CSci 555, Functional Programming Recursion Concepts and Terminology: Scala Version

H. Conrad Cunningham

10 February 2016 (minor formatting updates 3 August 2016)

Contents

Acknowledgements: I adapted the factorial, Fibonacci number, and exponentiation functions used below from the Scala functional programming examples developed for this class based on similar Elixir programs, which, in turn, were adapted from similar Lua programs, which, in turn, were adapted from the Scheme programs in Abelson and Sussman's classic SICP textbook. This set of notes was also adapted from a version using Elixir examples.

Recursion Concepts and Terminology

Linear and Nonlinear Recursion

Linear recursion

A function definition is *linear recursive* if at most one recursive application of the function occurs in a leg of the definition (i.e., along a path from an entry to a return). The various function clauses and branches of the conditional expressions `if` and `match` introduce a path.

The definition of the function `factorial` below is linear recursive because the expression in the second leg of the definition (i.e., `n * factorial(n-1)`) involves a single recursive application. The other leg is nonrecursive; it is the base case of the recursive definition.

```
def factorial(n: Int): Int = n match {
  case 0          => 1
  case m if m > 0 => m * factorial(m-1)
}
```

Scala checks for pattern matches for the clauses in the order given in the function definition. It executes the leg corresponding to the first successful match. If no pattern matches, then the function aborts and displays an error message.

Time and space complexity

How efficient is function `factorial`?

Function `factorial` recurses to a depth of `n`. It thus has *time complexity* O(`n`), if we count either the recursive calls or the multiplication at each level. The *space complexity* is also O(`n`) because a new runtime stack frame is needed for each recursive call.

Termination of recursion

How do we know that function `factorial` terminates?

To show that evaluation of a recursive function terminates, we must show that each recursive application *always* gets closer to a termination condition represented by a base case. For a call `factorial(n)` with `n > 0`, the argument of the recursive application always decreases to `n - 1`. Because the argument always decreases in integer steps, it must eventually reach 0 and, hence, terminate in the first leg of the definition.

Preconditions and postconditions

The *precondition* of a function is what the caller (i.e., the client of the function) must ensure holds when calling the function. A precondition may specify the valid combinations of values of the arguments. It may also record any constraints on the values of “global” data structures that the function access or modifies.

If the precondition holds, the supplier (i.e., developer) of the function must ensure that the function terminates with the *postcondition* satisfied. That is, the function returns the required values and/or alters the “global” data structures in the required manner.

The precondition of the `factorial` function requires that argument `n` be a nonnegative integer value. We could use Scala’s predefined `requires` method to ensure this precondition holds, but, in this version, if all pattern matches fail, then the function call aborts with a standard error message.

The postcondition of `factorial` is that the result returned is the correct mathematical value of `n` factorial. The function `factorial` neither accesses nor modifies any global data structures.

Nonlinear recursion

A *nonlinear recursion* is a recursive function in which the evaluation of some leg requires more than one recursive application. For example, the naive Fibonacci number function `fib` shown below has two recursive applications in its third leg. When we apply this function to a nonnegative integer argument greater than 1, we generate a pattern of recursive applications that has the “shape” of a binary tree. Some call this a *tree recursion*.

```
def fib(n: Int): Int = n match {
  case 0           => 0
  case 1           => 1
  case m if m >= 2 => fib(m-1) + fib(m-2) // double (tree) recursion
}
```

Termination: How do we know that `fib` terminates?

Complexity: Function `fib` is combinatorially explosive, having a time complexity O(`fib n`). The space complexity is O(`n`) because a new runtime stack frame is needed for each recursive call and the calls recurse to a depth of `n`.

An advantage of a linear recursion over a nonlinear one is that a linear recursion can be compiled into a *loop* in a straightforward manner. Converting a nonlinear recursion to a loop is, in general, difficult.

Backward and Forward Recursion

Backward recursion

A function definition is *backward recursive* if the recursive application is embedded within another expression. During execution, the program must complete the evaluation of the expression after the recursive call returns. Thus, the program must preserve sufficient information from the outer call's environment to complete the evaluation.

The definition for the function `factorial` above is backward recursive because the recursive application `factorial(n-1)` in the second leg is *embedded within the expression* `n * factorial(n-1)`. During execution, the multiplication must be done after return. The program must "remember" (at least) the value of parameter `n` for that call.

A compiler can translate a backward linear recursion into a loop, but the translation may require the use of a stack to store the program's *state* (i.e., the values of the variables and execution location) needed to complete the evaluation of the expression.

Forward recursion

A function definition is *forward recursive* if the recursive application is *not embedded within another expression*. That is, the *outermost expression is the recursive application* and any other subexpressions appear in the argument lists. During execution, significant work is done as the recursive calls are made (e.g., in the argument list of the recursive call).

