ECE251: Programming Languages & Translators Fall 2010

Lecture 25 — November 18, 2010

Patrick Lam *version 2*

Let's look at some examples of Scala code: first a largish example, then various features of functional languages as seen in Scala. I won't be marking syntax, but I will be marking ideas (as expressed in Scala).

Quicksort. Here is a literal translation from Java[1].

```
def sort (xs: Array[Int]) {
  def swap (i:Int, j:Int) {
    val t = xs(i); xs(i) = xs(j); xs(j) = t
  }
  def sort1(l:Int, r:Int) {
    val pivot = xs((l+r)/2)
    var i = l; var j = r
    while (i <= j) {
      while (xs(i) < pivot) i += 1
      while (xs(j) > pivot) j -= 1
      if (i <= j) {
        swap(i, j); i += 1; j -= 1;
      }
    }
    if (l < j) sort1(l, j)
    if (j < r) sort1(i, r)
  }
  sort1(0, xs.length - 1)
}

scala> var q = Array(2,3,4);
q: Array[Int] = Array(2, 3, 4)

scala> sort(q)

scala> q
res6: Array[Int] = Array(2, 3, 4)
```

So, you can just write "normal Java-style code" in Scala. (Hence Scala is interoperable with other lagnuages.)

[1] This is from "Scala by Example", which I mentioned last time.

Functional style. That was not really the point of Scala. Let's consider this alternate implementation.

```
def sort(xs:Array[Int]) : Array[Int] = {
  if (xs.length <= 1) xs
  else {
    val pivot = xs(xs.length/2)
    Array.concat(
      sort(xs filter (pivot >)),
           xs filter (pivot ==),
      sort(xs filter (pivot <)))
  }
}

scala> val r = Array(8,2,1,4)
r: Array[Int] = Array(8, 2, 1, 4)

scala> sort(r)
res7: Array[Int] = Array(1, 2, 4, 8)
```

Note how this expresses the essence of quicksort:

1. empty or single-element arrays are already sorted, return array itself.
2. otherwise, chose a pivot in the middle;
3. create sub-arrays: (3a) less than pivot; (3b) equal to pivot; (3c) greater than pivot;
4. recursively sort (3a) and (3c);
5. append all of the arrays together.

Analysis: The two versions have the same asymptotic complexity (which is?) The imperative version modifies the array in-place, while the functional version returns a new, sorted array. It therefore uses more working memory, but is also more parallelizable.

Big Idea: Higher-Order Functions. We have one higher-order function here, `filter`. We call it using the invocation:

```
    xs filter (pivot >)
```

What this means is that we are calling `filter` on `xs`, an object of type `Array[T]`.

```
class Array[T] {
  def filter(p: T => Boolean) : Array[T]
  // etc
}
```

The `filter` function is a *higher-order function* because it takes a function, `p`, as an argument.

We call `p` a *predicate function*, because it takes an object and returns `true` or `false`. Note that `p` has type `T => Boolean`, and that it's totally fine to pass `p` as a parameter to `filter`. The contract of `filter` is that it returns a subarray containing all objects o of the array which satisfy the predicate, i.e. `p(o) == true`.

Now, what about `pivot >`? This is a bit confusing, but it defines an anonymous function which returns `true` for all `int`s less than `pivot`. Another way of writing it is `x => pivot > x`.

But the syntax we have here is for a *partially-applied function*. In Scala, `>` is a method on the `Int` class which takes a parameter (equivalently, a two-parameter function which returns `true` if left > right). We've applied it to the left parameter, `pivot`, so now we get a method which returns `true` if `pivot` is greater than the remaining argument.

Bottom Line. The two implementations are relatively similar, but look quite different.

More on Higher-Order Functions

Higher-order functions, which either take a function as an argument, or return a function, allow you to parametrize over the code structure, helping to avoid the need to write boilerplate code. (Note that ASTs are particularly boilerplate-heavy).

The key is that higher-order functions usually take relatively simple, pure (no side-effect) functions: predicates (`T => Bool`) or other simple functions (`T => T'`) as specifications of what to do to a data structure.

Examples of higher-order functions. Here are some of the classical higher-order functions. There are also a couple more from math which we won't talk about, e.g. derivatives and integrals (if we did, we ought to talk about numerical stability).

- `map`: takes a sequence `Seq[T]` and a function `T => T'`, and returns a sequence `Seq[T']`.

  ```
  scala> val l = List(2,3,4)
  l: List[Int] = List(2, 3, 4)

  scala> l map (x => x + "foo")
  res16: List[java.lang.String] = List(2foo, 3foo, 4foo)
  ```

  Examples:

  ```
  scala> l map (2 *)
  res17: List[Int] = List(4, 6, 8)

  scala> l map (x => x.toFloat)
  res18: List[Float] = List(2.0, 3.0, 4.0)
  ```

- `filter`: we saw this before in the `sort` example.
- folding, general accumulators, currying, uncurrying: next time!