Programmiersprachen

Lernskript

Dozent:
Dr. Andreas Karwath

LaTeX von:
Sven Bamberger

Zuletzt Aktualisiert:
4. August 2014

Dieses Skript wurde erstellt, um sich besser auf die Klausur vorzubereiten.

Dieses Dokument garantiert weder Richtigkeit noch Vollständigkeit, da es aus Mitschriften und Vorlesungsfolien gefertigt wurde und dabei immer Fehler entstehen können. Falls ein Fehler enthalten ist, bitte melden oder selbst korrigieren und neu hoch laden.

Inhaltsverzeichnis

1 Haskell

1.1 Allgemein

Haskell ist eine Funktionale Programmiersprache und ist daher um einiges anders als die bisher gelernten Objektorientierten Sprachen. Diese verwenden verstärkt funktionale Eigenschaften um sich weiter zu entwickeln. Funktionale Sprachen sind frei von Seiteneffekten und die Methoden können sich Gegenseitig nicht beeinflussen. Eine Methode liefert bei den gleichem Input immer den gleichen Output. In diesem Dokument werden verstärkt Codebeispiele genutzt um die Sprache kurz zu erläutern.

1.2 Input, Main, Output

Zunächst ein kleines "Hello World"Beispiel, wie es zu jeder Sprache gehört. Hier sieht man direkt wie die main von Haskell aufgebaut ist.

```
main = putStrLn "Hello World"
```

Um ein kleines Programm zu schreiben, was mehr tut, als lediglich Sachen auszugeben, müssen wir Input verarbeiten. Dafür ist Haskell nie wirklich vorgesehen worden, da dies unvorhergesehenes Verhalten erzeugen kann, welche Haskell vermeiden soll. Daher sollte man dies nur verwenden, wenn es gefordert oder nicht anders möglich ist.

```
main =  do
        print "What is your name?"
        name <- getLine
        print("Hello" ++ name ++ "!")
        -- main
```

Wie man sieht, kann man den Input in eine vorher nicht definierte variable geben. In Haskell muss man variablen nicht vorher genauer definieren. Der Compiler versucht zur Compilezeit zu ermitteln, welchen Typ die Variable hat und setzt diesen fest. Wenn dies nicht möglich ist, wird die Variable zur Laufzeit erstellt und definiert und genau das kann zu unvorhergesehenem Verhalten führen.

1.2.1 IO

IO Programme aus der Fragestunde, härter als das wird es aller Voraussicht nach nicht werden.

```
putToConsoleAndCheck :: String -> IO Bool
putToConsoleAndCheck s = do
        if s == "" then return False
        else do
        putStrLn s
        return True
```

```
putToConsoleAndReturnNothing :: String -> IO ()
putToConsoleAndReturnNothing s = do
        if s == "" then return ()
        else do
        putStrLn s
        return ()
```

1.3 Funktionsdeklaration

Eine Funktion in Haskell sieht ungewohnt aus und erinnert mich immer stark an Mathe. Wobei die späteren Listen Verarbeitungsmöglichkeiten noch stärker an die Mathematik angelehnt sind.

```
f :: Int -> Int -> Int
f x y = x*x + y*y
```

dies ist alles was genötigt wird um eine Funktion zu deklarieren. Vom Schreibaufwand ist dies sehr gering. Jedoch ist es ungewohnt eine Funktion so zu sehen. Also nehmen wir diese einmal komplett auseinander.

$$\overbrace{f}^{\text{Funktionsname}} \quad \underbrace{x \quad y}_{\text{Variablennamen der Funktion}} \quad = \quad \overbrace{x * x + y * y}^{\text{was die Funktion macht}}$$

