Haskell Übersicht / Gedächtnisstütze

Daniel Pöllmann

26. Juli 2017

1 Allgemein

Diese Zusammenfassung soll nur eine kleine Gedächtnisstütze sein. Manches wird stark vereinfacht und verwendet Analogien um Sachverhalte leichter verständlich zu machen, die formal nicht ganz korrekt sind.

2 GHCi

Wichtige Befehle für GHCI:

```
:?              Hilfe
:quit           Beenden
:info (*)       Informationen zur * Funktion
:type 'x'       Typ von 'x' bestimmen
:load dat.hs    Laedt dat.hs
:set +t         Gibt den Typ von Ausdruecken nach deren Auswendung aus
:unset +t       Schaltet die automatische Ausgabe von Typen aus
:{              Mehrzeilige Code Eingabe
      CODE
:}
```

3 Typen und Typklassen

3.1 Allgemein

- Jeder Ausdruck hat einen Typ
- Keine impliziten Typumwandlungen
- Typklassen ermöglichen eine allgemeine Typdefinition
  Zum Beispiel

  ```
  (Num a) => ...
  ```

  legt fest, dass a ein numerischer Wert sein muss.

3.2 Konstruktoren

```
[]    nullstellig
(:)   zweistellig
```

```
[1, 2] = 1 : 2 : []
```

3.3 Typvariable

```
f :: Num a => a -> a -> a -> a
f x y z = x + 2 * y + 3 * z
```

In dem Beispiel ist »a« eine Typvariable, und mit »Num a« vor dem »=>« wird der Typ von a auf die Typklasse Num eingeschränkt.

⇒ Eine Typsignatur in der eine Typvariable vorkommt heisst:

- polymorph
- parametrisch

3.4 Hierarchie

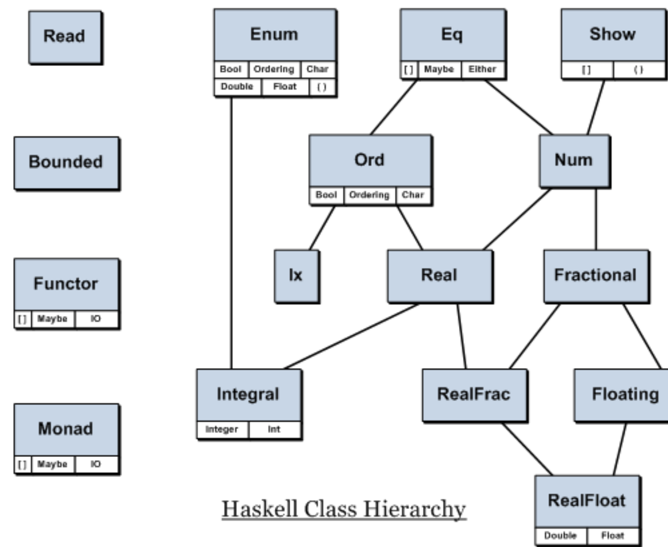

Abbildung 1: https://blogs.msdn.microsoft.com/saeed/2009/03/15/haskell-class-hierarchy-diagram/

3.5 Polymorphismus

3.5.1 Parametrischer Polymorphismus

mit Typ-Variablen, z.B. [a]

3.5.2 Ad-hoc Polymorphismus

mit Typ-Klassen

3.6 Grundlegende Typklassen

Eq: =, /=
Ord: <, >, <=, >=, **compare**
Num

Eine ganze Zahl ist eine polymorphe Konstante.

