hochschule coburg university of applied sciences

Fakultät Elektrotechnik und Informatik

Funktionale Programmierung am Beispiel Haskell

(Teil 2)

Verfasser: Jan Schöppach
Betreuender Dozent: Prof. Volkhard Pfeiffer
Abgabetermin: Dienstag, 23. Juni 2009

Funktionale Programmierung am Beispiel Haskell (Teil 2)

Jan Schöppach

23. Juni 2009

Inhaltsverzeichnis

0 Problemstellung **5**

1 Bedingungen **6**
1.1 Bewachte Gleichungen 6

2 Funktionen **7**
2.1 Die erste Funktion 7
2.1.1 Beispiel: Funktion in Java 8
2.2 Funktionen mit mehreren Argumenten 8
2.2.1 Currying 8
2.2.2 curry/uncurry 9
2.3 Rekursive Funktionen 9
2.3.1 Beispiel: Die Fibonacci-Folge 9
2.4 Polymorphismus 10
2.5 Überladen von Funktionen 11
2.5.1 Typklassen 11
2.5.2 Beispiel 11
2.6 Funktionen höherer Ordnung 12

3 Module **13**
3.1 Moduldefinierung 13
3.2 Import 13
3.3 Beispiel 14

4 Ein-/Ausgabe **15**
4.1 getChar & putChar 15

5 Fazit **16**

A Codebeispiele **18**

B Ehrenwörtliche Erklärung **19**

Abkürzungsverzeichnis

BASIC = **B**eginner's **A**ll-purpose **S**ymbolic **I**nstruction **C**ode

Anlagenverzeichnis

Anlage 1

Titelblatt

Anlage 2

Diese Arbeit, Quellen und Präsentation auf CD-ROM

Listings

1.1 Einfache Bedingung 6
1.2 Bewachte Gleichung 6
2.1 Einfache Funktion 7
2.2 Einfache Funktion in Java 8
2.3 Funktion mit mehreren Argumenten 8
2.4 Curried Funktion 9
2.5 Fibonacci-Folge 10
2.6 Fibonacci-Folge (Bedingung) 10
2.7 „bewachte“ Fibonacci-Folge 10
2.8 Überladene Funktion 12
2.9 Funktion höherer Ordnung 12
3.1 Quelltext-kopf Moduldefinierung 13
4.1 Einfache IO-Funktion 15
A.1 Alle Funktionen als Modul zusammengefasst 18

Kapitel 0

Problemstellung

Diese Seminararbeit führt in die funktionale Programmierung am Beispiel der rein funktionalen Programmiersprache Haskell ein. Es wird in Bedingungen, Funktionen, Module und die Ein- und Ausgabe von Daten eingegangen. Der Hauptaugenmerk liegt hierbei bei den verschiedenen Arten von Funktionen in Haskell.

Die funktionale Programmierung wird in diesem Teil der Arbeit nicht genauer erläutert, da dies in Teil 1 dieser Arbeit schon ausführlich behandelt wird.

Als Vorkenntnisse werden grundlegende Programmiersprachenkenntnisse, am besten Java, und die Seminararbeit „Funktionale Programmierung am Beispiel Haskell (Teil 1)“ von Dominik Wachter vorausgesetzt.

Kapitel 1

Bedingungen

Bedingungen in Haskell lassen sich BASIC-ähnlich schreiben:

Listing 1.1: Einfache Bedingung

```
1 if n == 1
2 then 1
3 else 0
```

„**Wenn** n gleich eins ist, **dann** gebe 1 aus, **ansonsten** 0."

Der Unterschied zu BASIC besteht darin, dass der else-Block in Haskell nicht weggelassen werden darf, sondern immer angegeben werden muss. [Hut07, S.31]

1.1 Bewachte Gleichungen

Eine zweite Möglichkeit der Bedingungen sind die sogenannten bewachten Gleichungen. Mit Hilfe dieser kann man unter mehreren Alternativen eine bestimmte auswählen, und dessen Definition zurückgeben lassen. Sie sind in etwa mit der „case"-Anweisung in Java gleichzusetzen und lassen sich wie folgt benutzen:

Listing 1.2: Bewachte Gleichung

```
1 |n == 1 = 1
2 |n == 10 = 2
3 |n == 100 = 3
4 |otherwise = 0
```

Diese Gleichung prüft der Reihe nach ob eine der Bedingungen „**True**" ist und gibt dann dessen Ergebnis zurück. Ist keiner der Fälle wahr wird die Definition des **otherwise**-Wächters benutzt. Dieser muss nicht zwingend vorhanden sein und gibt standardmäßig „**True**" zurück. [Hut07, S.31]

Kapitel 2

Funktionen

Funktionen sind die Basis funktionaler Programmierung. Wegen der Verhinderung von Nebenwirkungen geben diese in Haskell für die gleichen Argumente immer das gleiche Ergebnis. Dies vereinfacht die Programmierung und vor allem das Debugging ungemein.

