Cornell CS 3110 - Functional Programming

Ivin Lee

September 25, 2020

No, I'm not from Cornell. But I think this set of lectures on OCaml is really useful for learning functional programming, which is why I am including it in these notes. I studied this course before my Java course because I think it will add value to learn Java course with prior knowledge of the functional programming paradigm.

1 Introduction

Functional languages:

- define computations as mathematical functions
- avoid mutable states

The former means that every computation takes in values and outputs values (more on this later).

The latter means that you never write $x = x + 1$, because when you do so you are changing (muting) the value of x.

Imperative languages:

- mutable states
- functions have side effects

The latter means that in

```
int addValue(Wallet wallet, int change) {
    wallet.money += change;
    return wallet.money;
}
```

apart from returning the new value of the wallet, it also changes the value in the Wallet instance.

It is claimed that functional programming allows one to write correct code easier, because:

- variables never change values
- functions never change other things (and cause trouble!)

It is possible to write code following the functional programming paradigm in any programming language, but some languages have been designed to make this easier. e.g.OCaml

Benefits of OCaml:

1. Immutable programming inbuilt (can't accidentally re-write a variable after being declared)
2. Functions passed as values
3. **Automatic type inference** - quite useful from experience

2 Functions

Value - expression that does not need any further evaluation

```
let x = if e1 then e2 else e3
let y = e2 +. e3
```

There is no need to declare the type as the type is inferred from e2 and e3 (e.g. if e2 and e3 are ints, then the type of x is inferred to be int).

The +. is a binary infix operator that takes in two floats and returns a float. In order for the type of y to be inferred correctly, it is necessary to use the operator, which tells the compiler(?) the return type of this expression.

Function Definitions

```
let rec pow x y =
    if y = 0 then 1
    else x * pow x (y - 1)
```

Note that you don't put brackets around the arguments. Also, all functions only return one thing, which matches the mathematical definition of a function.

```
let rec f x1 x2 ... xn = e
val f : t1 -> t2 -> ... tn -> te
```

The second line is output in the command line. Type `t -> u` is the type of a function that takes an input of type t and returns an output of type u.

Anonymous Functions

```
fun x -> x + 1
let inc = fun x -> x = 1
```

Can be used when you don't need the function to have a name, e.g. defining the function in another function's argument like in

```
let f x y =
  x + y
f ((fun x -> x + 1) 2) 3 (* returns 6*)
```

A function is a value. In the above example, $x + 1$ is not evaluated until 2 was passed to the anonymous function.

Note that the anonymous expression syntax is analogous to lambda expressions in math:

$$\lambda x.e$$
$$\text{fun } x- > e$$

Function Application Operator

```
f e (*equivalent to*) e |> f
5 |> inc |> square
```

5 is passed to inc and square

Functions are Values

Implication: functions can take functions as arguments and can return functions as results

3 Lists

Lists are constructed recursively, so the following constructors are equivalent

```
let x = [1;2;3]
let x = 1::[2;3]
let x = 1::2::[3]
let x = 1::2::3::[]
let x = h::t (* h: head, t: tail)
```

The type of x is a' list, where a' is the type of the elements in the tail of the list (note: the type of h is a' but the type of t is a' list!)

Pattern Matching

```
let f inputList =
    match inputList with
    |[] -> -1
    |h::t -> h
```

Basically pattern matching allows you to look at the first element of the list, and if that doesn't satisfy you, you can recurse and look at the first element of the tail list.

Linked List
As expected, the linked list data structure is good for sequential access.

Function Keyword Another way to write this is:

```
let f = function
    |[] -> -1
    |h::t -> h
```

This function takes in one argument, matches the argument to a pattern and returns the corresponding value. The main difference is that using `match ... with` can take multiple inputs but `function` only takes in one input.

Patterns

```
a::[] (*matches lists with 1 element*)
a::b::[] (*matches all lists with 2 elements*)
x (*matches anything that has a value*)
_ (*matches everything*)
```

Pattern Matching Warnings
Not all cases considered: inexhaustive pattern-match warning
Duplicated cases: unused match case warning

4 Let Expressions

Let definitions have been used until now

```
let x = 2 in x + x
```

This returns the value $x + x$ so this statement is an expression

```
let x = 2 in x = 1
(fun x -> x + 1) 2
```

These two equations are the same. Basically all these expressions are secretly functions!

