

Real World Haskell

Tobias Höppner

SoSe 2013

Contents

1	VL I	2
1.1	Motivation	2
1.2	Was passiert hier?! - der kleine Webserver	2
1.2.1	der kleine Webserver	2
1.2.2	Einbinden von Modulen	3
1.2.3	Do-Notation	3
1.2.4	\$-Operator	3
1.2.5	!!-Operator	3
1.3	der größere Webserver	4
1.4	builds	4
1.4.1	mit ghc	4
1.4.2	mit cabal	4
1.5	.(Punkt)-Operator	4
1.6	Generics in Haskell	4
1.7	Stdlib - System.IO	4
1.8	Stdlib - System.Environment	4
1.9	Kommentare und Haddock	4
1.10	Keywords	5
2	VL II	6
2.1	Zustände in Haskell	6
2.1.1	Eval Funktion in "besser"	6
2.1.2	Zustandsveränderung	7
2.1.3	Verbesserte Eval-Funktion	7
2.1.4	Eval5, noch coolerer	8
2.1.5	Jetzt ohne Zustände	8
2.1.6	evals im Vergleich	8
2.2	Monaden	9
2.2.1	die letzte Eval-Funktion (wirklich!)	9

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 mit Zustandsveränderungen verbessern kann.

2.1.1 Eval Funktion in "besser"

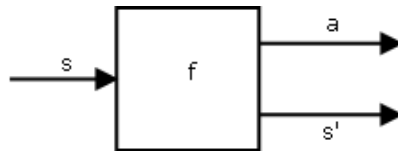
```
1 data Expr = Const Float | Add Expr Expr | Div Expr Expr
2
3 evalO :: Expr -> Float
4 evalO (Const x) = x = id x
5 evalO (Add e1 e2) = evalO(e1) + evalO(e2)
6 evalO (Div e1 e2) = evalO(e1) / evalO(e2)
7 evalO (Div (Const 1) (Const 0)) = Infinity
8
9 evalExpr fC fA fD (Const x) = fC x
10 evalExpr fC fA fD (Add e1 e2) = fA (evalExpr fC fA fD e1) (evalExpr
    fC fA fD e2)
11
12 data Maybe a = Nothing | Just a
13
14 eval1 = eval Expr id (+) fD
15 where fD x y = if y == 0
16         ...
17
18 eval2 :: Expr -> Maybe Float
19 eval2 = evalExpr Just fA fD
20 where
21     fA e1 e2 = case e1 of
22         Nothing -> Nothing
23         Just x -> case e2 of
24             Nothing -> Nothing
25             Just y -> Just (x + y)
26     fD e1 e2
```

```

27     Nothing -> Nothing
28     Just x -> case e2 of
29         Nothing -> Nothing
30         Just y -> Just (x 7 y)
31
32 — Abstrahieren
33
34 op :: Maybe a -> (a -> Maybe b) -> Maybe b
35 op val f = case val of
36     Nothing -> Nothing
37     Just x -> f x
38
39 eval3 = eval Expr Just fA fD
40 where
41     fA e1 e2 = e1 'op' (\x -> e2 'op' (\y -> (x + y)))
42     fA e1 e2 = e1 'op' (\x -> e2 'op' (\y -> (x / y)))

```

2.1.2 Zustandsveränderung



```

1 type State = Float
2
3 update :: Float -> State -> State
4 update f = max abs(f)
5
6 data St a = S(State -> (a, State))

```

"Folgezustand" berechnen

```

1 apply :: St a -> State -> (a, State)
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.

2.1.3 Verbesserte Eval-Funktion

```

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

```

2.1.4 Eval5, noch coolerer

```

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))))

```

2.1.5 Jetzt ohne Zustände

Zwei Hilfsfunktionen:

```

1 get :: St State
2 get = S (\s -> (s, s))
3
4 put :: State -> St ()
5 put s = S(\_ -> ((), s))
6
7 →
8
9 stAct :: (State -> State) -> St ()
10 stAct f = get 'ap' (put.f)
11
12
13 eval6 = evalExpr fC fA fD
14 where
15     fC = ret
16     fA sx sy = sx 'ap' (\x ->
17                     sy 'ap' (\y ->
18                         stAct (update(x+y)) 'ap' (\() -> ret (x+y))))

```

2.1.6 evals im Vergleich

- eval3 und eval6 machen das gleiche, jedoch auf unterschiedliche Wege

2.2 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
```

2.2.1 die letzte Eval-Funktion (wirklich!)

So implementiert man ungefähr immer eine Monade

```
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)
```

.

```
1 m1 >> m2
```

ist nichts anderes als

```
1 do m1
2    m2
```

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