

Real World Haskell

Tobias Höppner

SoSe 2013

Contents

1	VL I	3
1.1	Motivation	3
1.2	Was passiert hier?! - der kleine Webserver	3
1.2.1	der kleine Webserver	3
1.2.2	Einbinden von Modulen	4
1.2.3	Do-Notation	4
1.2.4	\$-Operator	4
1.2.5	!!-Operator	4
1.3	der größere Webserver	5
1.4	builds	5
1.4.1	mit ghc	5
1.4.2	mit cabal	5
1.5	.(Punkt)-Operator	5
1.6	Generics in Haskell	5
1.7	Stdlib - System.IO	5
1.8	Stdlib - System.Environment	5
1.9	Kommentare und Haddock	5
1.10	Keywords	6
2	VL II	7
2.1	Zustände in Haskell	7
2.1.1	Eval Funktion in "besser"	7
2.1.2	Abstrahieren	8
2.2	Zustandsveränderung	9
2.2.1	Verbesserte Eval-Funktion	9
2.2.2	Zustände nutzen	10
2.3	Monaden	10
2.3.1	die letzte Eval-Funktion (wirklich!)	11
3	VL III	12
3.1	Wiederholung Monanden	12
3.1.1	Maybe Monad	12
3.1.2	Die Id-Monade	13
3.2	Die IO-Monade	14
3.3	Arbeiten mit Monaden	14
3.3.1	Das Echo	14
3.3.2	handles	15

3.4	Gloss	15
3.4.1	komplexere Bilder	15
3.4.2	Animationen	16

Chapter 1

VL I

1.1 Motivation

Warum eigentlich Haskell?

Haskell Compiler ist mächtig. Weil die Semantik und Typsystem wilde Sachen erlaubt. Wilde Sachen ermöglichen korrekte Software und sind meist sogar effizienter.

1.2 Was passiert hier?! - der kleine Webserver

1.2.1 der kleine Webserver

```
1 import System.Environment (getArgs)
2 import System.IO (hFlush, hClose)
3 import Control.Monad (forever)
4 import Text.Printf (hPrintf, printf)
5 import Network (listenOn, accept, Socket, PortID (..))
6 import Control.Exception (handle, finally, SomeException)
7 import Control.Concurrent (forkIO)
8
9 main = getArgs >>= return . read . (!! 0)
10         >>= listenOn . PortNumber . fromIntegral
11         >>= forever . serve
12
13 serve socket = handle (\e -> print (e :: SomeException)) $ do
14   (sock, host, _) <- accept socket
15
16   forkIO $ flip finally (hClose sock) $ do
17     text <- readFile "index.html"
18
19     hPrintf sock "HTTP/1.1_200_OK\r\nContent-Length:_%d\r\n\r\n%s"
20               (length text) text >> hFlush sock
21
22   printf "Anfrage_von_%s_beantwortet\n" host
```

Was nicht behandelt wurde:

- Fehlerfälle, Exceptions Haskell unterstützt Exceptions
- Effizienz

1.2.2 Einbinden von Modulen

import am Anfang der Datei

- System.IO
- Control.Monad (forever)
- Text.Printf
- Network
- Control.Exception
- Control.Concurrent

1.2.3 Do-Notation

```
1 main = do
2   putStrLn "hallo user!!"
3   putStrLn "xxxx"
4   main = p "x" >> p "x"
```

ist das gleiche wie

```
1 main :: IO()
2 main = do
3   args <- getArgs
4   read (!!0) args
5   let x = read (!! 1) args
```

Typen

listenOn: $_ \leftarrow IO _$

1.2.4 \$-Operator

```
1 f a b
```

a ist eine Fkt. $g \times k$

b ist eine Fkt. $k \rightarrow v$

```
1 f $ g x k $ k f v
```

1.2.5 !!-Operator

Gibt das angegebene Element aus der Liste zurück.

```
1 (!! ) :: [a] -> Int -> a
2 let xs = []
3 ys = [1,2,4]
4 zs = [1..1378]
5
6 zs !! 0
```

1.3 der größere Webserver

1.4 builds

1.4.1 mit ghc

```
1 ghc x.hs
```

Wird unübersichtlich für mehrere Dateien / Module.

