COMP90048 Declarative Programming
Semester 1, 2018
Peter J. Stuckey

Declarative Programming

Answers to workshop exercises set 11.

QUESTION 1
Write a function fibs :: Int -> [Integer] which returns a list containing the first n numbers in the Fibonacci sequence: [0,1,1,2,3,5,8,...], where
the third and subsequent numbers are the sum of the two preceeding numbers
(0+1=1, 1+1=2, 1+2=3, 2+3=5, etc). We use Integer rather than Int because the numbers grow exponentially and therefore overflow native Ints quite quickly. Is the algorithmic complexity of your solution acceptable?

ANSWER
Here is one possible solution. It uses a helper function which generates a list of Fibonacci numbers given the previous two Fibonacci numbers and
the required list length. This makes it efficient; its complexity is O(n).
Naive codings can do *lots* of repeated computation and can have exponential
complexity.

```
>fibs :: Int -> [Integer]
>fibs 0 = []
>fibs 1 = [0]
>fibs n | n > 1 = 0:1:fibs1 0 1 (n-2)

>fibs1 fpp fp 0 = []
>fibs1 fpp fp n = (fpp+fp) : fibs1 fp (fpp+fp) (n-1)
```

QUESTION 2
If we do pairwise addition of the elements of the Fibonacci sequence and its tail, we get the tail of the tail of the sequence:

```
  0 1 1 2 3 5 8 ... fibs
+ 1 1 2 3 5 8 ...   tail fibs
= 1 2 3 5 8 ...     tail (tail fibs)
```

Use this property to write a definition of allfibs :: [Integer] which is
the (infinite) Fibonacci sequence (Hint: the zipWith Prelude function is useful). Define fibs in terms of allfibs. How efficient is this definition
of fibs compared to your previous one?

ANSWER

```
>allfibs :: [Integer]
>allfibs = 0 : 1 : (zipWith (+) allfibs (tail allfibs))
>fibs' n = take n allfibs
```

The two fibs definitions are likely to have quite similar efficiency: both are O(n), though the second will have a higher constant factor.

One subtlety (not covered in lectures) is that allfibs is what is known as a "constant applicative form", or CAF. These are constants defined at the top level of the program (not inside a let or where clause). Haskell guarantees that CAFs are never evaluated more than once.
Thus the first time we need a list of Fibonacci numbers, allfibs will compute what is needed and the result will be saved.
If we need another shorter list of Fibonacci numbers later, no further (re)computation of allfibs will be required. If we need a longer list, the lazy evaluation will continue from where it stopped previously.
This can all save time. However, it may use more space. If allfibs was defined inside a let or where clause (which could be e.g. in the definition
of fibs'), it would be recomputed in each different call to fibs', and the
space it used would be reclaimed after each time.

QUESTION 3
Consider the bottom-up merge sort implementation from workshop 2.

```
>mergesort xs = repeat_merge_all (merge_consec (to_single_els xs))
>
>to_single_els [] = []
>to_single_els (x:xs) = [x] : to_single_els xs
>
>merge [] ys = ys
>merge (x:xs) [] = x:xs
>merge (x:xs) (y:ys)
```

```
>   | x <= y = x : merge xs (y:ys)
>   | x >  y = y : merge (x:xs) ys
>
>merge_consec [] = []
>merge_consec [xs] = [xs]
>merge_consec (xs1:xs2:xss) = (merge xs1 xs2) : merge_consec xss
>
>repeat_merge_all [] = []
>repeat_merge_all [xs] = xs
>repeat_merge_all xss@(_:_:_) = repeat_merge_all (merge_consec xss)
```

With list xs of length n, what is the maximum additional space that is needed at any one time, assuming strict evaluation, for evaluating merge_consec (to_single_els xs)? What if lazy evaluation is used instead?

What is the maximum additional space is needed at any one time, assuming strict evaluation, for evaluating mergesort xs? Can we do significantly better than this?