3.7 Beispiel: Instanz einer Typklasse

```
data BBKM = L | MM a (BBKM a) (BBKM a) deriving (Show)

instance Eq a => BinBaum BBKM a where
        istIn x (MK w _ _) | w == x = True
        istIn x (MK _ links rechts) | otherwise = (istIn x links) || (istIn x rechts)
        istIn _ _ | otherwise = False
```

4 Benutzerdefinierte Typen

```
data StudentInfo = Student String String Int deriving (Show)
```

Hierbei ist:

- StudentInfo: Bezeichnung des Typs
- Student: Typkonstruktor

4.1 Typsynomyme

```
        type Vorname = String
```

4.2 Record Syntax

```
data Student = Student {
                                                vorname :: String,
                                                nachname :: String
                                                matrikelNr :: Int
                                                } deriving (Show, Eq)

anna = Student {vorname="Anna",nachname="Schmidt",matrikelNr=1234}
```

Zum extrahieren von Werten stehen dann automatisch Funktionen zur Verfügung, die nach den Feldnamen benannt sind.

4.3 newtype

```
newtype Polar = Polar (Double, Double)
```

- nur ein Typkonstruktor erlaubt
- Typkonstruktor darf nur ein Feld haben

⇒ führt zur schnelleren Auswertung, falls anwendbar

4.4 data vs. newtype

- data definiert eager types
- newtype definiert lazy types

5 Operatoren

http://www.imada.sdu.dk/~rolf/Edu/DM22/F06/haskell-operatorer.pdf

6 Grundlegende Funktionen

Übersicht über grundlegende Funktionen und einer möglichen Implementierung.

```
-- Prueft, ob ein eine Zahl ungerade ist.
odd :: Integral a => a -> Bool
odd n = mod n 2 == 1

-- Prueft, ob eine Zahl gerade ist.
even :: Integral a => a -> Bool
even n = mod n 2 == 0

-- Entfernt die ersten n Elemente aus einer Liste.
drop :: Int -> [a] -> [a]
drop n xs       | n <= 0 =  xs
drop _ []                =  []
drop n (_:xs)            =  drop (n-1) xs

-- Prueft, ob eine Element in einer Liste vorkommt.
elem :: (Foldable t, Eq a) => a -> t a -> Bool
elem _ []     = False
elem y (x:xs) = y == x || elem y xs

-- Vergleicht zwei Werte.
compare :: Ord a => a -> a -> Ordering

-- Summiert eine Liste.
sum :: (Foldable t, Num a) => t a -> a
sum []     = 0
sum (x:xs) = x + sum xs

-- Kehrt die Reihenfolge der Listenelemente um.
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]

-- Trennt einen String anhand der Leerzeichen.
words :: String -> [String]

-- Fuegt Listenelemente zusammen und fuegt nach jedem Element einen Zeilenumbruch ein.
```

```
unlines :: [String] -> String

-- Bestimmt die Laenge der Liste.
length :: Foldable t => t a -> Int

-- Das erste Element der Liste.
head :: [a] -> a
head (x:xs) = x

-- Die Liste ohne das erste Element.
tail :: [a] -> a
tail (x:xs) = xs

-- Rest der Ganzzahldivision.
mod :: Integral a => a -> a -> a

-- Die ersten n Elemente der Liste
take :: Int -> [a] -> [a]
take 0 _      = []
take n (x:xs) = x : take (n-1) xs

-- Das letzte Element der Liste.
last :: [a] -> [a]
last (x:[]) = x
last (x:xs) = last xs

-- Wandelt einen String in einen anderen Datentyp um
read :: Read a => String -> a

-- Wandelt einen Wert (Showable) in einen String um, sodass dieser ausgegeben werden kann
show :: Show a => a -> String

-- Prueft, ob eine Liste leer ist.
null :: Foldable t => t a -> Bool
null xs = xs == []
```

7 Listen

```
[1..4] = [1, 2, 3, 4] = 1 : (2 : (3 : (4 : [])))
```

8 List Comprehension

```
[ x*x | x <- [1..10], ..., x `mod` 2 == 0, ...]
```

- $x * x$: Ausdruck, der in die Liste aufgenommen wird. Hier kann zum Beispiel auch wesentlich komplizierteres stehen.
- $x < -[1..10]$: x wird jeden den Wert jedes Elements aus der Liste [1..10] annehmen. Wenn mehrere Zuweisungen vorhanden sind, wird jede mögliche Kombination auf die verschiedenen Variablen zugewiesen.

```
[ (x, y) | x <- [1..5], y <- [1..5]]
ergibt:
[(1,1),(1,2),(1,3),(1,4),(1,5),(2,1),(2,2),(2,3),
(2,4),(2,5),(3,1),(3,2),(3,3),(3,4),(3,5),(4,1),
(4,2),(4,3),(4,4),(4,5),(5,1),(5,2),(5,3),(5,4),(5,5)]
```

