Updated: 2021-03-04

4.5 Recursion

Northeastern University

Iteration

- You're all familiar with iteration in Java:

```
public class Iteration {

    public static int sum(int[] array) {
        int result = 0;
        for (int i : array) result += i;
        return result;
    }

    public static void main(String[] args) {
        int[] array = new int[10];
        for (int i = 0; i < 10; i++) array[i] = i+1;
        System.out.println(sum(array));
    }
}
```

- Note that this involves the use of a mutable variable called *result*.

Addition by recursion

- Iteration vs. Recursion
  - In functional programming we like to avoid the use of mutable variables so that we can prove programs using *referential transparency* (a.k.a. the substitution principle).
  - But what is the more natural way to do the sum? iteration or recursion?
  - A discrete series is often defined recursively (e.g. Fibonacci sequence). I suggest that iteration is a consequence of the Turing/von Neumann architecture.
  - A more mathematical way of defining the sum of a list *xs* of numbers would be:
    - sum(xs) = xs.head + sum(xs.tail)

Sum by recursion

- Here's the simplest way to sum in Scala:

```
object Sum extends App {

  def sum(xs: Seq[Int]): Int = xs match {
    case Nil => 0
    case h :: t => h + sum(t)
  }

  val xs = LazyList.from(1) take 10
  println(sum(xs.toList))
}
```

Recursively call sum

- But haven't we always been taught not to use recursion in our programs if we can avoid it?
  - What's wrong with recursion?
  - Think about what would happen if instead of 10 numbers, we summed 10 *billion* numbers.

Stack Overflow

- We'd get a stack overflow. Try it for yourself.
  - That's really bad news. That's why we were taught not to use recursion!
- But in functional programming we can actually avoid using the stack provided that the recursion is *tail* recursive.
  - Suppose that the *last* thing we do in our code is the recursive call itself (that's called *tail* recursion)? We'd also say that the recursive call is in *tail position.* In that case, there'd be nothing we'd need to store on the stack, right?
  - So, we can "unroll" a tail-recursive call into a kind of iteration.

Sum by recursion (take 2)

- Let’s take another look

```
object Sum extends App {

  def sum(xs: Seq[Int]): Int = xs match {
    case Nil => 0
    case h :: t => h + sum(t)
  }

  val xs = LazyList.from(1) take 10
  println(sum(xs.toList))
}
```

Recursively call* sum *but the* last *thing we do is to add* h *to the result of the recursive call— “+” is in tail position.

- How can we make this tail-recursive?
  - Actually, it’s quite easy…

Sum by tail-recursion*

- We create an “inner” method which is tail-recursive
  - Its signature is based on two† things:
    - the current value of the result (and which will be yielded when the recursion terminates, in this case when *work* is *Nil*);
    - the work still to do.
  - Here’s our new *sum* (note we use *BigInt* because we no longer have a restriction on the size of *xs*):

```scala
object Sum extends App {
  def sum(xs: Seq[Int]): BigInt = {
    def inner(result: BigInt, work: Seq[Int]): BigInt = work match {
      case Nil => result
      case h :: t => inner(result+h,t)
    }
    inner(0, xs)
  }
  val xs = LazyList.from(1) take 10000000
  println(sum(xs.toList))
}
```

Tail-recursively call* inner *which* is *the* last *thing we do.

*** also known as tail call recursion**

† three if you’re processing something 2-dimensional like a tree

How can we be <u>sure</u> it's *tail*-recursive?

- The compiler will optimize it provided that it really is tail-recursive. But what if it's not?
  - In that case, you risk a stack overflow at run-time!
  - But here's what you can do: just add the *tailrec* annotation which asserts that the method *is* tail-recursive and, if it's not, the compiler will warn you:

```
import scala.annotation.tailrec
object Sum extends App {
  def sum(xs: Seq[Int]): BigInt = {
    @tailrec def inner(result: BigInt, work: Seq[Int]): BigInt = work match {
      case Nil => result
      case h :: t => inner(result+h, t)
    }
    inner(0, xs)
  }
  val xs = LazyList.from(1) take 10000000
  println(sum(xs.toList))
}
```

Tail Recursion —factorial

- Let's take a look at perhaps the most obvious recursive function, *factorial*:

```
scala> def badFactorial(x: Int): Long = if (x<=1) 1 else x*badFactorial(x-1)
badFactorial: (x: Int)Long
```

  - This isn't tail-recursive. Why not?
  - But the following *is* tail-recursive (which we can assert with the annotation):

```
    def factorial(n: Int) = {
  @scala.annotation.tailrec def inner(r: Long, n: Int): Long =
    if (n <= 1) r
    else inner(n * r, n - 1)
  inner(1L, n)
}
```

When we create one of these tail-recursive inner methods, we usually have two parameters: the first (*r*) is the current result; the second (*n*) represents the work still to do. Sometimes, there is a third parameter which controls some other aspect of the recursion.

What else can we do that's tail-recursive?

- *sum* and *factorial*
  - In the *sum* case, we added *h* to *result*;
  - In the *factorial* case, we multiplied *n* by *r*.
- *FoldLeft* for a more general aggregation function
  - In general, we can provide our own function to aggregate the current result with the current head (we will also need a value to use as the starting result).

```
def foldLeft[X,Y](xs: Seq[X])(y: Y)(f: (Y,X)=>Y): Y = {
  @tailrec def inner(result: Y, work: Seq[X]): Y = work match {
    case Nil => result
    case h :: t => inner(f(result,h),t)
  }
  inner(y, xs)
}
def sum(xs: Seq[Int]): BigInt = foldLeft(xs)(0)(_+_)
val xs = LazyList.from(1) take 10000
println(sum(xs.toList))
```

Other recursive methods

- Of course, we can also define *foldRight* but it can't be efficient <u>and</u> tail-recursive unless we are operating on a backwards list.
- We can also define *reduce* like *foldLeft* but in *reduce*, the initial value is inferred to be the zero value in the *Y* type. This constrains *Y* such that we can create a *Y* based on zero. More on that when we get to implicits…
- Another important recursive method which is similar but is not reducing (as in *Seq[A] => A*) but composing (*Seq[A] => Seq[A]*) is *scanLeft* (and *scanRight*). Elements of the result are compositions of adjacent elements of the input, where the composition is defined by the given function. Remember this?

```
0L #:: f.scanLeft(1L)(_ + _)
```

Remember *sequence*, *traverse*?

- What if you had a *Seq[Option[X]]* and you wanted an *Option[Seq[X]]?*
  - *sequence*:

    ```
    def sequence[X](xos: Seq[Option[X]]): Option[Seq[X]] = ???
    ```

  - this method should iterate through *xos* and, if all elements are *Some(x)*, collect them into a sequence *xs* then return *Some(xs)*. If any of the elements are *None*, return *None*.
  - Now, we're ready to implement this one:

```
def sequence[X](xos: Seq[Option[X]]): Option[Seq[X]] = (Option(Seq[X]()) /: xos) {
    (xso, xo) => for (xs <- xso; x <- xo) yield xs :+ x
  }
```

I don't expect you to remember this!!!

***Deprecated* synonym for *foldLeft* where operands are swapped**

See also…

- http://scalaprof.blogspot.com/2016/05/transforming-iteration-into-tail.html
- http://scalaprof.blogspot.com/2016/11/a-generic-tail-recursive-method-for.html
- You can also transform a recursive call in tail position into a loop yourself: you can do it in Java or Scala. But keep in mind that the Java compiler does not optimize tail calls.
- Java8 also allows you to do tail-call recursion but it uses a technique called *trampolining* which is beyond our scope.