1.4.2 mit cabal

```
1 cabal configure
2 cabal build
3 cabal install
```

Projekte werden als **.cabal** gespeichert, sind eleganter und man kann schneller testen.

1.5 .(Punkt)-Operator

```
1 (.) :: (b -> c) -> (a -> b) -> (a -> c)
2 f . g
```

entspricht

```
1 (\x -> f ( g x))
```

1.6 Generics in Haskell

```
1 List e
2 m k v
```

1.7 Stdlib - System.IO

- Textinput / Textoutput
 - Print
 - getLine
 - getChar

1.8 Stdlib - System.Environment

- getArgs

1.9 Kommentare und Haddock

```
1 — einfacher Kommentar
2 {- mehrzeiliger Kommentar -}
3 {- mehrzeiliger Kommentar
4   {- verschachtelter Kommentar -}
```

```
5 -}  
6 - | haddock kommentar
```

1.10 Keywords

Programming Guidelines sind brauchbar

```
1 main = do  
2   args <- getArgs  
3   case args of  
4     [] -> ...  
5     ["-x"] -> ...  
6     ["-x",b] -> ...
```

Chapter 2

VL II

2.1 Zustände in Haskell

Gestern wurde der Taschenrechner implementiert, heute sehen wir uns an wie man die eval-Funktion weiter entwickeln kann.

2.1.1 Eval Funktion in "besser"

```
1 data Expr = Const Float | Add Expr Expr | Div Expr Expr
2
3 eval0 :: Expr -> Float
4 eval0 (Const x) = x = id x
5 eval0 (Add e1 e2) = eval0(e1) + eval0(e2)
6 eval0 (Div e1 e2) = eval0(e1) / eval0(e2)
7 eval0 (Div (Const 1) (Const 0)) = Infinity
```

Zunächst abstrahieren wir das pattern matching mithilfe einer neuen Funktion **evalExpr**. Die Hilfsfunktionen **fC**, **fA**, **fD** stehen jeweils für das ermitteln einer Konstanten, das Berechnen einer Addition oder das Berechnen einer Division.

```
1 fC :: Float -> Float
2 fA :: Float -> Float -> Float
3 fD :: Float -> Float -> Float
4
5 evalExpr fC fA fD (Const x) = fC x
6 evalExpr fC fA fD (Add e1 e2) = fA (evalExpr fC fA fD e1) (evalExpr
    fC fA fD e2)
7 evalExpr fC fA fD (Div e1 e2) = fD (evalExpr fC fA fD e1) (evalExpr
    fC fA fD e2)
```

Eine Fehlerbehandlung kann wie folgt realisiert werden:

```
1 eval1 = eval Expr id (+) fD
2 where fD x y = if y == 0 then error "div_by_0!" else (x/y)
```

Hier wird das Programm mit **error** abgebrochen sobald der Wert 0 für *y* angegeben wird. Besser wäre es eine Funktion zu formulieren die einen zusätzlichen Fehlerwert ausgeben kann (**Nothing**). Alle

erfolgreichen Ergebnisse sind in ein **Just** verpackt.

Zur Erinnerung **Maybe** ist wie folgt definiert:

```
1 data Maybe a = Nothing | Just a
```

In unserem Fall hat **Just** folgende Signatur:

```
1 Just :: Float -> Maybe Float
```

Damit kann man jetzt eine bessere **Eval**-Funktion schreiben:

```
1 eval2 :: Expr -> Maybe Float
2 eval2 = evalExpr Just fA fD
3   where
4     fA e1 e2 = case e1 of
5       Nothing -> Nothing
6       Just x -> case e2 of
7         Nothing -> Nothing
8         Just y -> Just (x + y)
9     fD e1 e2
10      Nothing -> Nothing
11      Just x -> case e2 of
12        Nothing -> Nothing
13        Just y -> Just (x / y)
```

2.1.2 Abstrahieren

fA und **fD** folgen dem selben Muster:

```
1 func x ... = case x of
2   Nothing -> Nothing
3   Just x -> ... some stuff with x ...
```