- x ‘mod‘ $2 == 0$: Eine Bedingung, die erfüllt werden muss, damit der Ausdruck in die Liste aufgenommen wird. Wenn mehrere Bedingungen angegeben werden, so müssen auch alle erfüllt sein.

9 IO

9.1 Auf Konsole ausgeben

```
putStrLn "Hello World"
putStr "Hello World"
print "Hello World"
```

9.2 Von Konsole einlesen

```
zeichenfolge <- getLine
zeichen <- getChar
```

9.3 do

Innerhalb eines do-Blocks können IO Aktionen durchgeführt werden. Der letzte Ausdruck in einem do-Block muss eine IO-Aktion sein (bzw. return()).

9.4 return

return packt einen Wert in eine Aktion (IO)

$$\underbrace{return\ \underbrace{\text{"Hallo"}}_{String}}_{IOString}$$

9.5 $< -$

$'' < -''$ entnimmt den Wert aus einer IO-Aktion. Dies wird benutzt um zum Beispiel den Wert von 'IO String' als String zu bekommen.

```
zeichenfolge <- getLine

-- zeichenfolge ist vom Typ String
-- getLine vom Typ IO String
```

9.6 IO ()

Dies wird benutzt, wenn IO zurückgegeben werden muss, es aber unerheblich ist, was genau der verpackte Wert ist.

9.7 Lesen und Schreiben

```
leseGriff <- openFile "quelle.text" ReadMode
inhalt <- hGetContents leseGriff
hClose lesegriff

schreibGriff <- openFile "ziel.txt" AppendMode
hPutStr schreibGriff inhalt
hClose schreibGriff

Kuerzer:
inhalt <- readFile "quelle.txt"
writeFile "ziel.txt" inhalt
```

Modi: ReadMode, AppendMode, WriteMode, ReadWriteMode

10 Wächter/Guards

```
stepFunction value
    | value <= 10 = 1
    | value <= 20 = 0
    | otherwise   = 2
```

Es werden die Bedingungen solange der Reihe nach geprüft, bis eine True ist. Der folgende Ausdruck wird dann für den Funktionsaufruf verwendet.
'otherwise' ist ein Synonym für True.

11 Maybe

Datentyp Kapsel, die angibt, dass es einen Wert nicht geben muss.
Beispiel:

```
fak :: Integer -> Maybe Integer
fak n | n < 0     = Nothing
      | n == 0    = Just 1
      | otherwise = let Just m = fak (n-1) in Just (n * m)
```

12 Rekursion

Rekursion wird verwendet um komplexe Probleme zu vereinfachen und zu lösen. Dabei wird die Funktion selbst im Funktionsaufruf immer wieder aufgerufen (mit veränderten Argumenten) bis eine Endbedingung erreicht wird.

```
-- Berechne die Summe einer Liste rekursiv
sum []     = 0
sum (x:xs) = x + (sum xs)
```

Endrekursion
Rekursion mit konstantem Speicherbedarf.

- Rekursionsaufruf ist nicht in einem anderen Ausdruck eingebettet
- letzter Aufruf in einer Rekursion ist die Rekursionsfunktion selbst

13 Terminierungsbeweise

1. Geltungsbereich der Funktion feststellen.
2. Abstiegsfunktion $a : \mathbb{G} \rightarrow \mathbb{N}$, sodass $f(n)$ im Bezug auf $f(m)$ definiert ist.

