The Art of Recursion: Problem Set 10
Due Tuesday, 20 November 2012

Tail recursion

A recursive function is tail recursive if every recursive call is in “tail position”—that is, the result of the recursive call is immediately the result of the whole function; there’s no extra work that needs to be done after the recursive call returns.

For example,

\[
\text{fact } 0 = 1 \\
\text{fact } n = n \times \text{fact } (n-1)
\]

is not tail recursive, because after the recursive call to \(\text{fact } (n-1)\) completes, \(\text{fact } n\) isn’t done: a multiplication by \(n\) still has to take place. We can make a tail-recursive variant of the factorial function by adding an extra “accumulating parameter” which accumulates the result as we descend through the recursive calls; when \(n\) reaches 0 we return the accumulated value.

\[
\text{factTR } r \ 0 = r \\
\text{factTR } r \ n = \text{factTR } (n \times r) \ (n-1)
\]

1. How can the original \(\text{fact}\) be defined in terms of \(\text{factTR}\)?

2. Write a tail-recursive version of \(\text{reverse}\), where

\[
\text{reverse } [] = [] \\
\text{reverse } (x:xs) = \text{reverse } xs \ ++ \ [x]
\]

Note that I don’t care whether you derive it in some “principled” way from the implementation of \(\text{reverse}\). I am just looking for \(\text{any}\) function which computes the same thing as \(\text{reverse}\) and is implemented using tail recursion.

3. Write a tail-recursive version of the usual Fibonacci function. (Again, it doesn’t matter how you come up with it; any tail-recursive Fibonacci function will do.)

The important point about tail recursion is that any tail-recursive function can easily be implemented iteratively. Since at every recursive call control simply passes to the recursive invocation and does not need to return, it can be implemented just by jumping to the top of a loop. And, as usual, in the iterative version we have one mutable variable corresponding to each argument to the tail-recursive function.
4. Turn the definitions of factTR and your tail-recursive Fibonacci function into iterative, imperative programs (using whatever language you like, or pseudocode).

**Continuation-passing style**

If we can convert any recursive function into an equivalent tail-recursive function, we can then compile any recursive function into iterative code. However, it is not clear whether this is always possible. The tail-recursive variants seen above have been rather ad-hoc. What we need are more principled methods for deriving tail-recursive variants of arbitrary recursive functions.

One such principled method is continuation passing style (or CPS for short). The idea is that instead of returning a result directly, we take an extra argument, called a continuation, which specifies “what to do next”: we pass the result to this function instead of returning it. For example, here is the usual factorial function again:

```haskell
fact 0 = 1
fact n = n * fact (n-1)
```

And here is factorial written using continuation-passing style:

```haskell
factCPS k 0 = k 1
factCPS k n = factCPS (k . (\r -> n*r)) (n-1)
```

In the case that \( n = 0 \), we know the answer should be 1, but instead of returning 1 directly, we pass it to the continuation \( k \). In the general case, when the argument to \( \text{fact} \) is \( n \), we call \( \text{fact} \) with \( n-1 \) and specify what needs to happen to the result: first, it needs to be multiplied by \( n \), and then that result should be passed to \( k \).

CPS can be tricky to wrap your head around at first; study the above example until you are sure you understand what it is doing!

