DM552 exercises

Department of Mathematics and Computer Science
University of Southern Denmark

October 4, 2017

1. Identify the redexes in the following expressions, and determine whether each redex is innermost, outermost, neither, or both:

   $1 + (2 * 3)$
   $(1 + 2) * (2 + 3)$
   $\mathit{fst}\ (1 + 2, 2 + 3)$
   $(\lambda x \to 1 + x)\ (2 * 3)$

2. Show why outermost evaluation is preferable to innermost for the purposes of evaluating the expression $\mathit{fst}\ (1 + 2, 2 + 3)$.

3. You are given

   $\mathit{cube}\ x = x * x * x$

   Reduce the expression

   $\mathit{cube}\ (\mathit{cube}\ 3)$

   to normal form, both by using

   - leftmost innermost reduction sequence (corresponds to eager evaluation)
   - leftmost outermost reduction sequence
   - leftmost outermost reduction sequence with sharing (corresponds to Haskell's evaluation strategy)

4. Show the evaluation steps Haskell performs when evaluating:

   $\mathit{map}\ (2*)\ (\mathit{map}\ (1+)\ [1, 2, 3])$

5. The built-in function $seq :: a \to b \to b$ "forces" a computation, in the sense that it converts an expression to Weak Head Normal Form (see the Lecture 5 slides for the definition of WHNF).

   Using the GHCi command `:sprint`, one can inspect which parts of a definition has been computed, and what is still an unevaluated "thunk" (represented by _). The following is a demonstration of the *map* function applied to a list of 10 elements - we see that only the outermost constructor (_ : _) is revealed when forcing the computation of *map*, and each element of the list are not computed if they are not used.

```
Prelude> let xs = map (+1) [1..10] :: [Int]
Prelude> :sprint xs
xs = _
Prelude> seq xs ()
()
Prelude> :sprint xs
xs = _ : _
Prelude> length xs
10
Prelude> :sprint xs
xs = [_,_,_,_,_,_,_,_,_,_]
Prelude>
```

   Implement a function $mapStrict :: (a \to b) \to [a] \to [b]$ which is equivalent to the original map-function, except that it will completely evaluate the result when forcing the computation of *mapStrict*. GHCi should output:

```
*Main> let xs = mapStrict (+1) [1..10] :: [Int]
*Main> :sprint xs
xs = _
*Main> seq xs ()
()
*Main> :sprint xs
xs = [2,3,4,5,6,7,8,9,10,11]
```

   You can use *seq* or the helper function for strict function application

$$(\$!) :: (a \to b) \to a \to b$$
$$f \ \$! \ x = x \text{ `}seq\text{` } f \ x$$

6. Using Haskells function *iterate* solve the following exercises:

   - give f such that $\mathit{iterate}\ f\ 0$ equals all natural numbers from 0 and up: $[0, 1, 2...]$
   - give f such that $\mathit{iterate}\ f\ 0$ equals all even numbers from 0 and up: $[0, 2, 4, 6...]$
   - give f such that $\mathit{iterate}\ f\ 1$ equals all two-powers from 1 and up: $[1, 2, 4, 8, ...]$
   - give f such that $\mathit{iterate}\ f\ (0, 1)$ equals $(n, \mathit{factorial}\ n)$ from $n = 0$ and up: $[(0, 1), (1, 1), (2, 2), (3, 6)...]$
   - give f such that $\mathit{iterate}\ f\ (0, 1)$ equals all consecutive pairs of Fibonacci numbers: $[(0, 1), (1, 1), (1, 2), (2, 3), (3, 5)...]$

7. You are given the following definition of a infinite list containing infinite lists:

$$\begin{aligned} &\mathit{pairs} :: [[(\mathit{Int}, \mathit{Int})]] \\ &\mathit{pairs} = [[(x, y) \mid y \leftarrow [1\,..]] \mid x \leftarrow [1\,..]] \end{aligned}$$

   (a) Write a function

$$\mathit{taketake} :: \mathit{Int} \to [[a]] \to [[a]]$$

   which on the call $\mathit{taketake}\ n$ returns the first n elements of the first n lists. Example: $\mathit{taketake}\ 2\ \mathit{pairs} = [[(1, 1), (1, 2)], [(2, 1), (2, 2)]]$

   (b) Write a function

$$\mathit{diags} :: [[a]] \to [[a]]$$

   which returns the diagonals of the input list.
   Example:

$$\begin{aligned} &\mathit{diags}\ \mathit{pairs} = [\\ &\quad [(1, 1)], \\ &\quad [(1, 2), (2, 1)], \\ &\quad [(1, 3), (2, 2), (3, 1)], \\ &\quad [(1, 4), (2, 3), (3, 2), (4, 1)]] \\ &\quad ... \\ &] \end{aligned}$$

8. An algebraic data type describing full binary trees (trees where all nodes have equally sized left and right subtrees) is

$$\textbf{data}\ \mathit{FTree}\ a = \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathit{FTree}\ (a, a))\ \textbf{deriving}\ \mathit{Show}$$

- Write explicit constructions of trees of 0 levels, 1 level and 2 levels.
- Similar to the function *take* for lists, define the function

  $$\mathit{levels} :: \mathit{Int} \to \mathit{FTree}\ a \to \mathit{FTree}\ a$$

  which returns the first n levels from the tree.
- Define a function $\mathit{split} :: \mathit{FTree}\ (a, a) \to (\mathit{FTree}\ a, \mathit{FTree}\ a)$
- Use *split* to define $\mathit{left}, \mathit{right} :: \mathit{FTree}\ a \to \mathit{Maybe}\ (\mathit{FTree}\ a)$
- Define a function $\mathit{join} :: (\mathit{FTree}\ a, \mathit{FTree}\ a) \to \mathit{FTree}\ (a, a)$
- Define a function $\mathit{gentree} :: \mathit{Integer} \to \mathit{FTree}\ \mathit{Integer}$ which returns the infinite tree

  $$\mathit{Cons}\ 1\ (\mathit{Cons}\ (2, 3)\ (\mathit{Cons}\ ((4, 5), (6, 7))...))$$

  Hint: Consider using *join*.