Funktionale Programmierung, WS 2010/11, Blatt 3

Dr. Jan Schwinghammer, Prof. Dr. Gert Smolka
www.ps.uni-saarland.de/courses/proseminar-ws10/

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.

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

```
> strlen
Enter a string: xyz
The string has 3 characters.
```

Aufgabe 3.3 (While) Deklarieren Sie eine Prozedur *while* :: *IO Bool* → *IO* () → *IO* (), so dass *while b c* die Aktion *c* wiederholt, solange die Bedingung *b* wahr ist. Beispielsweise soll sich für

```
beep     = putChar '\BEL'
continue = do putStr "Quit? (y or n)  "
              c <- getChar
              putChar '\n'
              return (c /= 'y')
```

die folgende Interaktion ergeben:

```
> while continue beep
Quit? (y or n)  n
Quit? (y or n)  n
Quit? (y or n)  y
```

Aufgabe 3.4 (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.

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 *eval* (*Div* (*Div* (*Const 8*) (*Const 2*)) (*Const 2*)) das Ergebnis *Val 2* liefern, während *eval* (*Div* (*Div* (*Const 8*) (*Const 0*)) (*Const 0*)) das Ergebnis *Err* "*division of 8 by 0*" liefert.

Beachten Sie, dass die Fehlerbehandlung in (b) eine Änderung des gesamten Programms erzwingt, obwohl nur der zusätzliche Fall für die Division Fehler einführen kann. Die folgende Aufgabe zeigt, wie die monadische Struktur des Typkonstruktors *Result* ausgenutzt werden kann, um eine modulare Auswertungsprozedur zu schreiben die sich leicht erweitern lässt. (Dieses Beispiel ist aus Phil Wadler's "Monads for functional programming.")

Aufgabe 3.5 (Error Monad) Sei *Result* wie in Aufgabe 3.3(b) gegeben.

a) Geben Sie eine Prozedur *return* :: *a* → *Result a*, die Werte vom Typ *a* in den Typ *Result a* einbettet.

b) Geben Sie eine (Infix-) Prozedur (≫=) :: *Result a* → (*a* → *Result b*) → *Result b* für die sequentielle Komposition zweier Ausdrücke: Falls *e* zu *Err s* auswertet, dann liefert auch *e* ≫= *f* das Ergebnis *Err s*, und falls *e* zu *Val v* auswertet, dann ist *e* ≫= *f* gleich *f v*.

c) Verwenden Sie (a), (b) und *raise*, um *Result* als eine Instanz der Typklasse *Monad* zu 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(a) zu vervollständigen:

```
eval :: (Monad m) => Exp -> m 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(a) übereinstimmt.

e) Erweitern Sie *eval* nun so, dass auch Division korrekt behandelt wird. Vergleichen Sie die Änderungen zu denen aus Aufgabe 3.3(b).