Funktionale Programmierung, SS 2019, Blatt 3

Fabian Kunze, Kathrin Stark,
Prof. Dr. Gert Smolka
https://courses.ps.uni-saarland.de/fp_ss19/

I/O und Monadische Typen

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. *Hinweis:* Seit ghc Version 7.10 müssen Monaden auch Applicative und Functor sein [1]. Dazu benötigen Sie anfangs die folgenden Bibliotheken:

```
import Control.Applicative (Applicative(..))
import Control.Monad (liftM,ap)
```

Für jede Monade M ergänzt man nach der Monaden-Instanz:

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

Aufgabe 3.1 (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 3.2 (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 $eval :: Exp \to Result\ Int$.

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

d) 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
⋮
```

Aufgabe 3.3 (State Monad) In Haskell gibt es keine Zustände. Betrachten Sie den folgenden Datentyp.

```
newtype State s a = State { runState :: s →  (a, s) }
```

State s a stellt eine Berechnung dar, die einen Zustand des Typs *s* manipuliert und ein Ergebnis vom Typ *a* zurückgibt.

a) Zeigen Sie, dass *State s* für jeden Typ *s* eine Monade ist.

b) Nutzen Sie die obige Defintion, um in Haskell einen Datentyp *Stack* von Stacks zusammen mit monadischen Operationen $pop :: State\ Stack\ Int$ und $push :: Int \to State\ Stack\ ()$ zu deklarieren.

c) Betrachten Sie den folgenden Datentyp von Kommandos:

```
data cmd = Const Int | Add
```

Definieren Sie einen Interpreter $eval :: [cmd] \to Int$, der Listen von Kommandos auswertet.
Hinweis: Benutzen Sie eine Hilfsfunktion $eval' :: [cmd] \to State\ Stack\ ()$.

Aufgabe 3.4 (Monad Laws) 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?