5. How can we define the original fact in terms of factCPS?

6. Write a tail-recursive version of map using CPS, where

```haskell
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

7. Write a tail-recursive version of the naive Fibonacci function using CPS. (The point is that the naive version can “mechanically”\(^1\) be made tail-recursive using CPS, without having any special flashes of insight about efficiency, memoization, etc.)

\(^1\) Once you understand how it works…
Dissecting call trees

CPS is a general method for turning any recursive function into a tail-recursive variant. However, in order to take a tail-recursive function using CPS and “compile” it to iterative code, we have to figure out how to compile lambdas. It can be done, and in fact, many compilers for functional languages work this way.\(^2\) However, in some cases we can do something easier.\(^3\)

Consider expanding out a recursive function call into a complete call tree, defined as the tree with one node for each recursive invocation of the function, where the children of a given node are all the recursive calls made by that node. We label each node with the argument(s) to that recursive call, and, just to help keep track of what is going on, we also decorate internal nodes with whatever computation needs to happen with the recursive results, and leaves with their values.

For example, the figure to the right shows the call tree for fact 5 (where fact is the standard implementation of the factorial function), and the figure below shows the call tree for the standard naïve implementation of the Fibonacci function, also at the argument 5.

Observe that evaluating a recursive function is equivalent to doing a traversal of its call tree. So, to implement it iteratively (i.e. tail-recursively), we just need to keep track of enough information during the traversal so that we know where we are in the tree and what to do next at each step of the iteration.

As a simple first example, consider the factorial function fact. At each step of the traversal, there will be a “current” node, but we

\(^2\) If you want to know more, look up “closure conversion”. For example, see http://matt.might.net/articles/closure-conversion.

\(^3\) For sufficiently large values of “easy”.

---

\[5 \times 5\]
\[4 \times 4\]
\[3 \times 3\]
\[2 \times 2\]
\[1 \times 1\]
\[0 \times 0\]
need to remember whether we have just recursed down the tree to
reach the current node, or whether we have just returned up the tree
after completing a recursive subcall. If we’re going down, we need to
know what the input to the current node is; if we’re coming back up,
we need to know what the output from the child node was. We can
keep track of this information with the Direction type:

```haskell
data Direction i o where
  Down :: i -> Direction i o
  Up :: o -> Direction i o
```

In fact, Direction is not specific to fact; this same pattern applies
to any recursive call tree.

We may also need to keep track of some extra information about
the nodes we’ve visited; this part is specific to each recursive func-
tion. In the case of fact, we need to remember what number to
multiply by once we’re done with the recursive calls. We’ll use the
FactNode type for this, which consists of just a single constructor
storing an Integer.\(^4\)

```haskell
data FactNode where
  -- what to multiply by after finishing the recursion
  FN :: Integer -> FactNode
```

At any given point in the traversal we need to remember informa-
tion about all the nodes above us, so we will actually use a stack of
FactNodes (which we can represent in Haskell using a list).

Finally, fact’s input is the same type as its output; to better distin-
guish them (and to prevent confusion when you later generalize to
examples where the input and output types are not the same), let’s
create two different type synonyms:

```haskell
  type FactIn = Integer
  type FactOut = Integer
```

Now we’re ready to write the traversal itself. The factT function
takes a Direction (indicating whether we’re going down or up the
tree, and containing either an input or an output) and a stack of node
information, and ultimately produces the result of the entire fact
computation.

```haskell
  factT :: Direction FactIn FactOut -> [FactNode] -> FactOut
```

First, if we’re traveling down and see an input of 0, we’ve reached
the base case of the recursion (i.e. a leaf of the call tree), so we can
immediately “return” 1 and begin traveling back up the stack.

