Grundlagen der Funktionalen Programmierung

WS2223

Torben Tietgen

FH Wedel

17. Oktober 2023

Organisatorisches

Vorlesung

- gehalten von Torben Tietgen
- 12 Vorlesungseinheiten
- Klausur am Ende des Semesters
- Dienstags 09:30 im HS02
- 2 ECTS

Übung

- geleitet von Malte Heins
- ca. 9 Übungseinheiten
- eine oder zwei Wochen Bearbeitungszeit
- Dienstags 12:30 im PC02 und PC03
- 3 ECTS

Kommunikation

- entsprechende Foren in den Moodle Kurse:
  - Kurs für die Vorlesung
  - Kurs für die Übung
- E-Mail:
  - torben.tietgen@fh-wedel.de
  - malte.heins@fh-wedel.de
- Bitte **nicht** per Teams!
  Das ist nicht um euch zu ärgern, sondern weil da Nachrichten schnell mal unter gehen.

Vorbereitung

- Einschreiben in die Kurse in Moodle
- Einschreiben in die Kurse in myCampus
  - Sonst ist die Teilnahme an der Klausur nicht möglich
- Installieren eines Haskell Compilers[1]
  - der verbreitetste ist der Glasgow Haskell Compiler (GHC)
  - mittels ghcup unter Linux, MacOS, FreeBSD und Windows (WSL2 und nativ) möglich
  - unter Linux gibt es das Package haskell-platform, etwas veraltet aber reicht uns
- entscheiden für einen Editor[1]
  - viele Editoren unterstützen Haskell
  - Syntax-Highlighting reicht

[1] Weitere Informationen im Moodlekurs der Übung unter Verwendete Werkzeuge.

Literatur

- Learn You a Haskell
  A Beginner's Guide
  von Miran Lipovača
  ISBN: 9781593272838
  online: learnyouahaskell.com
- Real World Haskell
  Code You Can Believe in
  von Bryan O'Sullivan, John Goerzen und Don Stewart
  ISBN: 9780596514983
  online: book.realworldhaskell.org
- Haskell
  Wikibooks
  online: Haskell - Wikibooks

Literatur

Diese habe ich nicht selbst gelesen, wurden mir aber empfohlen.

- Programming in Haskell
  von Graham Hutton
  ISBN: 9781316626221
- Get Programming with Haskell
  von Will Kurt
  ISBN: 9781617293764

Grobe Gliederung der Vorlesungsinhalte

- Einführung
- Grundlagen Haskell
  - Datentypen
  - Funktionen
  - Auswertung
  - Partielle Funktionsanwendung
- Typklassen
- Datentypen definieren
- Module
- Rekursion
  - Rekursive Funktionen
  - Rekursive Datentypen
- List Comprehension
- Funktionen höherer Ordnung

Was ist funktionale Programmierung?

- alles sind Funktionen im mathematischen Sinne
  - auch Werte und Typen
- Funktionen sind First-Class-Objekte (engl. first-class citizens)
  - können auf Variablen gespeichert werden
  - können als Parameter an Funktionen übergeben werden
  - können von Funktionen als Rückgabewert zurückgegeben werden
  - können zur Laufzeit erzeugt werden
- es gibt nur Definitionen, keine Zuweisungen
  - eine Variable hat immer den gleichen Wert
- es gibt nur Funktionsanwendungen und -kompositionen, keine Anweisungen bzw. Anweisungsfolgen

zusätzlich bei reinen funktionalen Sprachen:

- Funktionen sind rein
  - bei gleicher Eingabe kommt immer das gleiche Ergebnis raus
  - besitzen keine Seiteneffekte
    - alles was eine Funktion tut ist ihren Rückgabewert berechnen

Haskell

- ist eine reine funktionale Sprache.
- ist faul (lazy).
  - erst wenn etwas benötigt wird, wird es berechnet
  - und auch nur soweit wie nötig
- ist statisch und streng getypt.
  - zur Übersetzungszeit sind alle Typen bekannt
  - keine dynamischen Typen
- hat ein kurze und prägnante Syntax.
- entstand um die Entwicklung verschiedener funktionaler Sprachen zu bündeln.

Motivation

Oder anders: Warum ist diese Veranstaltung sinnvoll?

- eine andere Herangehensweise an Probleme kennenlernen
- viele Konzepte aus der funktionalen Programmierung werden in andere Sprachen übernommen
- Haskell (und andere funktionale Sprachen) findet Anwendung in verschiedenen Unternehmen

Umgang mit dem GHCi

Der GHCi ist die interaktive Variante des GHC.

- Direktes interagieren mit Programmcode
- Laden von Haskell Code ohne vorheriges Compilieren

Starten des GHCi:

```
ghci [Dateien und Optionen]
```

Es kann direkt Haskell Code eingegeben und ausgewertet werden:

```
> 8 + 3
11
```

GHCi Befehle

Befehle, die kein Haskell sind, sondern den GHCi steuern.

