Lab Sheet 6

Higher-Order Functions

STS Software Technology Systems

Functional Programming - Winter 2022/2023 - November 30, 2022 - 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.

For the remainder of this laboratory, whenever you have a type `Foldable t =>t a` you may assume it is equivalent to the type `[a]`, provided that `a` is the same type variable. Therefore, if a function has type `Foldable t =>(a -> Bool) -> t a -> Bool` then you can treat it as if it had the type `(a -> Bool) -> [a] -> Bool`.

Task 1 Provide the type of the higher-order function `foo`. Describe which parts of the implementation allow you to determine the number of arguments to `foo` and the type of each argument. What would be a better name for the function?
Hint: Use `:type foo` to double-check that you arrived at the correct result.

```
-- foo :: TODO
foo _ [] = False
foo p (x : xs) = p x || foo p xs
```

Task 2 The following expressions do not type check. Use the resulting type errors to explain how the Haskell type checker can determine this.

a) `any (++"e") "Test1"`

b) `map (all (/='e')) "Test2"`

Task 3 Express `[x * x | x <- [1..], even x]` using map and filter.

Task 4 Describe the result of the following functions:

a) `foo xs = map (+1) (map (+1) xs)`

b) `bar xs = sum (map (\_ -> 1) (filter (> 7) (filter (< 13) xs)))`

c) `baz xss = map (map (+1)) xss`

Task 5 Use the definition of foldl and foldr from the lecture notes and perform stepwise calculation of the following expressions.

a) `foldl (-) 0 [1, 2, 3]`

b) `foldr (-) 0 [1, 2, 3]`

Hint: You can use GHCi at each step to control your results.

Task 6 * For long expressions like the `bar` function from task 4, the number of parentheses can make the expression difficult to read. In such cases, you can use the function application operator (`$`). The operator is defined in the following way:

```
($) :: (a -> b) -> a -> b
f $ x = f x
infixr 0 $
```

The operator might not seem useful at first, since it only does function application, but allows you to replace an expression like `f (g (z x))` with `f $ g $ z x` which can be a lot easier on the eyes. Use `$` to simplify the functions `foo`, `bar`, and `baz` from task 4.

Task 7 * Implement map by using only the functions foldr, `(.)`, *and* `(:)`.

```
map f xs = foldr ? ? xs
```

Below are the types of the involved functions, where the names have been unified so that the type `a` represents the same type in all contracts. The types `c` and `d` can be expressed in terms of `a` and `b`. Can you figure out how?

```
map :: (a -> b) -> [a] -> [b]
foldr :: (a -> c -> c) -> c -> [a] -> c
(:) :: b -> [b] -> [b]
(.) :: (b -> d) -> (a -> b) -> a -> d
```

Hint: The result of `map f [x0, x1, ..., xN]` looks like this:

```
[f x0, f x1, ..., f xN] == ( (f x0) : ( (f x1) : ( ... : (f xN) ) ) )
```

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