Es sieht zwar nicht gerade überragend aus, jedoch ist hier schon sehr genau erklärt wie eine Funktion aufgebaut ist. $f :: Int- > Int- > Int$ ist die sogenannte Funktionstypendeklaration das erste Int steht für das x das zweite für y und das dritte für den Typ des Ergebnisses. Es ist zwar nicht notwendig dies anzugeben, jedoch fördert es das Verständnis und hilft einem dabei nochmals genau über die Funktion nachzudenken. Da diese Methode jedoch auch gut für alle Zahlen geeignet ist, sollte man den Funktionstypen umdefinieren

```
f :: Num a => a -> a -> a
```

Das sind nochmals kryptischer aus, jedoch bedeutet dies nur, dass die Funktion f eine Zahl ($Numa$ => ") vom Typ Num erwartet. Danach hätte es gerne noch eine Zahl und gibt eine Zahl zurück. Welche Typen diese Zahlen haben, wird zur Compilezeit definiert und entsprechend angewandt. Das $a- > a- > a$ bedeutet, dass alle vom Typ Num sein müssen. Also wenn wir als erstes ein Int eingeben, werden alle a zu einem Int und es werden nur noch Int's erwartet. Vorteil, wir können eine Funktion für unterschiedliche Zahlentypen verwenden, ohne diese jedes mal neu zu definieren.

Auch können Funktionen weitere Funktionen aufrufen.

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

g :: Num a => a -> a -> a
g = f 3
```

Wi funktioniert nun *g*? diese Funktion ruft nun *f* mit dem Wert 3 auf und *f* gibt sich die Funktion *f* zurück um noch einen Wert entgegen zu nehmen. Dies bezeichnet man auch als Currying. Um zu verhindern, dass Haskell weiterhin die Typ Transformationen von Variablen übernimmt, kann man diese vorher definieren. Davon wird abgeraten, da dies zu unvorhergesehen Fehlern führen kann. z.B. wird der nach führende Code nie erfolgreich ausgeführt werden.