Mit dieser Erkenntnis können wir eine weitere Hilfsfunktion definieren **op**:

```
1 op :: Maybe a -> (a -> Maybe b) -> Maybe b
2 op val f = case val of
3   Nothing -> Nothing
4   Just x -> f x
```

Die Funktion **f** überführt einen Wert vom Typ *a* in ein *Maybeb*. **op** betrachtet das Argument *val* und wendet entweder **f** auf den in **Just** verpackten Wert an, oder gibt **Nothing** zurück.

```
1 eval3 = eval Expr Just fA fD
2   where
3     fA e1 e2 = e1 'op' (\x -> e2 'op' (\y -> (x + y)))
4     fD e1 e2 = e1 'op' (\x -> e2 'op' (\y -> (x / y)))
```

Der Taschenrechner von gestern hat jetzt eine Fehlerbehandlung die das Programm nicht abbricht, sondern eine adequate Fehlerbehandlung durchführt. (Fehlerhafte Werte werden mit **Nothing** symbolisiert.

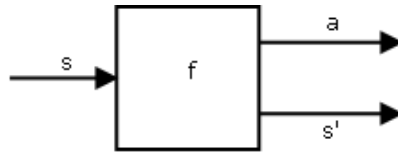


Figure 2.1: Zustandsveränderungen

2.2 Zustandsveränderung

Taschenrechner können sich meist Werte für verschiedene Variablen oder das letzte Ergebnis merken. Wir versuchen hier letzteres zu speichern:

```
1 type State = Float
```

type deklariert ein Type-Alias.

Jetzt brauchen wir eine Funktion die zustandsveränderungen konkret behandeln kann. Quasi als eine Art Objekte (keine OOP-Objekte).

```
1 update :: Float -> State -> State
2 update f = max abs(f)
3
4 data St a = S(State -> (a, State))
```

"Folgezustand" berechnen

```
1 apply :: St a -> State -> (a, State) — Startzustand -> (Ergebnis und
   Folgezustand)
2 appl (S f) s = f s
```

Unterschied data / type Listings:

type beschreibt wie man einen Zustand definiert. **data** beschreibt wie man einen Zustand verändert.

2.2.1 Verbesserte Eval-Funktion

```
1 eval4 :: Expr -> St Float
2 eval4 = eval Expr fC fA fD
3 where
4   fC x = S (\s -> (x, s))
5   fA sx sy = S (\s -> let (x, s1) = apply sx s
6                       (y, s2) = apply sy s1
7                       in (x + y, update(x+y) s2))
8   fA sx sy = S (\s -> let (x, s1) = apply sx s
9                       (y, s2) = apply sy s1
10                      in (x / y, update(x/y) s2))
```

Man erkennt, hier kann man **apply** durch eine neue Funktion **ap** verkürzen:

```
1 ret x = S (\s -> (x, s))
2
3 ap :: St a -> (a -> St b) -> St b
4 ap st f = S (\s -> let (x, s1) = apply st s
```

```

5         in apply (f x) s1)
6
7 eval5 = evalExpr fC fA fD
8     where
9         fC x = ret x
10        fA sx sy = sx 'ap' (\x ->
11            sy 'ap' (\y ->
12                s(\s -> (x+y, update abs (x+y) s))))
13        fD sx sy = sx 'ap' (\x ->
14            sy 'ap' (\y ->
15                s(\s -> (x/y, update abs (x/y) s))))

```

2.2.2 Zustände nutzen

Damit wir die Zustände auch verwenden können müssen wir sie auslesen und manipulieren können:

```

1 get :: St State
2 get = S (\s -> (s, s))
3
4 put :: State -> St ()
5 put s = S(\_ -> ((), s))

```

Unsere neue **Eval**-Funktion sieht danach so aus:

```

1 stAct :: (State -> State) -> St ()
2 stAct f = get 'ap' (put.f)
3
4 eval6 = evalExpr fC fA fD
5     where
6         fC = ret
7         fA sx sy = sx 'ap' (\x ->
8             sy 'ap' (\y ->
9                 stAct (update(x+y)) 'ap' (\() -> ret (x+y))))
10        fD sx sy = sx 'ap' (\x ->
11            sy 'ap' (\y ->
12                stAct (update(x\y)) 'ap' (\() -> ret (x\y))))

```

Man sollte erkennen das es immer wieder gleiche Muster gibt. Dieses Muster tritt so häufig in der funktionalen Programmierung auf, dass man es in eine einheitliche Schnittstelle verpackt hat.

