ECE251: Programming Languages & Translators Fall 2010

Lecture 26 — November 18, 2010

Patrick Lam *version 1*

More Higher-Order Functions

I promised to talk about fixing a parameter, currying and uncurrying, folding, and accumulators. These are various applications of higher-order functions.

Fixing a parameter. We've seen this already in Lab 1. Lab 1 included `map` expressions, which would apply a transformer $f(x) := x$ `op` c to all elements of the list. We allowed `op` to be one of `+`, `-`, `*` or `/`, and c a constant specified with `map`.

Operators take two parameters. By fixing a parameter, we get a one-parameter function:

```
scala> def plus5(x:Int) = x + 5
plus5: (Int) => Int

scala> plus5(2)
res0: Int = 7
```

Currying. Let's generalize this. For any two-parameter function $f(x, y)$, we can write f' which takes one parameter, x, and *returns* a function $f_x(y)$ which fixes x for f. This is called *currying*[1].

In other languages, you can write currying in the language itself, but Scala uses `Function.curried` to get a curried function. Here's how it works.

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

scala> def plusCurried = Function.curried(plus _)
plusCurried: (Int) => (Int) => Int

scala> def plus5 = plusCurried(5)
plus5: (Int) => Int

scala> plus5(2)
res5: Int = 7
```

The `plusCurried` function takes an `Int`, x, and returns the function $f_x(y)$. The returned function $f_x(y)$ fixes x and adds it to y.

Uncurrying. How would the inverse of currying work?

[1] Named after Haskell Curry, a logician.

Summations. We can also create a higher-order function to add numbers. In fact, we'll create a higher-order function to recursively evaluate:

$$\sum_{i=a}^{b} f(i).$$

```
scala> def sum(f:Int=>Int,a:Int,b:Int) : Int =
     | if (a > b) 0 else f (a) + sum(f, a + 1, b)
sum: ((Int) => Int,Int,Int) => Int

scala> sum(x => x, 3, 5)
res7: Int = 12

scala> sum(x => x*x, 3, 5)
res8: Int = 50

scala> sum(x => Math.pow(2,x).asInstanceOf[Int], 3, 5)
res16: Int = 56
```

This is still pretty limited. Let's see how to generalize this by allowing the user to specify an arbitrary sequence to sum over.

```
scala> def sum(f:Int=>Int, xs:Seq[Int]) =
     | xs.foldLeft(0)((acc, n) => acc+f(n))
sum: ((Int) => Int,Seq[Int]) => Int

scala> sum(x=>x,List(3,4,5))
res26: Int = 12

scala> sum(x=>x*2,List(1,2))
res25: Int = 6
```

What's going on here?

Reducing and folding. We've used the `foldLeft` operation. Here's what it does on our example.

- It first sets the accumulator to the start value, which we've provided as 0.
- For each element e in the list, it calls the supplied function with the accumulator and e, using the returned value as the new accumulator value.
- When there are no more elements in the list, return the accumulator.

Let's work out an example of this fold operation.

There is also an alternate syntax for `foldLeft`:

```
scala> def sum(f:Int=>Int, xs:Seq[Int]) = (0 /: xs)((acc, n) => acc+f(n))
```

The `reduceLeft` operation is similar to the `foldLeft` operation, except that it uses the first element of the list as the start value.

```
scala> def concat(xs:Seq[String]) =
     | xs.reduceLeft((acc, n) => acc + "," + n)
concat: (Seq[String]) => String

scala> concat(List("one", "two", "three"))
res27: String = one,two,three
```

The left fold and reduce operations associate operations to the left. Right fold and reduce operations associate to the right. For instance, in evaluating `sum(x=>x,List(3,4,5))` with a left fold, we evaluate $((0+3)+4)+5$. The right fold would return the same result, but would actually evaluate $3+(4+(5+0))$.

Other remarks

Scala type inference allows us to declare variables without stating their types. It's also quite useful for finding the types of functions, even when they are complicated higher-order functions.

Functional languages support infinite streams. You can give a rule for computing the next stream element from the previous ones. Some examples of streams include the integers, odd integers, prime numbers, etc. Streams behave just like lists, except that you can't print out the whole stream.