Code Review 4 Handout

Josh Seides

Topic Outline

This week we are looking at a very important, useful construct in OCaml that helps with abstraction and making code more reusable and scalable.

- modules
- signatures and interfaces
- polymorphic abstract types
- functors

Modules

Using **modules** is an important way to take advantage of abstraction in OCaml. Modules are basically a grouping of types, functions, and other values together under one value name. They provide a way to organize and scale repeated blocks of code.

Problem 1 *What are some example use cases for modules?*

Problem 2 *Why create modules in the first place?*

Making Modules

Problem 3 *Create a `TFJosh` module that includes a `demographic` type and value (which should be a record with age, gender, and hometown), a `name` value, a `favorite int -> int -> bool` function, and a `print_demographic` function of type `demographic -> string`.*

Alternatives Ways to Create Modules In Problem 2, we created an explicit module. There are also other ways to create modules.

- built-in modules
- files as modules

Accessing Modules There are also multiple ways to access modules

- top-level `open`
- local `open`

Problem 4 *How do you decide when to use top-level vs. local opens?*

Signatures and Interfaces

Consider our example for `TF`

```
module TF =
    struct
        type demographic = {
            age : int;
            gender : string;
            hometown : string;
        }
        let name : string = "Josh"
        let favorite = ( > )
        let print_demographic (d : demographic) : string =
            (string_of_int d.age) ^ " / " ^ d.gender ^ " / " ^ d.hometown
    end ;;
```

What if we wanted to factor out everything to reuse basic data types, since apparently the `Professor` type has the exact same values, except it also has a few additional parts such as a `lecture` function and `years_experience` value?

We should create a **module signature** (also called an **interface**). These basically explain the types of the exposed values inside a module of that module signature. It is, in a sense, a blueprint that people using your module can reference to know what functionality is available to use for that module.

Problem 5 *Why have module signatures?*

Problem 6 *How does this module vs. module signature distinction relate to something like the `List` module?*

Problem 7 *Make a type signature for our `TF` module.*

Problem 8 *Explicitly type the module implementation defined above as matching the module signature `TF`.*

It is difficult at first to understand why all of this discussion about modules and interfaces is relevant at all. Consider this example from lab

```
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
            | [] -> raise EmptyStack
            | h :: _ -> h

        (* Return a modified stack with the topmost element removed *)
        let pop (s : stack) : stack =
            match s with
            | [] -> raise EmptyStack
            | _::t -> t
    end ;;
```

Problem 9 *What is bad about the implementation of the `int stack` above?*

Problem 10 *How would we write a module signature for `int stack` to better abstract the implementation?*

After implementing Problem 10, we can create a more abstracted stack

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

Aside on Reverse Application Operator The reverse application operator is good design and easier to reason about. This

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

goes to this

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

Polymorphic Abstract Types

These are similar to polymorphic ADTs. This is especially useful for modules that can be implemented on multiple data types.

```
module type QUEUE =
    sig
        exception EmptyQueue
        type 'a queue
        val empty : unit -> 'a queue
        val enqueue : 'a -> 'a queue -> 'a queue
        val front : 'a queue -> 'a
        val dequeue : 'a queue -> 'a queue
    end ;;
```

Functors

Functors are simply functions that take in modules as input and return modules as ouput. A good way to review previous material and understand functors is to draw comparisons between records and modules.

Problem 11 *Create a record type* `integer` *and a module signature* `Integer` *each with one value of type* `int` *named* `x`*.*

Problem 12 *Create a record* `one` *and a module* `One` *to represent the number 1 in their respective formats.*

Problem 13 *Create a function for records that adds their* `x` *fields and a functor for modules that adds their* `x` *values.*

Problem 14 *(Next level stuff.) Create a higher-order function and functor that takes in a function/functor* `a -> a -> a` *and one value* `i` *of type* `a` *(where* `a` *is either* `integer` *or* `Integer` *for functions and functors, respectively) and returns the function/functor applied on* `i` *twice.*

Problem 15 *How can we use partial application on the function/functor we just created?*