CS3110 Spring 2016 Lecture 6: Modules

Modules are the O part of OCaml, influenced by objects in Java. We see the evolution: Classic ML, CambridgeML (CAML), and finally OCaml. David McQueen is writing the history of this evolution.

Motivations

1. Managing specifications and code for large projects, local recoding does not impact other "modules", can discuss the system structure and design.

2. Enriching the type system with module signatures and functors. Module structures are a new kind of object, like a tuple. The type theories of proof assistants are richer and motivate PL extensions. We will look briefly at the way rich type systems handle these concepts.

3. Connections to the area of Mathematical Knowledge Management (MKM) as conducted by modern proof assistants and theorem provers. The MKM efforts introduce a classification of mathematical topics.

   We see this in our refernce to the ring of integers and the field of *rationals* and the field of *reals*.

   nat – for Natural numbers

   int – extends *nat*, provides structure of a ring.

   rat – extends *int*, provides structure of a field

   reals – extends *rat* and provides a field

   complex – extends *reals* and provides a field

   Types corresponding to propositional logic:

- Logical types:
  $\alpha * \beta$ for & (*and*)
  $\alpha \rightarrow \beta$ for *implies*
  $L\alpha||R\beta$ for *or*
- $\alpha * \beta \Rightarrow \alpha$ is a valid logical principle, so are:
  $\alpha \Rightarrow L\alpha||R\beta$
  $\alpha * \beta \Rightarrow \beta * \alpha$, etc.

There are two aspects to modules:

- The type part, called the signature.
  `module type` $Name$ = `sig.....end`
  ↖ defintions
- The value part, called the structure.
  `module type` *modname* = `struct.....end`
  ↖ implementation

Example from PS2:

```
module type N = sig
   type t
   val zero : t
   val succ : t -> t
   val eq : t -> t -> bool
end
```

The centerpiece of PS2 is a module for the rational numbers, $\mathbb{Q}$.

```
module type Q = sig   defs   end
```

We have provided the signature, your job is to build the structure, called Rational.

Functional stacks and queues, dictionaries, fractions

Functional data structures

In this recitation, we look at examples of structures and signatures that implement data structures. We show that stacks and queues can be implemented efficiently in a functional style.

What is a functional stack, or a functional queue? It is a data structure for which the operations do not *change* the data structure, but rather create a new data structure, with the appropriate modifications, instead of changing it in-place. In imperative languages, data operations generally support *destructive update* — "destructive" in the sense that after the update is done, the original data structure is gone. Functional abstractions support*nondestructive updates*: the original value is still around, unmodified, so code that was using it is unaffected. For efficiency, it is important to implement nondestructive updates not by creating an entirely new data structure, but by sharing as much as possible with the original data structure.

Stacks

Recall a stack: a last-in first-out (LIFO) queue. Just like lists, the stack operations fundamentally do not care about the type of the values stored, so it is a naturally polymorphic data structure.

Here is a possible signature for functional stacks:

```
module type STACK =
  sig
    (* A stack of elements of type 'a. We write  to
     * denote a stack whose top element is a1, with successive
     * elements a2, a3,...an. *)
    type 'a stack

    exception EmptyStack

    (* The empty stack. *)
    val empty : 'a stack

    (* Whether this stack is empty. *)
    val isEmpty : 'a stack -> bool

    (* Returns a new stack with x pushed onto the top. *)
    val push : ('a * 'a stack) -> 'a stack

    (* Returns a new stack with the top element popped off. *)
    val pop : 'a stack -> 'a stack

    (* The top element of the stack. *)
    val top : 'a stack -> 'a

    (* map(f) maps one stack into a corresponding stack, using f. *)
    val map : ('a -> 'b) -> 'a stack -> 'b stack

    (* app(f) applies f to every element of the stack, top to bottom. *)
    val app : ('a -> unit) -> 'a stack -> unit
  end
```

This signature specifies a parameterized abstract type for stack. Notice the type variable `'a`. The signature also specifies the empty stack value, and functions to check if a stack is empty, and to perform push, pop and top operations on the stack. Moreover, we specify functions map and app to walk over the values of the stack.