```haskell
  factT (Down 0) stack = factT (Up 1) stack
```

\(^4\) In this case we could just use Integer
directly instead of going to the trouble
of making a new FactNode type; the
point is to make it easier to see how to
generalize to more complex examples.
Otherwise, if we’re traveling down and see an input of \( n \), we continue traveling down with an input of \( n-1 \), but push another node onto the stack in order to remember that when we get back to this node we need to multiply by \( n \).

\[
\text{factT (Down } n \text{) stack} = \text{factT (Down (} n - 1 \text{)) (FN } n : \text{stack)}
\]

When traveling up the tree, we look at the top of the stack, multiply by the number we find there, and keep traveling up (popping the stack).

\[
\text{factT (Up } res \text{) (FN } n : \text{stack)} = \text{factT (Up (} n * res \text{)) stack}
\]

When we run out of nodes on the stack, that means we’re back at the top of the tree, i.e. we’re done with the traversal, so we return the final result.

\[
\text{factT (Up } res \text{) []} = res
\]

And that’s it. Of course this version is quite a bit more complicated than the original! But as you can easily verify, factT is completely tail-recursive; and unlike the CPS version, it is completely first-order, that is, it does not require passing any functions as arguments. It is therefore quite straightforward to translate factT into a loop in an imperative language: we just need a variable for the direction (in practice we might use a boolean flag to indicate the direction, another variable to store the input, and a third to store the output) and another variable storing the stack of integers.

Let’s do one more example, this time the naïve fib function. This is a bit more complex than fact because we have something more interesting to remember at each node. In particular, since each non-leaf node has two children (i.e. makes two recursive calls), we have to remember which of the two subtrees we are currently traversing. Furthermore, when we are working on the left subtree, we have to also remember what the input to the recursive call at the root of the right subtree will be; when working on the right subtree, we have to also remember what the output from the left subtree was. The FibNode type encapsulates this idea: FibL indicates we are currently traversing the left subtree, and stores the input for the right; FibR means we are currently traversing the right subtree, and stores the output from the left.

\[
\text{type FibIn} = \text{Integer} \quad \text{type FibOut} = \text{Integer}
\]

\[
\text{data FibNode where} \quad \text{FibL :: FibIn} \rightarrow \text{FibNode} \quad \text{FibR :: FibOut} \rightarrow \text{FibNode}
\]

\(^5\) The CPS version and this version are actually related: in a sense we’ve simply recorded the different possible types of continuations as a data structure.
And now for the traversal. First, when we reach a leaf node (that is, we see a 0 or 1 input when traveling down), we turn around and go back up the tree with the answer.

```haskell
fibT :: Direction FibIn FibOut -> [FibNode] -> Integer
fibT (Down 0) stack = fibT (Up 0) stack
fibT (Down 1) stack = fibT (Up 1) stack
```

In general, when traveling down and we see input \( n \), we go down the left subtree with input \( n-1 \); we remember that we went down the left subtree, and that the input to the right subtree will be \( n-2 \), by pushing \( \text{FibL} \ (n-2) \) on the stack.

```haskell
fibT (Down n) stack = fibT (Down (n-1)) (FibL (n-2) : stack)
```

When traveling up, if we see that we have just completed traversing a left subtree, we turn around and go down the right subtree, using the stored input. We replace the \( \text{FibL} \) node with a \( \text{FibR} \) node to mark the fact that we’re working on the right subtree now, and to store the result \( \text{res} \) from the left subtree.

```haskell
fibT (Up res) (FibL n2 : stack) = fibT (Down n2) (FibR res : stack)
```

If we’re coming up and see that we just completed a right subtree, then we add the stored result from the left subtree with the current result and continue up.

```haskell
fibT (Up res) (FibR res1 : stack) = fibT (Up (res1 + res)) stack
```

Finally, when there are no nodes left on the stack, we’re done.

```haskell
fibT (Up res) [] = res
```

8. How can we define the original \( \text{fib} \) in terms of \( \text{fibT} \)?

9. Recall the \( h \) function, defined by

```haskell
h :: Integer -> Integer
h 0 = 1
h n | odd n = h (n ‘div’ 2)
    | even n = h (n ‘div’ 2) + h ((n ‘div’ 2) - 1)
```

Implement a tail-recursive version of \( h \).

10. Consider the usual type of binary trees with data at internal nodes, and the standard fold over such trees. Implement a tail-recursive version of \( \text{foldBTree} \).

```haskell
data BTree a = Empty | Branch (BTree a) a (BTree a)

foldBTree :: c -> (c -> a -> c -> c) -> BTree a -> c
foldBTree e _ Empty = e
foldBTree e b (Branch l a r) = b (foldBTree e b l) a (foldBTree e b r)
```