15 Tail Recursion

We discussed how we interpreted features of the object language using corresponding features of the metalanguage. For example, we used recursion in Scala to interpret recursion in our object language. This is a consequence of our use of a recursive evaluation method for interpreted our object language expressions.

For instance, suppose we are interpreting a call `(sum 10)` where `sum` is defined as usual:

```
(define sum
   (fun s (n) (tfun (int) int)
     (if (= n 0) 0 (+ n (s (- n 1))))))
```

If we evaluate `(sum 10)`, or more accurately the abstract representation of `(sum 10)`, we end up calling method `EApply.eval()` with function `sum` and argument `10`. During that call, we end up recursively calling method `EApply.eval()` with function `sum` and argument `9`. Thus, recursion in the object language is interpreted via recursion in the metalanguage.

The problem with this approach in our specific case is that our object language inherits the limitations on recursion of the metalanguage. For instance, if we try to evaluate `(sum 100000)`, we're liable to get an error from Scala, most likely that we've run out of stack space. The exact number where this problem will occur may vary from system to system, but it will happen.

Scala implements recursion the way many imperative languages implement recursion, by using a stack (the *call stack*) in which to save the context that existed at the time of a recursive call so that when the recursive call returns execution can pick up where it left off. This context is saved into something called an *activation frame*, which holds, among other things, the local variables of the function. Every recursive call adds an activation frame on the call stack. And Scala does not let the call stack get too tall.

This is not specific to our interpreter. Consider the Scala version of the above function sum:

```
def sum (n:Int):Int = {
  if (n == 0) {
    return 0
  } else {
    return n + sum(n-1)
    }
```

This will also fail with `sum(100000)`.

This seems unavoidable. Recursion seems to require a call stack to grow, and Scala has limitations on the size of the call stack.

It turns out that we can bypass this limitation of Scala in some cases at least. In order to do so, we need to study our object language a bit more closely.

The Scala code for `sum` above is not how most programmers accustomed to writing in an imperative language would write such a function. They would use iteration:

```
  def sum_iter (n:Int):Int = {
    var result = 0
    var curr = n
    while (curr > 0) {
      result += curr
      curr = curr - 1
    }
    return result
  }
```

Note that this does not grow the call stack.

How can we lift the iteration idea to our object language? It turns out that in a functional language, we can write a form of iteration as follows:

```
  (define sum_iter
    (fun (n result) (tfunc (int int) int)
      (if (= n 0)
          result
         (sum_iter (- n 1) (+ n result)))))
```

You can see the similarities. One disadvantage is that it requires the passing of two values, but that's easy enough to fix with a wrapper function `(fun (n) (tfun (int) int) (sum_iter n 0))`.

Clearly, the two functions `sum` and `sum_iter` compute the same thing. But they do so differently. To analyse them, we'll use *equational reasoning.* Roughly speaking, we'll reason algebraically about the behavior of the functions. That's something you often can do with a functional language. You can think of the definition of a function as defining an equation so that, for example:

```
(sum n) = (if (= n 0) 0 (+ n (sum (- n 1))))
```

and

```
(sum_iter n res) = (if (= n 0) res (sum_iter (- n 1) (+ res n)))
```

So let's use these equations to see what `(sum 6)` and `(sum_iter 6 0)` look like:

```
(sum 6) = (if (= 6 0) 0 (+ 6 (sum (- 6 1))))
        = (if false 0 (+ 6 (sum (- 6 1))))
        = (+ 6 (sum (- 6 1)))
        = (+ 6 (sum 5))
        = (+ 6 (if (= 5 0) 0 (+ 5 (sum (- 5 1)))))
        = (+ 6 (+ 5 (sum 4)))
        = ...
        = (+ 6 (+ 5 (+ 4 (+ 3 (+ 2 (+ 1 (sum 0)))))))
        = (+ 6 (+ 5 (+ 4 (+ 3 (+ 2 (+ 1 (if (= 0 0) 0 (+ 0 (sum (- 0 1))))))))))
        = (+ 6 (+ 5 (+ 4 (+ 3 (+ 2 (+ 1 0))))))    (*)
        = (+ 6 (+ 5 (+ 4 (+ 3 (+ 2 1)))))
        = (+ 6 (+ 5 (+ 4 (+ 3 3))))
        = ...
        = (+ 6 15)
        = 21
```

The key thing to notice is that we can't start simplifying the expression until we've basically expanded out all the recursive `sum` calls, in line (*).

Contrast to the equational simplification of `(sum_iter 6 0)`:

```
(sum_iter 6 0) = (if (= 6 0) 0 (sum_iter (- 6 1) (+ 6 0)))
               = (if false 0 (sum_iter (- 6 1) (+ 6 0)))
               = (sum_iter 5 6)
               = (if (= 5 0) 6 (sum_iter (- 5 1) (+ 5 6)))
               = (sum_iter 4 11)
               = (if (= 4 0) 11 (sum_iter (- 4 1) (+ 4 11)))
               = (sum_iter 3 15)
               = (if (= 3 0) 15 (sum_iter (- 3 1) (+ 3 15)))
               = (sum_iter 2 18)
               = (if (= 2 0) 18 (sum_iter (- 2 1) (+ 2 18)))
               = (sum_iter 1 20)
               = (if (= 1 0) 20 (sum_iter (- 1 1) (+ 1 20)))
               = (sum_iter 0 21)
               = (if (= 0 0) 21 (sum_iter (- 0 1) (+ 0 21)))
               = 21
```

While the terms in the equational simplification of `(sum n)` get longer and longer as `n` gets larger, the terms in `(sum_iter n 0)` remain essentially constant size.

This is not surprising. In a way that can be made precise (but which I won't do here), the size of the terms in the equational simplifications above correspond precisely to the growth of the call stack in the Scala execution model!

