Lab Sheet 10

Proofs

Functional Programming - Winter 2022/2023 - January 11, 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.

Task 1 Rewrite the following function[3] so that no overlapping patterns are used anymore.
"*Patterns that do not rely on the order in which they are matched are called non-overlapping.*" – Graham Hutton in *Programming in Haskell*[4].

```
ackermann :: Integer -> Integer -> Integer
ackermann m n | m < 0 || n < 0 = error "only␣positive␣arguments␣allowed"
ackermann 0 n = n + 1
ackermann m 0 = ackermann (m - 1) 1
ackermann m n = ackermann (m - 1) (ackermann m (n - 1))
```

Task 2 Recall the definition of natural numbers, the `add` function, and the `mult` function:

```
data Nat = Zero | Succ Nat deriving (Show, Eq)

add :: Nat -> Nat -> Nat
add Zero m = m
add (Succ n) m = Succ (add n m)

mult :: Nat -> Nat -> Nat
mult Zero _ = Zero
mult (Succ n) m = add (mult n m) m
```

Your task is to use either *equational reasoning* or *induction* to prove that `Succ Zero` is a neutral element to `mult`.

a) Prove that for any natural number `x` of type `Nat`, `mult (Succ Zero) x` is equal to `x`. In other words, you should prove the following: $\forall x :$ `mult (Succ Zero) x` $\equiv$ `x`.

b) Prove: $\forall x :$ `mult x (Succ Zero)` $\equiv$ `x`.

Task 3 recap: types and type classes What are the most general types of the following expressions? Answer using Haskell type notation.

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

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

[1]https://yaps.zll.tuhh.de/fp/lab10/examserver/client/instance/
[2]E.g., Google Chrome, Chromium, Microsoft Edge, etc.
[3]http://en.wikipedia.org/wiki/Ackermann_function
[4]https://katalog.tub.tuhh.de/Record/86551464X

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.”

Task 4 recap: recursion 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
```