Termination Analysis

As we have seen, programming systems which allow arbitrary recursion (such as the untyped $\lambda$-calculus, or Haskell) do not make for a good foundation for mathematics and proofs, because infinite recursion makes everything "provable". On the other hand, a language in which it is only possible to create terminating programs would be quite restrictive. So it becomes important to be able to analyze a given recursive function whether it will terminate (though, of course, it's not possible to decide this in general). Proof-oriented languages like Coq and Agda actually do such checks for every function you write, and will refuse to accept any function that can't be proved terminating.

Consider the following type of (good old-fashioned) binary (search) trees, along with a fold operation:

```haskell
data BST a where
  Empty :: BST a
  Node   :: BST a -> a -> BST a -> BST a

bstFold :: r -> (r -> a -> r -> r) -> BST a -> r
bstFold e _ Empty = e
bstFold e n (Node l a r) = n (bstFold e n l) a (bstFold e n r)
```

As we've seen, Haskell does in fact allow infinite values, such as

```haskell
weirdTree = Node weirdTree 1 weirdTree
```

From now on, however, unless explicitly stated, you may assume that all values of type `BST a` are finite. (In future weeks we'll explore the distinction in more detail, and what to do when we really do want to reason about infinite values.)

1. Prove that for all input trees $t$, $bstFold\ e\ f\ t$ will always terminate as long as $f$ terminates for all inputs. Give an informal argument why this sort of termination result will always be true for any fold.

2. Write a function `insert :: Int -> BST Int -> BST Int` which inserts a new value into a binary search tree.\(^1\)

   Recall that a binary search tree is one with the property that the value at every node is greater than all the values in its left subtree and less than all those in its right subtree. If you wish, and you understand what it means, you can give `insert` the more general type `Ord a => a -> BST a -> BST a`.

3. Explain why it would be difficult to implement `insert` using `bstFold`.\(^2\)

\(^1\) Of course, it is possible—anything is possible with folds—and you might like to try as an optional challenge.
4. Even though your `insert` is not implemented using `bstFold`, it should still be *structurally recursive.* Prove that `insert` terminates for all (finite) inputs.

Consider the *hailstone function* on natural numbers given by

\[
    f(n) = \begin{cases} 
        n/2 & \text{if } n \text{ is even} \\ 
        3n + 1 & \text{if } n \text{ is odd} 
    \end{cases}
\]

Of course, it is obvious that \( f \) terminates for all natural number inputs. That isn’t the point. Consider the following function which *iterates* \( f \) until it reaches 1:

\[
    h(0) = 1 \\
    h(1) = 1 \\
    h(n) = h(f(n))
\]

It is clear that \( h \) will always evaluate to 1 for any input...that is, *if* it terminates!

5. Is \( h \) structurally recursive? Why or why not? Explain informally why it might be difficult to prove that \( h \) terminates. The point of this problem is just to show how unobvious and tricky proving termination can be, even for “simple” functions.

6. Implement `mergesort`.

7. Is your implementation structurally recursive?

In order to prove termination for a recursive function \( f \) which is not structurally recursive, a *termination measure* is needed, which is a function \( m \) with the properties

- \( m \) assigns a “measure” to any arguments given to \( f \). For now we will take a “measure” to be a natural number.
- The measure of the arguments in each recursive call to \( f \) is smaller than the original measure.

Since the measure is always getting smaller, and the measure is a natural number, the recursion must eventually stop.

Consider the function

\[
    f \ x \ y \ \mid \ x <= 1 \quad = \ y \\
    \ \mid \ y <= 1 \quad = \ x \\
    \ \mid \ otherwise \quad = \ f \ (x-2) \ (y+1) \ + \ f \ (x+1) \ (y-2)
\]

It is not immediately obvious whether \( f \) always terminates—the arguments in the recursive calls sometimes get bigger and sometimes...
smaller. \( f \) is certainly not structurally recursive. However, as the termination measure for \( f \)'s arguments \( x \) and \( y \) we may take \( x + y \). The idea is that even though \( x \) and \( y \) individually may go up, their sum always goes down in every recursive call. In particular, both recursive calls have the sum \( x + y - 1 \), which is less than the original sum \( x + y \). Thus \( f \) terminates for all inputs.

8. Prove that your implementation of \texttt{mergesort} terminates for all (finite) input lists. What termination measure should you use?

9. Consider again the Ackermann function, defined by

\[
\begin{align*}
A(0,y) &= y+1 \\
A(x,0) &= A(x-1,1) \\
A(x,y) &= A(x-1,A(x,y-1))
\end{align*}
\]

The Ackermann function does indeed terminate for all natural number inputs, but there’s no way to give a natural number measure which proves it. Instead we need to use something more general.

In fact, as a measure we can use any well-founded relation. A well-founded relation is a set \( S \) along with some “ordering” relation \(<\) on \( S \) with the property that there are no infinite descending chains, that is, there are no sequences of \( s_0 > s_1 > s_2 > \ldots \) which go on forever. Of course, the usual \(<\) relation on natural numbers is well-founded, because there are no infinite sequences of natural numbers \( n_0 > n_1 > n_2 > \ldots \); every such sequence must eventually hit 0 after a finite number of steps.

If we always have \( m_1 > m_2 \) whenever the arguments to a function have measure \( m_1 \) and the arguments to a recursive call have measure \( m_2 \), then because there are no infinite descending chains the function must terminate.

Define some well-founded relation \((x_1, y_1) < (x_2, y_2)\) on pairs of natural numbers, and use it to prove that the Ackermann function terminates for all inputs. Be sure to prove that your relation is well-founded.

10. [\textbf{optional}] Explain what this code does:

\[
\begin{align*}
babbage \ [x] &= \text{repeat } x \\
babbage \ x@(x0:xs) &= \text{scanl } (+) \ x0 \ (\text{babbage } (\text{zipWith } (-) \ xs \ x))
\end{align*}
\]