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
  Ja, 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

```
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: _ ← IO _

1.2.4 $-Operator

```
f a b
```

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

```
f (g x k) (k f v)
```

kann man auch

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

schreiben.

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
  - Liest Porgrammargumente aus und liefert einen IO String

1.9 Kommentare und Haddock

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

1.10 Keywords

Programming Guidelines sind brauchbar, eine Main sollte möglichst immer folgendes Muster haben:

```
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.

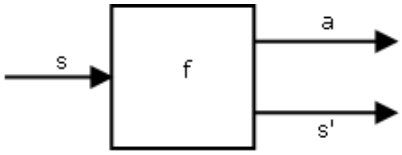

Figure 2.1: Zustandsveränderungen

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 Wiederholung Monanden

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

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

Zur Erinnerung, Monaden sind wie folgt typisiert

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

implementieren kann man das so:

```
instance Monad Entweder where
  return = Rechts
  m >>= f = case of
    Rechts wert -> f wert
    _ -> m
  m >> n = case of
    Rechts _ -> n
    Links message -> Links message
  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 aus der Haskell Community die behaupten, dass man diese Funktionen nicht braucht und man könnte diese auch durch eine extra Monade ersetzten. (Anmerk. Tobi: *Müssen wohl Puristen gewesen sein.*)

3.1.1 Maybe Monad

```
instance Monad Maybe where
  return Just
  m >>= f = case m of
    Just wert -> f wert
    _ -> Nothing
  fail _ = Nothing
```

Wie verwendet man diese Monade?

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

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

```
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

```
data Id a = Id a
```

```
instance Monad Id where
  return = Id
  (ID x) >>= f = f x
```

Warum braucht man die? Wir kennen die Id-Funktion

```
id :: a -> a
id x = x
```

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

```
id :: Monad m => a -> Id a
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

```
print :: Show a => a IO ()

main :: IO()
main = do
```

```
  let xss = ["Hallo", "Welt", "da drau ß en"]
  mapM print xss
```

oder auch direkt mit **forM**

```
forM :: [a] -> (a -> m b) -> m [b]

main = do
  let xss = ["Hallo", "Welt", "da drau ß en"]
  forM xss $ do
    ... -- irgendwas
    ... -- irgendwas anderes
    ... -- irgendwas weiteres
  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:

```
main :: [Response] -> [Request]
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:

```
main :: IO()
main = do
  map print [8,9,9]    --> [IO, IO, IO]
  mapM print [8,9,9]   --> 8
                       --  9
                       --  9
```

Die IO-Monade ist eine State-Monade. Die macht noch etwas mehr "magic" damit man "sauber" mit dem OS komunizieren 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. Dieses Konzept ist auch der ultimative Vorteil von Haskell, dass man mit Monaden zu bestimmten Situationen Seiteneffekte ausschliessen kann, einfach weil sie durch Monaden ausgeschlossen wurden. Das ist gerade für "sichere" Software interessant.
(Anmerk. Tobi: *Wie definiert man hier sichere Software?*)

3.3 Arbeiten mit Monaden

3.3.1 Das Echo

Zur Erinnerung, wichtige Funktionen die man braucht um mit der Außenwelt zu kommunizieren:

```
module Main where
  putStr :: String -> IO()
  putStrLn :: String -> IO()
  print :: Show a => a -> IO()
  getLine :: IO String
```

Einfaches Echo:

```
main :: IO()
main = do
  line <- getLine
  putStrLn line
```

einfaches Echo - Einzeiler:

```
main = getLine >>= putStrLn
```

unendliches Echo

```
main = do
  line <- getLine
  putStrLn line
  main
```

oder

```
main = forever $ do
  line <- getLine
  putStrLn line
```

cat - in einer Zeile

```
main = forever (getLine >>= putStrln)
```

3.3.2 handles

Werden meist von anderen Funktionen übergeben. Man kann Handels mit entsprechenden **h..**-Funktionen verwenden. Zum Beipsiel:

```
module Main where
-- hputStr :: Handle -> String -> IO()
-- hputStrLn :: Handle -> String -> IO()
```

3.4 Gloss

Gloss ist echtes, extrem vereinfachtes OpenGL. Einfaches Beispiel:

```
import Graphics.Gloss

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?

```
import Graphics.Gloss

main = display
  (FullScreen (1280,800))
  black
  (
    Pictures[
      Translate (-200) 0 (Color red (Circle 100)),
      Translate 100 0 (Color yellow (Circle 100)),
      Color white (ThickCircle 100 200),
      Translate 100 100 $ Color blue $ Circle 300
    ]
  )
```

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

```
import Graphics.Gloss

redCircle = Color red $ Cricle 100
yellowCircle = Color yellow $ Circle 100
whiteCircle = Color white $ Circle 200

main = display
  (FullScreen (1280,800))
  black
  (
    Pictures[
      Translate (-200) 0 redCricle,
      Translate 100 0 yellowCircle,
      whitecircle,
      Translate 100 100 $ Color blue $ Circle 300
    ]
  )
```

Wichtig: $0,0$ ist der Mittelpunkt des Bildschirms.

3.4.2 Animationen

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

Chapter 4

VL IV

Ich bin ein Fan von Einzeilern

4.1 Nachtrag: Der EchoServer

4.2 List-Monade

Listen in Haskell sind eigentlich auch Monaden:

```
data List a = Nil | Cons a (List a)
```

Wie funktioniert das?:

```
[1,2,3]    = Cons 1 ( Cos 2 ( Cons 3 Nil))
       = 1 : 2 : 3 : []
```

Die Monade ist wie folgt definiert:

```
instance Monad List where
  -- :: a -> List a
  return a = [a]
  -- :: List a -> (a -> List b) -> List b
  as >>= f =  concat $ map f as
```

Idee für Bind:
Ich hab eine Liste mit vielen Elementen von Typ a also [a,a,...,a]. Die Bindfunktion macht aus jedem a eine Liste mit Elementen von Typ b: [[b,..,b],...,[b,..,b]]. Am Ende müssen wir für jedes b eine eine Funktion anwenden die daraus wieder b's generiert.

Jetzt muss man zeigen das alle 3 Monadengesetze gelten.

4.2.1 Das 1. Monadengesetz

Wir wollen zeigen:

```
return a >>= f = f a
```

Was macht das?
return a steckt a in eine Monade. **(»=)**(Bind) nimmt a wieder aus der Monade und wendet **f** darauf an. Also muss das Ergebnis das gleiche sein wie einfach nur **f** auf a angewendet.

Beweis:

```
return a >>= f
= [a] >>= f
= concat (map f [a])
= concat ([f a])
= f a
```

q.e.d

4.3 Was ist eigentlich (»=)?

Zur Erinnerung man kann schreiben:

```
a <- m
f a
```

für

```
m >>= (\a -> f a)
```

Jetzt sehen wir uns am Beispiel von Listen an was der Bind-Operator eigentlich genau macht:

```
f :: [a] -> [b] -> [(a,b)]
f xs ys = d
  a <- xs
  b <- ys
  return (a, b)
```

Wir erhalten hier das kartesiche Produkt der 2 Listen. Folgend mit X dargestellt:

```
f xs ys = xs X ys
```

Intiutives Beispiel:

```
f [1,2] [3,4]
  = [(1,3),(1,4),(2,3),(2,4)]
```

4.4 Funktoren

Is nich so fancy wie es klingt...
Wir haben hoffentlich alle schonmal was geschrieben wie:

```
map :: (a -> b) -> List a -> List b
map f [x1,...,xn] = [f x1,...,f xn]
```

Zur Veranschaulichung definieren wir uns einen Baum:

```
data Baum a = Leaf | Node (Baum a) a (Baum a)
```

Wie kann ich jetzt map auf Bäume anwenden?
Man definiert sich eine eigene Map-Funktion für seine Datenstruktur:

```haskell
mapBaum :: (a -> b) -> Baum a -> Baum b
mapBaum _ Leaf = Leaf
mapBaum f (Node l a r) = Node (mapBaum f l) (f a) (mapBaum f r)
```

Fertig.

4.4.1 Beispiel Maybe

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

Oh, cool! **Maybe** ist auch polymorph! Also können wir uns **mapMaybe** definieren.

```haskell
mapMaybe :: (a -> b) -> Maybe a -> Maybe b
mapMaybe _ Nothing = Nothing
mapMaybe f (Just a) = Just $ f a
```

Wir erkennen ein Muster! Also abstrahieren wir.