We also declare an exception `EmptyStack` to be raised by top and pop operations when the stack is empty.

Here is the simplest implementation of stacks that matches the above signature. It is implemented in terms of lists.

```
module Stack : STACK =
  struct
    type 'a stack = 'a list
    exception EmptyStack

    let empty : 'a stack = []

    let isEmpty (l : 'a stack) : bool = l = []

    let push ((x : 'a), (l : 'a stack)) : 'a stack = x :: l

    let pop (l : 'a stack) : 'a stack =
       match l with
         [] -> raise EmptyStack
       | x :: xs -> xs

    let top (l : 'a stack) : 'a =
       match l with
         [] -> raise EmptyStack
       | x :: xs -> x

    let map (f : 'a -> 'b) (l : 'a stack) : 'b stack = List.map f l
    let app (f : 'a -> unit) (l : 'a stack) : unit = List.iter f l
  end
```

Up until now, we have been defining exceptions solely in order to raise them and interrupt the executing program. Just like in Java, it is also possible to catch exceptions, which is termed 'handling an exception' in OCaml.

As an example, consider the following example. In the above code, we have implemented top and pop respectively as functions that return the first element of the list and the rest of the list. OCaml already defines functions to do just that, namely `List.hd` and `List.tl` (for head and tail). The function `hd` takes a list as argument and returns the first element of the list, or raises the exception `Failure` if the list is empty. Similarly for `tl`. One would like to simply be able to write in `Stack`:

```
let top (l : 'a stack) : 'a = List.hd l
let pop (l : 'a stack) : 'a stack = List.tl l
```

However, if passed an empty stack, `top` and `pop` should raise the `EmptyStack`exception. As written above, the exception `Failure` would be raised. What we need to do is intercept (or handle) the exception, and raise the right one. Here's one way to do it:

```
let top (l : 'a stack) : 'a =
  try List.hd l with Failure _ -> raise EmptyStack
let pop (l : 'a stack) : 'a stack =
  try List.tl l with Failure _ -> raise EmptyStack
```

The syntax for handling exceptions is as follows:

```
try e with exn -> e'
```

where `e` is the expression to evaluate, and if `e` raises an exception that matches `exn`, then expression `e'` is evaluated instead. The type of `e` and `e'` must be the same.

Signatures

To successfully develop large programs, we need more than the ability to group related operations together in a module. We need to be able to use the compiler to enforce the separation between different modules, which prevents bad things from happening. Signatures are the mechanism that enforces this separation.

Signature declarations that have the following syntax:

```
module type SIGNAME = sig definitions end
```

By convention, the signature name *SIGNAME* is all in capital letters. The definitions of a signature declare a set of types and values that any module implementing it must provide. The definitions of a signature may be `type` definitions, `val` definitions to define the type signature of a name, and `exception` definitions to specify exceptions that module can raise.

A module that implements a particular signature specifies the name of that signature in its definition, after the module name and separated by a : as with types. The signature must be defined before the module is defined.

```
module ModuleName : SIGNAME = struct implementation end
```

A module that implements a signature must specify concrete types for the abstract types in the signature and provide all the declarations in the signature. Only the abstract types are accessible outside the module, unless the signature exposes the definition. Only declarations in the signature are accessible outside of the module (for instance functions defined in the implementation but not in the signature are not accessible).