The definition for the auxiliary function `factIter` within the `factorial2` definition below is forward recursive. The recursive application `factIter(n-1,n*r)` in the second leg is on the outside of the expression evaluated for return. The other legs are nonrecursive.

```
def factorial2(n: Int): Int = {

  def factIter(n1: Int, r: Int): Int = n1 match {
    case 0          => r
    case m if m > 0 => factIter(m-1,m*r)
  }

  factIter(n,1)
}
```

Termination: How do we know that `factIter` terminates?

Complexity: Function `factorial2` has a time complexity O(`n`). But, because, tail call optimization converts the `factIter` recursion to a loop, the time complexity's constant factor should be smaller than that of `factorial` and the space complexity of `factIter` is O(1).

Tail Recursion

A function definition is *tail recursive* if it is *both forward recursive and linear recursive.* In a tail recursion, the last action performed before the return is a recursive call.

The definition of the function `factIter` above is tail recursive because it is both forward recursive and linear recursive.

Tail recursive definitions are easy to compile into efficient loops. There is no need to save the states of unevaluated expressions for higher level calls; the result of a recursive call can be returned directly as the caller's result. This is sometimes called *tail call optimization* (or "tail call elimination" or "proper tail calls").

In converting the backward recursive function `factorial` to a tail recursive auxiliary function, we added the parameter `r` to `factIter`. This parameter is sometimes called an *accumulating parameter* (or just an *accumulator*).

We typically use an accumulating parameter to "accumulate" the result of the computation incrementally for return when the recursion terminates. In `factIter`, this "state" passed from one "iteration" to the next enables us to convert a backward recursive function to an "equivalent" tail recursive one.

Function `factIter` defines a more general function than `factorial`. It computes a factorial when we initialize the accumulator to 1, but it can compute some multiple of the factorial if we initialize the accumulator to another value. However, the application of `factIter` in `factorial2` gives the initial value of 1 needed for factorial.

Consider auxiliary function `fibIter` used by function `fib2` below. This function adds two "accumulating parameters" to the backward nonlinear recursive function `fib` to convert the nonlinear (tree) recursion into a tail recursion. This technique works for Fibonacci numbers, but the same technique will not work in all cases.

```
def fib2(n: Int): Int = {

  def fibIter(a: Int, b: Int, n1: Int): Int = n1 match {
    case 0 => b
    case m => fibIter(a+b,a,m-1)
  }

  if (n >= 0)
    fibIter(1,0,n)
  else
    sys.error("Fibonacci undefined for negative value " + n)
}
```

Termination: How do we know that `fibIter` terminates?

Complexity: Function `fib2` has a time complexity of O(`n`) in contrast to O(`fib(n)`) for `fib`. This algorithmic speedup results from the replacement of the very expensive operation `fib(n-1) + fib(n-2)` at each level in `fib` by the inexpensive operation `a + b` (i.e., addition of two numbers) in `fib2`. Because tail call optimization converts the `fibIter` recursion to a loop, the space complexity of `fib2` is O(1).

Logarithmic Recursive Algorithms

The backward recursive exponentiation function `expt1` below raises a number to a nonnegative integer power. It has time complexity O(`n`) and space complexity O(`n`).

```
def expt1(b: Double, n: Int): Double = n match {
  case 0          => 1
  case m if m > 0 => b * expt1(b,m-1)
  case _          =>
    sys.error("Cannot raise to a negative power " + n)
}
```

Termination: How do we know that `expt1` terminates?

Consider the tail recursive function `exptIter` called within function `expt2` below. This function has time complexity O(`n`) and space complexity O(1), assuming tail call optimization.

```
def expt2(b: Double, n: Int): Double = {

  def exptIter(b1: Double, n1: Int, p: Double): Double = n1 match {
    case 0 => p
    case m => exptIter(b1,m-1,b1*p)
  }

  if (n >= 0)
    exptIter(b,n,1)
  else
    sys.error("Cannot raise to negative power " + n )
}
```

Termination: How do we know that `exptIter` terminates?

The exponentiation function can be made computationally more efficient by squaring the intermediate values instead of iteratively multiplying. We observe that:

```
b^n = b^(n/2)^2   if n is even
b^n = b * b^(n-1) if n is odd
```

Function `expt3` below incorporates this observation in an improved algorithm. Its time complexity is O(`log(n)`) and space complexity is O(`log(n)`).

```
def expt3(b: Double, n: Int): Double = {

  def exptIter(b1: Double, n1: Int): Double = n1 match {
    case 0                 => 1
    case m if (m % 2 == 0) => // i.e. even
        val exp = exptIter(b1, m/2)
        exp * exp                   // backward recursion
    case m                 => // i.e. odd
        b1 * exptIter(b1,m-1)   // backward recursion
  }

  if (n >= 0)
    exptIter(b,n)
  else
    sys.error("Cannot raise to negative power " + n )
}
```

Termination: How do we know that `exptIter` terminates?