Code Review 4: Modules, Functors, and Priority Queues

CS51
Melissa Kwan

1 Logistics

1. **Midterm on Mon. 3/7 from 7:40 - 9:10pm.** Next week will be dedicated to review!

2 Modules and Functors (the short version)

1. A **module** is a collection of values.
2. A **functor** is a function from modules to modules.
3. Modules and functors allow us to **abstract functionality** and **avoid redundancy** (two big themes in this class and in software development in general).
4. The interface (signature) of a module is separate from its implementation.

A syntax note. Syntax for modules and functors can be tricky. Don't worry about the position of `sig` and `end` and the formatting of your `.mli` — when we ask you to implement these, we'll always give you a reference. Focus on the underlying concepts instead.

3 Invariants

Maintaining **invariants** allows us to simplify our use of the data, because we can assume that the invariant is true. It's like a stamp of quality.

Example: An invariant of a queue invariant is first-in, first-out (FIFO) behavior. The first item entered using enqueue will always be the first item removed using dequeue. If we don't enforce this invariant, then we've defeated the purpose of using a queue.

Exercise 3.1. Under what conditions could a user break this invariant?

Exercise 3.2. How could we prevent users from breaking this invariant?

4 Modules

A **module** is a collection of values. It's a package of useful related features that provide functionality without allowing the user to see their implementation. In OCaml, every piece of code is wrapped in a module.

4.1 Files are modules too!

OCaml takes the name of the module / signature directly from the file name (`module.ml` contains the module, and `module.mli` contains the signature.) As a result, if your module is the whole file, you don't need to bookend it with `struct` and `end` or `sig` and `end`. Below are two ways to use module files. Local opens are preferable if you only need the module definitions in a very small portion of the file.

```
(* Global Open *)
open ExpressionLibrary
parse "x^2" ;;

(* Local Open *)
let open ExpressionLibrary in parse "x^2" ;;
```

4.2 Anatomy of an in-file module

Defined in-file, modules are made of an interface (bookended by `sig end`) and an implementation (bookended by `struct end`). Here's an example of an integer queue.

```
module type INT_QUEUE =
    sig
        type int_queue
        val empty_queue : int_queue
        val enqueue : int -> int_queue -> int_queue
        val dequeue : int_queue -> int * int_queue
    end ;;
```

```
module IntQueue : INT_QUEUE =
    struct
        type int_queue = int list
        let empty_queue = []
        let enqueue elt q =
            q @ [elt]
        let dequeue q =
            match q with
            | [] -> raise (Invalid_argument "dequeue: empty queue")
            | hd :: tl -> hd, tl
    end ;;
```

Exercise 4.1. What would happen if we changed the implementation of `empty_queue` to return 0 instead of an empty list?

Exercise 4.2. What would happen if we got rid of the `INT_QUEUE` output type and try running the module implementation by itself?

Exercise 4.3. Suppose you wanted to create a queue with elements of a different type. Is there a easier way than copy pasting `INT_QUEUE` and `IntQueue` and changing a few lines?

5 Functors

A **functor** is a function from modules to modules. Functors allow us to create modules abstractly rather than hard-coding them. With a well designed functor, we can efficiently create several different modules that have similar functionality (perhaps only differing in type of element used).

Here, `TYPE` is a signature, `QUEUE` is a signature, and `MakeQueue` is a functor that takes in a module that fits the signature `TYPE` and outputs a module that fits the signature `QUEUE` (with a sharing constraint, which we'll talk about later.

Intuition: One natural metaphor to think about when using modules is the "blueprint-house" metaphor. A signature is a type, which means it's like a blueprint: it might tell you that a house has a bathroom, a front lawn, and a bedroom, but it won't tell you anything about what those look like. These are indicated with the header `module type`, and the contents are bookended by `sig` and `end`. A module is like a house: its header starts with `module`, not `module type`, and its contents are bookended by `struct` and `end`.

A functor, in this analogy, is a construction company that takes in a plan and outputs a house. Why does it make sense to have a "plan"? Let's say most houses are pretty similar: even though they have different versions of bathrooms, front lawns, and bedrooms, the functions `doChores` and `cleanYard` are effectively the same for all houses. It would be annoying to have to rewrite the same `doChores` and `cleanYard` functions again and again and again when creating houses that are almost the same. To avoid this, we create a "mini module" signature that only specifies the type of the bathroom, front yard, and bedroom. We then create a mini module that follows that mini module signature and give it to the construction company, and the construction company builds the entire house.

Let's translate this metaphor to the example at hand. `QUEUE` is the blueprint for the house, `MakeQueue` is the construction company, and `TYPE` is the outline for the plan. `MakeQueue` takes in an actual plan (that follows the outline) and builds an actual house (that follows the blueprint).

```
module type TYPE =
    sig
        type t
    end ;;

module type QUEUE =
    sig
        type element
        type queue
        val empty : queue
        val enqueue : element -> queue -> queue
        val dequeue : queue -> element * queue
    end ;;

module MakeQueue (Elements: TYPE)
                : (QUEUE with type element = Elements.t) =
    struct
        type element = Elements.t
```

```
        type queue = element list
        let empty = []
        let enqueue (elt : element) (q : queue) : queue = q @ [elt]
        let dequeue (q : queue) : element * queue =
            match q with
            | [] -> raise (Invalid_argument "dequeue: empty queue")
            | hd :: tl -> hd, tl
    end ;;
```

Exercise 5.1. Use this functor to define a `FloatQueue` and an `IntQueue`.

5.1 Sharing constraints

In our implementation of the functor `MakeQueue`, why did we need to specify `QUEUE with type element = Elements.t`?

In the textbook, we created a module `IntQueue` that was actually implemented as an `int list`. But we wanted to hide this fact from the user to prevent them from violating the invariant.

For the same reasons, if we define a `QUEUE` with a type `element`, the queue won't reveal the implementation of the `QUEUE.element`. So even if we called `MakeQueue` with `struct type t = int end`, it won't accept `int` inputs into its functions. But in this case, instead of being helpful, it ends up being self-defeating. We *want* the user to know that the elements are `int` values — that was the entire point of queue!

We can get around this problem with sharing constraints, which allows you to control exactly which part of the implementation you want the user to see. You can specify exactly what is revealed with the syntax `SIGTYPE with [specify implementation here]`. This is exactly what the sharing constraint `QUEUE with type element = Elements.t` does. Note that `Elements` is the name we assigned to the variable struct in the input. If we named the input `T`, the sharing constraint would be `QUEUE with type element = T.t`.

The user needs to interface with a module of type `IntQueue` using `int` values. However, `QUEUE` currently doesn't reveal the type of the element. The type of `QUEUE.element` is technically an `int`, but `QUEUE` only recognizes elements of type `QUEUE.element`.

5.2 A "Smol" Functor (Inspired by Benton Liang (inspired by Ezra))

```
module type IntM =
    sig
        val x : int
    end

module Seven : IntM =
    struct
        let x = 7
    end
```

Exercise 5.2. Write a functor called `Increment` that takes in an `IntM` module and returns a new module that represents that value incremented by 1.

6 Priority Queues (Heaps)

Priority Queues 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. Here are the important operations of a priority queue.

- `empty`: Returns an empty priority queue.
- `is_empty`: 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.