Lazy evaluation

Most programming languages use \textit{strict evaluation}: when evaluating a function call, the arguments to the function are completely evaluated \textit{before} calling the function. In a \textit{lazy} language such as Haskell, function arguments are \textit{not} evaluated before being passed to a function. The only time anything is evaluated is when it is \textit{pattern-matched}, and even then it is evaluated only just enough to see whether it matches the pattern.

For now, it’s not too important to go into the details of how this works (though I’m happy to discuss it if you’re interested). Just keep in mind the intuition that expressions are only evaluated when—and not before—they are actually needed.

Laziness in conjunction with recursion has some very interesting consequences; in this assignment we’ll explore a few of them.

Infinite data structures

Up until now we have been pretending that all data types are finite (so we can use induction), but in a lazy language like Haskell this is not actually true.\footnote{To see why induction breaks down in the presence of infinite data structures, consider this “proof” by induction that all Haskell lists have a finite length: the empty list has length 0 which is certainly finite; and if we assume \texttt{l} has finite length, then \texttt{(x:l)} also has a finite length for any \texttt{x}. This is a valid proof by induction, but in fact all we have proved is that all \textit{finite} lists in Haskell have a finite length (duh), because induction only applies to finite things. Proving things about potentially infinite data structures requires something called \textit{coinduction}, but this semester is too narrow to contain it.}

Recall the Calkin-Wilf tree, the (infinite!) binary tree with 1/1 at its root node, and where a node containing \(a/b\) always has \(a/(a+b)\) as its left child and \((a+b)/b\) as its right child.

1. Create a polymorphic binary tree type \texttt{BTree a}. Then define \texttt{cw :: BTree Rational} as the infinite Calkin-Wilf tree.\footnote{To get \texttt{Rational}, import the \texttt{Data.Ratio} module. To construct a \texttt{Rational}, use \texttt{(\%)}. You may also find the \texttt{numerator} and \texttt{denominator} functions useful.}

2. Write a function \texttt{flatten :: BTree a \rightarrow [a]} which yields a list of the elements of a binary tree in top-bottom, left-right order. Thus \texttt{flatten cw} should be an infinite list containing every rational number exactly once at a finite index.\footnote{Warning! Don’t print out this list directly. Try using \texttt{take}.}

Dynamic programming for free

Recall once again the problem of computing the number of tilings of a \(1 \times 100\) rectangle using tiles of length 1, 2, 3, and 4. As we have seen, in many languages an efficient solution can be obtained by storing the values of \(t(n)\) in an array.

Using mutable arrays in Haskell is purposely rather difficult (although it is possible). However, there is a better way: it turns out that
mutable arrays are not really necessary at all! The idea is to use a recursive, immutable array: we simply initialize it with its final values, constructing the values by recursively referring to the array itself! For example, here is how we might define an array containing the factorials of all the numbers from 0 to 100, using the array function, which takes an index range and a list of associations between indices and values:

```haskell
-- an array indexed by Integers, storing Integers
factsArray :: Array Integer Integer
factsArray =
  array
    -- index lower and upper bounds
    (0,100)
    -- list of (index, value) pairs
    [(0,1) : [ (k, k * factsArray!(k-1)) | k <- [1..100]]]
```

There are a few things to note about the above definition:

- You are probably used to recursive functions, but `factsArray` is a recursive value. It’s only possible to have true recursive values in a lazy language.
- The order of the list of (index, value) pairs doesn’t matter! Because of Haskell’s laziness, entries in the array are only evaluated when needed, and the runtime system figures out the correct order to evaluate them, just by following references. For example, if you ask for the 100th element of the above array, it sees that it needs the value of the 99th, which requires the value of the 98th, … and so on, all the way down to the 0th element, which is already known to be 1. Then it will go back through the array computing the elements as it goes.

So this is better than the traditional mutable-array solution in two ways: one, we can avoid mutable arrays (which are more difficult to reason about). Even better, we don’t have to worry about filling in the array entries in the correct order! We just list the entries and their definitions. For factorial this is not that big of a deal. But for more complicated dynamic programming tasks, figuring out the correct order can be nontrivial.

In fact, “dynamic programming” hardly deserves a special name in Haskell: it’s just computing a recursive function with lazy memoization.

Note that you can create a “two-dimensional” array by simply using pairs as indices. For example,

\[
\text{array } \begin{pmatrix} (0,0), (9,9) \end{pmatrix} \rightarrow \begin{pmatrix} ((i,j), i+j) | i \gets [0..9], j \gets [0..9] \end{pmatrix}
\]

creates a $10 \times 10$ array where the $(i,j)$ entry contains $i+j$.

**Tying the knot**

One might think that it is impossible to create cyclic lists, graphs, or other sorts of pointer-based data structures in Haskell. In fact, it *is* possible: because of purity (no mutation), values can always be *shared* (i.e. everything is passed by reference instead of being copied). Combined with laziness and recursion, this means that cyclic structures can be created in memory by sharing values in their own definition.

For example, consider the definition

\[c = 1 : 2 : c\]

This represents the infinite list $1 : 2 : 1 : 2 : \ldots$, but in memory it actually consists of a cyclic list, with two $(:)$ constructors that each point to the other.

Consider the following definitions.

\[
data \text{ BTree } a = \text{Empty} \\
\quad \mid \text{Branch } (\text{BTree } a) \ a \ (\text{BTree } a)
\]

\[
type \text{ Parent } a = \text{Maybe } (\text{PBTree } a)
\]

\[
data \text{ PBTree } a = \text{PEmpty } (\text{Parent } a) \\
\quad \mid \text{PBranch } (\text{PBTree } a) \ a \ (\text{PBTree } a) \ (\text{Parent } a)
\]

**BTree** represents a usual binary tree type. **PBTree** $a$ represents binary trees where each node has a “pointer” to its parent (note that **Maybe** is used since the root node has no parent). Of course, it looks like each node actually has an entire copy of its parent tree stored—the point is that using some tricks we can arrange things so that this really is represented just by a pointer back to the parent instead of by a separate copy.

4. Implement a function

\[
\text{linkParents :: BTree } a \rightarrow \text{PBTree } a
\]

which turns a normal **BTree** into a **PBTree** with back links from each node to its parent. Such a **PBTree** could be used to efficiently traverse a static tree both downwards and upwards, or to maintain a “current pointer” into the middle of a tree giving fast access to the current node and the nodes right around it.

You can check what structure is produced in memory using the [ghc-vis](http://hackage.haskell.org/package/ghc-vis) package.