Praktische Informatik 3: Funktionale Programmierung
Vorlesung 1 vom 13.10.2014: Einführung

Christoph Lüth

Universität Bremen

Wintersemester 2014/15

Personal

- Vorlesung:
  Christoph Lüth `cxl@informatik.uni-bremen.de`
  MZH 3110, Tel. 59830
- Tutoren:
  Jan Radtke `jradtke@informatik.uni-bremen.de`
  Sandor Herms `sanherms@informatik.uni-bremen.de`
  Daniel Müller `dmueller@informatik.uni-bremen.de`
  Felix Thielke `fthielke@informatik.uni-bremen.de`
  Sören Schulze `sschulze@informatik.uni-bremen.de`
  Henrik Reichmann `henrikr@informatik.uni-bremen.de`
- Fragestunde:
  Berthold Hoffmann `hof@informatik.uni-bremen.de`
- Webseite: `www.informatik.uni-bremen.de/~cxl/lehre/pi3.ws14`

Termine

- Vorlesung: Di 12 – 14, MZH 1380/1400
- Tutorien:

Mi	08 – 10	MZH 1110	Sören Schulze
Mi	10 – 12	MZH 1470	Sandor Herms
Mi	12 – 14	MZH 1110	Henrik Reichmann
Mi	14 – 16	SFG 1020	Felix Thielke
Do	08 – 10	MZH 1110	Jan Radtke
Do	10 – 12	MZH 1090	Daniel Müller

- Fragestunde : Di 10 – 12 Berthold Hoffmann (Cartesium 2.048)
- Anmeldung zu den Übungsgruppen über stud.ip

Übungsbetrieb

- Ausgabe der Übungsblätter über die Webseite Dienstag abend
- Besprechung der Übungsblätter in den Tutorien
- Bearbeitungszeit: eine Woche
- Abgabe: elektronisch bis Freitag nächste Woche 12:00
- Elf Übungsblätter (voraussichtlich) plus 0. Übungsblatt
- Übungsgruppen: max. drei Teilnehmer (nur in Ausnahmefällen vier)

Scheinkriterien

- Von n Übungsblättern werden $n-1$ bewertet (geplant $n = 11$)
- Insgesamt mind. 50% aller Punkte
- Notenspiegel (in Prozent aller Punkte):

Pkt.%	Note	Pkt.%	Note	Pkt.%	Note	Pkt.%	Note
		89.5-85	1.7	74.5-70	2.7	59.5-55	3.7
≥ 95	1.0	84.5-80	2.0	69.5-65	3.0	54.5-50	4.0
94.5-90	1.3	79.5-75	2.3	64.5-60	3.3	49.5-0	n/b

- Fachgespräch (Individualität der Leistung) am Ende

Spielregeln

- Quellen angeben bei
  - Gruppenübergreifender Zusammenarbeit;
  - Internetrecherche, Literatur, etc.
- Erster Täuschungsversuch: Null Punkte
- Zweiter Täuschungsversuch: Kein Schein.
- Deadline verpaßt?
  - Triftiger Grund (z.B. Krankheit mehrerer Gruppenmitglieder)
  - Vorher ankündigen, sonst null Punkte.

Fahrplan

- Teil I: Funktionale Programmierung im Kleinen
  - Einführung
  - Funktionen und Datentypen
  - Rekursive Datentypen
  - Typvariablen und Polymorphie
  - Funktionen höherer Ordnung I
  - Funktionen höherer Ordnung II
  - Typinferenz
- Teil II: Funktionale Programmierung im Großen
- Teil III: Funktionale Programmierung im richtigen Leben

Warum funktionale Programmierung lernen?

- Denken in Algorithmen, nicht in Programmiersprachen
- Abstraktion: Konzentration auf das Wesentliche
- Wesentliche Elemente moderner Programmierung:
  - Datenabstraktion und Funktionale Abstraktion
  - Modularisierung
  - Typisierung und Spezifikation

The Future is Bright — The Future is Functional

- Blick über den Tellerrand — Blick in die Zukunft
- Studium $\neq$ Programmierkurs — was kommt in 10 Jahren?
- Funktionale Programmierung ist bereit für die Herausforderungen der Zukunft:
  - Nebenläufige Systeme (Mehrkernarchitekturen)
  - Vielfach vernetzte Rechner („Internet der Dinge“)
  - Große Datenmengen („Big Data“)

Warum Haskell?

- Moderne Sprache
- Standardisiert, mehrere Implementationen
  - Interpreter: `ghci`, `hugs`
  - Compiler: `ghc`, `nhc98`
- Rein funktional
  - Essenz der funktionalen Programmierung

Geschichtliches

- Grundlagen 1920/30
  - Kombinatorlogik und λ-Kalkül (Schönfinkel, Curry, Church)
- Erste funktionale Programmiersprachen 1960
  - LISP (McCarthy), ISWIM (Landin)
- Weitere Programmiersprachen 1970– 80
  - FP (Backus); ML (Milner, Gordon); Hope (Burstall); Miranda (Turner)
- Konsolidierung 1990
  - CAML, Formale Semantik für Standard ML
  - Haskell als Standardsprache
- Kommerzialisierung 2010
  - Scala, Clojure, F#

Programme als Funktionen

- Programme als Funktionen

$$P : \mathit{Eingabe} \to \mathit{Ausgabe}$$

- Keine veränderlichen Variablen — kein versteckter Zustand
- Rückgabewert hängt ausschließlich von Werten der Argumente ab, nicht vom Aufrufkontext (referentielle Transparenz)
- Alle Abhängigkeiten explizit

Beispiel: Programmieren mit Funktionen

- Programme werden durch Gleichungen definiert:

```
fac n = if n == 0 then 1
        else n* fac(n–1)
```

