EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

Functional Programming
Wintersemester 2021/2022
Prof. Dr. T. Grust, D. Hirn

Assignment 6 (03.12.2021)

Handin until: Friday, 10.12.2021, 00:00

> **Die Vorlesungsevaluation im WS 2021/2022 kommt – Bitte gebt uns eure Rückmeldung. Danke!**
>
> Bitte nehmt euch ein paar Minuten Zeit und meldet euch zurück. Jedes Bit an Rückmeldung zählt – das gilt insbesondere für die Freitextkommentare. Die gewonnenen Einsichten sind Gold wert für uns, um einzuschätzen, ob wir die *Functional Programming* in die richtige Richtung steuern und wie wir die Veranstaltung in den verbleibenden Wochen des Semesters noch besser machen können.
> Die Feedbackformulare der Lehrevaluation können **ab dem 6. Dezember 2021** ausgefüllt werden. Nochmals: Von uns schon jetzt ganz herzlichen Dank dafür!

Exercise 1: Fold **[7 Points]**

Formulate the following functions without using explicit recursion. Instead, make use of the prelude function

`foldr :: (a -> b -> b) -> b -> [a] -> b`[1]

Applying the function (`foldr f z xs`), we can right-associatively fold a list (`xs :: [a]`) using a binary operator (`f :: (a -> b -> b)`). `foldr` starts with initial value `z` and then proceeds to apply `f` while walking `xs` right to left:

```
foldr f z [x1, x2, ..., xn] ≡ x1 `f` (x2 `f` ... (xn `f` z)...)
```

Take for example a function `sum'` which sums all elements of a list. It can be written using `foldr`:

```
sum' :: [Integer] -> Integer
sum' xs = foldr (+) 0 xs
```

Applied to a list `[4,2,6]` the sum is evaluated as follows:

```
sum' [4,2,6] ≡ foldr (+) 0 [4,2,6] ≡ 4 + (2 + (6 + 0)) ≡ 12
```

Note: Obviously you must not use specialized prelude or module functions (e.g. `length`, `intercalate`, etc.) to solve the following problems.

1. `length' :: [a] -> Integer` – to determine the size of a list.

   Hint: You can use `let` or `where` to locally define the operator to fold with.

2. `reverse' :: [a] -> [a]` – to reverse a list.

3. `commaSep :: [String] -> String` – to concatenate a list of strings, separated by commas (`','`).

   Example: `commaSep ["Hello","World"]` `"Hello,World"`

4. Rewrite the following function, using `foldr` instead of explicit recursion. The function removes duplicate elements from a sorted list:

```
removeDups :: [Integer] -> [Integer]
removeDups []                  = []
removeDups [x]                 = [x]
removeDups (x:y:ys)  | x == y    = removeDups (y:ys)
                     | otherwise = x:removeDups (y:ys)
```

[1] Actually the prelude function `foldr` has a more general type. However, applied to lists its type can be described in this concrete version.

Exercise 2: Implementation of Typeclasses **[13 Points]**

Let us look at an example. Lines 3-7 of Figure 1a define a simplified version of type class `Eq` and an instance of `Eq` for type `Int`. Further, function `member` is defined which traverses a given list `[a]` and checks whether the list contains a specific element. The type signature of `member` is read "`member` has type `[a] -> a -> Bool`, for every type `a` such that `a` is an instance of class `Eq`." Without the constraint `Eq a`, we would not be allowed to compare `x` and `y` in line 11.

```haskell
1  import GHC.Int (eqInt)
2
3  class Eq a where
4    (==) :: a -> a -> Bool
5
6  instance Eq Int where
7    (==) = eqInt
8
9  member :: Eq a => [a] -> a -> Bool
10 member [] y     = False
11 member (x:xs) y = x == y
12                 || member xs y
```

(a) Simplified version of type class `Eq`.

```haskell
1  import GHC.Int (eqInt)
2
3  data EqD a = EqDict (a -> a -> Bool)
4  eq (EqDict e) = e
5
6  eqDInt :: EqD Int
7  eqDInt = EqDict eqInt
8
9  member :: EqD a -> [a] -> a -> Bool
10 member eqDa [] y     = False
11 member eqDa (x:xs) y = eq eqDa x y
12                      || member eqDa xs y
```

(b) Equivalent program without type class and instance declarations.

The Haskell compiler translates any program containing type class and instance declarations into an equivalent program that does not. Figure 1b shows the translation of the declarations in Figure 1a. Here is how it works[2]:

- A new data type is defined for each type class declaration. This data type is the so called *"method dictionary"* for that class. Each field corresponds to a method of the type class. Additionally, functions to access the fields of the dictionary are defined.

  Example: For class `Eq` the new data type `EqD` is defined. Values of this type can be created using the constructor `EqDict`, which has one entry for method `(==)`. Accessor function `eq` uses pattern matching to extract the field for method `(==)`.

- Each instance of a type class is translated into a declaration of a value of the method dictionary.

  Example: For `instance Eq Int`, dictionary `eqDInt :: EqD Int` is defined.

- Finally, functions with type class constraints like `(Eq a, Ord b, ...) => ...` are transformed into functions without. Each constraint turns into an additional parameter: `EqD a -> OrdD b -> ....` All invocations of methods are replaced by invocations of the corresponding entry in the method dictionary.

  Example: The type signature of `member` changes to `member :: EqD a -> [a] -> a -> Bool`. Invocations of `x == y` are replaced by the corresponding expression `eq eqD x y`, where `eqD` is an appropriate dictionary for type `a`.

File `typeclass.hs` contains type class declarations, instances, and function definitions. Translate the program to an equivalent one without type classes by following the method described above.

1. Define dictionaries and accessor functions for classes `Comparable` and `Printable`.
2. Translate instances `Comparable Integer`, `Printable Weekday`, and `Comparable Weekday` into values of the corresponding dictionary.
3. Translate functions `table`, and `qsort`.

[2] The type class approach to ad-hoc polymorphism has been proposed by Philip Wadler and Stephen Blott: https://dbworld.informatik.uni-tuebingen.de/staticfiles/teaching/ws2122/FP/wadler-typeclasses.pdf