(Because of this feature, a function such as `sum_iter` is usually said to implement an *iterative process* in the context of a functional language. Note that it is still syntactically recursive: it is still a function that refers to itself in its body. But when it executes, it doesn't grow the stack. In contrast, a function such as `sum` is said to implement a *recursive process*.)

By this correspondence,`sum_iter` in our object language doesn't seem to grow the stack. It's enlightening to go back to Scala and see how that manifests itself there. Here's `sum_iter` in Scala, using recursion:

```
def sum_iter (n:Int,result:Int):int {
  if (n == 0) {
    return result
  } else {
    return sum_iter(n-1,result+n)
  }
}
```

If you execute `sum_iter(100000)`, you once again blow the stack! So what's going on? I claim that the stack doesn't grow based on the equational simplification above, yet Scala still can't deal with.

That's because the actual claim is that the stack doesn't *need* to grow. If Scala implemented something called *tail-call optimization*, then `sum_iter` would execute just fine. Tail-call optimization is the realization that if the very last thing that a function A does before returning is calling another function B, then we don't need to put a new activation frame for B on the call stack, but instead we can just reuse the activation frame for A. (An activation frame is meant to record the context of a call so that the system know what to return to after the call is done, but if the next thing to do when function B returns is to return from A itself, then we don't need to record the context of A — it doesn't serve any role.)

If Scala implemented tail-call optimization, then we could execute

```
sum\_iter(100000)
```

without growing the stack, and it would execute without problem.

That's a problem for us because if you look at, for instance, the `eval()` method in `EApply`, you see the following:

```
def eval (env : Env[Value]) : Value = {
   val vf = f.eval(env)
   val vargs = args.map((e:Exp) => e.eval(env))
   return vf.apply(vargs)
}
```

and in `VRecClosure`:

```
override def apply (args: List[Value]) : Value = {
   // Type system ensures that right # args is passed
   var new_env = env
   for ((p,v) <- params.zip(args)) {
      new_env = new_env.push(p,v)
```

```
    }

    // push the current closure as the value bound to identifier self
    new_env = new_env.push(self,this)
    return body.eval(new_env)
  }
```

and note the last line: the last thing `eval()` does before returning is calling `vf.apply.vargs()`, which itself calls `body.eval()` as its last thing it does before returning. If Scala implemented tail-call optimization, it would not need to grow the stack here. If you do a bit of reasoning and scribble some diagrams, you can convince yourself that such a thing would allow you to interpret `(sum_iter n)` in our object language for very large `n`s.

Most functional languages implement tail-call optimization. This allows them to execute recursive functions in which every recursive call is in tail position (that is, where every recursive call is the last thing that the function does before returning — those functions often called *tail recursive*) without growing the stack, as an iterative process.

But Scala does not implement tail-call optimization. So if we want to be able to execute an iterative process implemented as a recursive function in our object language, we have to work harder. We can modify our interpreter so that `eval()` for `EApply` does not use an activation frame for evaluating the body of the function being called.

The trick is to basically turn a tail call to f into a jump to f. Scala doesn't let us do jumps to particular points in the code (and that's a *good* thing!) so we have to fake jumps using a loop. The easiest way to implement a tail-call optimized form of evaluation is to pull evaluation *out* of the concrete subclasses of `Exp` and put it in the `Exp` base class, where we basically do a case analysis on the kind of expression we have in order to do the evaluation for each case. When we need to evaluate recursively, we evaluate recursively as usual. In the situations where evaluation is in tail position, we simply replace the expression we are currently evaluating, and jump to the top of our evaluation function. Here's the code. Can you figure out why this works?

```
abstract class Exp {
  ...

  def evalTail (env : Env[Value]) : Value = {
```

```
var currExp = this
var currEnv = env

while (true) {
  currExp match {

     case EIf(ec,et,ee) => {
       val ev = ec.evalTail(currEnv)
       if (!ev.getBool()) {
         currExp = ee
       } else {
         currExp = et
       }
     }

     case EApply(f,args) => {
        val vf = f.evalTail(currEnv)
        val vargs = args.map((e:Exp) => e.evalTail(currEnv))
        if (vf.isPrimOp()) {
          return vf.applyOper(vargs)
        } else {
          // defined function
          // push the vf closure as the value bound to identifier self
          var new_env = vf.getEnv().push(vf.getSelf(),vf)
          for ((p,v) <- vf.getParams().zip(vargs)) {
             new_env = new_env.push(p,v)
          }
          currEnv = new_env
          currExp = vf.getBody()
        }
     }

     case ELet(bindings,body) => {
         var new_env = currEnv
         for ((n,e) <- bindings) {
           val v = e.evalTail(currEnv)
           new_env = new_env.push(n,v)
         }
         currEnv = new_env
         currExp = body
     }

     // every other expression type evaluates normally
     case _ => return currExp.eval(currEnv)
```

```
            }
        }
        return new VInteger(0) // needed for typechecking
    }
}
```

This relies on a couple of things:

1. every concrete subclass of `Exp` is a *case class* so that we can apply Scala pattern matching;
2. every expression node still has a method `eval()`, used for those cases that don't have recursive evaluation calls.

With these changes to the interpreter, once we have the shell use `evalTail()` instead of `eval()` to evaluate typed expressions, we can evaluate `(sum_iter 100000)` and more without any problem:

```
TFUNC> (define sum-iter
         (fun s (n result) (tfun (int int) int)
           (if (= n 0) result (s (+ n -1) (+ result n)))))
sum-iter defined with type (fun (int int) int)
TFUNC> (sum-iter 100000 0)
705082704 : int
TFUNC> (sum-iter 1000000 0)
1784293664 : int
```