- Auswertung durch Reduktion von Ausdrücken:

```
fac 2  → if 2 == 0 then 1 else 2* fac (2-1)
       → if False then 1 else 2* fac 1
       → 2* fac 1
       → 2* if 1 == 0 then 1 else 1* fac (1-1)
       → 2* if False then 1 else 1* fac 0
       → 2* 1* fac 0
       → 2* 1* if 0 == 0 then 1 else 0* fac (0-1)
       → 2* 1* if True then 1 else 0* fac (-1)
       → 2* 1* 1 → 2
```


Beispiel: Nichtnumerische Werte

- Rechnen mit Zeichenketten

```
repeat n s = if n == 0 then ""
             else s ++ repeat (n–1) s
```

- Auswertung:

```
repeat 2 "hallo "
→ if 2 == 0 then "" else "hallo " ++ repeat (2-1) "hallo "
→ "hallo "++ repeat 1 "hallo "
→ "hallo "++  if 1 == 0 then ""
              else "hallo "++ repeat (1-1) "hallo "
→ "hallo "++ ("hallo "++ repeat 0 "hallo ")
→ "hallo "++ ("hallo "++  if 0 == 0 then ""
                          else repeat (0-1) "hallo ")
→ "hallo "++ ("hallo " ++ "")
→ "hallo hallo "
```


Auswertung als Ausführungsbegriff

- Programme werden durch Gleichungen definiert:

$$f(x) = E$$

- Auswertung durch Anwenden der Gleichungen:
  - Suchen nach Vorkommen von f, e.g. $f(t)$
  - $f(t)$ wird durch $E\left[\begin{array}{c} t \\ x \end{array}\right]$ ersetzt
- Auswertung kann divergieren!

Ausdrücke und Werte

- Nichtreduzierbare Ausdrücke sind Werte
- Vorgebenene Basiswerte: Zahlen, Zeichen
  - Durch Implementation gegeben
- Definierte Datentypen: Wahrheitswerte, Listen, ...
  - Modellierung von Daten

Typisierung

- Typen unterscheiden Arten von Ausdrücken und Werten:

  `repeat n s = ...` `n` Zahl
  `s` Zeichenkette

- Verschiedene Typen:
  - Basistypen (Zahlen, Zeichen)
  - strukturierte Typen (Listen, Tupel, etc)
- Wozu Typen?
  - Typüberprüfung während Übersetzung erspart Laufzeitfehler
  - Programmsicherheit

Signaturen

- Jede Funktion hat eine Signatur

```
fac      :: Integer→ Integer
```

```
repeat :: Int→ String→ String
```

- Typüberprüfung
  - `fac` nur auf `Int` anwendbar, Resultat ist `Int`
  - `repeat` nur auf `Int` und `String` anwendbar, Resultat ist `String`

Übersicht: Typen in Haskell

Typ	Bezeichner	Beispiel		
Ganze Zahlen	`Int`	`0`	`94`	`-45`
Fließkomma	`Double`	`3.0`	`3.141592`	
Zeichen	`Char`	`'a' 'x'`	`'\034'`	`'\n'`
Zeichenketten	`String`	`"yuck"`	`"hi\nho\"\n"`	
Wahrheitswerte	`Bool`	`True`	`False`	
Funktionen	`a -> b`			

- Später mehr. Viel mehr.

Das Rechnen mit Zahlen

Beschränkte Genauigkeit, konstanter Aufwand ⟷ beliebige Genauigkeit, wachsender Aufwand

Haskell bietet die Auswahl:

- Int - ganze Zahlen als Maschinenworte (≥ 31 Bit)
- Integer - beliebig große ganze Zahlen
- Rational - beliebig genaue rationale Zahlen
- Float, Double - Fließkommazahlen (reelle Zahlen)

Ganze Zahlen: Int und Integer

- Nützliche Funktionen (überladen, auch für Integer):

```
+, *, ^, - :: Int→ Int→ Int
abs        :: Int→ Int — Betrag
div, quot  :: Int→ Int→ Int
mod, rem   :: Int→ Int→ Int
```

  Es gilt $(\text{div } x\ y) * y + \text{mod } x\ y == x$

- Vergleich durch $==, \neq, \leq, <, \ldots$
- Achtung: Unäres Minus
  - Unterschied zum Infix-Operator $-$
  - Im Zweifelsfall klammern: abs (-34)

Fließkommazahlen: Double

- Doppeltgenaue Fließkommazahlen (IEEE 754 und 854)
  - Logarithmen, Wurzel, Exponentation, π und e, trigonometrische Funktionen
- Konversion in ganze Zahlen:
  - fromIntegral :: Int, Integer→ Double
  - fromInteger :: Integer→ Double
  - round, truncate :: Double→ Int, Integer
  - Überladungen mit Typannotation auflösen:

```
round (fromInt 10) :: Int
```

- Rundungsfehler!

Alphanumerische Basisdatentypen: Char

- Notation für einzelne Zeichen: 'a',...
- Nützliche Funktionen:

```
ord :: Char → Int
chr :: Int → Char

toLower :: Char→ Char
toUpper :: Char→ Char
isDigit :: Char→ Bool
isAlpha :: Char→ Bool
```

- Zeichenketten: String

Zusammenfassung

- Programme sind Funktionen, definiert durch Gleichungen
  - Referentielle Transparenz
  - kein impliziter Zustand, keine veränderlichen Variablen
- Ausführung durch Reduktion von Ausdrücken
- Typisierung:
  - Basistypen: Zahlen, Zeichen(ketten), Wahrheitswerte
  - Strukturierte Typen: Listen, Tupel
  - Jede Funktion `f` hat eine Signatur `f :: a → b`