4.4.2 fmap

Das f vor dem **map** steht für Funktor. Wir kennen ja bereits dass man Buchstaben hinter bestimmte Funktionen schreibt. Zum Beispiel **mapM** mit M für Monaden.

```haskell
fmap :: Functor f => (a -> b) -> f a -> f b

class Functor a where
  fmap ...
```

Es gibt zwei **Funktorengesetze**.

4.4.3 Funktorengestze

1. Ich möchte nichts verändern, wenn ich nichts mache!

   ```haskell
   fmap id = id
   ```

2. Wende ich mehrere Funktionen mit fmap an, so soll die Reihenfolge keinen Einfluss auf das Ergebnis haben:

   ```haskell
   fmap (f.g) = fmap f . fmap g
   ```

4.4.4 ein eigener Funktor

Man kann sich auch zusätzliche Funktoren definieren. Wenn man sich einen eigenen Datentyp erstellt hat. Dann kann man sich so einen Funktor dazu definieren.

```haskell
instance Functor Baum where
  fmap = mapBaum
```

4.4.5 Beispiel für die Verwendung

Wir wollen sowas wie:

```
main = do
  getArgs :: IO[String]
  head :: [a] -> a

  head $ getArgs -- geht nicht, aber

  head 'fmap' getArgs -- geht
  head <$> getArgs -- und das hier auch!
  -- machen beide das gleiche
```

4.5 DataRecords

4.5.1 DataRecords definieren

DataRecords sind Datenstrukturen wo die Parameter konkrete Namen haben. Der Vorteil ist dabei, das es auch gleich Konstruktoren sind und man kann sie verwenden.

```
data Client = Client{
  clientHandle :: Handle
  , clientHost :: Hostname
  , clientPort :: PortNumber
}
```

4.5.2 DataRecord update syntax

sind quasi setter

```
client{clientPort = 9000}
```

ändert den Port auf 9000.

Chapter 5

VL V

5.1 Parallel Haskell

5.1.1 Möglichkeiten

- par-Monade
- eval-Monade
- Strategys

5.1.2 par-Monade

```
-- schon implementiert und unter Control.Monad.Par verfügbar
-- muss vorher installiert werden
-- cabal install monad-par
data Par a
```

Zunächst braucht man einen NFD-Datentyp, damit kann man die Funktionen fork, spawn verwenden.

```
fork :: Par () -> Par ()
spawn :: Par a -> Par (IVar a)
```

fork erlaubt uns nicht mit dem neuen Prozess zu "reden", wir bekommen hier kein Ergebnis zurück. **fork** ist also nur ein Nebeneffekt. Für das parallele Berechnen als nicht brauchbar, aber für dienliche andere Aufgaben (Ausgaben auf stdio z.B.).
Besser ist **spawn**. Die aus **spawn** resultierende **IVar a** Variable dient der IPC (inter process communication). **IVar a** ist also nichts anderes als ein Future.
Wie verwendet man das?

```
eval :: Par a
eval = do
  future <- spawn(return(f x))  -- erzeugt neuen Thread der f x
      berechnet
  result <- get future      -- erzwingt die Ausführung von f x und
      blockiert ggf. das Programm
  return result

erg = runPar eval
```

In der Variabel future steht eventuell schon das Ergebnis, dass wird aber nicht zugesichert und es kann sein, das der Wert noch nicht berechent wurde. Wenn man das Ergebnis sofort benötigt, dann muss man diese Variable mit **get** aufrufen. **get** blockiert das Programm solange bis das Ergebnis der Berechnung fest steht.
Wir zeigen hier dem Compiler quasi wo er bestimmte Aufgaben parallelisieren kann. Das OS und der Compiler können dann entscheiden, ob ein zusätzlicher Thread erzeugt wird. Wichtig ist die Datenunabhängigkeit der Threads.

5.2 Concurrent Haskell

```
1 -- aus dem Paket Control.Concurrent
2 forkIO :: IO() -> IO ThreadID
```

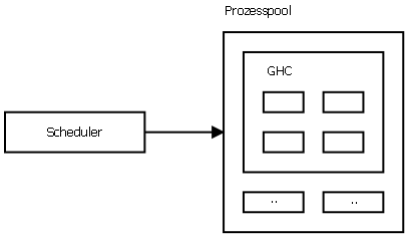

Figure 5.1: GHC hat eigene kleinere Prozesse

GHC hat seinen eigenen Scheduler. Der OS-Scheduler sieht nur den GHC-Prozess.
Das hat den Vorteil, das GHC geteilten Speicher ermöglicht. Damit kommen dann alle Probleme wie aus ALP4 bekannt sind wieder hoch.

