Prof. Dr. Peter Thiemann
Winter Term 2024/25

Hannes Saffrich
saffrich@informatik.uni-freiburg.de

Functional Programming

https://proglang.informatik.uni-freiburg.de/teaching/functional-programming/2024/

Exercise Sheet 4

Exercise 1 (Type Classes)

In this exercise, we keep our tradition of reimplementing functionality from the standard library. In particular, we are going to look at functions and instances related to the `Semigroup`, `Monoid`, and `Foldable` type classes:

- A *semigroup* is a set A with a binary operation $\otimes$, which is associative, i.e.

  $$\forall x, y, z \in A.\ (x \otimes y) \otimes z = x \otimes (y \otimes z)$$

  In Haskell, we can capture the concept of a semigroup by the following type class:

  ```
  class Semigroup' a where
    (<+>) :: a -> a -> a
  ```

  Note, that the law of associativity is not enforced by the type system, but instead the programmer needs to make sure to only create `instance`s of `Semigroup`, which satisfy the law of associativity, for this abstraction to make sense.

- A *monoid* is a semigroup $(A, \otimes)$ with a neutral element ϵ, i.e.

  $$\forall x \in A.\ x + \epsilon = x = \epsilon + x$$

  In Haskell, we can capture the concept of a monoid by the following type class:

  ```
  class Semigroup' a => Monoid' a where
    mempty' :: a
  ```

Your tasks are:

1. Find suitable binary operations and neutral elements to make `Semigroup'` and `Monoid'` instances for the following types and implement them:

   a) `Bool`;

   b) `Int`;

   c) `[a]` for all types `a`;

   d) `Maybe a` for all types `a`;

   e) `(a, b)` for all types `a` and `b`, where `a` and `b` also have semigroup and monoid instances;

   f) `a -> b` for all types `a` and `b`, where `b` also has a semigroup and monoid instance.

   For some of the above types multiple binary operations are suitable for defining law-abiding `Semigroup'` and `Monoid'` instances. Just pick one for now – we will revisit this problem later in this exercise.

2. Write a function `mconcat'`, which for a given monoid `m` takes a list of type `[m]` and combines all elements using the monoid operations.

   Example: `mconcat [1, 5, 2]` could evaluate either to `8` or `10` depending on which monoid instance for `Int` you used in exercise 1.1.

3. In the lecture, we looked at the `foldr` function for lists:

   ```
   foldr :: (a -> b -> b) -> b -> [a] -> b
   ```

   The `Foldable'` typeclass generalizes this concept to other datatypes and is defined as follows:

   ```
   class Foldable' f where
     foldr' :: (a -> b -> b) -> b -> f a -> b
   ```

   Note, that the `f` in this definition does *not* represent a type, but a function from types to types, i.e. if `a` is a type, then `f a` is a type. This is the same as you cannot have a value of type `Maybe`, but you can have a value of type `Maybe Int`. For this reason, `Maybe` is also called a *higher-kinded type*.

   Write `Foldable'` instances for the following higher-kinded types:

   a) `[]`;

   b) `Maybe`;

   c) `BTree` (one of the binary tree data types from the lecture).

   Note, that the list type `[a]` is actually syntactic sugar for `[] a`, so you can use `[]` to talk about the higher-kinded list type, e.g.

   ```
   ex1 :: [] Int     -- this is the same as `[Int]`
   ex1 = [1,2,3]
   ```

4. By utilizing `Foldable'` and `Monoid'` together, we can generalize `mconcat` to work for arbitrary containers, which contain elements of a monoid:

   ```
   fold' :: (Foldable' f, Monoid' a) => f a -> a
   ```

   Implement this function.

   Example: `fold' [1, 5, 2]` could evaluate either to `8` or `10` depending on which monoid instance for `Int` you used in exercise 1.1.