```
x :: Int
x = 3

y :: Float
y=2,4

print (f x)
```

1.4 Notation

1.4.1 Logik

|| ⇒ or && ⇒ and == ⇒ Gleichheit /= ⇒ Differenz

1.4.2 Potenz

xˆn x**y

Hierbei gilt es zu beachten, dass n ein Int || Integer sein muss, und y irgend eine Variable vom Typ Num sein darf.

1.4.3 Listen

- sind getypt, es können lediglich Elemente eines Typs in einer Liste vorkommen (Mischen von Typen ist nicht möglich). Dies ergänzt sich mit Tupeln: Listen sind theoretisch beliebig lange Folgen von Elementen eines Typs, Tupel sind definiert lange Folgen von Elementen, die verschiedene Typen haben können. Typsignatur von Listen: z.B. [Int], [Char], usw.
- können polymorph sein, d.h., der Typ einer Liste kann parametrisiert werden. Typsignatur in diesem Fall: z.B. [a]
- können per Aufzählung der Elemente definiert werden (Beispiel: [1,2,3]) oder mittels des Listenkonstruktors (Beispiel: 1:(2:(3))).
- werden per Rekursion (x:xs beschreibt das erste Element x einer Liste und die Restliste xs) oder per Indizierung (Operator !!) verarbeitet (die Indizierung beginnt bei 0)
- können mittels Erzeugungsschemata beschrieben werden (list comprehensions): z.B. [transform neueListe | neueListe <- alteListe, tollerTest neueListe], in dem die Elemente einer existierenden Liste traversiert, optional mit Prädikaten gefiltert und ebenfalls optional durch weitere Funktionen transformiert in eine neue Liste übernommen werden. In diese Rubrik fällt auch eine abgekürzte Schreibweise für Aufzählungen: z.B. [1..10] um eine Liste mit Elementen von 1 bis 10 zu erzeugen.
- können dank Haskell´s Lazy Evaluation Strategie unendlich lang sein (ihrer Beschreibung nach), wie z.B. bei [1..]. Solche Listen können verwendet werden, solange für den gewünschten Zweck die Auswertung nur eines Teils der Liste ausreicht.

Beispiele

[ ] ⇔ leere Liste
[1,2,3] ⇔ Liste von integralen
["foo","bar","baz"] ⇔ Liste aus Strings
1:[2,3] ⇔ [1,2,3] : ist das Symbol zum vorne anhängen an eine Liste
1:2:[ ] ⇔ [1,2]
[1,2]++[3,4] ⇔ [1,2,3,4] ++ ist für das konkatinieren zuständig
[1,2,3]++["foo"] ⇔ Error, die Inahlte einer Liste müssen immer vom selben Typ sein.
[1..4] ⇔ [1,2,3,4]
[1,3, .. 10] ⇔ [1,3,5,7,9]
[2,3,5,7,11 ... 100] ⇔ Haskell ist leider nicht so klug ;-)
[10,9 ..1] ⇔ [10,9,8,7,6,5,4,3,2,1]

1.4.4 Strings ⇒ Liste aus Char

'a' :: Char
"a":: [Char]
""⇔ []
"ab"⇔ ['a','b'] ⇔ 'a':"b"⇔ 'a':['b'] ⇔ 'a':'b':[]
"abc"⇔ "ab"++ "c"

1.4.5 Tupel

Tupel können unterschiedliche Typen enthalten und beliebig groß sein, jedoch funktionieren die Standard Implementationen lediglich auf Tupel der Größe 2

1.4.6 Beispiele

Tupelkonstruktionen

(2,"foo") Unterschiedliche Typen sind legitim in Tupel
(3,'a',[2,3]) Listen sind auch möglich in Tupel
((2,"a"),"c",3) Auch können Tupel in Tupel geschrieben werden

Standardimplementationen

fst(x,y) ⇒ x
fst(x,y,z) ⇒ Error, zu viele Elemente
snd(x,y) ⇒ y
snd(x,y,z) ⇒ Error, zu viele Elemente

1.5 Currying

Grundsätzlich sind alle Funktionen in Haskell curried. Curried bedeutet, dass die Funktion eine Funktion zurück liefert welche eine weitere Variable erwartet.

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

Bei diesem Beispiel wird die Funktion mit $f\ 4\ 5$ aufgerufen. Als erstes Ergebnis liefert die Funktion sich selbst zurück und nimmt die 5 als y auf. Dadurch ist es möglich Funktionen kürzer zu gestalten. In den folgenden Beispielen als erstes die gecurriede Version und direkt im Anschluss die nicht gercurriede Version.

```
Klassisches Currying
sum x y = x + y

Klassische ugecurriede Version
sum (x,y) = x + y
```

Auch lassen sich die Funktionen einfach Verkürzen.

```
Ungecurriede Version
square :: Num a => a -> a
square x = x^2

Gecurriede Version
square :: Num a => a -> a
square' = (^2)
```

1.6 if .. then .. else und Guards

Hier einfach ein Beispiel der gleichen Funktion einmal mit if .. then .. else Konstrukt und einmal als Guards (diese erinnern mich immer sehr stark an Switches) welche auch mit Pattern Matching zusammenarbeiten. Bei einem if .. then .. else Konstrukt muss immer ein else Teil angegeben werden. Da der Compiler sonst streikt. Denn Funktionen sollen ohne Nebeneffekte ausgeführt werden können. Und ein Vergessenes else führt zu nicht vorhersagbaren Nebeneffekten.

```
absolute :: (Ord a, Num a) => a -> a
absolute x = if x >= 0 then x else -x

absolute' :: (Ord a, Num a) => a -> a
absolute' x
        | x >= 0 =x
        | otherwise = -x
```

1.7 Typen und Typ Klassen

1.7.1 Standard Typen

Int ist ein ganzzahliger, an das System gebundener Wert. 32-Bit / 64-Bit

Integer ist auch ein ganzzahliger, nicht an das System gebundener Wert, es kann so groß werden, wie der Speicher groß ist.
Float eine echte Gleitkommazahl mit einfacher Genauigkeit.
Double eine echte Gleitkommazahl mit doppelter Genauigkeit.
Bool Boolean halt.
Char ein einzelnes Zeichen in Einfachen Anführungszeichen.

1.7.2 Standard Typ Klassen

Eq

```
(==) :: (Eq a) => a -> a -> Bool
```

Ord

```
(>) :: (Ord a) => a -> a -> Bool
```

Show wandelt Zahlen und boolische Ausdrücke in Strings um

```
show :: Show a => a -> String
```

Read wandelt Strings in Zahlen oder boolische Ausdrücke um.