5.2.1 MVar

M steht vermutlich für mutable.

```
1 takeMVar -- liest aus MVar wenn etwas drin ist, sonst wartet es bis
      etwas drin ist
2 putMVar -- schreibt in MVar wenn diese leer sind, sonst wartet es bis
      diese MVar leer ist
```

Beides sind blocking Methoden. MVars sind quasi ungepufferte, asynchr., beschränkte Kanäle (Puffergröße 1).
Weiterhin kann man MVars als Locks betrachten. Ebenfalls kann man MVars als binäre Semaphore verwenden, jedoch sollte man hier aufpassen wie man ein binäres Semaphore definiert hat.

5.2.2 Beispiel

Parallel Quicksort

```
1 module ParQuickSort where
2
3 import Control.Monad.Par
```

```
4
5 -- Grundidee für QS sequentiell
6 quickSort [] = []
7 quickSort (x:xs) = quickSort [y | y <- xs, y <- x] ++ [x] ++
      quickSort [y | y <- xs, y >= x]
8
9 parQuicksort [] = []
10 parQuicksort (x:xs) = runPar $ do
11   f1 <- spwan (return (parQuicksort [y | y <- xs, y < x]))
12   f2 <- spwan (return (parQuicksort [y | y <- ys, y >= x]))
13   left <- get f1
14   right <. get f2
15   return $ left ++ [x] ++ right
```

Es ist schon sinvoller sich ein günstiges Pivot-Element zu wählen. Den Median kann man in O(n) bestimmen.

5.3 STM - Software Transactional Memory

kurzer Abriss, für ausführliche Infos siehe Mitschrift auf Homepage

5.3.1 Die Idee

STM verwendet Transkationen. Transkationen können mehrere Befehle sein, die hintereinander ausgeführt werden sollen. Sollte während der Ausführung eine Störung (also irgendeine Art Fehler, inkonsitenz o.Ä.) auftretten, wird diese Transaktion zurück gerollt (roll-back) und später nochmal versucht.
Beispiel:

```
1 BEGIN
2   Hebe x von KTO A ab
3   Zahle x auf KTO B ein
4 END
```

5.3.2 TVar

Sind Variablen auf die man Transkationen definieren kann. Diese werden Atomar auszuführen.

```
1 atomically $ do
2   a <- readTVar x
3   if a < 0    -- Fehlerbedingung
4   then retry    -- nochmal versuchen
5   else -- weiter
```

retry ist (GHC-spezifisch) so realisiert, das der Thread erstmal schlafen gelegt wird. Man versucht den Thread dann "klug" zu wecken, also immer dann wenn in einer der TVars in der Transaktion geschrieben wurde.

5.4 Laziness vs. Strictness

Haskell ist faul... (ich auch)
Sowas wie

```
let x = [1,2..]
```

ist offensichtlich eine Endlosschleife. Da Haskell aber faul ist wird diese Liste aber nur soweit berechnet, wie benötigt.
Das klingt erstmal wie ein Vorteil, hat aber auch Nachteile. Zum einen braucht man ziemlich viel Speicher und zum anderen schießt man sich größere Lücken in den Speicher.

5.4.1 Was heißt eigentlich strict?

f ist strikt in x $\leftrightarrow$ f

```
f x = const 5 x
f (error "_") -> 5
```

Das Problem:

```
foldl (+) 0 [1,2,3,4]
= foldl (+) (0+1) [2,3,4]
= foldl (+) (0+1)+2 [3,4]
= foldl (+) ((0+1)+2)+3 [4]
= foldl (+) (((0+1)+2)+3)+4 []
= (((0+1)+2)+3)+4
= ((1+2)+3)+4
= (3+3)+4
= 6+4
= 10
```

Wie wir sehen können wird dieser Ausdruck "unglaublich" groß. Diese Phänomen haben wir sehr oft in Haskell. Man verbraucht sehr viel Speicher und das ist ein Nachteil.

```
f $ x
```

$\rightarrow$ f $! x

a`seq`b = b

```
const 5$ [error"_"]
```

$\rightarrow$ 5

```
const 5$! [error"_"]
```

$\rightarrow$ error

Das strickte foldl`:

```
foldl (+) 0 [1,2,3,4]
= foldl (+) 1 [2,3,4]
= foldl (+) 3 [3,4]
= foldl (+) 6 [4]
```

```
= foldl (+) 10 []
= 10
```

spart Speicherplatz! Man hat Lineare Speichermenge in der Größe der Eingabe.

```
const 5$ [error"_"]
```

