Code Review 4: February 28, 2020

Modules, Functors, and Prioqueues

Hakeem Angulu, hangulu@college.harvard.edu

Learning Goals

- Understand why we use modules
- Understand the difference between a module's type and its implementation
- Relate functors to functions
- Create and understand the use of priority queues

Logistical Notes

- Midterm 1 is next Monday from 7:40 pm - 9:10 pm.
    - practice, practice, practice
    - the best thing you can do is work on the labs (redo them if possible), then the exercises in the book, then the problem sets

---

Modules

A module is a package of types and values (eg. variables and functions). Modules have type signatures, which are defined separately from the module's implementation. Technically it isn't necessary to define a module's type signature, but it is often helpful to do so. The syntax for defining a module type signature and a module implementation is shown below.

```
module type Example_Signature =
  sig
    type weird_type
    type weird_type_2
    val x : weird_type
    val y : float
    val flip : float -> float
end
```

```
module Example_Implementation : (Example_Signature with type weird_type = int) =
  struct
    type weird_type = int
    type weird_type_2 = int
    type weird_type_3 = int
    let x = 0
    let y = 0.
    let z = 3.6
    let flip l = l +. z
end
```

Exercise 1: True or false: every type and value defined in the module type must be implemented in the module implementation.

True.

Exercise 2: True or false: you cannot implement types and values not defined in the module type in the module implementation.

False

Pay note to the type signature of `Example_Implementation`. We specify that it is of type `Example_Signature with type weird_type = int`. By specifying this, now whenever we refer to `Example_Implementation.weird_type` outside of the module's definition, OCaml knows that we mean `int`. This is not the same for `weird_type_2`. If we refer to `Example_Implementation.weird_type_2`, OCaml does not know that we mean `int`.

This is a key feature of module implementations that you saw in lab, and that will be useful for the problem set. It allows the developer to mask the inner workings of the implementation for security reasons.

Exercise 3: What would the following return when run in an OCaml repl after defining the module above?

```
 >> Example_Implementation.x ;;
0

>> Example_Implementation.y ;;
0.

>> Example_Implementation.z ;;
(* Nothing. This would throw an error because z is not in the signature. *)

>> Example_Implementation.flip 0.4 ;;
4.0
```

Files as Modules

How do we build large projects? Hopefully as you've seen, you can input the following into utop.

```
(* #mod_use "fibonacci.ml" *)
module Fibonacci :
  sig
    type length = Infinite | Finite of int
    type info = { name : string; length : length; inventor : string; }
    val name : string
    val length : length
    val inventor : string
    val info : info
    val eval : int -> int option
    val exists : int -> bool
  end
```

Modules are an excellent way to build mechanics and abstract or package them for easy use. Data structures, objects, function collections, and more can all be built and abstracted and re-used as modules.

Local Open

```
let open Math in
max([cos(pi); sin(pi)]) ;;
```

You'll notice that these functions are written with parentheses to wrap the arguments. This is mainly to differentiate them from the builtin functions (you can write them with or without the parentheses). Used tastefully, this can make code cleaner but for the most part it might be easier to just open it globally or use dot notation.

You can build large programs in OCaml across multiple files by using functions you develop as modules.

Interfaces

Let's talk through the implementation of `IntListStack`.

```
module IntListStack =
  struct
    exception EmptyStack
    type stack = int list

    (* Returns an empty stack *)
    let empty () : stack = []

    (* Add an element to the top of the stack *)

    let push (i : int) (s : stack) : stack = i :: s

    (* Return the value of the topmost element on the stack *)
    let top (s : stack) : int =
      match s with
      | hd :: tl -> hd
      | _ -> raise EmptyStack

    (* Return a modified stack with the topmost element removed *)
    let pop (s : stack) : stack =
      match s with
      | hd :: tl -> tl
      | _ -> raise EmptyStack

  end ;;
```

We later defined the interface for this module in `INT_STACK`.

```
module type INT_STACK =
  sig
    type stack
    exception EmptyStack
    val empty : unit -> stack
    val push : int -> stack -> stack
    val top : stack -> int
    val pop : stack -> stack
  end ;;
```

How do we define a "safe" `IntListStack`?

```
module SafeIntListStack = (IntListStack : INT_STACK) ;;
```

How do we create a new stack? Use the functions specified in the interface.

```
let safe_stack () : SafeIntListStack.stack =
   let open SafeIntListStack in
   empty () |> push 5 |> push 1 ;;
```

Exercise 4: Say you're given the implementation of an int stack, as above. Write a function `multi_stack` that returns the first pair of consecutive equal elements in the stack if it exists, and throw an `EmptyStack` error otherwise.

```
open SafeIntListStack

let rec multi_stack (s : stack) : int * int =
  let first = top s in
  let rest = pop s
  let second = top rest in
  if first = second then (first, second)
  else multi_stack rest ;;
```

Functors

Functors are objects in OCaml that take modules as arguments and return a new module. They are essentially functions for modules. An example of a functor `Inc` is shown below.

```
(* Define a new module signature *)
module type Value =
  sig
    val x : int
  end

(* define a new module called Value_Zero of type Value *)
module Value_Zero : Value =
  struct
    let x = 0
  end

(* Define a functor which takes a Value and returns a Value *)
module Inc (V: Value) : Value =
  struct
    let x = V.x + 1
  end

(* Use Inc to create new modules Value_One and Value_Two *)
module Value_One = Inc (Value_Zero)
module Value_Two = Inc (Value_One)
```

Exercise 5: Similarly, define a functor Square_Value which takes a Value module as an argument and returns a new Value module whose x-value is the square of the given x-value. Example:

```
 # module New_Value = Square_Value (Value_One) ;;
# New_Value.x ;;
- : int = 4
```

```
module Double_Value (V : Value) : Value =
  struct
    let x = V.x ** V.x
  end
```

Priority Queues

Priority Queues (or prioqueues for short) are data structures that allow you to insert elements and remove them based on their priority.

The highest priority element will always be removed first, regardless of whether it was inserted after a lower priority element. We specify that if a new element has the same priority as an element already in the priority queue, then the new element is inserted into the queue after the element that was already there.

For example, let's say we're inserting letters into a priority queue, where a letter's position in the alphabet is its priority (A comes before B). If we inserted B, then C, then A, our priority queue would be: A, B, C.

Some important functions for a Priority Queue are: - empty – returns an empty priority queue - is_empty – which returns true if a priority queue is empty and false otherwise - add – adds an element to a priority queue - take – returns the first element from a priority queue along with the priority queue minus the first element

There are many ways to represent a priority queue. One such way is through a list: we store elements with higher priority at the front of the list and elements with lower priority at the end of the list.

Exercise 6: Implement the addition function for an integer priority queue, represented by an int list, for whom the highest priority elements are the lowest in value.

```
let rec add (lst : int list) (el : int) : int list =
  match lst with
  | [] -> [el]
  | hd :: tl -> if hd > el then el :: lst else hl :: add tl el ;;
```

For this and future code reviews, please do not hesitate to reach out to me with any questions or concerns. My email can be found at the top of this document.