Lab Sheet 11

Lazy Evaluation

How to succeed with the labs and exercises?

Labs and exercise sheets are published every week on the course homepage at StudIP. As described in the first lecture, each successfully completed lab and exercise earns you bonus points towards your final score in this semester's exam. Keep in mind that you only get bonus points if you would pass the exam without them. **Cheating does not help you - but we will!**

How to complete a lab successfully?

In the lab, you will solve the lab sheet with fellow student(s). To find out who your teammates are, please look at the group pdf in StudIP's download area. **During the session, one participant of every group will be selected, who must then explain one of their task solutions to a lab assistant**.

How to complete an exercise successfully?

In order to complete an exercise sheet successfully, you must upload your answers using DOMjudge **before the deadline** printed on the exercise sheet. We will not consider any solutions handed in after the deadline! Furthermore, you must solve and hand in the exercises **individually** and your Haskell code **must compile** and **pass certain amounts of tests** as specified. During the exercise session, we develop possible solutions together. Please participate! We encourage you to ask and answer questions from fellow students.

Technically, Haskell files you submit using DOMjudge must have the format as specified in the task sheets(usually ".hs", ".lhs", or ".txt"). Furthermore, DOMjudge will only consider your last submission. Therefore, if you first submit successfully (your code compiles and tests are passed) and afterwards unsuccessfully (your code does not compile or certain tests fail again), your last submission counts, and - if it does not compile - will therefore be ignored. Make sure your last submission was successful!

How to get additional information?

We encourage you to discuss past and present exercise sheets with us. Either approach us during the exercise session, or visit us during the weekly office hours. We are also available via e-mail or on the StudIP forum. We try to reply as quick as possible and in general, you should get a reply the next weekday, but we cannot guarantee this.

This lab is about lazy evaluation, which roughly means an expression is only evaluated if needed. The module `Debug.Trace`, with its function `trace :: String -> a -> a`[1], can be used to make evaluation visible. Note that for reasons of caching, if the evaluation of a particular `trace` was triggered once, **GHCi must be restarted** before it can be triggered again. In WinGHCi, this can also be done by clicking `Actions -> Clear Modules` and then opening your `.hs`/`.lhs` file again. `Note:` The show function forces the evaluation of its argument. Therefore, whenever GHCi displays a result, this result is evaluated fully.

Exercise 1 Consider the operator `&.&` defined below. This operator forces the evaluation of its first argument. Assume you know that the first argument is usually very expensive to compute. Reimplement `&.&` to be lazy in its first argument.

```
(&.&) :: Bool -> Bool -> Bool
True &.& True = True
_ &.& _ = False
```

Exercise 2 Read through the definitions of `sumTwo` and `xs` defined below. Think about which arguments of `xs` would be evaluated in the expression `sumTwo xs`. How can you test this with `trace`?

```
xs :: [Int]
xs = zipWith (+) [0..] $ repeat (-10)

sumTwo :: (Num n, Ord n) => [n] -> n
sumTwo (a:_:c:_:ds)
    | a + c < 0 = a + c + sumTwo ds
    | otherwise = 0
```

Exercise 3 Download the file "mean.hs" from StudIP and place it into a fresh directory. In there, you find the inefficient `mean` function that takes a list of Integer and computes their mean value. Your task is to implement the function `meanOpt` that should do the same as `mean`, but use less memory. **Hint:** Use the `$!` operator or seq to force evaluation. **Hint:** You can either implement your function recursively as in `mean`, or use foldl' from `Data.List`.

a) Open a shell (`Start -> Run -> cmd`) and navigate to your fresh directory containing "mean.hs"

b) To compile your program, execute "`ghc -fforce-recomp -prof -rtsopts --make mean`".

c) Execute "`mean 2500000 slow +RTS -K1000M -s`". The Haskell runtime system will now print a report about the execution. Specifically, look at how much memory the program needed.

d) Now implement your optimized mean function, recompile and execute "`mean 2500000 fast +RTS -K1000M -s`". This will use your mean function instead of the original one. Compare the results. Can you get below 10MB of total memory usage?

[1] http://hackage.haskell.org/packages/archive/base/latest/doc/html/Debug-Trace.html

Exercise 4 * Recap Redefine the Prelude function reverse using either foldl or foldr

Exercise 5 * Recap Are the following functions polymorphic and/or overloaded?

```
f1 :: Int -> Int
f2 :: Eq a => a -> a -> Bool
f3 :: (a -> b -> a) -> a -> [b] -> a
f4 :: (Integral i, Eq e) => e -> [e] -> [i]
```