Universität Siegen
Lehrstuhl Theoretische Informatik
Carl Philipp Reh

Übungsblatt 3

Aufgabe 1. Betrachten Sie folgende `data`-Deklaration:

```
data Error e a = Fail [e] | Ok a
```

Im Vergleich zu `Maybe` benutzen wir hier `Fail [e]` statt `Nothing`. Die Idee ist, dass `[e]` eine Liste von aufgetretenen Fehlern enthält.

(a) Implementieren Sie die Funktion

```
stringToInt :: String -> Error String Int
```

Diese soll im Fall, dass ein `String` nicht in einen `Int` konvertieren werden kann, eine geeignete Fehlermeldung als `String` liefern. Zum Konvertieren eines `String` in einen `Int` können Sie die Funktion

```
readMaybe :: String -> Maybe Int
```

aus dem Modul `Text.Read` benutzen. Schreiben Sie hierzu

```
import Text.Read(readMaybe)
```

an den Anfang Ihrer Datei.

(b) Implementieren Sie `Functor` für `Error e`.

(c) Implementieren Sie `Applicative` für `Error e`. Hierbei sollen alle aufgetretenen Fehler akkumuliert werden. D.h., es soll gelten, dass

```
(Fail x) <*> (Fail y) = Fail (x ++ y)
```

Aufgabe 2. Bei `Applicative` hat man die Wahl, `<*>` oder `liftA2` zu implementieren.

(a) Implementieren Sie `<*>` über `liftA2`, was wir `apply` nennen:

```
apply :: Applicative f =>
         f (a -> b) -> f a -> f b
```

Hinweis: Sie müssen vorher `liftA2` importieren:

```
import Control.Applicative(liftA2)
```

(b) Implementieren Sie `liftA2` über `<*>`, was wir `liftA2'` nennen:

```
liftA2' :: Applicative f =>
           (a -> b -> c) -> f a -> f b -> f c
```

Hinweis: Sie müssen `fmap` verwenden.

Aufgabe 3. Man könnte `Monad` auch über `join` statt über `>>=` implementieren, was allerdings in Haskell nicht vorgesehen ist.

(a) Implementieren Sie `join` über `>>=`, was wir `join'` nennen:

```
join' :: Monad m => m (m a) -> m a
```

(b) Implementieren Sie `>>=` über `join`, was wir `bind` nennen:

```
bind :: Monad m => m a -> (a -> m b) -> m b
```

Hinweis: Sie müssen vorher `join` importieren:

```
import Control.Monad(join)
```

Sie müssen außerdem `fmap` verwenden.

Aufgabe 4. Betrachten Sie folgende Data-Deklaration:

```
data Log a = MkLog a [String]
```

Die Idee von `Log` ist, dass man neben einem Wert vom Typ `a` auch noch eine Liste von Log-Meldungen berechnet.

(a) Implementieren Sie `Functor` für `Log`.

(b) Implementieren Sie `Applicative` für `Log`. Achten Sie darauf, dass Sie Log-Meldungen akkumulieren müssen.

(c) Implementieren Sie `Monad` für `Log`. Achten Sie darauf, dass Sie Log-Meldungen akkumulieren müssen.