Variant

```
type day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
let int_of_day d =
    match d with
    |Sun -> 1
    |Mon -> 2
    |Tue -> 3
    |Wed -> 4
    |Thu -> 5
    |Fri -> 6
    |Sat -> 7
```

Each of the Sun/Mon/Tues are called constructors, which are already a value

Records

Need to define a record type for type inference

```
type contact = {name: string; hp: int}
let nick = {name="Nick"; hp=81234567}
nick.name (*returns "Nick"*)
let get_hp m =
    match m with
    | {name=_; hp = h} -> h
let get_hp m =
    match m with
    |{name; hp} -> hp
let get_hp m =
    match m with
    |{hp} -> hp
let get_hp m = m.hp
```

All the functions are the same, check if you understand how this pattern matching works!

Tuples

```
(1,2,10) : int*int*int
(true, "Hello") : bool*string
([1;2;3], (0.5, 'X')) : int list * (float*char)
```

```
let f t =
    match t with
    | (x, y, z) -> z
let f t =
    let (x, y, z) = t in z
let f t =
    let (_, _, z) = t in z
let f (_, _, z) = z
f (1, 2, 3) = 3
```

Extended Syntax for Let

Previously x was used as a variable, but actually any pattern will work

```
let x = e1 in e2
let rec f x1 ... xn = e1 in e2
```

so replacing all the x's with patterns will work as well

```
let add t =
    let (x, y, z) = t
    in x + y + z
let add (x, y, z) = x + y + z
```

There are built-in functions for accessing first and second element of a tuple (not sure why you need it...)

```
let fst (x, _) = x
let snd (_, y) = y
```

Type synonyms

Can define types so you don't have to type float list list for matrix etc

```
type point = float * float
type vector = float list
type matrix = float list list
```

Of course, you will have to specify the type because the compiler isn't smart enough to read your mind!

```
type point = float * float
let f x : point =
  let y = x +. x in
  (y, y)
 (* if : point is not written, the type of f will be float -> float*float instead of float -> point!*)
```

Back to variants

Recall: variants are enumerated sets of values. In particular, you can have a variant that enumerates values that of different types.

```
type point = float * float
type shape =
    | Point of point
    | Circle of point * float
    | Rect of point * point

let pt1 : point = (1., 2.)
(*pt1 = point(1., 2.) is invalid!! *)
let pt2 : shape = Point(1., 2.)
let pt3 = Circle (pt1, 2.)
let pt4 = Circle ((1., 2.), 3.)
(*note that since Circle's type is point * float, you
   don't need to define the tuple as a point since it is
     already defined!*)
```

The data type of shape is called an **algebraic data type** because it is a **tagged union** (Google nonintersecting union if you're not sure)

This allows you to classify things under other things. This is similar to inheritance with classes in Java. We may combine this with functions to allow a function to take in arguments of various types (patterns), which is similar to overloading in Java.

```
let area = function
    | Point _ -> 0.0
    | Circle (_,r) -> pi *. (r ** 2.0)
    | Rect ((x1,y1),(x2,y2)) ->
    let w = x2 -. x1 in
    let h = y2 -. y1 in
    w *. h
```

Recursive Variants

We may implement the linked list data type with a recursive variant definition.

```
type intlist = Nil | Cons of int * intlist

let emp = Nil
let l3 = Cons (3, Nil) (* 3::[] or [3]*)
let l123 = Cons(1, Cons(2, l3)) (* [1;2;3] *)

let rec sum (l:intlist) =
    match l with
    | Nil -> 0
    | Cons(h,t) -> h + sum t
```

```

let rec length = function
    | Nil -> 0
    | Cons (_,t) -> 1 + length t
    (* length : intlist -> int *)

let empty = function
    | Nil -> true
    | Cons _ -> false
    (* empty: intlist -> bool *)
```

We may also use `'a` to allow our list to accept other types.

```
type 'a mylist = Nil | Cons of 'a * 'a mylist
int mylist
```

Mylist is known as a **type constructor**, because it takes a type as an input and returns a type.