`:quit`	beendet den GHCi, alternativ `Ctrl + D`
`:load`	lädt die angegebene Datei
`:reload`	lädt die aktuelle Datei neu
`:type`	ermittelt den Typen eines Wertes oder einer Funktion
`:info`	gibt Information zu einer benannten Entität aus
`:help`	gibt einen Hilfetext aus

Die Befehle beginnen immer mit `:`.
Zusätzlich können einige Befehle abgekürzt werden.
Beispiel: `:quit` als `:q`

Syntax

Haskell ist einrückungssensitive, das heißt anhand der Einrückung werden Blöcke definiert.

- gleiche Einrückungstiefe $\rightarrow$ gleicher Block
- weniger eingerückt $\rightarrow$ umliegender Block
- weiter eingerückt $\rightarrow$ eingeschlossener Block

Groß- und Kleinschreibung ist relevant.

- `isdigit` ist nicht das selbe wie `isDigit`.
- Funktionen und Variablen werden klein geschrieben und Typen werden groß geschrieben. Es gibt ein paar Ausnahmen, aber die behandeln wir wenn wir dazu kommen.

zwei Arten von Haskell

Es gibt zwei Formen von Haskell Quellcode Dateien:

- „normales“ Haskell
  - alles ist Code, außer Kommentare
  - Kommentare beginnen mit `--` oder sind umgeben mit `{-` und `-}`
  - Dateiendung: .hs
- Literate Haskell
  - nichts ist Code, außer es ist extra markiert
  - Codezeilen beginnen mit `>` oder sind umgeben mit `\begin{code}` und `\end{code}`
  - es muss immer eine Leerzeile zwischen Code und Text stehen
  - Dateiendung: .lhs

Datentypen

Ein Datentyp definiert eine bestimmte Menge von Werten.

Haskell ist statisch getypt, heißt während der Kompilierung werden alle Typen bestimmt.
Haskell ist streng getypt, heißt Werte können nicht implizit von einem Datentyp in einen anderen umgewandelt werden.

Beispiel: Wahrheitswerte

Datentyp:	`Bool`
Werte:	`False` und `True`

Notation:

```
e :: T
```

Der Ausdruck `e` besitzt den Typen `T`.

```
False :: Bool
True :: Bool
```

Einfache Datentypen

Einfach Werte, die für sich stehen.

- Wahrheitswerte
  - `Bool`
  - `False` (Falsch, Unwahr), `True` (Richtig, Wahr)
- Ganze Zahlen
  - `Int` : $[-2^{29} .. 2^{29} - 1]$
    (genaue Grenzen abhängig von der Implementierung)
  - `Word` : $[0 .. 2^{30} - 1]$
    (genaue Grenzen abhängig von der Implementierung)
  - `Integer` : keine obere und untere Grenze
  - 1, -125, 3168763
- Gleitkommazahlen
  - `Float` : einfache Genauigkeit
  - `Double` : doppelte Genauigkeit
  - 1.0, -0.3, 1.5e-2
- Zeichen
  - `Char` : Unicode Zeichen
  - 'a', 'ö', '\n', '\169', '©'

Komplexe Datentypen

Kombination aus mehreren Werten von (anderen) Typen.

- Tupel
  - Eine endliche Folge von Werten mit fester Größe.
  - Werte können unterschiedliche Typen haben.
  - `(1, 2)`, `('a', 'b', 'c')`, `(1, 'a')`, `(1, 'a', (True, False))`
- Listen
  - Eine Folge von Werten mit flexibler Größe.
  - Alle Werte haben den gleichen Typ.
  - `[]`, `[1, 2, 3]`, `[(1, 'a'), (2, 'b')]`
- Strings
  - Liste von Char
  - `['H', 'a', 'l', 'l', 'o']`, `"Hallo"`

Werte von komplexen Datentypen

Wie sehen die konkreten Typen von den Werten der vorherigen Folie aus?
Wir gehen an dieser Stelle davon aus, dass alle Zahlen vom Typ `Int` sind.

```
(1, 2) :: (Int, Int)
('a', 'b', 'c') :: (Char, Char, Char)
(1, 'a') :: (Int, Char)
(1, 'a', (True, False)) :: (Int, Char, (Bool, Bool))

[1, 2, 3] :: [Int]
[(1, 'a'), (2, 'b')] :: [(Int, Char)]
['H', 'a', 'l', 'l', 'o'] :: String oder [Char]
"Hallo" :: String oder [Char]
```

Den Typ von `[]` gucken wir uns später an, da uns noch etwas fehlt, um den Typ richtig angeben zu können.

