Cornell CS 3110 - Functional Programming

Ivin Lee

September 23, 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.

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