Unit 4e: Higher Order Functions

1 Introduction

You've seen that functions in Scala can be assigned to variables, just like values can (either explicitly or as a result of partial application of a function). Functions can be used like values in other ways, for example:

- passed as parameters to functions
- returned from functions

A function that takes other functions as parameters, or whose result is a function, is known as a higher-order function. These can be extremely useful and are now supported and widely used in many languages, including mainstream ones (for example, they are used to implement callback functions that are very important in JavaScript).

2 Functions as function parameters

Here are three simple function definitions (two named, the other anonymous and assigned to a variable), each takes a single **Int** parameter and returns an **Int** (function type is *Int => Int*).

```
def square(n:Int):Int = {n*n}

def cube(n:Int):Int = {n*n*n}

val double = (x:Int) => x * 2
```

The following higher-order function takes a function of type *Int => Int* as a parameter, and applies that to the other parameter, which is an **Int**.

```
def calculate(f: Int => Int, x: Int) = f(x)
```

We can call *calculate* with any of these functions as a parameter, and it will apply that function to the other parameter:

```
calculate(square, 5)
calculate(cube, 5)
calculate(double, 10)
```

Note that *calculate* doesn't need to know anything about the function that it applies other than its parameter and return types – it will quite happily apply any function of type *Int => Int*.

3 Functions as return types

This function adds a prefix to a **String** and returns the result as a **String**:

```
def saySomething(prefix: String) = {
  prefix + " " + s
}
```

The return type is not explicitly stated; the compiler works it out from type of expression returned. The function definition (after =) is a block that is evaluated when the function is called.

This higher-order version returns a function that will add a prefix to a **String**.

```
def saySomething(prefix: String) = (s: String) => {
  prefix + " " + s
}
```

The function definition (after =) is a lambda expression. The code block in {} is not evaluated immediately when this function *saySomething* is called. Again the return type is not explicitly stated; the compiler works it out from the lambda expression.

Let's call the higher-order *saySomething* function:

```
scala> val sayHello = saySomething("Hello")
sayHello: String => String = <function1>
```

Nothing has been "said" yet; but calling *saySomething* with the parameter "Hello" has given a function, referenced by the variable *sayHello*, that will add the specific prefix "Hello " to its parameter. Let's call that:

```
scala> sayHello("World!")
res0: String = Hello World!
```

Now the code block has been evaluated and a **String** returned with the result. So *saySomething* is a "function factory" that can be called to create a function that will add a specific prefix to a **String** when that function is called. What would *saySomething("Hey")* create?

4 Practical application - filtering

We have stated that HO functions are extremely useful; here's a practical example that shows how we can use a HO function to filter a collection of integers.

We want to start with a collection and get back another collection that only contains the ones we are interested in. In this case this will be the even numbers only.

The imperative solution would involve the following steps:

- Create empty collection for result
- Iterate through initial collection
- For each item in initial collection, check whether it is even
- If so, add to result collection
- When iteration is complete, return result collection

Functional solution notes that there are actually just two transforms here:

- Transform a value to a Boolean that is true if the value is even, false otherwise
- Transform the collection to another collection by applying the first transform and including only the values which return true

So, we can solve the problem by defining functions to do each of these transforms and composing them.

Remember, when thinking functionally you don't think about performing the required action by executing a series of steps. Instead, you think about expressing the required action as a description of the result you want.

The first transform becomes a predicate i.e. a function that returns a Boolean:

```
def mypredicate(x: Int): Boolean = {
  if(x%2 == 0) true
  else false
}
```

The second transform becomes a filter i.e. a function that selects only the elements of a collection for which the predicate is true. The *predicate* function is a parameter for the *filter* function.

```
def filter(l:List[Int], f:Int => Boolean): List[Int] = {
  for(x <- l if f(x)) yield x
}
```

The parameter **`f:Int => Boolean`** is a function of predicate type. The predicate function is *applied* in the filter function. Note that the filter function doesn't know anything about this specific predicate function; we could apply any predicate.