5 Higher-order Programming

Recall that functions are values, so we can pass them as arguments. This is an example of functions being 'first-class citizens' lol.

```
let square x = x * x
let quad x = (square x) * (square x)
let twice f x = f (f x)
let quad2 x = twice square x
```

The slides didn't mention this, but I believe that the lack of distinction between functions and values is why braces around function arguments were not implemented.

Map and Fold

Note: map is not a method to store key - value pairs!

```
let rec map f = function
    |[] -> []
    | h::t -> (f h) :: (map f t)

let add1 = List.map (fun x -> x + 1)
let list1 = [1; 2; 3; 4]
add1 list1 (*returns [2; 3; 4; 5]*)
```

Filter

```
let rec filter f = function
    |[] -> []
    | h::t ->
        if f h
        then h::(filter f t)
        else filter f t

let filter1 = List.filter (fun x -> x > 2)
let list1 = [1; 2; 3; 4]
filter1 list1 (*returns [3; 4]*)
```

These are called iterators.

Combining Elements

```
let rec combine init op = function
    | [] -> init
    | h::t -> op h (combine init op t)

let sum = combine 0 (+)
let concat = combine "" (^)
```

Notice once again how all these functions can be ordered around as values. With braces, the code would be more confusing than without, which is quite interesting! Notice also the use of `()` to convert an infix operator to a prefix operator.

OCaml has a `fold_right / fold_left` operator to do this explicitly.

```
let rec fold_right f t acc =
    match t with
    | [] -> acc
    | h2::t2 -> f h2 (fold_right f t2 acc)

List.fold_right f [a;b;c] init
(*computes f a (f b (f c init))*)

let rec fold_left f acc t =
    match t with
    | [] -> acc
    | h2::t2 -> fold_left f (f acc h2) t2
(*note the difference between fold_left and fold_right's
    implementation - fold_right has to reach the
   rightmost element before evaluating f but fold_left
   evaluates the left-most element straight away*)
```

6 Modular Programming

OCaml uses **functional data structures** which means that data structures never change (again, to avoid having multi-valued things)

```
module MyStack = struct
  type 'a stack =
    | Empty
    | Entry of 'a * 'a stack
  let empty = Empty
  let is_empty s = s = Empty
  let push x s = Entry (x, s)
  let peek = function
    | Empty -> failwith "Empty"
    | Entry(x,_) -> x
  let pop = function
    | Empty -> failwith "Empty"
    | Entry(_,s) -> s
end

let test1 = MyStack.empty;;
let test2 = MyStack.push 1 test1;;
let test3 = MyStack.push 2 test2;;

(*returns*)
val test3 : int MyStack.stack =
  MyStack.Entry (2, MyStack.Entry (1, MyStack.Empty))
```

- Note the `module...struct` keywords; which is equivalent to classes in Java. Specifically, a module creates a new **namespace**.
- In particular, you need to define the type which is needed for type inference.
- Note the last 3 lines: it is not possible to push an element into the original stack, because the stack is not mutable.

Function Signature

It is possible to define the function signature, i.e. what types of variables it can input and output, as well as what functions to be accessed from outside the module.

The 2 benefits are:

- input and output types are correct

- unnecessary methods are hidden

```
module type S1 = sig
  val x:int
  val y:int
end
module M1 : S1 = struct
  let x = 42
end
(*Error: Signature mismatch:
Modules do not match: sig val x : int end is not
   included in S1
The value ‘y’ is required but not provided*)
module type S2 = sig
  val x:int
end
module M2 : S2 = struct
  let x = 42
  let y = 7
end
M2.y
(*Error: Unbound value M2.y*)
```