Funktionenstypen

Eine Funktion bildet Argumente auf ein Resultat ab.
Der Typ einer Funktion ist diese Abbildung.

Dieser kann vom Compiler ermittelt oder manuell annotiert werden.
Wenn annotiert, kann der Compiler uns auf Fehler hinweisen!

Beispiele:

```
not :: Bool -> Bool
```
`not` bildet einen Wert vom Typ `Bool` auf einen Wert vom Typ `Bool` ab.

```
add :: Int -> Int -> Int
```
`add` bildet zwei Werte vom Typ `Int` auf einen Wert vom Typ `Int` ab.

Funktionensaufruf

Eine Funktion wird mit ihrem Namen referenziert.
Die Werte für die Parameter werden hinter den Funktionsnamen geschrieben.
Diese werden dabei nicht in Klammern zusammengefasst.
Die einzelnen Werte werden durch Leerzeichen getrennt.

```
> not True
False

> add 5 3
8
```

Unterschied zu Operatoren wie +?
Beispiel: `5 + 3`

Operatoren

Sind auch Funktionen, aber in infix Notation.
Sie stehen zwischen den Operanden, statt davor.

Jede Funktion mit (mindestens) zwei Parametern kann in der infix Notation verwendet werden.
Dafür muss der Funktionsname mit Backticks ` umgeben werden.

```
> 5 `add` 3
8
```

Jeder Operator kann in prefix Notation verwendet werden.
Dafür muss der Operator mit runden Klammern umgeben werden.

```
> (+) 5 3
8
```

Funktionsdefinition

Angabe des Typen gefolgt von der Definition, wie die Abbildung berechnet wird.

```
succ :: Int -> Int
succ x = x + 1

add :: Int -> Int -> Int
add x y = x + y

one :: Int
one = 1
```

Auf der linken Seite können Variablen definiert werden, auf der rechten Seite können diese dann genutzt werde.

Ein paar vordefinierte Funktionen und Operatoren

Auf die Angabe der Typen wird an dieser Stelle verzichtet, um nicht unnötig vor zu greifen.

Vergleiche:

`(==)`, `(/=)`	-	Gleichheitsprüfung
`(>)`, `(<)`	-	größer / kleiner als
`(>=)`, `(<=)`	-	größer / kleiner als oder gleich

Logik:

`(&&)`, `(||)`	-	und, oder
`not`	-	Negation

Arithmetik:

`(+)`, `(-)`, `(*)`	-	Addition, Subtraktion/Negation, Multiplikation
`(/)`	-	Division ohne Rest
`div`, `mod`	-	Division mit Rest, mathematisch[1]
`quot`, `rem`	-	Division mit Rest, symmetrisch[1]
`even`, `odd`	-	gerade, ungerade

[1] Für die Division mit Rest mit negativen Zahlen gibt es unterschiedliche Definitionen. Programmiersprachen unterscheiden sich, welche Definition implementiert ist. Für weitere Informationen siehe den entsprechenden Wikipedia Artikel.

Anonyme Funktionen

Anonyme Funktionen sind Funktionen, denen kein Name gegeben wurde.
Das ist dann sinnvoll, wenn eine Funktion nur an einer Stelle benötigt wird.

```
-- entspricht succ
\x -> x + 1

-- entspricht add
\ x y -> x + y
```

Die Idee dahinter entstammt dem Lambda-Kalkül. In der Mathematik wird deswegen das kleine griechischen Lambda λ genutzt, um eine anonyme Funktion zu beginnen.
\ ist die „vereinfachte Darstellung“, die leicht mit einer Tastatur eingegeben werden kann.

$\lambda x.x + 1$

Typvariablen

Wenn der Typ von einem Wert für eine Funktion nicht relevant ist, kann dieser mit einem Platzhalter angegeben werden.
Diese Platzhalter werden **Typvariablen** genannt.
Typischerweise werden für diese Platzhalter die Buchstaben am Anfang des Alphabets genutzt, weswegen diese meistens nicht als Variablenbezeichner genutzt werden.

Beispiele:

```
fst :: (a, b) -> a   -   Zugriff auf das erste Element eines 2-Tupel
snd :: (a, b) -> b   -   Zugriff auf das zweite Element eines 2-Tupel
```

Ein paar vordefinierte Listenfunktionen und -operatoren

`(:) :: a -> [a] -> [a]`
ein Element vorne an eine Liste anfügen

`(++) :: [a] -> [a] -> [a]`
zwei Listen hintereinander hängen

`head, last :: [a] -> a`
liefert das erste bzw. letzte Element einer Liste

`tail, init :: [a] -> [a]`
liefert die Liste ohne das erste bzw. letzte Element

`null :: [a] -> Bool`
Prüfung, ob eine Liste leer ist

`length :: [a] -> Int`
liefert die Länge einer Liste