Real World Haskell

Tobias Höppner

SoSe 2013

Contents

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 unterstütz 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

```
main = do
  putStrLn "hallo user!!"
  putStrLn "xxxx"
  main = p "x" >> p "x"
```

ist das gleiche wie

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

Typen

listenOn: $\_ \leftarrow IO \_$

1.2.4 $-Operator

```
f a b
```

a ist eine Fkt. g x k
b ist eine Fkt. k fv

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

1.2.5 !!-Operator

Gibt das angegebene Element aus der Liste zurück.

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

zs !! 0
```

1.3 der größere Webserver

1.4 builds

1.4.1 mit ghc

```
ghc x.hs
```

Wird unübersichtlich für mehrere Dateien / Module.

1.4.2 mit cabal

```
cabal configure
cabal build
cabal install
```

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

1.5 .(Punkt)-Operator

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

entspricht

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

1.6 Generics in Haskell

```
List e
m k v
```

1.7 Stdlib - System.IO

- Textinput / Textoutput
  - Print
  - getLine
  - getChar

1.8 Stdlib - System.Environment

- getArgs

1.9 Kommentare und Haddock

```
-- einfacher Kommentar
{- mehrzeiliger Kommentar -}
{- mehrzeiliger Kommentar
  {- verschachtelter Kommentar -}
```

```
-}
-- | haddoc kommentar
```

1.10 Keywords

Programming Guidelines sind brauchbar

```
main = do
  args <- getArgs
  case args of
    [] -> ...
    ["-x"] -> ...
    ["-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"

```
data Expr = Const Float | Add Expr Expr | Div Expr Expr

eval0 :: Expr -> Float
eval0 (Const x) = x = id x
eval0 (Add e1 e2) = eval0(e1) + eval0(e2)
eval0 (Div e1 e2) = eval0(e1) / eval0(e2)
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.

```
fC :: Float -> Float
fA :: Float -> Float -> Float
fD :: Float -> Float -> Float

evalExpr fC fA fD (Const x) = fC x
evalExpr fC fA fD (Add e1 e2) = fA (evalExpr fC fA fD e1) (evalExpr
    fC fA fD e2)
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:

```
eval1 = eval Expr id (+) fD
  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:

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

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

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

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

```
eval2 :: Expr -> Maybe Float
eval2 = evalExpr Just fA fD
  where
    fA e1 e2 = case e1 of
      Nothing -> Nothing
      Just x -> case e2 of
        Nothing -> Nothing
        Just y -> Just (x + y)
    fD e1 e2
      Nothing -> Nothing
      Just x -> case e2 of
        Nothing -> Nothing
        Just y -> Just (x / y)
```

2.1.2 Abstrahieren

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

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

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

```
op :: Maybe a -> (a -> Maybe b) -> Maybe b
op val f = case val of
  Nothing -> Nothing
  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.

```
eval3 = eval Expr Just fA fD
  where
    fA e1 e2 = e1 `op` (\x -> e2 `op` (\y -> (x + y)))
    fA 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.

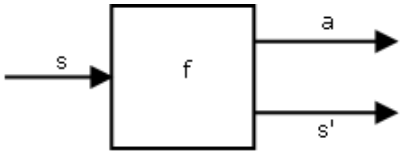

Figure 2.1: Zustandsveränderungen

2.2 Zustandsveränderung

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

```
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).

```
update :: Float -> State -> State
update f = max abs(f)

data St a = S(State -> (a,State))
```

"Folgezustand" berechnen

```
apply :: St a -> State -> (a, State) -- Startzustand -> (Ergebnis und
    Folgezustand)
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

```
eval4 :: Expr -> St Float
eval4 = eval Expr fC fA fD
  where
    fC x = S (\s -> (x,s))
    fA sx sy = S (\s -> let (x, s1) = apply sx s
                           (y, s2) = apply sy s1
                       in (x + y, update(x+y) s2))
    fA sx sy = S (\s -> let (x, s1) = apply sx s
                           (y, s2) = apply sy s1
                       in (x / y, update(x/y) s2))
```

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

```
ret x = S (\s -> (x,s))

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

```
            in apply (f x) s1)

eval5 = evalExpr fC fA fD
  where
    fC x = ret x
    fA sx sy = sx 'ap' (\x ->
          sy 'ap' (\y ->
          s(\s -> (x+y, update abs (x+y) s))))
    fD sx sy = sx 'ap' (\x ->
          sy 'ap' (\y ->
          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:

```
get :: St State
get = S (\s -> (s,s))

put :: State -> St ()
put s = S(\_ -> ((),s))
```

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

```
stAct :: (State -> State) -> St ()
stAct f = get 'ap' (put.f)

eval6 = evalExpr fC fA fD
  where
    fC = ret
    fA sx sy = sx 'ap' (\x ->
          sy 'ap'  (\y ->
          stAct (update(x+y)) 'ap' (\() -> ret (x+y)))))
    fD sx sy = sx 'ap' (\x ->
          sy 'ap'  (\y ->
          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 funktionalten Programmierung auf, dass man es in eine einheitliche Schnittstelle verpackt hat.

2.3 Monaden

```
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
```

ap ist eine Monade!
In Haskell schreibt man das so:

```
instance Monad Maybe where
  return = Just
  (>>=) = op
```

2.3.1 die letzte Eval-Funktion (wirklich!)

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

```
evalM :: Monad m => Expr -> m Float
evalM = eval Expr fC fA fD
  where
    fC = return
    fA m1 m2 = m1 >>= (x ->
          m2 >>= (y ->
            ... -- irgendwas spezifisches
            return (x+y))))
    fD m1 m2 = m1 >>= (x ->
          m2 >>= (y ->
            if y == 0 -- irgendwas spezifisches
            then Nothing
            else return (x/y))))
```

Für

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

schreibt man auch

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

Und

```
m1 >> m2
```

ist nichts anderes als

```
do  m1
    m2
```

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

Chapter 3

VL III

3.1 und weiter...