5. In Exercises 1.1 we mentioned that some types have multiple binary operations which make them a monoid.

   The problem however is, that type classes allow at most one instance for each type, because the type checker needs to select the instance based on the type. For example, in

   ```
   example :: Int
   example = mconcat [True, False, True]
   ```

   the typechecker knows that the argument to `mconcat` has type `[Bool]`. If there would be multiple instances of `Monoid Bool`, then it wouldn't be clear, which instance to use, as all the instances are equally viable.

   The usual workaround for this problem is as follows: if we want to define multiple instances for a type, then for each instance we create a "wrapper type" for `Bool`, and then we define each instance for its corresponding wrapper type:

```
data Bool1 = Bool1 { unBool1 :: Bool }
data Bool2 = Bool2 { unBool2 :: Bool }

instance Monoid Bool1 where ...
instance Monoid Bool2 where ...
```

This allows us to select, which instance we want to use for a `Bool` value, by wrapping it with the corresponding wrapper type for that instance, e.g.

```
x1 = unBool1 $ mconcat $ map Bool1 [ True, False, True ]
x2 = unBool2 $ mconcat $ map Bool2 [ True, False, True ]
```

Your task is to use this approach to define two `Semigroup'` and `Monoid'` instances for `Bool` and `Int`.

Note: There is a more ergonomic way of using these wrapper types, which is implemented in the `newtype` library on hackage. However, this approach relies on more advanced concepts, which we didn't cover yet.

Exercise 2 (IO)

1. Write the following definitions without `do`-notation:

   a)
   ```
   echoLine :: IO ()
   echoLine = do
     l <- getLine
     putStrLn l
   ```

   b)
   ```
   greet :: IO ()
   greet = do
     putStrLn "What's your name"
     l <- getLine
     putStrLn ("Hello, " ++ l ++ "!")
   ```

   c)
   ```
   greetFormal :: IO ()
   greetFormal = do
     putStrLn "What's your first name?"
     firstName <- getLine
     putStrLn "What's your surname?"
     surName <- getLine
     let greeting = "Hello, " ++ firstName ++ " " ++ surName ++ "!"
     putStrLn greeting
   ```

   d)
   ```
   choices :: IO ()
   choices = do
     putStrLn "Do you want to go outside?"
     l <- getLine
     if l == "yes" then do
       putStrLn "It rains!"
       putStrLn "Too, bad!"
     else
       putStrLn "It still rains! Not so bad!"
   ```

2. Write an IO function

```
choose :: String -> [String] -> IO String
```

which takes a question and a list of choices and asks the user to choose an option.

Example:

```
>>> choose "Do you want to quit?" ["yes", "no"]
Do you want to quit? [yes|no]
asfasfasf
Invalid input. Try again!

Do you want to quit? [yes|no]
yes
```

Hint: You can use the `getLine` and `putStrLn` functions from the standard library for terminal I/O.

3. Write an IO action

```
todo :: IO ()
```

which runs a small todo-list simulator.

It should support the functionality shown in the example output below:

```
>>> todo

What do you want to do? [add|display|remove|quit]
add
Enter text of todo item:
Feed the fishes

What do you want to do? [add|display|remove|quit]
add
Enter text of todo item:
Sleep

What do you want to do? [add|display|remove|quit]
add
Enter text of todo item:
Sleep some more...

What do you want to do? [add|display|remove|quit]
display

0) Sleep some more...
1) Sleep
2) Feed the fishes

What do you want to do? [add|display|remove|quit]
remove
Index of item to remove:
1
Removed item: Sleep

What do you want to do? [add|display|remove|quit]
display
```

```
0) Sleep some more...
1) Feed the fishes

What do you want to do? [add|display|remove|quit]
quiiet
Invalid input. Try again!

What do you want to do? [add|display|remove|quit]
quit
```

Hint: You might want to define `todo` in terms of another function `run :: [String] -> IO ()`, which takes the list of todo-items as an argument, and then define `todo = run []`.