  let push s x = s.c <- x :: s.c

  let pop s =
    match s.c with
      hd::tl -> s.c <- tl
    | []     -> raise EmptyStackException

  let top s =
    match s.c with
      hd::_ -> hd
    | []    -> raise EmptyStackException

  let is_empty s = (s.c = [])
end;;
```

```
module BoundedStack = struct
  type char_stack = {
      mutable c: char array;
      mutable top: int }
  exception EmptyStackException

  let empty = {top=0; c=Array.make 10 ' '}
  let push s x =
     s.c.(s.top) <- x; s.top <- s.top+1

  let pop s =
    match s.top with
      0 -> raise EmptyStackException
    | _ -> s.top <- s.top -1

  let top s =
    match s.top with
      0 -> raise EmptyStackException
    | _ -> s.c.(s.top)

  let is_empty s = (s.top = 0)
end;;
```

Both implementations adhere to the **StackADT** interface.


```
[18:15]cazzola@surtur:~/lp/ml>ocaml
# module M0 = Matcher(BoundedStack);;
module M0 : sig val is_balanced : string -> bool end

# module M1 = Matcher(UnboundedStack);;
module M1 : sig val is_balanced : string -> bool end

# M0.is_balanced "a(b)+c(a+d(e-f))";;
- : bool = true
# M0.is_balanced "a(b(+c(a+d(e-f))";;
- : bool = false
# M1.is_balanced "a(b)+c(a+d(e-f))";;
- : bool = true
# M1.is_balanced "a(b(+c(a+d(e-f))";;
- : bool = false
```

References

- Davide Ancona, Giovanni Lagorio, and Elena Zucca.
  Linguaggi di Programmazione.
  Città Studi Edizioni, 2007.
- Greg Michaelson.
  An Introduction to Functional Programming through λ-Calculus.
  Addison-Wesley, 1989.
- Larry C. Paulson.
  ML for the Working Programmer.
  Cambridge University Press, 1996.