Lab Sheet 11

Lazy Evaluation

STS Software Technology Systems

Functional Programming - Winter 2022/2023 - January 18, 2023 - Schupp/Lübke

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 tasks on lab sheet. You are encouraged to talk to your neighbors and find solutions together, as well to ask the tutor for help. **Towards the end of the session, the tutor will briefly discuss your solutions with you. To pass the lab, you should complete two thirds of the tasks (rounding half up)**.

How to complete an exercise successfully?

In order to complete an exercise sheet successfully, you must upload your answers using INGInious **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 INGInious must have the format as specified in the task sheets (usually ".hs", ".lhs", or ".txt"). Furthermore, INGInious 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.

The *Functional Programming* exam is carried out in electronic form, which is why this lab is a bit different from the ones you're used to. Today's lab uses the same electronic exam system as the e-exam, so you can get acquainted with the system. If you have any feedback (positive or negative), please let us know (as always)!

Please click this link[1] and open it in a Chromium-based[2] web browser. Follow the on-screen instructions and solve the tasks. As always, your tutors are available for help.

In case you'd like to redo or finish the tasks at home, the same tasks which are in the online system are also printed here.

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`[3], 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. `Note:` The `show` function forces the evaluation of its argument. Therefore, whenever GHCi displays a result, this result is evaluated fully.

Task 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
```

Task 2 Read through the definitions of `sumTwo` and `xs` defined below. Think about which elements 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
```

Task 3 Take the (enclosed) files *mean.hs* and *mean.cabal* and place them 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`.

[1] https://yaps.zll.tuhh.de/fp/lab11/examserver/client/instance/
[2] E.g., Google Chrome, Chromium, Microsoft Edge, etc.
[3] http://hackage.haskell.org/packages/archive/base/latest/doc/html/Debug-Trace.html

a) Open a terminal and navigate to your fresh directory containing *mean.hs* and *mean.cabal*

b) To compile your program, execute `cabal build`. Note the compiler flags used in the compilation: `-fprof-auto "-with-rtsopts=-K1000M -s"`:

- `-fprof-auto`: Compile for const-centre profiling with automagic const-centres on all top-level functions
- `"-with-rtsopts=-K1000M -s"`: Add the following runtime system options :
  - `-K1000M`: Set the maximum stack size
  - `-s`: Produce runtime system statistics

c) Execute `cabal run mean -- 2500000 slow`. The Haskell runtime system will now print a report about the execution. Specifically, look at how much memory the program needed.
Note: Older versions of `cabal` (as installed in the Linux pools) may not have the `run` command. In that case you need to execute to program "manually". You can either

- use the command `find . -name "mean"` to locate the program, copy its path and execute it like `./path/to/mean 2500000 slow` or
- also build manually with `ghc -fprof-auto "-with-rtsopts=-K1000M -s" mean.hs` which should place the executable in the current directory, so you can execute it via `./mean 2500000 slow`.

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

In the following, you are supposed to revisit topics that you had problems with in the past, or that interest you. Therefore, the following tasks are just a recommendation, not something you are required to work on. You may also choose old exercise or lab tasks, or come up with a task of your own.

Task 4 * recap: higher-order functions

a) Reimplement the following `foo` function once using the function application operator (`$`), and once using composition (`.`). `foo x y = bar x (baz y)`

b) Redefine the Prelude function reverse using either foldl or foldr

Task 5 * recap: user-defined types As part of its standard library, Haskell provides a data structure for sets, appropriately called `Set`. Its type constructor takes one variable that denotes the type of the included elements. Therefore, `Set Char` is a set of characters.

The following is the Haskell definition for a set structure, represented by a binary tree:

```
data Set a = Leaf -- Empty set
        | Node {
         left :: (Set a) -- Set with elements smaller than x
        ,elem :: a -- One element (x)
        ,right :: (Set a) -- Set with elements larger than x
        } deriving (Ord, Eq, Show)
```

The associated set library exports the following functions:

Function	Purpose
insert	Takes an element x and a set xs and returns a set that contains all elements of xs and x.
delete	Takes an element x and a set xs and returns a set that contains all elements of xs except x.
`member`	Takes an element x and a set xs and returns True exactly when x is a member of xs.

a) Provide the most general type signatures for the functions insert and `member`.

b) Provide a definition for the function `member`.

* This is an optional task, but we strongly advise you to give it a try!