2.3 Monaden

```

1 class Monad m where
2     return :: a -> m a
3     (>>=) :: m a -> (a -> m b) -> m b

```

ap ist eine Monade!

In Haskell schreibt man das so:

```

1 instance Monad Maybe where
2     return = Just
3     (>>=) = op

```

2.3.1 die letzte Eval-Funktion (wirklich!)

So implementiert man ungefähr immer eine Monade, **eval3**, **eval5** und **eval6** lassen sich so vereinfachen:

```
1 evalM :: Monad m => Expr -> m Float
2 evalM = eval Expr fC fA fD
3   where
4     fC = return
5     fA m1 m2 = m1 >>= (x ->
6       m2 >>= (y ->
7         ... — irgendwas spezifisches
8         return (x+y)))
9     fD m1 m2 = m1 >>= (x ->
10      m2 >>= (y ->
11        if y == 0 — irgendwas spezifisches
12        then Nothing
13        else return (x/y)))
```

Für

```
1 m >>= \x -> fx
```

schreibt man auch

```
1 do x <- m
2   return (f x)
```

Und

```
1 m1 >> m2
```

ist nichts anderes als

```
1 do m1
2     m2
```

Für Monaden gibt es 3 Gesetze, die stehen in der Doku.

Chapter 3

VL III

3.1 Wiederholung Monanden

Wir definieren uns unsere eigene Monade, dazu brauchen wir einen Datentyp:

```
1 data Entweder a = Links String | Rechts a
```

Zur Erinnerung, Monaden sind wie folgt typisiert

```
1 class Monad m where
2   return :: a -> m a
3   (>>=) :: m a -> (a -> m b) -> m b
4   (>>) :: m a -> m b -> m b
5   fail :: String -> m a
```

implementieren kann man das so:

```
1 instance Monad Entweder where
2   return = Rechts
3   m >>= f = case of
4     Rechts wert -> f wert
5     _ -> m
6   m >> n = case of
7     Rechts _ -> n
8     Links message -> Links message
9   fail str = Links str
```

Im Grunde braucht man nur **return** und **(>>=)**(bind). **(>>)** und **fail** sind nettigkeiten die uns das Leben in bestimmten Situationen vereinfachen können. Es gibt stimmen die behaupten, man braucht diese Funktionen nicht und man könnte diese auch durch eine extra Monade darstellen.

3.1.1 Maybe Monad

```
1 instance Monad Maybe where
2   return Just
3   m >>= f = case m of
4     Just wert -> f wert
```

```

5  _ -> Nothing
6  fail _ = Nothing

```

Wie verwendet man diese Monade?

```

1  f :: Int -> Maybe k
2  f i = if i == 0 then Nothing else Just i

```

kann man mit der **Maybe** Monade auch so schreiben:

```

1  f i = if i == 0 then fail "not_0" else return i

```

Nice to know: Listen in Haskell sind auch Monaden.

3.1.2 Die Id-Monade

```

1  data Id a = Id a

1  instance Monad Id where
2    return = Id
3    (ID x) >>= f = f x

```

Warum braucht man die? Wir kennen die Id-Funktion

```

1  id :: a -> a
2  id x = x

```

Für die Monaden gibt es noch keine ID Funktion. Deswegen definiert man sich die wie folgt:

```

1  id :: Monad m => a -> Id a
2  id x = return

```

Haskell kennt 2 Welten, alles ohne Monaden und alles mit Monaden. Beispiele: Ohne Monaden:

- id
- map
- ...

Mit Monaden:

- return
- mapM
- ...

Beispiel funktionale For-Schleife

```

1  print :: Show a => a IO ()
2
3  main :: IO()
4  main = do
5    let xss = ["Hallo", "Welt", "da_drau ß en"]
6    mapM print xss

```

oder auch direkt mit **forM**

```

1  forM :: [a] -> (a -> m b) -> m [b]
2
3  main = do
4    let xss = ["Hallo", "Welt", "da_drau ß en"]

```

```

5  forM xss $ do
6      ... — irgendwas
7      ... — irgendwas anderes
8      ... — irgendwas weiteres
9  putStrLn "xxx"

```

Wie ist **forM** implementiert?

