

Haskell Übersicht / Gedächtnisstütze

Daniel Pöllmann

9. Juni 2017

1 Allgemein

Diese Zusammenfassung soll nur eine kleine Gedächtnisstütze zu den Sachen die wir gemacht haben sein und vereinfacht manches stark 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

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

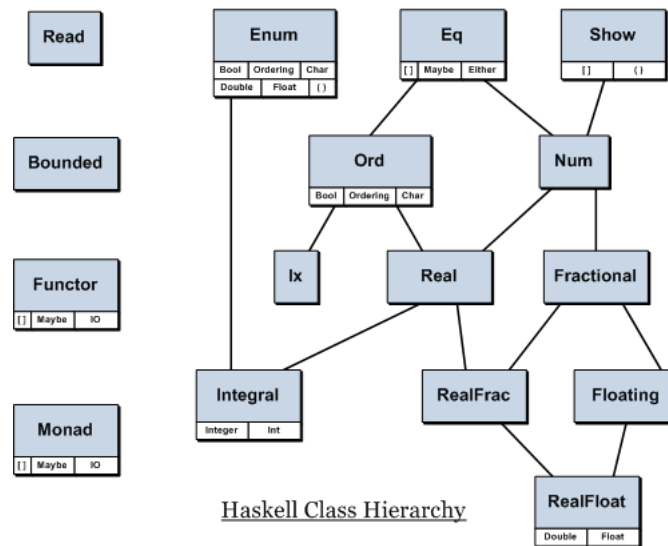


Abbildung 1: <https://blogs.msdn.microsoft.com/saeed/2009/03/15/haskell-class-hierarchy-diagram/>

4 Operatoren

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

5 Grundlegende Funktionen

Übersicht über grundlegende Funktionen und einer möglichen Implementierung (die jedoch nicht immer der internen Haskell Implementierung entspricht!).

— *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 \Rightarrow a \rightarrow **Bool**
even n = **mod** n 2 == 0

— *Entfernt die ersten n Elemente aus einer Liste.*

drop :: **Int** \rightarrow [a] \rightarrow [a]
drop n xs | n <= 0 = xs
drop _ [] = []
drop n (x:xs) = **drop** (n-1) xs

— *Prueft, ob eine Element in einer Liste vorkommt.*

elem :: (Foldable t, **Eq** a) \Rightarrow a \rightarrow t a \rightarrow **Bool**
elem _ [] = **False**
elem y (x:xs) = y == x || **elem** y xs

— *Vergleicht zwei Werte.*

compare :: **Ord** a \Rightarrow a \rightarrow a \rightarrow **Ordering**

— *Summiert eine Liste.*

sum :: (Foldable t, **Num** a) \Rightarrow t a \rightarrow a
sum [] = 0
sum (x:xs) = x + **sum** xs

— *Kehrt die Reihenfolge der Listenelemente um.*

reverse :: [a] \rightarrow [a]
reverse [] = []
reverse (x:xs) = **reverse** xs ++ [x]

— *Trennt einen String anhand der Leerzeichen.*

words :: **String** \rightarrow [**String**]

— *Fuegt Listenelemente zusammen und fuegt nach jedem Element einen Zeilenumbruch ein.*

unlines :: [**String**] \rightarrow **String**

— *Bestimmt die Laenge der Liste.*

length :: Foldable t \Rightarrow t a \rightarrow **Int**

— *Das erste Element der Liste.*

head :: [a] \rightarrow a
head (x:xs) = x

— *Die Liste ohne das erste Element.*

tail :: [a] \rightarrow a
tail (x:xs) = xs

— *Rest der Ganzzahldivision.*

mod :: **Integral** a \Rightarrow a \rightarrow a \rightarrow a

— *Die ersten n Elemente der Liste*

take :: **Int** \rightarrow [a] \rightarrow [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 == []

6 Listen

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

7 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)]

- $x \text{ 'mod' } 2 == 0$: 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.

8 IO

8.1 Auf Konsole ausgeben

putStrLn "Hello_World"

putStr "Hello_World"

print "Hello_World"

8.2 Von Konsole einlesen

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

8.3 do

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

8.4 return

return packt einen Wert in eine Aktion (IO)

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

8.5 < -

"< -" 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*

8.6 IO ()

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

8.7 Lesen und Schreiben

```
leseGriff <- openFile "quelle.txt" 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

9 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.

10 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)
```

11 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

12 Terminierungsbeweise

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

$$a(m) < a(n)$$

13 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 Bezeichnungen: Inside-in 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 (reducible 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