Functional Programming in Scala
Summer 2008

Functional programming

- One of the most interesting aims of Scala is the attempt to unify object-oriented and functional programming
- What is functional programming then? A many different answers. Martin Odersky divides these into two categories[1]:
  1. *Exclusive* functional programming – programming without side-effects (e.g. Haskell)
  2. *Inclusive* functional programming – programming style, which composes functions in interesting ways
- *Scala follows the latter approach*

Contents

- Contents of this talk
  1. What's a function
  2. Local functions
  3. First-class functions
  4. Partially applied functions
  5. Closures
  6. Currying
- Based on chapters 8 and 9 of the book "Programming in Scala"

What's a function?

- Technically, a function in Scala is an object with a method with signature *apply*(…)
- E.g:
  - apply(Int) is a function, which returns an Int
  - apply(String, Int) is a function, which takes one String parameter and returns an int
  - apply(String, Int, Int) is a function, which takes one String parameter, one Int parameter and returns an Int
  - and so on.

Interpreter exercise

```
scala> class func {
     |   def apply(): Int = 6;
     | }
defined class func

scala> val f = new func();
f: func = func@c192f

scala> f();
res11: Int = 6
```

Function$_n$ traits

- For programmer convenience, the standard library defines *n* traits, which are named as

  Function0

  Function1

  ..

  Function15

  ..

  Function22
- Why up to Function22?
- Twenty-one parameters ought to be enough for anybody..

A function object, defined with traits

```
scala> class f extends Function1[Int, Int] {
                    def apply(arg: Int) = 6;
          }
defined class f
scala> val f = new f();
f: f = <function>

scala> f(9);
res19: Int = 6
```

Theme 2/6: Local functions

- With method placing, Java took the road of flat, one namespace, similar to C
- The primary construct for reducing namespace pollution is the have private methods in classes
- In Scala, there's an alternative: local functions
- Lexical scoping restricts the visibility of a local function

Long lines example – the Java style

```
import scala.io.Source object LongLines {
  def processFile(filename: String, width: Int) {
    val source = Source.fromFile(filename)
    for (line <- source.getLines)
      processLine(filename, width, line)
  }
  private def processLine(filename: String, width: Int, line:
     String) {
    if (line.length > width)
      println(filename +": "+ line.trim)
  }
}
```

Translation to local functions

```
object LongLines {
  def processFile(filename: String, width: Int) {
    def processLine(filename: String, width: Int, line: String){
      if (line.length > width)
        print(filename +": "+ line)
    }
    val source = Source.fromFile(filename)
    for (line <- source.getLines) {
      processLine(filename, width, line)
    }
  }
}
```

Local functions can access the names in the enclosing function

```
object LongLines {
  def processFile(filename: String, width: Int) {
    def processLine(line: String) {
      if (line.length > width)
        print(filename +": "+ line)
    }

    val source = Source.fromFile(filename)
    for (line <- source.getLines)
      processLine(line)
  }
}
```

Theme 3/6: first-class functions

- **Functions are ’first-class citizens’: in addition to defining and calling functions**
  - **Functions can be passed around (easily)**
  - **Writing functions as literals is possible**
- **Previously, we defined functions as objects with a method of certain signature**
- **A function literal makes this much easier:**

```
scala> (x: Int) => x + 1
res0: (Int) => Int = <function>
```

- **More than one expression in the function is done by curly braces (not so surprisingly!)**

First-class functions and library support

- **A lot of Scala library has been written to with the support for first-class functions in mind**
- **E.g the method *foreach* in the trait Iterable: it takes a function as an argument and invokes the function to each of the elements**

```
scala> val someNumbers = List(-11, -10, -5, 0, 5, 10)
    someNumbers: List[Int] = List(-11, -10, -5, 0, 5, 10)
scala> someNumbers.foreach((x: Int) => print(x))
-11 -10 -5 0 5 10
```

Interesting methods taking function parameters

- *foreach* – as discussed on the previous slide
- *filter* – takes a predicate function; drops falses
- *partition* – takes a predicate function; returns two lists
- map – apply the function to all elements and return a list containing the values of each invocation
- foldRight, foldLeft – higher order magic ☺

- A lot of expressiveness of functional programming comes from the ability of defining 'interesting' functions, and applying the functions to a collection of values

Filtering example - repeated

```
scala> someNumbers.filter((x: Int) => x > 0)
res6: List[Int] = List(5, 10)
scala> someNumbers.filter((x) => x > 0)
res7: List[Int] = List(5, 10)
scala> someNumbers.filter(x => x > 0)
res8: List[Int] = List(5, 10)
scala> someNumbers.filter(_ > 0)
res9: List[Int] = List(5, 10)
```

The placeholder parameter _

- **In function definitions, it is possible to use the underscore as a placeholders for one or more characters**
  - **As long as one parameter is used only once in the function definition**
- **Multiple underscores mean multiple parameters, not reuse of a single parameter repeatedly.**
  - **The first _ refers to the first parameter**
  - **The second _ refers to the second parameter**
- **So,** (_ op _) **is equivalent to** (x, y => x op y)

