Lecture 13

The *Same Fringe* Problem

Given a binary tree:

```
sealed trait BinTree[+A]
case class Node[A](lhs: BinTree[A], rhs: BinTree[A]) extends BinTree[A]
case class Leaf[A](x: A) extends BinTree[A]
```

The *fringe* of a binary tree are the values in left-to-right order. For example, the fringe of the following tree:

```
val t1 = Node(Leaf(10), Node(Leaf(20), Leaf(40)))
```

is 10, 20, and 40, in that order. The *same-fringe problem* is to write a function to determine if two trees have the same fringe. For example the following tree:

```
val t2 = Node(Node(Leaf(10), Leaf(20)), Leaf(40))
```

Has the same fringe as `t1`.

We can write a simple recursive function to calculate the fringe of a tree, as shown in fig. 23.1, so a simple solution to the same-fringe problem is:

```
fringe(t1) == fringe(t2)
```

But, this is not a very efficient solution:

- If the trees are very large, we'll spend a lot of time appending lists, and
- If the fringes are different, we'll generate the entire fringe instead of terminating as soon as a difference is detected.

An Imperative Solution

A typical Java solution to this problem, which is straightforward to reproduce in Scala, is to use an iterator. You've seen simple iterators for lists and arrays before. You can also write an iterator that iterates over the fringe of a tree. The key idea is this: before we descend into the left-hand side of a node to generate its fringe, we need to store the right-hand side of the node in a variable so that we don't forget to generate its fringe. Since the tree may be arbitrarily deep, we may need to remember a stack of nodes.

> **Think!** What would happen if we used a queue of nodes?

An iterator that uses this idea is shown in fig. 23.2 along with a solution to the same-fringe problem that doesn't have the two issues we identified above. An unusual feature of this solution is that it uses two iterators simultaneously in a single loop.

What's interesting about this problem is that the solution seems to be fundamentally imperative. In particular, each invocaation of `.next` returns the next value, which means that the iterators *must* use mutable state internally. Is it possible to solve this problem without state?

```
def fringe[A](t: BinTree[A]): List[A] = t match { case Leaf(x) =>
  List(x) case Node(lhs, rhs) => fringe(lhs) ++ fringe(rhs) }
```

Figure 23.1: Recursively calculating the fringe of a binary tree.

```
class FringeIterator[A](tree: BinTree[A]) extends Iterator[A] {
  private val stack = collection.mutable.Stack(tree)
  def hasNext = stack.isEmpty
  def next() = {
    val top = stack.pop()
    top match {
      case Leaf(x) => x
      case Node(lhs, rhs) => { stack.push(rhs); stack.push(lhs); next() }
    }
  }
}

def sameFringe[A](t1: BinTree[A], t2: BinTree[A]): Boolean = {
  val iter1 = new FringeIterator(t1)
  val iter2 = new FringeIterator(t2)
  while (iter1.hasNext && iter2.hasNext) {
    val x = iter1.next
    val y = iter2.next
    if (x != y) {
      return false
    }
  }
  return !iter1.hasNext && !iter2.hasNext
}
```

Figure 23.2: An imperative solution.

```
def g(): Int = {
  println("Evaluating g")
  10
}

def f1(x: Int) = {
  println("Evaluating f1")
  x + 10
}
f1(g())
```

Figure 23.3: Which line prints first?

> **Think!** Try to solve the problem before proceeding further.

By-Name Arguments

When you apply a function in Scala (and in most other programming languages), the argument of the function is first reduced to a value and then that value is passed to the function. For example, in `f(2 + 3)`, the expression `2 + 3` is first reduced to the value `5` and then `f(5)` is applied.

> **Think!** Is the statement above true? Can you think of a way to experimentally validate the claim that arguments are reduced to values before they are passed to functions?

Consider the two functions in fig. 23.3 and the expression `f1(g())`. If the argument is evaluated first, then "Evaluating g" will print before "evaluating f". Conversely, if the expression `g()` is passed to `f1` unevaluated and only evaluated when it is needed, then we'd expect the lines to print in the opposite order. If you try this out, you'll find that the former is true, thus validating the claim.

However, Scala allows you to change the order of evaluation. In the following function, the `=>` annotation means that the argument is only evaluated when it is *needed*:

```
def f2(x: => Int) = {
  println("Evaluating f2")
  x + 10
}
```

Therefore, the expression `f2(g())` prints "Evaluating f2" before it prints "Evaluating g", since the value of `g()` is not needed until after "Evaluating f2" is printed.

The following function ignores its argument:

```
def f3(x: => Int) = {
  println("Evaluating f3")
  10
}
```

Therefore, `f3(g())` does not evaluate `g()` at all, so "Evaluating g" is not printed. In fact, you can even write `f3(throw new Exception("bad"))` and the exception will not be thrown.

By-name arguments have several applications and are used extensively in the Scala standard library. For example, Scala maps have a `.get` method that optionally returns a value stored in a map:

```
val m = Map("X" -> 1 )
val v = m.get(key) // produces Some(1) if key is "X", otherwise None
val v2 = m.get("Z") // produces None
```

Suppose we were using maps in a program and that we wanted to return a default value 0 instead of `None`. We could write:

```
m.get(key) match {
  case Some(v) => v
  case None => 0
}
```

Alternatively, we could simply write `m.getOrElse(key, 0)`. It turns out that the second argument of this method is a by-name argument. Therefore, we can write `m.getOrElse(key, f())` and be assured that `f` will not be applied unless it is needed. It is safe to write, even if `f()` is an expensive computation that should only be run if `key` is not in the map. We could even write `m.getOrElse(key, throw new Exception(...))` to throw a custom exception if `key` is not found.

