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 9

In this exercise we are going to look at Generalized Algebraic Data Types (GADTs).

GADTs are not available in standard Haskell, and need to be enabled by putting the following pragma at the top of the file:

```
{-# LANGUAGE GADTs #-}
```

Exercise 1 (Safe Lists)

The type of Haskell lists does not tell us about whether lists are empty or non-empty. Consequentally, the only ways we can write a function that gives us the first element of a list are

```
head :: [a] -> a
head []      = error "Calling head on an empty list is not allowed"
head (x : _) = x

head' :: [a] -> Maybe a
head' []      = Nothing
head' (x : _) = Just x
```

The `head` function is not always safe to use: the type system allows us to call it on an empty list, which will crash the program.

The `head'` function is safe to use, but requires us to perform extra checks, even when we know that the list is non-empty, e.g.

```
xs = [1, 2, 3]
case head' xs of
  Just x  -> ...  -- Do something with x
  Nothing -> ...  -- IMPOSSIBLE, but we still need to write it...
```

1. Use GADTs to define your own list datatype `SafeList`, which supports a `safeHead` function that only type checks if the list it is called on is non-empty, e.g.

   ```
   >>> safeHead Nil
   <TYPE ERROR>
   >>> safeHead (Cons True Nil)
   True
   ```

2. Implement a `safeLast` function, i.e. a safe version of `last`, which only type checks if the list is non-empty, analogously to `safeHead`.

3. Implement a `safeAppend` function, i.e. a safe version of `(++)`. The `safeAppend` function should work on two `SafeLists` (non-empty or empty) and produce another `SafeList` as a result.

   Hint: To correctly describe whether the resulting list is empty or not, you need the *type families* extension, which allows defining functions on the type level, e.g.

```
{-# LANGUAGE TypeFamilies #-}

type family F a b where
  F Int  Int = Bool
  F Char x   = x

exFam1 :: F Int Int
exFam1 = True

exFam2 :: F Char Int
exFam2 = 1

exFam3 :: F Char String
exFam3 = "foo"
```

Exercise 2 (Stack Machine)

Stack machines are low-level languages, where a program is described by a list of instructions, which operate on a stack of values.

Stack machines are sometimes used as a simpler and cross-platform alternative to assembly code. Famous examples of such languages are web assembly and Java bytecode.

In this exercise, we will look at an interpreter for a simple stack machine language, and then use GADTs to improve it.

To illustrate how stack machines work, we start with an informal example program:

```
-- Stack: []
push 1
-- Stack: [1]
push 2
-- Stack: [2, 1]
push 3
-- Stack: [3, 2, 1]
add
-- Stack: [5, 1]
sub
-- Stack: [4]
```

The comments (lines starting with `--`) display the stack during evaluation. At the beginning, the stack is empty (`[]`). The `push` instruction, pushes a value onto the stack, so after executing the three `push` instructions, the stack is `[3, 2, 1]`. The `add` instruction, replaces the top two values of the stack (`3` and `2`) with their sum (`5`). If the stack would contain less than two values or if the top two values would be of non-integer type, then the program would crash. The `sub` instruction, replaces the top two values of the stack (`5` and `1`) with their subtraction (`4`). The output of the program are the values of the stack after running the last instruction, i.e. `[4]` for this particular example program.

In the supplementary code you can find an implementation of a simple stack language supporting operations on boolean and integer values.

The evaluation function takes an instruction and a stack and returns either and error or an updated stack:

```
eval :: Instr -> Stack -> Either Error Stack
```

Errors can occur if not enough elements are on the stack, e.g. when running the `add` instruction on an empty stack, or if the types of the values on the stack do not match the expected types of the instruction, e.g. when running the `add` instruction on a stack that has two boolean values on the top.

Your task is to use GADTs to rewrite the data types such that only those instructions type check, which do not cause an error when evaluated.

For this purpose you need to add two type parameters to the `Instr` type, which represent the types of the values on the stack before and after running the instruction, respectively. Similarly you also need to define your own datatype to represent stacks and adjust the datatypes for unary and binary operators.

After adjusting the data types, write an evaluation function that cannot fail:

```
eval :: Instr s1 s2 -> Stack s1 -> Stack s2
```