20160107-Thursday-ML

ryansharif

January 8, 2016

Contents

1 January 7th, 2016 - Lecture Notes: ML **1**
1.1 Course Outline Topics 1
1.2 Tension between performance and functionality 2
1.2.1 2 random questions from other professors 2
1.2.2 Issues at Google 3
1.3 Historical motivations for functional programming 3
1.3.1 Clarity 3
1.3.2 Performance 3
1.4 Practice: OCaml 4
1.4.1 basic properties 4
1.5 using OCaml 4
1.5.1 tuples 4
1.5.2 lists 4
1.5.3 simple functions 5
1.5.4 shorthand vs long form 6
1.5.5 patterns 6
1.5.6 functions and matching 7

1 January 7th, 2016 - Lecture Notes: ML

1.1 Course Outline Topics

There are several general topics that we will cover in this class:

- theory (mostly syntax)
- syntax & semantics

- functions
- names
- types
- control
- objects
- exceptions
- concurrency
- scripting

We will use five programming languages:

- OCaml (syntax and functions)
- Prolog (control)
- Java (concurrency and exceptions)
- Scheme (control and functions)
- Python (scripting)
- a mystery language

1.2 Tension between performance and functionality

1.2.1 2 random questions from other professors

1. Why are there so many programming languages?
   (a) Domains make languages diverge
2. My AI software (LISP) is too slow.
   (a) Rewrote the program in C/C++ but the result was a nightmare (memory management issues)

These are metaprogramming issues/questions for Computer Science.

1.2.2 Issues at Google

Google had a problem:

- big backend queries

In order to solve this problem, Google took the *functional* model. This solution is implemented by new solutions like *Hadoop*.

1.3 Historical motivations for functional programming

There were two general motivations for functional programming:

1.3.1 Clarity

Mathmeticians have studied functions for centuries. So why not build on it!

We know that a simple programming variable assignment doesn't work the same in mathematics:

$ i = i + 1 $ is false

Variable reassignment is not something we do generally in math.

referential transperancy: we want the same answer anytime we ask a programing what the value of a variable is.

Essentially, we want to avoid mutations of objects. We don't want a notion of time in our functions.

Furthermore, we want to capture the level of abstraction common in mathematics, e.g., $y = \int_0^2 f(x)dx$ We capture this notion using higher order functions and we use this pattern regularly.

1.3.2 Performance

There was a desire to escape from the Von Neumann style, i.e., a set of instructions for our computer to follow.

Similarly, there was a desire to escape from the Von Neumann bottleneck: only one word can be communicated at a time. We want to encourage parallelism to escape the Von Neumann style. A functional style allows for easier parallelization.

Suppose we have C code:

```
int a = f(x);
int b = g(x);

return a + b;
```

Since, f or g might access variables that the compiler does not know about, the compiler cannot automatically perform a parallel optimization.

1.4 Practice: OCaml

1.4.1 basic properties

OCaml has several basic properties:

1. functional
2. compile type checking (like C, C++, Java, ...)[safety]
3. you need not specify the types

(like Python, Scheme,...)[convenience]

1. no need to worry about storage management (has garbage collection)
2. good support for higher order functions (encourage by the language)

1.5 using OCaml

When using an *if-then-else* statement, the *then* and *else* must have the same type.

1.5.1 tuples

```
(1,2)
```

This is an example of a tuple, where what we get back is an approximation of the cross-product symbol:

```
int * int
```

1.5.2 lists

```
[1,2]
```

List must be homogenous, whereas tuples can be heterogeneous.

1. empty lists (generic types) If we input the following into OCaml, we get back a generic type:

   ```
   []

   - : 'a list = []
   ```

1.5.3 simple functions

```
let f x = x + 1;;
f: int -> int = <fun>
```

We can call this function, if we want:

```
f 27
- : int = 29
```

When Eggert was first learning how to use OCaml, he wanted to implement a function that he always used in LISP:

```
let cons(x,y) = x :: y;;
val cons: 'a * 'a list -> 'a list = <fun>

cons(3,[4;])
- : int list = [3;4;2]
```

The problem with his original solution was that by using a tuple, he was telling OCaml to call the function using a tuple. A better way to do this would be to define the function in a slightly different fashion:

```
let ccons x y = x :: y;;

val ccons: 'a -> 'a list = 'a list = <fun>
ccons 12 [3;9]
- : [12;3;9]
```

The function is called as if you have parantheses:

```
(ccons 12) [3;9]
```

This type of function calls are called currying. Currying helps write clearer programs.

1. More currying We can take our ccons function and can make a derivative that always has 12 as its first argument:

   ```
   let ccons12 = ccons 12;;

   val ccons12 : 'a -> 'a list = <fun>

   ccons12 [1;9]
   - : int list = [12;1;9]
   ```

1.5.4 shorthand vs long form

OCaml generally allows us to use shorthand form. But sometimes we need to give a long form:

```
fun x -> x + 2;;
- : int -> int = <fun>

let inc = fun x -> x + 1;;
inc: int -> int = <fun>

(* shorthand version: *)
let inc x = x + 1;;

fun x y -> x + y ;;
- : int -> int -> = <fun>
```

1.5.5 patterns

Based on the success or failure of a match, we can do work:

patter	matches
0	0
	[]
x (var)	anything
_	acts like anything (can't look at the value later)
p1,p2,p3	tuples, whose items, match p1,p2,p3 respectivelyp1;p2;p3>p1;p2;p3
	matches items of the list
p1::p2	matches non-empty lists, whose head matches p1 and the rest matches p2

```
match v with
| [] -> false
| _ -> true

(* original *)
let cons(x,y) = x::y;;

(* long form *)
let cons = fun (x,y) -> x :: y;;
```

```
(* longer form *)
let cons = fun z -> match z with (x,y) -> x :: y ;;
```

1.5.6 functions and matching

1. compute the length of a list:

   ```
   let rec len = fun x ->
     match x with
       | [] -> 0
       | h::t -> 1 + len t
   ```

2. reverse of a list

   ```
   (* 1978 ML program *)
   let rec rev = function
   | [] -> []
   | h::t -> (rev t) @ [h];;
   ```

3. first item of a list

   ```
   let gwdcar d = function
   | h::_ -> h
   | _ -> d
   ```

4. miniumum item in a list

   ```
   let minlist = function
   | h::t -> let v = minlist t
     in if h < v then h else v
   | _ -> biggest_int

   let rec gminlist lt max = function
   | h::t -> let v = gminlist t
     in if lt h v then h else v
   | _ -> max
   ```