```
read :: (Read a) => String -> a
```

Num ist eine numerische Typ Klasse. Ihr Mitglieder lasen sich wie Zahlen benutzen und verhalten sich so.
Enum ist eine Typ Klasse, welche alle auf zählbaren / durchzunummerierende Sachen enthalten. dadurch lassen sich erst die unendlichen Listen darstellen. Auch können succ und pred als Funktionen auf dessen Mitglieder angewandt werden.

1.7.3 Eigene Typklassen

type Name = AnotherType
Damit kann man dem Kind AnotherType einen neuen Namen geben. Wenn man z.B. zwischen Nachname und Vorname unterscheiden möchte. Jedoch macht der Compiler hier keinen Unterschied.

data Name = NameConstructor AnotherType
Hier macht der Compiler eine Unterscheidung. Der genaue Unterschied zwischen beiden type und data wird gleich an einem Beispiel erläutert.

data
Damit kann man sich rekursive Strukturen definieren.

deriving
damit kann man sich Funktionen erzeugen lassen.

Beispiele

Bei dem nachfolgenden Beispiel, muss man sehr aufpassen, denn die Type und Data Version haben starke Unterschiede zur Laufzeit. Wenn man die Type Version nutzt, kann man als Eingabe Color und Name

vertauschen und das Programm arbeitet trotzdem weiter, da es lediglich Strings erwartet.
Bei Data erwartet Haskell wirklich Datenstrukturen Name und Color. Wenn man hier zuerst Color und danach Name eingibt, meckert der Compiler.

```
Type Version
type Name  = String
type Color  = String

Data Version
data Name  = NameConstr String
data Color  = ColorConstr String

showInfos :: Name ->  Color -> String
showInfos (NameConstr name) (ColorConstr color) =
"Name: " ++ name ++ ", Color: " ++ color
```

Als nächstes wird eine Rekursive Datenstruktur definiert und mit deriving(Show) erweitert, was dazu führt, das diese Datenstruktur direkt gedruckt werden kann.

```
data Tree a = EmptyTree | Node a (Tree a) (Tree a) deriving (Show, Read, Eq)
```

1.8 Pattern Matching

Beim Pattern Matching handelt es sich um die Möglichkeit in eine Funktion Abzweigungen einzubauen, aufgrund von bestimmten Inhalten einer Variable. Davon wird exzessiv in Prolog[2] gebraucht gemacht. Das Pattern Matching ist eine effiziente Art zu programmieren, wenn man wenige unterschiedliche Werte hat, auf die geprüft werden muss. Auch kann man damit gut Abbruchbedingungen erstellen für rekursive Funktionen. Dabei gilt zu beachten, dass von oben nach unten gearbeitet wird.

```
factorial :: (Integral a) => a -> a
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

1.9 List Comprehension

Mit List Comprehensions kann man einfach Listen nach zuvor definierten Regeln erstellen. Dies ist besonders nützlich, um viele Werte schnell zu generieren.

1.9.1 Beispiele

```
[x^2|x<-[1..5]]
Ergebnis: [1,4,9,16,25]
```

Der Bereich nach | ist der sogenannte Generatorbereich und wird dazu genutzt, die Regeln für das erstellen der einzelnen Variablen zu definieren.