Functional Generators

The reason that iterators cannot be functional is evident in the type for iterators. The `Iterator<T>` interface is part of the Java standard library and it corresponds to the following Scala trait:

```
trait Iterator[T] {
  def hasNext(): Boolean
  def next(): T
  def remove(): Unit // not relevant for our discussion
}
```

Recall that a key property of functional programs is that a function (or method) always produces the same result when applies to the same arguments. In this interface, the `next()` method takes zero arguments. Therefore, if an implementation of `Iterator[T]` were written functionally, it could only ever produce one result.

To allow functional programming to work, we're going to have to work with a new kind of iterator, which we'll call a *generator*, where the `next` method produces the next value and *a new generator that generates the rest of the collection*:

```
trait Generator[T] {
  def hasNext(): Boolean
  def next(): (T, Generator[T])
}
```

An important detail of iterators that isn't obvious from the type, is that the `next` method will throw an exception if `hasNext` produces `false`. Instead of throwing an exception, we can make this behavior manifest in the type, by eliminating `hasNext` method and changing the type of `next` to optionally produce an element and a generator.

```
trait Generator[T] {
  def next(): Option[(T, Generator[T])]
}
```

We can now write some a functional generator. For example, here is a generator that produces three numbers:

```
val agen = new Generator[Int] {
  def next() = Some((1, new Generator[Int] {
                          def next() = Some((2, new Generator[Int] {
                                                  def next() = Some((3, new Generator[Int] {
                                                                          def next() = None
```

```
                                                                    }))
                                                            }))
                                    }))
}
```

Here is a simple function that prints all the elements generated by any generator, including the one above:

```
def printAll[T](gen: Generator[T]): Unit = gen.next() match {
  case None => ()
  case Some((x, rest)) => println(x); printAll(rest)
}
```

Combining Generators You probably agree that the definition of `agen` is too unwieldy. To make generators easier to write, we are going to define three generator building functions:

1. The function `zero[T]()` produces a generator that generates zero values of type `T`:

   ```
   def zero[T](): Generator[T] = new Generator[T] { def next() = None }
   ```

2. The function `one(x)` produces a generator that only generates the value `x`:

   ```
   def one[T](x: T): Generator[T] = new Generator[T] { def next() = Some((x, zero[T]())) }
   ```

3. The function `append(gen1, gen2)` takes two generators as arguments and first generates the values of `gen1` and then the values of `gen2`. Furthermore, to avoid generating values we don't need, the arguments are passed by-name:

   ```
   def append[T](gen1: => Generator[T], gen2: => Generator[T]): Generator[T] = new Generator[T] {
     def next() = gen1.next match {
       case None => gen2.next
       case Some((x, gen1Rest)) => Some((x, append(gen1Rest, gen2)))
     }
   }
   ```

   To make `append` easier to use, we can modify the `Generator[T]` trait to include an operator that invokes `append`:

   ```
   trait Generator[T] {
     def next(): Option[(T, Generator[T])] // as before
     def ++(other: => Generator[T]): Generator[T] = append(this, other)
   }
   ```

Using these functions, we can write a generator that generates the fringe of a binary tree:

```
def fringe[T](t: BinTree[T]): Generator[T] = t match {
  case Leaf(x) => one(x)
  case Node(lhs, rhs) => fringe(lhs) ++ fringe(rhs)
}
```

Notice that it is very similar to the function that generates the fringe as a list. However, it behaves quite differently. Since the append (`++`) operation evaluates its arguments only when needed, it doesn't immediately visit all the leaves of the tree, which was the problem with the original function.

The following function takes two generators and produces `true` if the generate exactly the same values:

```
def sameGen[T](gen1: Generator[T], gen2: Generator[T]): Boolean = (gen1.next, gen2.next) match {
  case (None, None) => true
  case (Some((x, rest1)), Some((y, rest2))) => x == y && sameGen(rest1, rest2)
  case _ => false
}
```

If the supplied generators only generate values on need, the `sameGen` function will abort early and produce `false` if (1) it encounters two values `x != y` or (2) one generator runs out of values before the other. Notice that `sameGen` has the exactly the same structure as the obvious recursive function that checks if two lists are equal.

Finally, we can write the `sameFringe` function that we wanted as follows:

```
def sameFringe[T](t1: BinTree[T], t2: BinTree[T]) = sameGen(fringe(t1), fringe(t2))
```

Think! How would you verify that this version of `sameFringe` truly aborts early and doesn't incorrectly traverse both trees when it isn't necessary?

More Generators It is staightforward to define other handy generator-building functions. For example, it is easy to define functions that are analogous to `map`, `filter`, and `fold` for lists. You can even write a function to flatten a generator-generator into a simple generator:

```
def flatten[T](gen: Generator[Generator[T]]): Generator[T] = new Generator[T] {
  def next() = gen.next match {
    case None => None
    case Some((x, rest)) => (x ++ flatten(rest)).next
  }
}
```