Funktionale Programmierung, SS 2018, Blatt 3

Fabian Kunze, Steven Schäfer, Kathrin Stark,
Prof. Dr. Gert Smolka
https://courses.ps.uni-saarland.de/fp_ss18/

I/O und Monadische Typen

Aufgabe 3.1 (I/O und Monads) Lesen Sie Kapitel 7.1–7.2, 7.5 und 9 in *A Gentle Introduction to Haskell* (www.haskell.org/tutorial/) von Hudak, Peterson und Fasel. Experimentieren Sie mit einigen Beispielen aus dem Text.

Hinweis: Seit ghc Version 7.10 müssen Monaden auch Applicative und Functor sein. [1]. Für *M* ergänzt man dafür nach der Monaden-Instanz:

```
instance Functor M where
  fmap = liftM
instance Applicative M where
  pure = return
  (<*>) = ap
```

Aufgabe 3.2 (StrLen) Deklarieren Sie eine Aktion *strlen* :: *IO* (), die wiederholt einen String einliest und anschließend dessen Länge ausgibt:

```
> strlen
Enter a string: xyz
The string has 3 characters. Would you like to measure another string? y
Enter a string: spaghetti
The string has 9 characters. Would you like to measure another string? n
Exiting ...
```

Deklarieren Sie hierfür zunächst eine Prozedur *doWhile* :: *IO* () → *IO Bool* → *IO* (), so dass *doWhile c b* die Aktion *c* ausführt und solange wiederholt, wie die Bedingung *b* zu "True" auswertet. Für die Ein- und Ausgabe von Text können Sie die Prozeduren *putStrLn* :: *String* → *IO* () und *getLine* :: *IO String* verwenden.

Aufgabe 3.3 (Naiver Interpreter für arithmetische Ausdrücke) Betrachten Sie den folgenden Datentyp für arithmetische Ausdrücke

```
data Exp = Const Int | Mult Exp Exp
```

a) Schreiben Sie eine Prozedur *eval* :: *Exp* → *Int* zum Auswerten solcher Ausdrücke.

[1] https://ghc.haskell.org/trac/ghc/wiki/Migration/7.10#base-4.8.0.0

b) Erweitern Sie *Exp* um einen Konstruktor *Div*, der Division repräsentiert. Beim Auswerten soll Division durch 0 mit einer Fehlerbehandlung abgefangen werden. Verwenden Sie dazu die Deklarationen

```
data Result a = Val a | Err String   deriving Show

raise :: String -> Result a
raise s = Err s
```

und ändern Sie die Prozedur aus (a) zu einer Prozedur *eval* :: *Exp* → *Result Int*. Beispielsweise soll sich ergeben:

```
> eval (Div (Div (Const 8) (Const 2)) (Const 2))
Val 2
> eval (Div (Div (Const 8) (Const 0)) (Const 0))
Err "division of 8 by 0"
```

Aufgabe 3.4 (Error Monad) Sei *Result* wie in Aufgabe 3.3(b) gegeben.

a) Deklarieren Sie *Result* als eine Instanz der Typklasse *Monad* zu deklarieren, indem Sie *return* :: *a* → *Result a* und die Komposition (>>=) :: *Result a* → (*a* → *Result b*) → *Result b* deklarieren. Machen Sie sich klar, dass diese Deklaration nun erlaubt, *do*-Notation für *Result a* (analog zu *IO a*) zu benutzen.

b) Verwenden Sie die *do*-Notation, um die folgende monadische Variante der Auswertungsprozedur aus Aufgabe 3.3(b) zu vervollständigen:

```
eval :: Exp -> Result Int
eval (Const n)    = return n
eval (Mult e1 e2) = do v1 <- eval e1
                       ⋮
```

Vergewissern Sie sich an einigen Beispielen, dass diese Prozedur mit *eval* aus Aufgabe 3.3(b) übereinstimmt.

Aufgabe 3.5 (Monad Laws)

a) Welche Operatoren benötigt man in Haskell, um eine Monade zu deklarieren? Welche Eigenschaften sollen diese Operatoren erfüllen? Inwieweit wird sichergestellt, dass diese Eigenschaften erfüllt werden?

b) Der Kleisli-Operator für Monaden hat folgende Signatur:

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

Überlegen Sie sich, welche Eigenschaften er erfüllen sollten.

c) Stellen Sie >=> mittels >>= dar und umgekehrt. Zeigen Sie, dass die Eigenschaften des einen Operators jeweils aus denen des anderen folgen.