Introduction to Functional Programming (SCALA)

John Woodward
Object Oriented Programming

- In OO the 3 principles are
- Inheritance (extend objects)
- Polymorphism (process different objects)
- Encapsulation (hiding data e.g. Y2K bug).
Not another (paradigm) language

• Imperative programming languages have a common design goal: to make efficient use of computers which use the von Neumann Architecture. (assembly languages, C)

• Efficiency is not always a priority. depends on application - properties such as low maintenance cost, easy debugging, formally provable correctness. e.g. in safety critical systems (medical, transport, nuclear).

• Big systems are often build with different languages.
Functional Programming

• **Immutable**, Stateless – (good news?)
• Function are first class objects (**higher order functions**).
• Programs in functional languages are generally shorter, easier to understand, design, debug, and maintain, than imperative counterpart.
• Modern functional languages are **strongly typed**. guarantees no type errors at runtime (trapped by the compiler). **Type Inference**
• Typically **recursion** is used in functional programming languages, iteration is used in imperative languages.
State & Referential Transparency

• Java object store state information (instance variables).
• E.g. in a constructor Person("John", 45, "UK", "Lecturer").
• This stores Name, Age, Nationality, Job
• Print Function_A(x)
• Print Function_B(x)
• Print Function_A(x)
• (side effects).
Data Type and Type Signatures

• In maths a function f maps between two sets A and B type signature A->B (domain and co-domain, input and output) type signatures.
• (Boolean, real, float, natural, integers, ) B,R, F, N, Z, (a pair of Booleans) BXB or B^2
• Type inference can be used to infer facts about types.
• e.g. composition - theorems for free
Can Your Programming Language Do This?

Motivating Example


```javascript
// A trivial example:
alert("I'd like some Spaghetti!");
alert("I'd like some Chocolate Moose!");
```

The repeated code looks wrong, of course, so you create a function:

```javascript
function SwedishChef( food )
{
    alert("I'd like some " + food + "!");
}

SwedishChef("Spaghetti");
SwedishChef("Chocolate Moose");
```
What is repeated here?

What is the abstraction?

```javascript
alert("get the lobster");
PutInPot("lobster");
PutInPot("water");

alert("get the chicken");
BoomBoom("chicken");
BoomBoom("coconut");
```
We repeat the 2nd action TWICE

function Cook( i1, i2, f )
{
    alert("get the " + i1);
    f(i1);
    f(i2);
}

Cook( "lobster", "water", PutInPot );
Cook( "chicken", "coconut", BoomBoom );

alert("get the lobster");
PutInPot("lobster");
PutInPot("water");

alert("get the chicken");
BoomBoom("chicken");
BoomBoom("coconut");
list of functional programming languages

- Pure: Charity Clean Curry Haskell Hope Miranda Idris
- Impure: C# Erlang F#, Java (since version 8) Lisp Clojure Scheme Mathematica ML OCaml R Scala
- There are more impure languages!
Higher Order Functions (HOF)

• In most programming languages we pass around integers, Booleans, strings, as argument to function and return types
• For example Boolean isPrime(Integer n)
• Takes an integer and returns true/false depending on if it is prime or not.
• With functional languages we can pass around functions. Type signatures
Everyday Examples of higher order functions

• At college you had to differentiate $y=mx+c$
• Also integration (returns a function).
• In physics...law of refraction/reflection.
• **Fermat's principle** states that light takes a path that (locally) minimizes the optical length between its endpoints.
• Mechanics...hanging chain (a ball rolls down a slope – a chain takes on a shape).
• Droplets minimize surface area, aerodynamics.
Naming a String (lambda)

- String name = “John”
- Print(name);//will print “John”
- Or we can just print the name directly
- Print(“John”)
- If we only use “John” once – we could use a string literal (a one-off use).
- If we use “John” multiple times – define a variable – name – and use that.
Anonymous Functions/lambda (PYTHON)

- def f (x): return x**2
- print f(8)
- Or we can do
- g = lambda x: x**2
- print g(8)
- Or just do it directly
- print (lambda x: x+2) (4)
- y = (lambda x: x*2) (4)
- print y
Anonymous Functions (SCALA)

- `println(((a: Int) => a + 1)(4))`
- An increment function with the argument 4
- `println(((a: Int, b: Int) => a + b)(4, 6))`
- The addition function with the arguments 4 and 6.
Lambda – filter, map, reduce (PYTHON)

