Computer Science I
Fruitful Functions 2

CSCI141
Lecture (2/2)

09/19/2011

1 Fruitful Functions

Recall the two mathematical functions that computer scientists like to use as examples: the factorial function and the Fibonacci function. The factorial of a natural number n is written as $n!$, the nth Fibonacci number is written F_n. The mathematical definitions are below.

$$\begin{array}{rcl} 0! & = & 1 \\ n! & = & n \times (n-1)! \end{array} \qquad \begin{array}{rcl} F_0 & = & 0 \\ F_1 & = & 1 \\ F_n & = & F_{n-1} + F_{n-2} \end{array}$$

When translating these definitions to code, we get the following code.

```
def fact(n):
    """fact: NatNum -> NatNum"""
    if n == 0:
        return 1
    else:
        return n * fact(n-1)

def fib(n):
    """fib: NatNum -> NatNum"""
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

2 Tail-Recursive Fruitful Functions and Iteration

2.1 Tail-Recursive Formulation

We can also characterize fruitful functions as being tail-recursive or not. The examples above are *not* tail-recursive. When calling `fact(3)`, there is work to do after the recursive call `fact(2)`: we need to multiply by three. When calling `fib(3)`, there is work to do after the recursive calls `fib(1)` and `fib(2)`: we need to add those results together.

It turns out, though, that there is a tail-recursive way of writing each of those functions. Doing so involves introducing a new accumulation parameter. Coming up with the accumulative formulation of a fruitful function is often tricky. The tail-recursive code is below.

```
def factAccum(n, a):
    """factAccum: NatNum * NatNum -> NatNum"""
    if n == 0:
        return a
    else:
        return factAccum(n-1, n*a)

def fact(n):
    """fact: NatNum -> NatNum"""
    return factAccum(n, 1)
```

Please note that the variable n was doing double duty. We used it to count down to 0 so that our recursive function knew when to stop, i.e., apply the base case. We also used it in our calculation of the factorial; we multiplied n by our answer-so-far a.

How did we determine this definition for `factAccum`? We know that we need to multiply from the definition of factorial; however, if we want a tail-recursive definition, we can't multiply on the outside, i.e., after the recursive call has returned. Thus we need one more parameter. Hence we are led to a generalization of factorial: `factAccum(`n`, `a`)` $= n! \times a$. From this definition, we can infer that `factAccum(0, `a`)` $= a$, and `factAccum(`n`, `a`)` $= n! \times a = n \times (n-1)! \times a = (n-1)! \times (n \times a) =$ `factAccum(`$n-1$`, `$n \times a$`)`. We were basically *accumulating* the product as we went along. In the original factorial function, all the multiplying happened at the end when the instances of the function were returning their values.

Please note that the variable n was doing double duty. We used it to count down to 0 so that our recursive function knew when to stop, i.e., apply the base case. We also used it in accumulating the answer; we multiplied n by our answer-so-far a. This is not the case with the fibonacci calculation. The accumulated value and the counter must be kept separate.

```
def fibAccum(n, a, b):
    """fibAccum: NatNum * NatNum * NatNum -> NatNum"""
    if n == 0:
        return a
    elif n == 1:
        return b
    else:
        return fibAccum(n-1, b, a+b)

def fib(n):
    """fib: NatNum -> NatNum"""
    return fibAccum(n, 0, 1)
```

How did we determine this definition for `fibAccum`? We know that we need to add from the definition of the Fibonacci function; however, if we want a tail-recursive definition, we can't add on the outside. Further, we suspect that since the previous two values are involved, that we need two more parameters. At this point, we will resort to intuition and note that the two additional parameters can be viewed as a window into the sequence. Unlike the factorial algorithm, as the counter `n` *decreases*, the window moves *forward* in the sequence.

2.2 Example: Substitution Trace of Tail-Recursive Factorial

$$\begin{aligned}\texttt{fact(3)} &= \texttt{factAccum(3, 1)}\\ &= \texttt{factAccum(2, 3)}\\ &= \texttt{factAccum(1, 6)}\\ &= \texttt{factAccum(0, 6)}\\ &= 6\end{aligned}$$

2.3 Example: Substitution Trace of Tail-Recursive Fibonacci

$$\begin{aligned}\texttt{fib(3)} &= \texttt{fibAccum(3, 0, 1)}\\ &= \texttt{fibAccum(2, 1, 1)}\\ &= \texttt{fibAccum(1, 1, 2)}\\ &= 2\end{aligned}$$

2.4 Iterative Formulation

Since we can understand tail-recursion as iteration, we can replace the recursive call with a loop. Note that `return` transfers control not only out of the loop but also out of the function. Also, because we now only have one copy of the variables since everything is done in a single call to the function, we must be somewhat careful about how we go about changing the values of those variables.

```
factAccum(n, a):
    """factAccum: NatNum * NatNum -> NatNum"""
    while True:
        if n == 0:
            return a
        else:
            a = n * a
            n = n - 1
```

```
def fact(n):
    """fact: NatNum -> NatNum"""
    return factAccum(n, 1)
```

Lets simplify things.

1. The accumulator `a` can be defined within the function, before the loop.
2. The test for the old base case can become the termination condition for the `while` loop.

```
def fact(n):
    """fact: NatNum -> NatNum"""
    a = 1
    while n > 0:
        a = n * a
        n = n - 1
    return a
```

For the Fibonacci function as well, we can replace the recursive call with a loop. Notice that we must be particularly careful about the way in which we change the values of the variables. Here we must introduce new variables when updating; changing the order is insufficient to ensure that the new values are correct.

```
def fib(n):
    """fib: NatNum -> NatNum"""
    a = 0
    b = 1
    if n == 0:
        return a
    else:
        while n > 1:
            n = n - 1
            newa = b # Assigning b to a here would mess up the next line.
            b = a+b
            a = newa
    return b
```

Let's do a trace of the execution of this loop-based algorithm in the form of a timeline. We will do it for `fib(6)` and we will write the values of the variables each time the while loop test is about to be executed.

n	a	b
6	0	1
5	1	1
4	1	2
3	2	3
2	3	5
1	5	8

... and 8 is returned.