2.1 Die erste Funktion

Funktionen lassen sich in Haskell wie folgt deklarieren:

Listing 2.1: Einfache Funktion

```haskell
toByte :: Int -> Int
toByte mb = mb * 1024 * 1024
```

Dieses einfache Codebeispiel wandelt die Größeneinheit Megabyte in Byte um und gibt dieses zurück.

In Zeile eins — der Funktionssignatur — werden der Funktionsname und die Datentypen der Eingangs- und Ausgangswerte definiert, was in Java oder C++ etwa dem Funktionskopf entspricht. Das erste „**Int**“ in dieser Zeile ist der Eingangswert, das zweite der Ausgangswert unserer Funktion.
Codezeile zwei ist die eigentliche Funktion. Hier wird festgelegt, dass „toByte“ die Variable „mb“ mit der darauf folgenden Definition zurückliefert.

Die Reihenfolge der Funktionssignatur (Z.1 des Beispiels) und der eigentlichen Funktion ist übrigens egal. Die Signatur kann auch weggelassen werden und wird dann vom Interpreter intern hinzugefügt, eine Angabe ist der Lesbarkeit halber aber zu empfehlen.
Das erste Zeichen eines Funktionsnamen muss immer ein kleingeschriebener Buchstabe sein, alle Folgenden können Klein- oder Großbuchstaben, Ziffern, Unterstriche oder einfache Anführungszeichen (') sein. [Hut07, S.14-15]

2.1.1 Beispiel: Funktion in Java

In Java wäre der Quelltext für dieses Beispiel folgender:

Listing 2.2: Einfache Funktion in Java

```
1 public int toByte (int mb)
2 {
3   return mb * 1024 * 1024;
4 }
```

2.2 Funktionen mit mehreren Argumenten

Funktionen mit mehreren Argumenten lassen sich durch Angabe dieser als ein Tupel[1] realisieren.

Listing 2.3: Funktion mit mehreren Argumenten

```
1 difference :: (Int, Int) → Int
2 difference (n,x) = x − n
```

Diese Funktion liefert die Differenz zwischen den Beiden übergebenen Werten.

Aufrufen lässt sich diese dann wie folgt:

```
> difference(10,20)
10
```

2.2.1 Currying

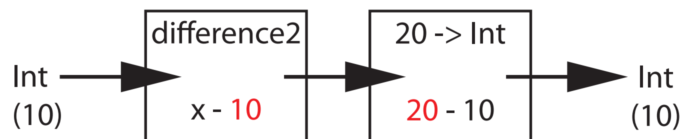
Mehrargumentige Funktionen lassen sich in Haskell auch als curried Funktionen definieren. Das Currying[2] bezeichnet die Umwandlung von Funktionen mit mehreren Argumenten der Form

```
(Int → Int) → Int
```

in Funktionen die als Rückgabewert wieder eine Funktion haben:

```
Int → Int → Int
```

Die Funktion bekommt also ein „**Int**“ als Eingangswert und gibt eine Funktion mit der Signatur „**Int** → **Int**“ zurück. Diese wird danach mit dem zweiten Argument ausgeführt. [Hut07, S.21-22]



[1] endliche Aneinanderreihung von Komponenten - siehe [Dom09, S.21]

[2] Der Ausdruck „Currying“ ist auf den Nachnamen des Mathematikers und Haskell-Namensgebers Haskell Brooks Curry zurückzuführen. [Hut07, S.6] [Wiki:Brooks]

Hier die „difference“-Funktion als Curried Funktion:

Listing 2.4: Curried Funktion

```
1 difference2 :: Int → Int → Int
2 difference2 n x = x − n
```

Diese lässt sich dann, analog der Standard Funktionen („**div**, **take**, **drop**“ etc.), folgendermaßen aufrufen:

```
difference2 10 20
```

oder

```
10 `difference2` 20
```

2.2.2 curry/uncurry

Im Prelude[3] gibt es die Befehle „**curry**“ und „**uncurry**“ um aus „curried“ Funktionen „uncurried“ Funktionen zu machen und umgekehrt.
So lässt sich die „difference“-Funktion zum Beispiel auch mit

```
curry difference 10 20
```

aufrufen und die „difference2“-Funktion mit:

```
uncurry difference2(10,20)
```

2.3 Rekursive Funktionen

Rekursive Funktionen sind Funktionen, welche sich selbst aufrufen um somit Probleme mit wiederkehrenden gleichen Lösungswegen zu Lösen. Sie sind in der funktionalen Programmierung essentiell, da Schleifen und Nebenwirkungen[4] nicht erlaubt sind.