→ 5

```
const 5$! [error"_"]
```

→ 5

→ Weak head normal form:
$(5+7) \to 12$ ist nicht in Normalform
$= (+)57$
$= ((\lambda xy.x+y)5)(7)$
$(\lambda x \to 2+x)$ ist in NF // $(\lambda x.2+x)$ ist in WHNF

$[4, 2+2]$ ist nicht in NF// =Cons4(Cons$(2+2)$) ist aber in WHNF

5.4.2 deepseq

Wenn wir sicherstellen wollen, dass eine Datenstruktur vollständig bearbeitet wird, brauchen wir ein deepsec.

```
a 'deepseq' b = $!!
```

Für parallele Ausführungen braucht man also Daten auf die eine Normalform definiert ist.

```
NFData a =>
```

sichert genau das zu.

5.4.3 Beispiel: Remote Key-Value-Store

Sequentiell

```
main = do
  s <- listenOn ... -- socket
  let serve d = do
    (h,_,_)<-accept s
    ... ... >>= case cmd of
    Get -> lookup pilt >> serve d
```

Nicht Sequentiell

```
main = do
  s <- ... -- socket
  m <- newMVar Map.empty
  forever $ do
```

```
5       (h,_,_)<-accept s
6       forkIO $ do
7         ... case h of
8           GET =
9           PUT =
```

Das Problem hier ist der Flaschenhals (bottleneck) beim Put. Da es alles blockiert wenn geschrieben werden soll. Damit werden auch alle Leser am Zugriff gehindert.

Nicht Seq. mit STM

```
1 main = do
2   s <- ... -- socket erzeugen
3   st <- atomically $ newTVar $ Map.empty -- speicher holen
4   forever $ do
5     (h,_,_)<-
6     forkIO $ do
7     ... case h of
8     GET -> do {atomically $ readTVar st >>= {-Ausgabe-}}
9     PUT -> do {atomically $ readTVar st >>= {-verändern-} >>=
          writeTVar m >>= {-ACK-}}
```

atomically und das STM-System in Haskell nehmen uns viel Arbeit hab. Möchte man Transaktionen selbst bedienen, braucht man **orElse** und **retry**

Chapter 6

VL VI

6.1 GHC unter der Haube

Der GHC bringt die "Effizienz" von Haskell. Das Typsystem von Haskell erlaubt Compileroptimierungen viel schneller als andere Compiler.

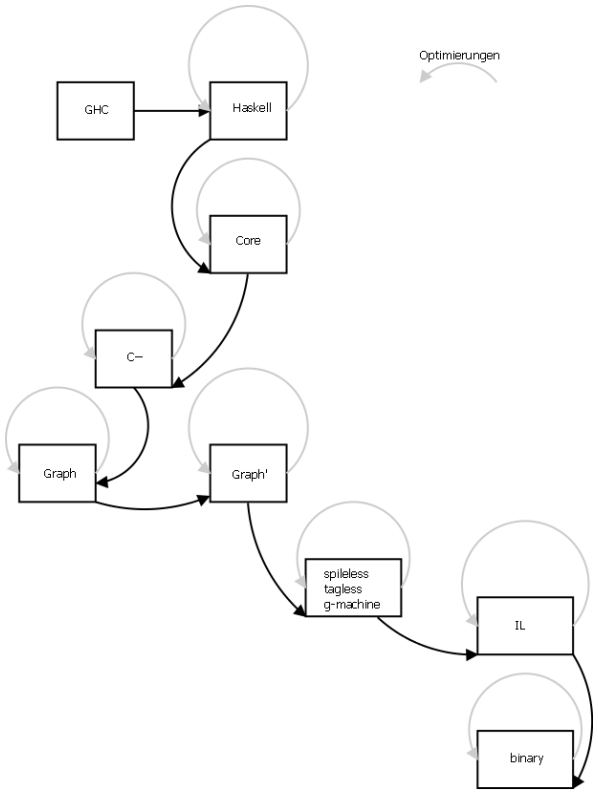

Figure 6.1: Optimierungen im GHC

6.2 Monadentransformation

6.2.1 Lambda Interpreter in Haskell

```haskell
data Exp = Lit Integer
  | Var Name
  | Plus Exp Exp
  | Abs Name Exp
  | App Exp Exp
  deriving (Show)
```

```haskell
data Value = IntVal Integer
  | FunVal Env Name Exp
```

```haskell
type Env = Map Name Value
```