For example, here is a signature for a simple set data abstraction, together with two implementations of that interface using lists:

```
(* Set data abstraction with union and intersection *)

module type SET = sig
  type 'a set
  val empty : 'a set
  val mem : 'a -> 'a set -> bool
  val add : 'a -> 'a set -> 'a set
  val rem : 'a -> 'a set -> 'a set
  val size: 'a set -> int
  val union: 'a set -> 'a set -> 'a set
  val inter: 'a set -> 'a set -> 'a set
end

(* Implementation of sets as lists with duplicates *)

module Set1 : SET = struct
  type 'a set = 'a list
  let empty = []
  let mem = List.mem
  let add x l = x :: l
  let rem x = List.filter ((<>) x)
  let rec size l =
    match l with
    | [] -> 0
    | h :: t -> size t + (if mem h t then 0 else 1)
  let union l1 l2 = l1 @ l2
  let inter l1 l2 = List.filter (fun h -> mem h l2) l1
end

(* Implementation of sets as lists without duplicates *)

module Set2 : SET = struct
  type 'a set = 'a list
  let empty = []
  let mem = List.mem
  (* add checks if already a member *)
  let add x l = if mem x l then l else x :: l
  let rem x = List.filter ((<>) x)
  let size = List.length (* size is just length if no duplicates *)
  let union l1 l2 = (* check if already in other set *)
    List.fold_left (fun a x -> if mem x l2 then a else x :: a) l2 l1
  let inter l1 l2 = List.filter (fun h -> mem h l2) l1
end
```

Queues

Let us write an example more interesting than stacks. After all, from the above, one can see that they are just lists. Consider the queue data structure, a first-in first-out data structure. Again, we consider functional queues. Here is a possible signature:

```
module type QUEUE =
  sig
    type 'a queue
    exception EmptyQueue

    val empty : 'a queue
    val isEmpty : 'a queue -> bool

    val enqueue : ('a * 'a queue) -> 'a queue
    val dequeue : 'a queue -> 'a queue
    val front : 'a queue -> 'a

    val map : ('a -> 'b) -> 'a queue -> 'b queue
    val app : ('a -> unit) -> 'a queue -> unit
  end
```

The simplest possible implementation for queues is to represent a queue via two stacks: one stack A on which to enqueue elements, and one stack B from which to dequeue elements. When dequeuing, if stack B is empty, then we reverse stack A and consider it the new stack B.

Here is an implementation for such queues. It uses the stack structure `Stack`, which is rebound to the name `S` inside the structure to avoid long identifier names.

```
module Queue : QUEUE =
    struct

      module S = Stack

      type 'a queue = ('a S.stack * 'a S.stack)
      exception EmptyQueue

      let empty : 'a queue = (S.empty, S.empty)
      let isEmpty ((s1, s2) : 'a queue) =
        S.isEmpty s1 && S.isEmpty s2

      let enqueue ((x : 'a), ((s1, s2) : 'a queue)) : 'a queue =
        (S.push (x, s1), s2)

      let rev (s : 'a S.stack) : 'a S.stack =
        let rec loop ((prev : 'a S.stack), (curr : 'a S.stack))
        : 'a S.stack =
          if S.isEmpty prev
          then curr
          else loop (S.pop prev, S.push (S.top prev, curr))
      in
        loop (s, S.empty)

      let dequeue ((s1, s2) : 'a queue) : 'a queue =
        if S.isEmpty s2
        then try (S.empty, S.pop (rev s1))
             with S.EmptyStack -> raise EmptyQueue
        else (s1, S.pop s2)

      let front ((s1, s2) : 'a queue) : 'a =
        if (S.isEmpty s2)
        then try S.top (rev s1)
             with S.EmptyStack -> raise EmptyQueue
        else S.top s2

      let map (f : 'a -> 'b) ((s1, s2) : 'a queue) : 'b queue =
        (S.map f s1, S.map f s2)

      let app (f : 'a -> unit) ((s1, s2) : 'a queue) : unit =
        S.app f s2;
        S.app f (rev s1)

    end
```

We learned about folding last week. In the above implementation, the stack reversal could have been done using fold. However, since the Stack module does not specify a fold operation, and the implementation of the Stack as a list is hidden from the Queue module, we need something more. The Stack signature should specify a fold operation that will help its users to iterate over its elements.