3.2 Die IO-Monade

Bereits eine Ausgabe auf eine Konsole ist ein Seiteneffekt. In Haskell ist man aber eher exakt und möchte Seiteneffekte vermeiden. Dafür gibt es die IO-Monade, der Ansatz:

```

1  main :: [Response] -> [Request]
2  main (x:xs) = Print "Hello World" : main xs

```

Das ist ziemlich umständlich zu implementieren. Wurde trotzdem in Haskell 1.3 vorgeschlagen. Wie können wir jetzt elegant die Haskell-Welt verlassen? Richtig mit der IO-Monade:

```

1  main :: IO()
2  main = do
3      map print [8,9,9] —> [IO, IO, IO]
4      mapM print [8,9,9] —> 8
5                          — 9
6                          — 9

```

Die IO-Monade ist eine State-Monade. Die macht noch etwas mehr "magic" damit man "sauber" mit dem OS kommunizieren kann.

Monaden sind sehr mächtig. Man kann sich mit Monaden zwingen bestimmte Dinge nicht zu tun. Das ist hilfreich und gerade wenn man Monaden miteinander verbindet. Der Vorteil von Haskell ist gerade, das man mit Monaden zu bestimmten Situationen Seiteneffekte ausschliessen kann. Das ist gerade für "sichere" Software interessant.

3.3 Arbeiten mit Monaden

3.3.1 Das Echo

```

1  module Main where
2  — putStr :: String -> IO()
3  — putStrLn :: String -> IO()
4  — print :: Show a => a -> IO()
5  — getLine :: IO String
6
7  main :: IO()
8  main = do
9      line <- getLine
10     putStrLn line
11     — main — für endlosschleife , alternativ zu main = forever $ do
12     — main = getLine >>= putStrLn
13

```

```
14 main = forever (getLine >>= putStrLn) — cat in einer Zeile
```

3.3.2 handles

Werden meist von anderen Funktionen übergeben. Man kann Handels mit entsprechenden h.. Funktionen.

```
1 module Main where
2 — hputStr :: Handle -> String -> IO()
3 — hputStrLn :: Handle -> String -> IO()
```

3.4 Gloss

Gloss ist echtes, extrem vereinfachtes OpenGL. Einfaches Beispiel:

```
1 import Graphics.Gloss
2
3 main = display(InWindow "Nice Window" (200,200) (10,10)) white (
    Circle 80))
```

GHCI mag nicht unbedingt Gloss. Also erstmal bauen und dann ausführen.

3.4.1 komplexere Bilder

Wie zeichne ich ein komplexes Bild?

```
1 import Graphics.Gloss
2
3 main = display
4     (Fullscreen (1280,800))
5     black
6     (
7         Pictures[
8             Translate (-200) 0 (Color red (Circle 100)),
9             Translate 100 0 (Color yellow (Circle 100)),
10            Color white (ThickCircle 100 200),
11            Translate 100 100 $ Color blue $ Circle 300
12        ]
13    )
```

toll ist: man kann in Pictures einfach funktionen übergeben, die Funktionen sollte man natürlich vorher definieren. Das macht den Code lesbarer.

```
1 redCircle = Color red $ Circle 100
2 yellowCircle = Color yellow $ Circle 100
3 whiteCircle = Color white $ Circle 200
4
5 main' = display
6     (Fullscreen (1280,800))
7     black
```



```

8  (
9    Pictures[
10     Translate (-200) 0 redCricle ,
11     Translate 100 0 yellowCircle ,
12     whitecircle ,
13     Translate 100 100 $ Color blue $ Circle 300
14   ]
15 )

```

Der Bildmittelpunkt ist entscheidend. 0,0 ist der Mittelpunkt des Bildschirms.

3.4.2 Animationen

```

1 main = animate
2   (InWindow "Titel" (400, 300) (100, 100))
3   white
4   (\t -> Pictures [ Line [(0, 100), (0, -100)] , Translate (10*t) 0 (
      Circle 80) ] )

```