Das Problem

```haskell
eval0 :: Env -> Exp -> Value
eval0 _ (Lit i) = IntVal i
eval0 env (Lit n) = env ! n
eval0 env (Plus e1 e2) = let IntVal i1 = new eval0 env e1
                             IntVal i2 = new evak0 env e2
                         in IntVal (i1 + i2)
eval0 env (Abr n exp) = FunVal env n exp
eval0 env (App e1 e2) = let v1 = eval0 env e1
                            v2 = eval0 env e2
                        in case v1 of (FunVal e n x) -> eval0 env x
```

Hier fehlt die Fehlerbehandlung. Zum Beispiel könnte man zwei Funktionen übergeben.

Lösung Schritt 1: umbauen in Monandischen Stil

```haskell
eval1 :: Monad m => Env -> Exp -> Value
eval1 _ (Lit i) = return $ IntVal i
eval1 env (Lit n) = env 'lockup' n
eval1 env (Plus e1 e2) = do IntVal i1 <- new eval0 env e1
                            IntVal i2 <- new evak0 env e2
                            in return $ IntVal (i1 + i2)
eval1 env (Abr n exp) = return $ FunVal env n exp
eval1 env (App e1 e2) = do v1 <- eval0 env e1
                           v2 <- eval0 env e2
                           in case v1 of (FunVal e n x) -> eval0 env x
```

Jetzt haben wir durch das **lookup** bereits eine kleine Fehlerbehandlung realisiert. Je nach verwendeter Monade bekommt man z.B. **Nothing** zurück.

Beispiel: Id Monade

Implementierung siehe:

```haskell
import Control.Monad.Identity
```

wir verwenden Id um Eval weiter zu verbessern.

```
type Eval a = Identity a
runEval :: Eval a -> a
runEval x = runIdentity x
```

Lösung Schritt 2: ordentliche Fehlerbehandlung

Wir verschachteln jetzt die **ErrorT**-Monade um die Id-Monade.

```
import Control.Monad.ErrorT
-- :k ErrorT
-- * -> (* -> *) -> * -> *

type Eval a = Identity a

runEval :: Eval a ->
runEval x = runIdentity (runErrorT x)
```

Wir "erben" jetzt die Funktionen von ErrorT. Zum Beispiel **throwError**. Damit können wir unser Programm ändern und mehr Sachen machen.

```
newtype Exc =   Exc Exp String
```

```
eval1 :: Monad m => Env -> Exp -> Value
eval1 _ (Lit i) = return $ IntVal i
eval1 env (Lit n) = env 'lockup' n of {
                        Just v -> return n;
                                _ -> throwError $ Exc e "not Found"}
eval1 env (Plus e1 e2) = do IntVal i1 <- new eval0 env e1
                            IntVal i2 <- new evak0 env e2
                          in return $ IntVal (i1 + i2)
eval1 env (Abr n exp) = return $ FunVal env n exp
eval1 env (App e1 e2) = do v1 <- eval0 env e1
                            v2 <- eval0 env e2
                          in case v1 of (FunVal e n x) -> eval0 env x
```

Lösung Schritt 3: readerT

Wir fügen die readerT Monande zu unserem Konstrukt hinzu.

```
import Control.Monad.Identity
import Control.Monad.ErrorT
import Control.Monad.Reader

type Eval a = a
runEval :: Eval a -> a
runEval x = runIdentity (runErrorT (runReaderT x))
```

```
eval1 :: Monad m => Env -> Exp -> Value
eval1 (Lit i) = return $ IntVal i
eval1 (Lit n) =  do{env <- ask}'lockup' n of {
                      Just v -> return n;
                             _ -> throwError $ Exc e "not Found"}
eval1 (Plus e1 e2) = do IntVal i1 <- new eval0 e1
                           IntVal i2 <- new evak0 e2
                        in return $ IntVal (i1 + i2)
eval1 (Abr n exp) = return $ FunVal n exp
eval1 (App e1 e2) = do v1 <- eval0 e1
                           v2 <- eval0 e2
                        in case v1 of (FunVal e n x) -> eval0 x
```