$a(m) < a(n)$

14 Auswertung

Substitutionsmodell
Substitutionsmodell lässt offen in welcher Reihenfolge die Teilausdrücke ausgewertet werden

Vereinfachte Syntax:
- Funktionsdefinitionen nur als Lambda-Ausdrücke
- Fallunterscheidung nur über if-then-else

Funktionale Ausdrücke:
- Konstante wie quadrat = \ x -> x*x (evtl. mit lok. Def.)
- Funktionsanwendungen (quadrat 2)

Sonderausdrücke
- Wertdeklarationen (quadrat = \ x -> x*x)
- If-then-else

Parameter
- Formaler Parameter (z.B. x bei quadrat = \ x -> x*x)
- Aktueller Parameter in Funktionsanwendung (z.B. (2+1))

Umgebung
Die Umgebung ist eine List die von links her (vom Anfang) her gelesen wird

AUSWERTUNG
- Alle Teilausdrücke werden durch ihre Definition ersetzt
  - Formale Parameter -> aktuelle Parameter
- Sonderausdruck if B then A1 else A2
  - Nur B wird zuerst ausgewertet
  - Wenn B wahr, dann wird A1 ausgewertet und ist Wert des Sonderausdrucks
  - Wenn B falsch, dann wird A2 ausgewertet und ist Wert des Sonderausdrucks
- let A in B
  - Definition von A wird der Umgebung hinzugefügt
  - B wird in dieser erweiterten Umgebung ausgewertet
  - Definition von A wird von der Umgebung beseitigt

Auswertungsreihenfolge

Applikative Reihenfolge
Parameterauswertung vor Funktionsanwendung
andere Bezeichungen: Inside-out Auswertung, Call-by-value Auswertung, strikte Auswertung, Eager Auswertung

Normaler Reihenfolge
Funktionsanwendung vor Parameterauswertung
andere Bezeichnungen: Outside-in Auswertung, Call-by-name Auswertung

Verzögerte Auswertung ohne Konstruktoren
"Wenn eine Variable zweimal auftaucht, dann wird diese durch einen Zeiger auf die Variable ersetzt"

Sei y = komplizierter Ausdruck
quadrat y
^y * ^y (wobei eben ^y ein Zeiger auf y ist, sodass y nur einmal berechnet werden muss)
y = c (werte in einer Ebene tiefer y aus)

Verzögerte Auswertung verbindet Vorteile der applikativen und normalen Auswertung -> ermöglicht unendliche Datenstrukturen wie [1..]

andere Bezeichnungen: lazy evaluation, call-by-need Auswertung

Verzögerte Auswertung mit Konstruktoren
head [0..] -- benötigt nur 0
take 5 [0..] --benötigt nur [0,1,2,3,4]

Graph-Darstellung eines Ausdrucks
Darstellung eines Ausdrucks mit Zeigern, der die verzögerte Auswertung benutzt.
Graph-Reduktion
Schrittweise Vereinfachung eines Ausdrucks in Graph-Darstellung
Redux (reducable expression, thunk)
Teilgraph der Graphdarstellung der noch vereinfacht werden kann.
Die Graphdarstellung eines Ausdrucks ist in Normalform, wenn:
- ganz ausgewertet ist
- endlich ist
- keinen Zyklus enthält

Weak Head Normalform (WHNF)
- Ausdruck ist in WHNF, wenn äußerstes Konstrukt ein Konstruktor (z.B. (:) oder (\x->x*x)) ist.
- Anwendung einer vordefinierten Funktion auf zu wenige, evtl. keine Argumente ist (z.B. (+) 1 oder (++))

Jeder Ausdruck in Normalform ist auch in WHNF

Die Verzögerte Auswertung wird beendet, sobald eine WHNF ermittelt wird!

Lambda-Ausdrücke

quadrat x = x*x
quadrat = \ x -> x*x

Fallunterscheidung nur über if-else (keine Guards, etc.)

