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.

Lösung.

```
stringToInt s = case readMaybe s of
  Just x -> Ok x
  Nothing -> Fail ["Failed␣to␣convert␣"
                   ++ s ++ "␣to␣Int"]
```

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

Lösung.

```
instance Functor (Error e) where
  fmap f m = case m of
    Fail x -> Fail x
    Ok x -> Ok (f x)
```

(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)
```

Lösung.

```
instance Applicative (Error e) where
  pure = Ok
  f <*> m = case f of
    Fail x -> case m of
      Fail y -> Fail (x ++ y)
      Ok y -> Fail x
    Ok g -> case m of
      Fail y -> Fail y
      Ok y -> Ok (g 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)
```

Lösung.

```
apply f x = liftA2 (\g y -> g y) f x
```

(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.

Lösung.

```
liftA2' f x y = (fmap f) x <*> y
```

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
```

Lösung.

```
join' x = x >>= id
```

Hier ist `id :: a -> a` die Identitätsfunktion.

(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.

Lösung.

```
bind x f = join ((fmap f) x)
```

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`.

Lösung.

```
instance Functor Log where
  fmap f x = case x of
    (MkLog y l) -> MkLog (f y) l
```

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

Lösung.

```
instance Applicative Log where
  pure x = MkLog x []
  f <*> x = case f of
    (MkLog f' l) -> case x of
      (MkLog x' l') -> MkLog (f' x') (l ++ l')
```

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

Lösung.

```
instance Monad Log where
  x >>= f = case x of
    (MkLog x' l) -> case f x' of
      (MkLog y l') -> MkLog y (l ++ l')
```