- $\text{foo} = [2, 18, 9, 22, 17, 24, 8, 12, 27]$
- $\text{print filter(lambda x: x % 3 == 0, foo)}$
  - $\#[18, 9, 24, 12, 27]$
- $\text{print map(lambda x: x * 2 + 10, foo)}$
  - $\#[14, 46, 28, 54, 44, 58, 26, 34, 64]$
- $\text{print reduce(lambda x, y: x + y, foo)}$
  - $\#139$
- (more detail in a few slides)
Lambda – filter, map, reduce (SCALA)

• foo = [2, 18, 9, 22, 17, 24, 8, 12, 27]
• println(foo.filter((x: Int) => x % 3 == 0))
  ///[18, 9, 24, 12, 27]
• println( foo.map((x: Int) => x*2+10))
  ///[14, 46, 28, 54, 44, 58, 26, 34, 64]
• println( foo.reduce((a:Int, b:Int)=>a+b))
  #139
• (more detail in a few slides)
Simple example: Higher order Function (Python)

• Takes a function \( f \) and value \( x \) and evaluates \( f(x) \), returning the result.
• ```
    def apply(f, x):
    return f(x)
```
Simple example: Higher order Function (Scala)

• Takes a function \( f \) and value \( x \) and evaluates \( f(x) \), returning the result.

```scala
• def apply(f:Int=>Int, x:Int):Int = {
•   f(x)
• }
```
f(x) and f

• In maths f and f(x) are different.
• f is a function!
• f(x) is the value of f at value x
• Never say “the function f(x)”
• Say – The function f that takes a variable x (i.e. f takes var x)
• Or – the function f at the points x (i.e. f given x has the value ??)
Example: A linear function (PYTHON)

• #return a function
• #note return result, not result(x)
• def linear(a, b):
  •   def result(x):
  •     return a*x + b
  •   return result
• myLinearFunction = linear(4,3)
• #make a function 4*x+3
• print myLinearFunction(8)
• print apply(myLinearFunction, 8)
Example: A linear function (SCALA)

• //return a function
• def linear(a: Double, b: Double): Double => Double = {
    • x:Double => a*x + b
• }myLinearFunction = linear(4,3)
• //make a function ?*?++?
• println(linear(2,1)(9))
Summation (PYTHON)

• In maths \( \sum_{i=1}^{10} f(x) = f(1)+f(2)+...+f(10) \)
• \( \Sigma \) takes 3 arguments, and upper and lower bound and a function to sum over.
• e.g. \( \sum_{i=1}^{10} 2x = 2.1+2.2+3.2 = 2+4+6 = 12 \)
• `def sum(f, a, b):
  total = 0
  for i in range(a, b+1):
    total += f(i)
  return total`
Summation (SCALA)

- In maths $\sum_{i=1}^{10} f(x) = f(1) + f(2) + \ldots + f(10)$
- $\sum$ takes 3 arguments, and upper and lower bound and a function to sum over.
- e.g. $\sum_{i=1}^{10} 2x = 2 \cdot 1 + 2 \cdot 2 + 3 \cdot 2 = 2 + 4 + 6 = 12$
- `def sum(f: Int => Int, a: Int, b: Int): Int = {
  \text{if} \ (a > b) \ 0
  \text{else} \ f(a) + \text{sum}(f, a + 1, b)
  \} \`
Product Symbol in Maths

• In mathematics we often use the product symbol. $\prod_{i=1}^{3} f(x)$ means $f(1)\times f(2)\times f(3)$.

• You can do this for the labs 😊
Composition as HOF (PYTHON)

• `def compositionValue(f, g, x):`
• `    return f(g(x))`
• This take two functions f and g.
• And a value x
• And calculates the values f(g(x))
• Picture this 😊
• What restrictions are there?
Composition as HOF (SCALA)

• def compositionValue(f:Int=>Int, g:Int=>Int, x:Int):Int = {
  f(g(x))
}
• println( compositionValue(dec, dec, 10))
Some simple functions

- def timesOne(x):
  - return 1*x
- def timesTwo(x):
  - return 2*x
- def timesThree(x):
  - return 3*x
- def apply(f, x):
  - return f(x)
Returning a Function (PYTHON)

```
• def compositionFunction(f, g):
•     def result(x):
•         return f(g(x))
•     return result
• myFunction = compositionFunction(timesThree, timesTwo)
• print myFunction(4)
• print apply(compositionFunction(timesThree, timesTwo), 4)
• print (lambda x, y: compositionFunction(x, y))(timesThree, timesTwo)(4)
```
Returning a Function (SCALA)

• def compositionFunction(f: Int => Int, g: Int => Int): Int => Int = {
  • x: Int => f(g(x))
  • }

Timing a function 1 (PYTHON)

• Suppose we want to time a function.
• We could do the following