Vergleich: applikative vs normale Reihenfolge
Wenn alle Funktionsanwendungen eines Ausdrucks terminieren, liefern normale und applikative Auswertungsreihenfolge das Gleiche Ergebnis

Beispiel:
null = \ x -> 0
f = \x -> f (x + 1)

Auswertung von null (f 0)

applikativ	terminiert nicht
normal	terminiert

15 Monoid

Ein Monoid wird definiert durch:

- einen Typ
- einen Operator über diesen Typ mit Eigenschaften:
  - neutrales Element
  - assoziativ
  - optional: kommutativ

```
class Monoid m where
    mempty :: m
    mappend :: m -> m -> m
```

```
    mconcat :: [m] -> m
    mconcat = foldr mappend mempty
```

———

```
newtype Sum n = Sum n

instance Num n => Monoid (Sum n) where
    mempty = Sum 0
    mappend (Sum x) (Sum y) = Sum (x + y)
```

```
mempty `mappend` x liefert x (mempty ist Links-Neutrum)
x `mappend` mempty liefert x (mempty ist Rechts-Neutrum)
(x `mappend` y) ``mappend` z =
x `mappend` (y `mappend` z) (mappend ist assoziativ)
```

16 Funktionen höherer Ordnung

Monoid Gesetze Dies sind Funktionen, die eine Funktion als Parameter haben oder eine Funktion als Wert haben.

Ordnungen

- Ordnung 0: Konstanten
- Ordnung 1: Funktionen deren Parameter und Werte der Ordnung 0 sind
- höhere Ordnung: Alle weiteren Funktionen

```
map
filter
foldr
foldl
```

17 Binärbäume

```
data BB a = L | K a (BB a) (BB a) deriving Show
b = K 1 (K 2 L L) (K 3 L L)
```

Bei einem geordneten Binärbaum ist das linke Kind kleiner und das rechte Kind größer als die Wurzel.

17.1 Durchläufe

Wichtige vordefinierte Funktionen

17.1.1 Tiefendurchlauf

Infix Knoten des linken Teilbaums, Wurzel, Knoten des rechten Teilbaums

Präfix Wurzel, Knoten des linken Teilbaums, Knoten des rechten Teilbaums

Postfix Konten des linken Teilbaums, Knoten des rechten Teilbaums, Wurzel

```
data BB a = L | K a (BB a) (BB a) deriving (Show)

tief :: b -> (a -> b -> b -> b) -> BB a -> b
tief fL fK L = fL
tief fL fK (K w l r) = fK w (tief fL fK l) (tief fL fK r)

infixCollect = tief [] (\ w l1 l2 -> l1 ++ [w] ++ l2)
prefixCollect = tief [] (\ w l1 l2 -> [w] ++ l1 ++ l2)
postfixCollect = tief [] (\ w l1 l2 -> l1 ++ l2 ++ [w])

anzahlKnoten = tief 0 (\ w l1 l2 -> 1 + l1 + l2)
anzahlBlaetter = halb . tief 1 (\ w l1 l2 -> l1 + l2)
                                                  where halb x = quot x 2

baumTiefe = tief 0 (\ w l1 l2 -> 1 + max l1 l2)

istIn x = tief False (\w l1 l2 -> x == w || l1 || l2)
```

17.1.2 Breitendurchlauf

Erst die Wurzel, dann Wurzeln der Nachfolger

```
breit :: BB a -> [a]
breit baum = hb [baum]
                                    where   hb [] = []
                                                  hb (L:list) = hb list
                                                  hb (K w l r:list) = w:hb(list ++ [l,r])
```

18 Funktor

Functor verallgemeinert Listen mit map. (Datentypen, die wie eine Box sind können Funktoren sein.)

```
class Functor f where
        fmap :: (a -> b) -> f a -> f b
```

Beispiel für Maybe Datentyp:

```
instance Functor Maybe where
        fmap f (Just x) = Just (f x)
        fmap f Nothing = Nothing
```

18.1 Funktor Gesetze

```
fmap id ist gleich id
fmap (p.q) ist gleich (fmap p) . (fmap q)
```

⇒ Functor ist ein Berechnungskontext für Werte, z.B. [Int], Maybe String, BB Char

