Marius Furter
Institut für Mathematik
Universität Zürich

FS 2025

Logic and Foundation with Haskell

Exercise sheet 5

In this sheet, we will be defining our own datatype for sets. The type is declared as

```
data Set a = Set [a] deriving Show
```

This essentially defines `Set` as a wrapper around lists. However, we will be defining functions on them with behavior that differs from that of lists. There are two new features compared to our definition of `CoolBool` in sheet 3. First, the data constructor `Set` on the right hand side of the equality has a field that is filled with the type `[a]`. Second, the `Set` type is parametric in `a`, which is why `a` occurs on the left side of the equality. This will be explained in more detail in the lecture. If you are curious, you can learn more here.

Exercise 1. Test the `Set` type by doing the following:

(i) You can create the set $\{a, b, c\}$ by `Set ['a','b','c']`. What is the type of the resulting set?

(ii) What happens if you include duplicate elements?

(iii) What happens if you try to create the set $\{a, 1\}$? Can you explain the problem?

It is possible to pattern match our new type using `(Set xs)` to extract the underlying list `xs`. For example, the following function calculates the number of elements in a set:

```
card :: Set a -> Int
card (Set xs) = length xs
```

Exercise 2. Write conversion functions

```
unSet :: Set a -> [a]
toSet :: (Eq a) => [a] -> Set a
```

that convert a set to a list and conversely. For the second, you can use the function `nub` which removes duplicates from a list. It can be imported by adding `import Data.List ( nub )` to the start of your file. From now on we can use `toSet` to avoid producing sets with duplicates.

Exercise 3. Write functions

```
inSet :: (Eq a) => a -> Set a -> Bool
subSet :: (Eq a) => Set a -> Set a -> Bool
```

where the first checks if and element lies in a given set, and the second checks for inclusion.

Exercise 4. Implement set theoretic operations:

```
unionSet :: (Eq a) => Set a -> Set a -> Set a
intersectSet :: (Eq a) => Set a -> Set a -> Set a
```