Chapter 1

Higher Order Functions

1.1 Intro

We cover this topic in OCaml so the examples here will be mostly written in OCaml.

1.2 Functions as we know them

Let us first define a function. A function is something that takes in input, or an argument and then returns a value. As programmers, we typically think of functions as a thing that takes in multiple input and then returns a value. Technically this is syntactic sugar for the most part but that’s a different chapter. The important part is that we have this process that has some sort of starting values, and then ends up with some other final value.

In the past, functions may have looks liked any of the following

\[
\begin{align*}
\text{java} & \\
\text{int area(int length, int width)\
\quad \text{\quad return length * width;} & \\
\} & \\
\text{C} & \\
\text{int max(int* arr, int arr_length)\
\quad \text{int max = arr[0]} & \\
\quad \text{for(int i =1; i < arr_length; i++)} & \\
\quad \quad \text{if arr[i] > max} & \\
\quad \quad \quad \quad \text{max = arr[i];} & \\
\quad \quad \text{return max;} & \\
\end{align*}
\]
CHAPTER 1. HIGHER ORDER FUNCTIONS

Ruby

def char_sum(str)
    sum = 0
    str.each_char{|i| sum += i.ord}
    sum
end

(* OCaml *)

let circumference = 3.14 *. 2. *. radius

In these functions, our inputs were things like data structures, or 'primitives'. Ultimately, our inputs were some sort of data type supported by the language. Our return value is the same, could be a data structure, could be a 'primitive', but ultimately some data type that is supported by the language.

This should hopefully all be straightforward, a review and pretty familiar. The final note of this section is there are 3 (I would say 4) parts of a function. We have the function name, the arguments, and the body (and then I would include the return type or value as well). Again this shouldn’t be new, just wanted this here so we are all on the same page.

1.3 Higher Programming

As we said, functions take in arguments that can be any data type supported by the language. A higher order programming language is one where functions themselves are considered a data type. We saw this in OCaml, but let’s take a deeper look at it now.

Let us consider the following C program:

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int add1(int x){
    return x + 1;
}

int sub1(int x){
    return x - 1;
}

// return a function pointer
int* getfunc(){
    int (*funcs[2])(int) = {sub1, add1};
    return funcs[rand()%2];
}

// take in a function pointer
### 1.4. ANONYMOUS FUNCTIONS

This program has one function that returns a function pointer, and one function that takes in a function pointer. The idea of this is the basis of allowing functions to be treated as data. For most languages we have the ability to bind variables to data.

```c
int x = 3; // C, Java
y = 4 # Ruby
let z = 4.2;; (* OCaml *)
// idea
// variable = data
```

If we consider what is going on in the machine (Maybe recall from 216), then we know that any piece of data is just 1s and 0s stored at some memory address. The variable name helps us know which memory address we are storing things (so we don't have to remember what we stored at address 0x012f or something). When we want to then refer to that data, we use the memory address (variable name) and we retrieve that data. Why should a function be any different? We previously saw a pointer to a function being passed around, which just means the pointer to a list of procedures that are associated with the function. So in the case of higher order programming, we are just allowing functions to be passed in function data as arguments or be returned.

Thus we can say that a higher order function is one which takes in or returns another function. We can also avoid all these void pointers and casting and stuff in most functional languages like OCaml:

```ocaml
(* takes in a function) let apply f x = f x;;
(* returns a function *) let get_func = let add1 x = x + 1 in add1;;
```

#### 1.4 Anonymous Functions

So we just said that we bind data to variables if we want to use them again. Sometimes though, we don't want to use them again, or we have no need to store a function for repeated use. So we have this idea of anonymous functions. It is anonymous because it has
no (variable) name, which also means we cannot refer to it later. The syntax of an anonymous function is

\[(*) \text{add1}\text{*)}\]
\[
\text{fun}\ x \rightarrow x + 1
\text{(* add *)}
\]
\[
\text{fun}\ x\ y \rightarrow x + y
\text{(* general syntax *)}
\]
\[
(*\ \text{fun}\ \text{var}_1 : \text{t}_1 \ \text{var}_2 : \text{t}_2 \ \ldots \ \text{var}_x : \rightarrow \ \text{e} : \text{ty} \text{*)}
\]
\[
(*\ \text{has type} (\text{t}_1 \rightarrow \text{t}_2 \rightarrow \ldots \rightarrow \text{t}_x \rightarrow \text{ty})\text{*)}
\]

This is no different that just saying something like \(2 + 3\) instead of saying something like \((\ast\ \text{let}\ x = 2 + 3\ast)\). This means that we can do the same thing by doing something like

\[
\text{let add1}\ x = \text{fun}\ x \rightarrow x + 1
\text{(* function by itself, no variable *)}
\]

Which means \(\text{let add1}\ x = \text{fun}\ x \rightarrow x + 1\text{ is just syntactic sugar of let add1 fun x \rightarrow x + 1}\). This is because OCaml and other functional programming languages are based on this thing called lambda calculus, which is another chapter. But if we think about our mathematical definition of a function: it is something that takes in \(1\) input, and returns \(1\) output. So if each function should have \(1\) input, then what about something like \(\ast\ \text{let}\ \text{plus}\ x\ y = x + y\ast\)?

### 1.5 Partial Applications

Recall a section or something ago when we said that higher order functions can take in functions as arguments, and return functions as return values. Consider:

\[
\text{let plus}\ x\ y = x + y
\text{(* int \rightarrow int \rightarrow int *)}
\]

We said earlier that functions have types where the last thing in the type is the return value, and the first few items are the input types. We kinda lied. Let us consider:

\[
\text{let plus}\ x\ y = x + y
\text{(* int \rightarrow int \rightarrow int *)}
\]
\[
\text{let plus}\ x = \text{fun}\ y \rightarrow x + y
\text{(* int \rightarrow int \rightarrow int *)}
\]
\[
\text{(* int \rightarrow (int \rightarrow int) *)}
\]

This last function does have type \(\text{int} \rightarrow \text{int} \rightarrow \text{int}\) but consider what the syntax says. plus is a function that takes in an \(\text{int}\) but then returns a function that itself takes in an \(\text{int}\) and returns an \(\text{int}\). Which means we can actually define plus as

\[
\text{let plus = fun}\ x \rightarrow \text{fun}\ y \rightarrow x + y;;
\]

If we can define functions like this then we can do things like
1.6. CLOSURES

```ml
let plus = fun x -> fun y -> x + y
let add3 = plus 3 (* returning fun y -> 3 + y *)
add3 5 (* returns 8 *)
```

This is called a partial application of a function, or the process of currying. Not all functional languages support this unless the function is specifically defined as one which returns a function.

1.6 Closures

1.7 Common HOF

Here are some common higher order functions:

1.7.1 Map

1.7.2 Fold