Lab Sheet 9

Recapitulation

Functional Programming - Winter 2018/2019 - December 19, 2018 - Schupp/Lehmann

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.

In this lab, 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. Discuss what you want to do with your lab partners and start working.

0.1 Types and Type Classes

Exercise 1 What are the most general types of the following expressions? Answer using Haskell type notation.

a) `['f', 'p']`

b) `([23,42], ())`

c) `([])`

d) `[foldr (:) [] "foldr"]`

Haskell type notation means the way Haskell accepts types in contracts. Therefore, the type of the expression `[True]` is denoted as `[Bool]`, not as "A list of Booleans."

0.2 Recursion

Exercise 2 Why does the expression `isSubsetOf xs ys` (for arbitrary finite lists of integers `xs` and `ys`) terminate?
Refer to the recursive definition of the `helper` function in your answer.

```
import Data.List
isSubsetOf :: Ord a => [a] -> [a] -> Bool
xs `isSubsetOf` ys = sort (nub xs) `helper` sort (nub ys)
    where
    [] `helper` _ = True
    _ `helper` [] = False
    (x:xs) `helper` (y:ys) | x == y = xs `isSubsetOf` ys
                           | x > y = (x:xs) `isSubsetOf` ys
                           | otherwise = False
```

Exercise 3 * foldl is a flexible higher-order function that can be used to implement many operations on lists. However, it will always iterate its input list completely before producing a result. This is suboptimal if the result of the fold only depends on the first n elements of the input list.

Implement the function `foldEither` that behaves similarly to foldl, but has a different type:

```
    foldEither :: (a -> b -> Either a a) -> a -> [b] -> a
```

If the result of the parameterizing function is Right `acc`, then `foldEither` proceeds as usual. If, however, the result is Left `acc`, then `acc` is returned and no recursive call is made.

Therefore, foldl can be re-implement in the following way:

```
    foldl f = foldEither (\acc x -> Right $ f acc x)
```

0.3 Higher-Order Functions

Exercise 4 Reimplement the following `foo` function once using the function application operator (`$`), and once using composition (`.`).

```
foo x y = bar x (baz y)
```

Exercise 5 When using what is called points-free style, which usually means that one creates functions from the composition of other functions, one needs to think from the solutions instead from the input data. The reason for this is that one usually writes from left to right, and `foo . bar . baz` means first apply `baz`, then `bar`, and then `foo`. Therefore, the order of application is inverse to the order in writing.

Implement an operator `($$)` such that `a $$ b $$ c` is equal to `c . b . a`.

0.4 User-Defined Types

Exercise 6 Define a type in Haskell to represent JSON values as described below:

There are six ways to construct a JSON value:

	JSON kind	JSON value description
1.	JSEmpty	An empty value.
2.	JSString	Value representing a string.
3.	JSNumber	Value representing a double precision floating point.
4.	JSBool	A Value representing either true or false.
5.	JSArray	A sequence of JSON values.
6.	JSObject	A collection of tuples that contain a string and a JSON value.

Exercise 7 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`.

0.5 Recursive Types

Exercise 8 *

a) Re-implement the list type and call it `MList`. This list must be parameterized by exactly one type variable.
Impelement functions that convert from Prelude lists to your lists and back.
Test your implementations using generalized test cases.

b) Additionally implement versions of length, map, foldl, foldr, and filter for your `MLists`. Also provide generalized tests for these implementations.

Exercise 9 * Recall the definition of natural numbers from the lecture.

a) Extend this definition so that it can also represent negative numbers. Provide a specification that ensures that each integral number has only one representation in your implementation.

b) Define the functions `intToMInt`, `mIntToInt`, add, and `mult` for your type. `intToMInt` and `mIntToInt` convert between integrals and your type and add and `mult` implement addition and multiplication respectively. Additionally, provide generalized test cases for your implementations.