2.3.1 Beispiel: Die Fibonacci-Folge

Ein geradezu prädestiniertes Beispiel für Rekursivität ist die Berechnung beliebiger Fibonacci-Zahlen.

„Die Folge 1, 1, 2, 3, 5, 8, 13, 21, 34, ... heißt die Fibonacci-Folge. Dabei erhält man das nächste Glied der Folge, indem man die zwei davor stehenden Zahlen addiert.“ [Beck04, S.4]

Mathematisch lässt sich das Ganze so schreiben:

$F_0 = 0$
$F_1 = 1$
$F_n = F_{n-1} + F_{n-2}$

[3] Die Prelude ist die Standard Funktionsbibliothek von Haskell. [Hut07, S.10]
[4] siehe [Dom09, S.12]

Diese mathematische Funktion lässt sich leicht auf Haskell portieren:

Listing 2.5: Fibonacci-Folge

```
fibonacci :: Int → Int
fibonacci 0 = 0
fibonacci 1 = 1
fibonacci n = fibonacci(n−1) + fibonacci(n−2)
```

Die Funktion ruft sich jedes Mal zweimal selbst auf, und führt dies solange fort, bis n kleiner oder gleich eins ist.

Mithilfe einer if-Bedingung lässt sich das Ganze auch schöner schreiben:

Listing 2.6: Fibonacci-Folge (Bedingung)

```
fibonacci :: Int → Int
fibonacci n = if n ≤ 1
              then n
              else fibonacci(n−1) + fibonacci(n−2)
```

Alternativ lässt sich die Funktion auch mithilfe von bewachten Gleichungen erstellen:

Listing 2.7: „bewachte“ Fibonacci-Folge

```
fibonacci :: Int → Int
fibonacci n
          |n ≤ 1 = n
          |otherwise = fibonacci(n−1) + fibonacci(n−2)
```

2.4 Polymorphismus

Polymorphismus[5] bezeichnet die Tatsache, dass Funktionen auch so Definiert werden können, dass sie für verschiedene Typen gültig ist.

Dies trifft zum Beispiel auf die meisten Standard Funktionen in Haskell zu. Hier ein paar Beispiele:

```
fst     :: (a, b) → a
head    :: [a] → a
take    :: Int → [a] → [a]
zip     :: [a] → [b] → [(a, b)]
id      :: a → a
```

[Hut07, S.23]

[5] Polymorphismus = „of many forms“ [Hut07, S.23] = vielförmig

Als expliziteres Beispiel geht Hutton in seinem Buch auf die „**length**“-Funktion ein. Diese liefert für Listen beliebigen Typs die Anzahl der Elemente.

```
> length [1, 3, 5, 7]
4
> length ["Yes", "No"]
2
> length [isDigit, isLower, isUpper]
3
```

[Hut07, S.23]

Der Typ der Funktion „**length**“ ist wie folgt definiert:

```
length :: [a] → Int
```

Als Eingangswert eine Liste beliebigen Typs und ein „**Int**“ als return-Wert. Typvariablen werden normalerweise einfach a, b, c usw. genannt. Generell lassen sich aber auch beliebige Variablennamen mit einem Kleinbuchstaben an erster Stelle benutzen. [Hut07, S.23]

2.5 Überladen von Funktionen

In Haskell gibt es Funktionen die verschiedene Typen aufnehmen, zum Beispiel der Operator „+“. Dieser kommt sowohl mit ganzzahligen Integer Werten als auch mit Fließkommazahlen zurecht. Die Funktionssignatur sieht dann so aus:

```
(+) :: Num a ⇒ a → a → a
```

Diese Definition ist ein polymorpher Typ, der aber noch den Zusatz hat, dass a von der Typklasse „**Num**“, d.H. eine numerische Zahl, sein muss.

