Functional Programming
Summer Term 2025
Prof. Torsten Grust, Björn Bamberg
WSI — Database Systems Research Group

Assignment 6

Hand in this assignment until **Tuesday, June 24, 12pm** at the latest.

> 🤔 Running out of ideas?
> Are you hitting a roadblock? Are some of the exercises unclear? Do you just need that one hint to get the ball rolling? Refer to the `#forum` 🌐 channel on our Discord server—maybe you'll find just the help you need.

> Ⓔxam-style Exercises
> Exercises marked with Ⓔ are similar in style to those you will find in the exam. You can use these to hone your expectations and gauge your skills.

Task 1: Fold

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 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.

A Ⓔ `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.

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

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

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

D Ⓔ 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 specialized to this concrete version.

Task 2: Implementation of Typeclasses

The Haskell compiler translates any program containing type class and instance declarations into an equivalent program that does not. Figure b shows the translation of the declarations in Figure a. 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 linebreak `x == y` are replaced by the corresponding expression `eq eqD x y`, where `eqD` is an appropriate dictionary for type `a`.

Let us look at an example. Lines 3-7 of (a) 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.

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.

A Define dictionaries and accessor functions for classes `Comparable` and `Printable`.

B Translate instances `Comparable Integer`, `Printable Weekday`, and `Comparable Weekday` into values of the corresponding dictionary.

C Translate functions `table`, and `qsort`.

[2]The type class approach to ad-hoc polymorphism has been proposed by Philip Wadler and Stephen Blott: https://db.cs.uni-tuebingen.de/teaching/ss23/FP/wadler-typeclasses.pdf