Abstract Types

There is one more step to abstraction: not revealing the data structure in the module signature.

```
module type ListStackSig = sig
  val empty : 'a list
  val is_empty : 'a list -> bool
  val push : 'a -> 'a list -> 'a list
  val peek : 'a list -> 'a
  val pop : 'a list -> 'a list
end

module ListStack = struct
  let empty = []
  let is_empty s = s = []
  let push x s = x :: s
  let peek = function
    | [] -> failwith "Empty"
    | x::_ -> x
  let pop = function
    | [] -> failwith "Empty"
```

```
      | _::xs -> xs
end
  (*client's code*)
let x = ListStack.empty;;
let y = ListStack.push 1 x;;
let z = (2 :: y);;
```

In this example, after realising that ListStack is implemented with list, the client decides to use the :: function to concatenate lists. However, if you decide to change the type used in ListStack to something else, all of the client's code that uses :: will break. Hence, the type used in implementation shouldn't be revealed.

```
module type Stack = sig
  type 'a t
  val empty : 'a t
  val is_empty : 'a t -> bool
  val push : 'a -> 'a t -> 'a t
  val peek : 'a t -> 'a
  val pop : 'a t -> 'a t
end

module ListStack : Stack = struct
  type 'a t = 'a list (*this line informs the compiler
     that all the ts refer to lists*)
      ...
```

Here, we give an arbitrary type stack so the client would not know what type we have used. By convention t is used to refer to an abstract type.

7 Functors

Interface Inheritance

```
module type Ring = sig
  type t
  val zero : t
  val one : t
  val add : t -> t -> t
  val mult : t -> t -> t
  val neg : t -> t
end
module type Field = sig
  include Ring
```

```
  val div : t -> t -> t
end

module FloatRing = struct
  type t = float
  let zero = 0.
  let one = 1.
  let add = (+.)
  let mult = ( *. )
  let neg = (~-.)
end
module FloatField = struct
  include FloatRing
  let div = (/.)
end
```

So `include` works for both module signatures and modules.

Functors

Functors take structures as inputs and output another structure. So even modules are treated like functions!

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

module IncX (M : X) = struct
  let x = M.x + 1
end

(*in anonymous form as well*)
module IncX = functor (M : X) -> struct
  let x = M.x + 1
end

module A = struct let x = 0 end
module B = IncX(A) (* B.x is 1 *)
module C = IncX(B) (* C.x is 2 *)
```

One use of functors is to test whether modules of a certain signature are working.

```
assert (MyStack.(empty |> push 1 |> peek) = 1)
(*The output of MyStack.empty is piped to MyStack.push 1
    using the pipe operator. Assert throws an exception
   if the expression evaluates to false.*)
```

```

(*instead of*)
assert (MyStack.(
    empty |> push 1 |> peek) = 1) ;;
assert (ListStack.(
    empty |> push 1 |> peek) = 1);;
assert (WhateverStack.(
    empty |> push 1 |> peek) = 1)

(*we could write*)
module StackTester (S:StackSig) = struct
  assert (S.(empty |> push 1 |> peek) = 1)
end
module MyStackTester
  = StackTester(MyStack)
module ListStackTester
  = StackTester(ListStack)

(*note that this works because when instantiating the
   StackTester module, it evaluates each expression
   within it, even if it isn't a let x = ... expression.
   *)
```

include works with functors as well... But to be fair it isn't surprising as a functor is just another module.

```
module type Sig = sig ... end
module Ext (M:Sig) = struct
  include M
  let f =
    ...
end
```

8 Abstraction and Specification

You should document your code properly if you want it to be readable. The following sections have guidelines on how to write specifications to make it easy for someone else to read.

```
(**
 * returns: [hd lst] is the head of [lst].
 * example: hd [1; 2; 3] is 1.
 * requires: [lst] is non-empty.
 * raises: [Failure "hd"] if [lst] is empty.
```

```
 * effects: ...
 *)
val hd : 'a list -> 'a
(*this last line would be in your module signature*)
```

The starry formatting allows html or others to parse the comments. The square brackets will format the words as `words`.

9 Abstraction Functions and Representation Invariants

Not critical to the Cambridge OCaml course. Will add in later if I feel like it.

10 Testing

Not critical to the Cambridge OCaml course. Will add in later if I feel like it.

11 Streams and Laziness