2.5.1 Typklassen

Neben „**Num**“ gibt es noch weitere Typklassen. Hier eine Auswahl:

Klasse	Erklärung	Beispiel
Num	Numerische Zahlen	1, 2, 1.0, 1.5
Eq	Vergleichbare Typen	Bool, String, Int, Float
Read	Einlesbare Typen	Alles was man aus einem String ziehen kann
Show	Zeigbare Typen	Alles was man zu String konvertieren kann
Fractional	Brüche	$\frac{7.0}{2.0}$, $\frac{2}{3}$

[Hut07, S.23-28]

2.5.2 Beispiel

Mithilfe der Überladung können wir unsere „difference2“-Funktion auch allgemeiner definieren:

Listing 2.8: Überladene Funktion

```
1 difference2 :: Num a ⇒ a → a → a
2 difference2 n x = x − n
```

So nimmt diese nicht mehr nur Integer Werte entgegen, sondern alle aus der „**Num**"-Typklasse.

2.6 Funktionen höherer Ordnung

Wie schon beim Currying[6] beschrieben gibt Haskell die Möglichkeit Funktionen zu schreiben die eine Funktion als Returnwert haben. Daraus lässt sich leicht schließen dass es auch Funktionen geben muss die eine Funktion als Argument annehmen. Funktionen die eine Funktion zurückgeben, oder diese als Eingangswert erwartet nennt man Funktionen höherer Ordnung. Um den Sinn dieser Funktionen zu erklären hat Hutton in seinem Buch ein gutes Beispiel gebracht:

For example, a function that takes a function and a value, and returns the result of applying the function twice to the value, can be defined as follows:

Listing 2.9: Funktion höherer Ordnung

```
1 twice :: (a → a) → a → a
2 twice f x = f (f x)
```

For example:

```
> twice (*2) 3
12
```

[Hut07, S.61]

[6] siehe 2.1.1

Kapitel 3

Module

Ein Modul in Haskell ist eine Sammlung von Klassen, Funktionen und Typen auf die über ein festgelegtes Interface zugegriffen werden kann. In der Moduldefinierung lässt sich genau angeben auf was Zugriff gewährt wird und auf was nicht.

3.1 Moduldefinierung

Ein Modul lässt sich mit

module Modulname **where**

am Anfang der Datei einleiten. Darauf folgen die Klassen, Funktionen und Typen des Moduls. Dabei muss der Dateiname der *.hs-Datei genauso lauten wie der Name des Moduls, dass der Haskell-Interpreter dieses beim importieren findet.
Wollen wir nur bestimmte Elemente zum Zugriff freigeben, werden diese in einer Klammer hinter dem Modulnamen angeben:

Listing 3.1: Quelltext-kopf Moduldefinierung

```
1 module Modulname(
2             funktion1,
3             funktion2
4 ) where
```

Dieses Beispiel exportiert nur „funktion1“ und „funktion2“.

3.2 Import

Ist das Modul gespeichert, lässt es sich jetzt mit

import Modulname

in andere Haskell Quelldateien importieren und die freigegebenen Funktionen nutzen.

Will man nur bestimmte Elemente importieren lassen sich diese auch in Klammern festlegen:

```
import Modulname (funktion1)
```

Parallel dazu lässt durch den „**hiding**"-Tag auch angeben, dass alle bis auf bestimmte Elemente importiert werden sollen:

```
import Modulname hiding (funktion1)
```

Dies importiert — wie sicher schon angenommen — alle Elemente, die der Modul-Autor freigegeben hat, bis auf „funktion1".

3.3 Beispiel

Ein komplettes Beispiel zu Modulen befindet sich im Anhang[1]. Es zeigt unsere Funktionen die wir in der Arbeit definiert haben als Modul zusammengefasst. Nach außen hin sichtbar sind aber nur die „fibonacci2"-, die „difference2"- und die „twice"-Funktion.
Diese Datei lässt sich jetzt mit

```
import SimpleMath
```

in andere Codes importieren, und die 3 veröffentlichten Funktionen nach Belieben nutzen.

[1] siehe Codebeispiel A.1

Kapitel 4

Ein-/Ausgabe

Die Ein- und Ausgabe von Daten stellt in der funktionalen Programmierung eine Diskrepanz dar, da es wohl oder übel Nebenwirkungen geben muss. Dies verstößt eigentlich gegen die Prinzipien funktionaler Programmierung, weswegen ich diesen Bereich hier nur oberflächlich behandle.[1]

4.1 getChar & putChar

Für die Ein- und Ausgabe gibt es in Haskell die Funktionen „**getChar**“ und „**putChar**“. „**getChar**“ liest einen Char-Wert ein, und „**putChar**“ gibt Einen in der Konsole aus.