```python
start = time.time()
result = g(*args)
elapsed = (time.time() - start)
The we time the function and execute it.
We could write a higher order function (HOF) to achieve this.
```
import time

def timer(g,*args):
    start = time.time()
    result = g(*args)
    elapsed = (time.time() - start)
    print "elapsed "+str(elapsed)
    print "result = " + str(result)

#the function g is called with a set of args *args
#call the “add” with 1,2 timer(add,1,2)
Maximum of two numbers (PYTHON)

• you can find the max of two numbers
• def max(x,y):
  • if (x>y):
    • return x
  • else:
    • return y
• what is the data-type signature
Maximum of two function (PYTHON)

• We can find the max of two functions (PICTURE)
• def maximumValue(f1, f2, x):
  • return max(f1(x), f2(x))
• #The inputs are ???
• #The output is ???
• #what is the data-type signature
• #Can we return a function?
• #What is the signature of the returned function
Returning the Max Function (PYTHON)

- `def maxFunction(f1, f2):
-   def maxFun(x):
-     return max(f1(x), f2(x))
-   return maxFun
- biggerFun = maxFunction(fun1, fun2)
- print biggerFun(2)`
Max Function (SCALA)

- `def maxFunction(f: Int => Int, g: Int => Int): Int => Int = {
  x: Int =>
  {
    if (f(x) > g(x))
      f(x)
    else
      g(x)
  }
`
Your turn

• Write a function \( f \) which returns a function which is the minimum of functions \( f_1, f_2 \)
• i.e. \( f(x) = \min \{ f_1(x) \text{ and } f_2(x) \} \)
• Write a function which returns a function which is the average of two functions
• i.e. \( f(x) = \frac{f_1(x) + f_2(x)}{2} \)
Partial Application

• Motivating example - diagram
• Addition (+) takes two arguments (arg1 + arg2)
• What if we only supply one argument?
• We cannot compute e.g. (1+) 1+WHAT
• But (1+) is a function of one argument
• (1+ could be called what???)
Inc as partial add (PYTHON)

- `#add (2 args), inc(x)=add(1,x)`
- `#inc is a special case of add`
- `from functools import partial`
- `def add(a,b):`
  - `return a+b`
- `inc = partial(add, 1)`
- `print inc(4)`
Inc as partial add (SCALA)

- `def add(a:Int, b:Int):Int = {
  a+b
}
- `val inc = add(1, _: Int)
- `def incFun: Int=>Int = {
  x: Int => add(x,1)
}
- `print(incFun(4))`
Inc defined with add (PYTHON)

• #add takes 2 arguments
• def add(x,y):
  •     return x+y
• #we can define a new function by hardcoding one variable
• def inc(x):
  •     return add(1,x)
• print inc(88)
double as partial mul (PYTHON)

• #mul(2 args), double(x)=mul(2,x)
• #double is a special case of mul
• from functools import partial
• def mul(a,b):
  • return a*b
• double = partial(mul, 2)
• print double(10)
Bit of History

• LISp (1950s John McCarthy) LISP Processing is the first/ancestor of programming languages. Higher order functions, functions that take functions as input and/or return functions as output. (type signatures) (Clojure has been described as LISP on the JVM)

• Twitter core written in scala - why
referential transparency

• Haskell is a pure functional language referential transparency - the evaluation of an expression does not depend on context.
• The value of an expression can be evaluated in any order (all sequences that terminate return the same value) (1+2)+(3+4) we could reduce this in any order.
• In the 2nd world war, Richard Feynman was in charge of making calculation for the atomic bomb project.
• These calculations were done by humans working in parallel. e.g. calculate exponential
Evaluation

• **call by name**: when a function is called arguments are evaluated as needed. (*lazy evaluation*)

• **call by value**: the arguments are evaluated before a function is called. (*strict evaluation*)

• Call by name can be inefficient - Haskell shares evaluations sub-expressions when used many times in an function

• call by value may not terminate

• (a little like short-circuit in java)
Map, Filter, Reduce

• Let us now look at 3 higher order functions which are used a lot in functional programming.
• First we will look LISTS
• Then we will look at Map, Filter and Reduce.
Lists (PYTHON)

- Functional programming uses LISTS as its primary data structure. E.g. [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

