Code Review 2 Handout

Josh Seides

Topic Outline

This week we are moving from basic (first-order) functional programming to a very powerful technique we will use for the rest of the course: higher-order functional programming. We will also look at a few other advanced concepts that are often used with higher-order programming and functional programming in general.

- higher-order functional programming
- partial application
- polymorphism
- option types
- abstract data types and records

Higher-Order Functional Programming

A higher-order function either takes a function as input or returns a function as output (or both).

Problem 1 The function below is a higher-order function. What is its type?

```
let not_actually_composition f g a b =
    (f (a +. 3.5)) ^ (g (a +. 2.5)) ^ (b ^ "Hello") ;;
```

There are three really important higher-order functions that we will see over and over again. First, a motivation. What is similar about the below functions?

```
let rec double l =
    match l with
    | [] -> []
    | h :: t -> (h * 2) :: double t ;;

let rec prods l =
    match l with
    | [] -> []
    | (fst, snd) :: t -> (fst * snd) :: prods t ;;
```

Why is this bad design? We can remedy this with `List.map`

```
let rec map f l =
    match l with
    | [] -> []
    | h :: t -> (f h) :: map f t ;;
```

There are two other important higher-order functions as well: `List.fold_right` and `List.filter`.

```
let rec fold_right f l acc =
    match l with
    | [] -> acc
    | h :: t -> f h (fold_right f t acc) ;;
```

```
let rec filter f l =
    match l with
    | [] -> []
    | h :: t -> if (f h) then h :: filter f t else filter f t ;;
```

Problem 2 Implement a `sum` function that sums all elements in a list.

Problem 3 Implement a `tuple_generator` function that changes an `int list` into an `int list` of the original number and its negative.

Problem 4 Remember `gcd` from Lecture 1?

```
let rec gcd_euclid a b =
    if b = 0
    then a
    else gcd_euclid b (a mod b) ;;
```

Implement the `gcd_finder` function that when given two integers and a list returns a list of all integers that are equal to the greatest common divisor of the two integers.

Partial Application

Partial application is a very powerful way to improve the design of your code. Basically, every function that seems to be taking in multiple arguments in reality only takes in one at a time. These two function definitions are equivalent (the first is syntactic sugar for the second)

```
let compare x y =
    if x > y then 2 else 3 ;;

let compare_long =
    (fun x ->
        (fun y ->
            if x > y then 2 else 3)) ;;
```

We can use this to partially apply functions. By providing only one (or a few) of the inputs to a function but not all at once, instead of returning the normal output type of a function we return another function. For example, if a function type is

```
int -> int -> int
```

Partially applying only the first input would result in an output of a function of type

```
int -> int
```

Problem 5 How can you take advantage of partial application in Problems 3 and 4?

Partial application is the foundation for currying vs. uncurrying. The main difference is

- currying: making functions take arguments separately, one at a time (example: `compare`)
- uncurrying: making functions take arguments together as one tuple (example: `compare_long`)

Problem 6 Implement `curry` and `uncurry` from lab. What are their types?

Polymorphism

If a function can be applied to inputs of multiple types, polymorphism is involved. This basically happens when the compiler cannot determine from an expression what the exact types of the inputs and/or outputs are.

Problem 7 What is the type of the following function?

```
let patriots_suck_question_mark a b =
    if a > b then "Yes" else "YES" ;;
```

Problem 8 What is the type of the following function?

```
let actually_composition f g a b =
    f (g (a ^ b)) ;;
```

Problem 9 What is the type of `List.map`?

Option Types

Options allow you to account for the possibility that some value can be an "error" or simply not definable. They are compound types (what are other examples of these kind of types?). The value constructor is

```
| None
| Some _
```

and the type constructor is

```
_ option
```

Problem 10 Implement `max_option` from lab.

Abstract Data Types and Records

Abstract data types are user-defined types. These are really helpful when you need to model data in a certain way and know the form of this data. An example would be mathematical expressions (!) that are easily modeled recursively as such

```
type binop = Add | Sub | Mul | Div | Pow

type unop = Sin | Cos | Ln | Neg

(type expression =
    | Num of float
    | Var
    | Binop of binop * expression * expression
    | Unop of unop * expression)
```

When defining ADTs, the definition resembles match patterns. Each pattern is a compound data type that takes in zero or more "inputs" (such as `Num` taking in one `float`). You can easily match on ADTs in an almost identical pattern to the type definition.

Problem 11 Define an ADT for `patriot` denoting the different types a New England Patriot can assume. Players should have numbers and names, coaches should have names, and there quarterbacks should be separate and have no information associated (because, honestly, Nick Foles got you good).

Problem 12 Implement a function that, given a `patriot`, prints information about the Patriot.

Records are also important. They are similar in a very loose way to objects in JavaScript. They have various fields that map to values. Records must start with a lowercase letter. For example

```
type student = {first : string; last : string; age : int; gpa : float} ;;
let student1 = {first="Bob"; last="Smith"; age=19; gpa=1.5} ;;
```

Record fields can be accessed with the dot syntax (such as `student1.first`).