Listing 4.1: Einfache IO-Funktion

```
1 input :: IO ()
2 input = do c ← getChar
3            putChar '\n'
4            putChar c
```

Hier definieren wir eine Funktion „input“. Der Typ IO gibt an, dass es sich um eine Aktion handelt, und ist etwa mit dem „void“ in Java zu vergleichen.

Das „**do**“ Schlüsselwort startet eine Aneinanderreihung von Befehlen die dann der Reihe nach ausgeführt werden. Hierbei muss auf den richtigen Einschub geachtet werden, da der Compiler sonst die nachfolgenden Befehle nicht dem „**do**“ zugehörig macht.

Mit „←“ wird der „**getChar**“-Befehl an die Variable c „gebunden“, und mit „**putChar**“ am Ende wieder ausgegeben.

```
> input
a -- Das ist die Eingabe
a -- Das die Ausgabe
```

Wir starten die Funktion und der Interpreter wartet auf die Eingabe eines „Char“-Wertes. Wird dies eingegeben springt er mit '\n' in eine neue Zeile und gibt den eingegebenen Wert aus. [Hskl99, S.31/32]

[1] Mehr Informationen siehe [Hut07, ab S.87] und [Hskl99, ab S.31]

Kapitel 5

Fazit

Ich finde Haskell ist eine interessante Programmiersprache deren Grundlagen mit grundlegenden Programmierkenntnissen recht einfach zu erlernen ist und trotzdem einige neue Aspekte der Programmierung mit sich bringt.
Die Programmierung ohne Schleifen und Nebenwirkungen war am Anfang etwas gewöhnungsbedürftig, aber macht die Programme um einiges übersichtlicher und verständlicher.
Vor allem die Umwandlung mathematischer Funktionen in Haskell Funktionen hat sich als relativ einfach erwiesen.

Man muss aber trotzdem auch die negativen Effekte funktionaler Programmierung berücksichtigen, weswegen die Einsatzbereiche ziemlich eingeschränkt sind.

Literaturverzeichnis

[Hut07] Hutton, Graham: Programming in Haskell. 1. Aufl., Cambridge University Press, New York, 2007
(ISBN-13: 978-0-511-29615-4)

[OSu09] O'Sullivan, Bryan; Stewart, Donald Bruce; Goerzen, John: Real World Haskell. 1. Aufl., O'Reilly Media, Sebastopol, 2009
(ISBN-13: 978-0-596-51498-3)
Leider nur Onlinefassung als Quelle vorhanden:
http://book.realworldhaskell.org/read/

[Hskl99] Hudak, Paul; Peterson, John; Fasel Joseph: A Gentle Introduction to Haskell 98, 1999
http://haskell.org/tutorial/haskell-98-tutorial.pdf

[Dom09] Wachter, Dominik: Seminararbeit: Funktionale Programmierung am Beispiel Haskell (Teil 1), 2009

[Beck04] Becker, Johannes: Fibonacci und der goldene Schnitt, 2004
http://www.uni-giessen.de/~g013/goldfibo/goldfibo.pdf

[Wiki:Brooks] Wikipedia: o. V.: Haskell Brooks Curry.
Letzte Änderung am 2009-04-30T23:14:00+02:00:00
http://de.wikipedia.org/wiki/Haskell_Brooks_Curry

Anhang A

Codebeispiele

Listing A.1: Alle Funktionen als Modul zusammengefasst

```
module SimpleMath(
          fibonacci2,
          difference2,
          twice
) where

toByte :: Int → Int
toByte mb = mb * 1024 * 1024

difference :: (Int, Int) → Int
difference (n,x) = x - n

difference2 :: Num a ⇒ a → a → a
difference2 n x = x - n

fibonacci :: Int → Int
fibonacci n
          |n ≤ 1 = n
          |otherwise = fibonacci(n-1) + fibonacci(n-2)

fibonacci2 :: Int → Int
fibonacci2 n =  if n <= 1
                then n
                else fibonacci2(n-1) + fibonacci2(n-2)

twice :: (a → a) → a → a
twice f x = f (f x)
```

Dateiname: SimpleMath.hs

Anhang B

Ehrenwörtliche Erklärung

Ich versichere hiermit, dass ich meine Seminararbeit mit dem Thema

Funktionale Programmierung am Beispiel der Programmiersprache Haskell (Teil 2)

selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ort .... Datum ....

Unterschrift ....