  ```python
  listNumbers = [1, 2, 3] # [1, 2, 3]
  listNumbers.append(4) # [1, 2, 3, 4]
  listNumbers.insert(2, 55) # [1, 2, 55, 3, 4]
  listNumbers.remove(55) # [1, 2, 3, 4]
  listNumbers.index(4) # 3
  listNumbers.count(2) # 1
  ```
Map (PYTHON)

- Map takes a function and a list and applies the function to each elements in the list
- `def cube(x): return x*x*x`
- `print map(cube, range(1, 11))`
- `print map(lambda x :x*x*x, range(1, 11))`
- `print map(lambda x :x*x*x, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10])`
- # [1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
- #what is the type signature
Filter (PYTHON)

• Filter takes a function (what type) and a list, and returns items which pass the test
  • def f1(x):  return x % 2 != 0
  • def f2(x):  return x % 3 != 0
  • def f3(x):  return x % 2 != 0 and x % 3 != 0

• print filter(f1, range(2, 25))
• print filter(f2, range(2, 25))
• print filter(f3, range(2, 25))
Filter 2 (PYTHON)

• def f1(x): return x % 2 != 0
• def f2(x): return x % 3 != 0
• def f3(x): return x % 2 != 0 and x % 3 != 0

• print filter(f1, range(2, 25))
• # [3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23]
• print filter(f2, range(2, 25))
• # [2, 4, 5, 7, 8, 10, 11, 13, 14, 16, 17, 19, 20, 22, 23]
• print filter(f3, range(2, 25))
• # [5, 7, 11, 13, 17, 19, 23]
Reduce (PYTHON) - exercise

• \( \text{reduce}( \lambda x, y: x + y), [1, 2, 3, 4]) \)

• \( \text{reduce}( \lambda x, y: x - y), [1, 2, 3, 4]) \)

• \( \text{reduce}( \lambda x, y: x * y), [1, 2, 3, 4]) \)

• \( \text{reduce}( \lambda x, y: x / y), [1, 2, 3, 4]) \)
Reduce 2 (PYTHON)

- \texttt{reduce( (lambda x, y: x + y), [1, 2, 3, 4] )}  
  \#10

- \texttt{reduce( (lambda x, y: x - y), [1, 2, 3, 4] )}  
  \#-8

- \texttt{reduce( (lambda x, y: x * y), [1, 2, 3, 4] )}  
  \#24

- \texttt{reduce( (lambda x, y: x / y), [1, 2, 3, 4] )}  
  10

- 8

- 24

- 0

- Is the last one correct?
The last one again.

- `print reduce((lambda x, y: x / y), [1.0, 2.0, 3.0, 4.0])`
- 0.0416666666667
- # what is the type signature?
Map - scala

- var nums: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)
- println(nums)
- println(nums.map(x => x * x * x * x))
- Will print
- List(1, 2, 3, 4, 5, 6, 7, 8, 9)
- List(1, 8, 27, 64, 125, 216, 343, 512, 729)
Filter – (Exercises) Scala

- `var nums: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)`
- `println(nums.filter(x => x % 2 == 0))`
- `println(nums.filter(x => x % 3 == 1))`
- `println(nums.filter(x => (x % 2 == 0) && (x % 3 == 0)))`
Filter – (answers) Scala

```scala
println(nums.filter(x => x % 2 == 0)) // List(2, 4, 6, 8)
println(nums.filter(x => x % 3 == 1)) // List(1, 4, 7)
println(nums.filter(x => (x % 2 == 0) && (x % 3 == 0)))
List(6)
```
Reduce – (Exercise) Scala

• `nums = List(1, 2, 3, 4)`
• `println(nums.reduce((x: Int, y: Int) => x + y))`
• `println(nums.reduce((x: Int, y: Int) => x - y))`
• `println(nums.reduce((x: Int, y: Int) => x * y))`
• `println(nums.reduce((x: Int, y: Int) => x / y))`
Reduce – (Answers) Scala

• println(nums.reduce((x: Int, y: Int) => x + y))//10
• println(nums.reduce((x: Int, y: Int) => x - y))//-8
• println(nums.reduce((x: Int, y: Int) => x * y))//24
• println(nums.reduce((x: Int, y: Int) => x / y))//0
How does reduce work?

- //what does the following print
- `def add(x: Int, y: Int): Int = {
  println(x + " " + y)
  x + y
}`
- `nums = List(1, 2, 3, 4)`
- `println(nums.reduce(add))`
• 1 2
• 3 3
• 6 4
• 10
• //try with mul (*), sub(-) and div(/)
• def mut(x: Int, y: Int): Int = {
  println(x + " " + y)
  x * y
}
Tutorials

• We can combine these operations.
• You will have examples in the tutorials.
Websites

- [http://stackoverflow.com/](http://stackoverflow.com/) (and links at bottom – a very good forum)
- And of course ... Google.