The type inferer is not always smart enough

- **The type inferer is not always able to know the types of the parameters.**
- **Usually they're blaming the Java's erasure**
- **Fix:**

```
(_: Int) + (_: Int)
(x: Int, y: Int => x + y)
```

- **Beautiful? Or perl?**

Theme 4/6: Partially applied functions

- **When using the underscore as the placeholder for a parameter, we're actually using *partially applied functions***
- **Usually, when a function is invoked, all of its arguments are given**

```
scala> def sum(a: Int, b: Int, c: Int) = a + b + c
sum: (Int,Int,Int)Int
scala> sum(1, 2, 3)
res12: Int = 6
```

- **A partially applied function is expression, where not all of the arguments are supplied**

Defining a partial sum of 1, x and 3

- **Let's define a function *b*, which calculates the sum**

```
scala> val b = sum(1, _: Int, 3)
b: (Int) => Int = <function>
```

- **Now, we can call the function with just one argument:**

```
scala> b(6)
res36: Int = 10
```

When to leave out the underscore

- **When used in a context, where a partially applied function is expected, it is possible to leave out the underscore**

```
scala> someNumbers.foreach(println _)
```

- **Can be also written as**

```
scala> someNumbers.foreach(println)
```

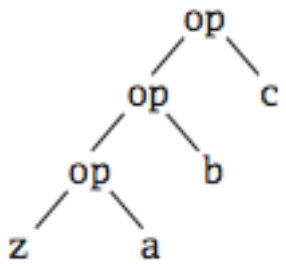
- **When the context doesn't mandate the partially applied function, leaving out the underscore is a compilation error (which is fixed by adding the _)**

```
scala> sum
<console>:5: error: missing arguments for method sum...
```

Time for foldLeft

- **A fold left operation** (startValue /: list) (binOp)
- **Result of the operation is successive application of *binOp*, prefixed with *startValue***
- ***So,*** (z /: List(a, b, c)) (op) **equals** op(op(op(z, a), b), c)

- **Or graphically:**



Theme 5/6: Closures

- **Java7 – the Java version with Closures. Except that the latest news say that they're dropping the closures.**
- **Scala – has closures, available today**
- **What's a closure? A function, in which the free variables are bound.**
- **E.g.**

```
scala> (x: Int) => x + more
<console>:5: error: not found: value more (x: Int) => x +
    more
```

A closure binds its free variables

```
scala> var more = 1
more: Int = 1
scala> val addMore = (x: Int) => x + more
addMore: (Int) => Int = <function>
scala> addMore(10)
res0: Int = 11
```

What happens, if the referred variables have changed?

scala> more = 9999 more: Int = 9999

scala> addMore(10) res21: Int = 10009

- **Scala's closures capture the variables themselves**
- **This is different than in Java's inner classes, which do not allow accessing modifiable variables in the surrounding scopes..**
  - **Making it indistinguishable from capturing the values of the variables**
- **If there are multiple variables with the same name, which one is beign accessed in closures?**
  - **The one that was active at the time of closure creation**

Some closures examples

- **Calculating the sum of integers in a list**

```
scala> var sum = 0
sum: Int = 0
scala> someNumbers.foreach(sum += _)
```

- **A new increaser closure is constructed at each invocation:**

```
scala> def makeIncreaser(more: Int) = (x: Int) => x + more
makeIncreaser: (Int)(Int) => Int
scala> val inc1 = makeIncreaser(1)
inc1: (Int) => Int = <function>
```

Theme 6/6: Currying

- **Currying is a functional programming technique, in which calculation is defined by a chain of function applications**
- **E.g. The plain old way to calculate sum of two integers:**

```
scala> def plainOldSum(x: Int, y: Int) = x + y
plainOldSum: (Int,Int)Int
scala> plainOldSum(1, 2)
res4: Int = 3
```

Calculating the sum via currying

- **In currying, the calculation is contains two function invocations**
  1. **Invoking a function with the first argument, returning a partially applied function back**
  2. **Invoking the partially applied function with the second argument**

```
scala> def curriedSum(x: Int)(y: Int) = x + y
curriedSum: (Int)(Int)Int
scala> curriedSum(1)(2)
res5: Int = 3
```

Currying example – step-by-step

- **Defining the first function**

```
scala> def first(x: Int) = (y: Int) => x + y
first: (Int)(Int) => Int
```

- **Applying 1 to the first function yields the second function**

```
scala> val second = first(1)
second: (Int) => Int = <function>
```

- **Applying 2 to the second function yields the end result**

```
scala> second(2)
res6: Int = 3
```

Wrapping it up – a final example

- Example problem: to calculate the distance of *n* places, which are contained in a list
- Assume that we have the function

  distance($place_1$, $place_2$)

- Lists are defined as List(1, 5, 9)

Distance calculation in Java-style

```
var sum = 0;
var currentPlace = list.head;
for(arg <- list.tail) {
    val distance = distance(currentPlace, arg);
    sum = sum + distance;
}

return sum;
```

Distance calculation in a more functional style

- An exercise left to the reader ☺
  - Hints: you need partial application, foldLeft /: and List.map2

References

1. M. Odersky – In defence of pattern matching
   http://www.artima.com/weblogs/viewpost.jsp?thread=166742