19 Applicative (Functor)

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

zum Beispiel:

```
fmap (*2) [1, 2, 3] liefert [2, 4, 6]
fmap (*2) (Just 2) liefert Just 4

(fmap (*)) [1, 2, 3] <*> [0, 2] liefert [0, 2, 0, 4, 0, 6]

wobei eben (fmap (*)) [1, 2, 3] = [(*1), (*2), (*3)] ist
```

Ein Applicative Functor ist ein Functor mit Operator ¡*¿, der Funktionsanwendungen innerhalb eines Funktors ermöglicht.

pure Wandelt "normalen Wertïn einen Functor wert um:

```
pure 1 :: Maybe Int ist Just 1
pure 1 :: [Int] ist [1]
pure "bcd" :: IO(String) ist IO Aktion mit String Wert

pure (+) <*> Nothing <*> Just 1 liefert Nothing
pure (+) <*> pure 1 <*> Just 1 liefert Just 2

f <*> pure x wendet Funktor-Funktion f auf das "Funktor-Gegenstueck" eines "normalen Wertes"

pure f <*> x ist das selbe wie fmap f x
```

19.1 Applicative Funktor Gesetze

```
pure f <*> x ist gleich fmap f x
pure id <*> v ist gleich v-identity
pure (.) <*> u <*> v <*> w ist gleich u <*> (v <*> w)
pure f <*> pure x ist gleich pure (f x)
u <*> pure y ist gleich pure ($ y) <*> u
```

19.2 Beispiel

```
instance Applicative Maybe where
        pure = Just
        Nothing <*> _ = Nothing
        (Just f) <*> something = fmap f something
```

20 Monad

Monade ist Applicative Functor mit

- Typkonstruktor m
- Bind-Operator (>>=) mit Typsignatur m a → (a → m b) → m b
- Einer Funktion return (=pure) mit Typ a → m a
- Einer Funktion fail mit Typsignatur String → m a

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

20.1 Monadengesetze

1. ```
   return Rechtseinheit von (>>=)
   -> m >>= return ist gleich m

   return Linkseinheit von (>>=)
   -> return x >>= f ist gleich f x
   ```

3. ```
   (>>=) ist linksassoziativ
   (m >>= f) >>= g ist gleich m >>= (\x -> f x >>= g)
   ```

21 Fehlerbehandlung

21.1 Maybe

- rein funktional

```
data Maybe a = Just a | Nothing

summe :: Maybe Int -> Maybe Int -> Maybe Int
summe m1 m2 = do
          n1 <- m1
          n2 <- m2
          return (n1 + n2)
```

21.2 Either a b

- rein funktional
- ```
  a: "Right Datentyp"
  b: Fehler-Datentyp (Left)
  ```
  ```

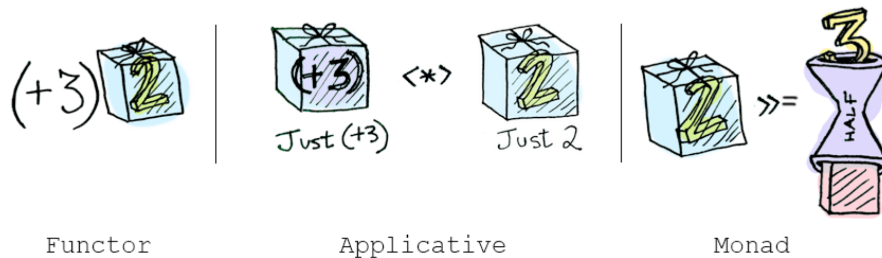

Abbildung 2:

```
Left "Fehler"
Right 5

summe :: Either Int String -> Either Int String -> Either Int String

summe e1 e2 = do
          n1 <- e1
          n2 <- e2
          return (n1 + n2)

summe (Right 1) (Right 2) => Right 3
summe (Left "Fehler") (Right 2) => Left "Fehler"
summe (Left "F1") (Left "F2") => Left F1
```

21.3 Exceptions

mit Nebeneffekten

22 Funktor, Applicative, Monade in Bildern

Super Erklärung zu diesen Themen:
http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html