ANSWER
Strict evaluation of merge_consec (to_single_els xs) proceeds as follows (we make n a power of two for simplicity):

```
    [4,1,6,2,8,7,3,5]           xs
->  [[4],[1],[6],[2],[8],[7],[3],[5]]   to_single_els
->  [[1,4],[2,6],[7,8],[3,5]]       merge_consec
```

The result of to_single_els has n additional cons cells (2n total), and this is completely constructed before merge_consec is evaluated.

With lazy evaluation, the merge_consec and to_single_els evaluations are interleaved: [1,4] is constructed as soon as [4] and [1] have been constructed (and the cons cells for [4] and [1] can be reclaimed). The maximum extra space needed is n/2 cons cells (plus a constant because the suspensions/thunks take a bit of space; the strict version probably has a smaller constant overhead).

For mergesort the evaluation proceeds on from the merge_consec result as
follows:

```
->  [[1,2,4,6],[3,5,7,8]]
->  [[1,2,3,4,5,6,7,8]]
```

-> [1,2,3,4,5,6,7,8]

The maximum space is needed when the result of to_single_els is computed. Determining what happens with Haskell is very tricky. However, there are
versions of bottom-up merge sort which require only O(log n) extra space. Coding them in a language which uses strict evaluation is quite a bit of work
but this complexity bound can be achieved with relatively simple code if non-strict evaluation is used.

QUESTION 4
Consider an interpreter for a language which produces a pair containing the result of the computation plus some debugging information, which is a string containing information about all assignment statements and function calls. Compare the efficiency of the following:

a) Execution of the interpreter using strict evaluation and printing the debugging string.
b) Execution of the interpreter using lazy evaluation and printing the debugging string.
c) Execution of the interpreter using strict evaluation but not printing the debugging string.
d) Execution of the interpreter using lazy evaluation but not printing the debugging string.
e) Execution of a similar interpreter which doesn't produce the debugging
string at all.

ANSWER

a) The debugging string will be entirely computed then printed. This will take time (calls to show, for example, and ++ for concatenating sub-strings) and, much more importantly, space. The space used by the computation will be at least as large as the debugging string; for a long running computaion this may be huge.

b) Printing the debugging string will be interleaved with interpreting the program (assuming a reasonable structure for the code). The time spent
computing the string will be comparable to a) but the space overhead for
debugging is likely to be constant (at least bounded by the size of a single

line of debug output instead of the entire debug output).

c) This will be as bad as a), but the extra time and space will be wasted.

d) This will be similar to b) but without the execution overhead of computing the string and with somewhat less space overhead (there may be
some suspensions generated, but these can probably be garbage collected if the program has reasonable structure).

e) This will be similar to d) but without the extra space overhead and with some constant factor speed increase due to not having to create suspensions, pass around the string, etc.

QUESTION 5
Here are some students' answers to one of the questions on a sample mid-semester test, which were posted on the LMS (thanks to the authors). The question asked for a Haskell function to print out Mtrees
with indentation showing the structure. Compare and contrast these solutions. Can you come up with something better than all three?

```
>data Mtree a = Mnode a [Mtree a]

>print_mtree :: Show a => Mtree a -> IO()
>print_mtree tree = indent_mtree 0 tree
>    where
>        indent_mtree :: Show a => Int -> Mtree a -> IO()
>        indent_mtree i (Mnode val children) = do
>            putStrLn $ (replicate i ' ') ++ (show val)
>            foldl (>>) (return ()) (map (indent_mtree (i+1)) children)

>type Line = String
>
>print_mtree' :: Show a => Mtree a -> IO ()
>print_mtree' t =
>    let
>        toLines :: Show a => Mtree a -> [Line]
>          toLines (Mnode val cs) = show val : map (' ':) (concatMap
(toLines) cs)
>    in  foldl (\acc str -> acc >> (putStrLn str)) (return ()) (toLines
t)
>
>-- A clearer version
```

```
>print_mtree2 :: Show a => Mtree a ->  IO ()
>print_mtree2 t =
>    let
>        toLines :: Show a => Mtree a -> [IO ()]
>        toLines (Mnode val cs) =
>            print val : map (putChar ' ' >>) (concatMap (toLines) cs)
>    in  foldl1 (>>) (toLines t)
```