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

Man kann auch mehrere Variablen im Generator Bereich verwenden und dadurch eine Liste von Tupeln generieren wie im oberen Beispiel gezeigt. Jedoch sollte man sehr auf die Reihenfolge der Generatoren achten, da diese das Ergebnis stark beeinflussen.

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

Hier werden Dependant Generators verwendet. Die zu erzeugenden Werte für y hängen von x ab. Dadurch kann man sehr gut doppelte Werte vermeiden und andere Spielereien machen.

1.10 Lambda (λ) Funktionen

Lambda Funktionen sind Namenlose Funktionen welche lediglich einmal verwendet werden. Dies macht es möglich Funktionen in Methoden zu definieren und hat unter anderem auch Einzug in Java gefunden. Lambda Funktionen werden mit einem \ eingeleitet. Hier ein Beispiel für eine einfache Addition.

```
(\x y -> x+y) 4 3
Ergebnis: 7
```

Auf diese Art und Weise kann man sich beliebige Funktionen überall einbauen. Wodurch besonders foldl und foldr stark werden.

1.11 Higher Order Functions

Als Higher Order Functions werden Funktionen bezeichnet welche eine Funktion als erstes Argument entgegennehmen. **map**
nimmt eine Funktion und eine Liste entgegen und verwendet diese Funktion auf jedes Element der Liste und generiert so eine neue.

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

filter
ist eine Funktion welche eine Funktion entgegen nimmt, welche einen booleschen Wert zurück gibt. z.B. even. Dazu nimmt es noch eine Liste auf. Das Ergebnis ist eine Lsite, welche zu die Übergebene Funktion mit true ergibt.

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs)
| p x       = x : filter p xs
| otherwise = filter p xs
```

foldl
foldl wendet eine Funktion von **links** beginnend auf eine gesamte Liste an.

```
foldl (\x y -> x^2 + y) 0 [1,2,3]
Ergebnis: 12
```

Was genau vor sich geht:

```
foldl (\x y -> x^2 + y) 0 [1,2,3]
foldl (\x y -> x^2 + y) (0^2+1) [2,3]
foldl (\x y -> x^2 + y) (1^2+2) [3]
foldl (\x y -> x^2 + y) (3^2+3) []
foldl (\x y -> x^2 + y) 12 []
```

foldr
foldr wendet eine Funktion von **rechts** beginnend auf eine gesamte Liste an.

```
foldr (\x y -> x^2 + y) 0 [1,2,3]
Ergebnis: 14
```

Was genau vor sich geht:

```
foldr (\x y -> x^2 + y) 0 [1,2,3]
foldr (\x y -> x^2 + y) (3^2+0) [1,2]
foldr (\x y -> x^2 + y) (2^2+9) [1]
foldr (\x y -> x^2 + y) (1^2+13) []
foldr (\x y -> x^2 + y) 14 []
```

1.12 Lazy Evaluation

Haskell ist Faul! es berechnet nie das Ergebnis einer Funktion komplett sondern erst, wenn dies notwendig ist. Erst dadurch ist es möglich die Ranges zu verwenden um unendliche Listen zu erstellen. Es gibt zwei Methoden der Reduktion um Funktionen abzuarbeiten Innermost und Outermost. Haskell verwendet Outermost, da dies auch terminiert, sofern Innermost terminiert und es bleibt nicht in einer Endlosschleife hängen. Jedoch ist Outermost Reduktion ineffizient und benötigt mehr Schritte. Um diesen Fehler zu umgehen benutzt Haskell Sharing. Dadurch wird es genauso effizient wie Innermost Reduktion und endet nicht in einer Endlosschleife.

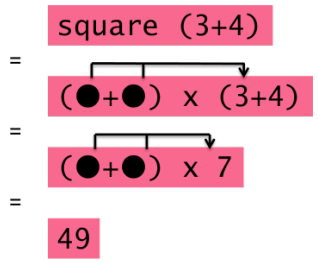

Abbildung 1.1: Sharing + Outermost

2 Prolog

3 C++

Abbildungsverzeichnis