Praktische Informatik 3: Funktionale Programmierung Vorlesung 1 vom 16.10.2012: Einführung

Christoph Lüth

Universität Bremen

Wintersemester 2012/13

Rev. 1935

1 [21]

Personal

- ▶ **Vorlesung:**
Christoph Lüth <cx1@informatik.uni-bremen.de>
MZH 3110, Tel. 59830
- ▶ **Tutoren:**
Marcus Ermler <@maermler@informatik.uni-bremen.de>
Christian Maeder <Christian.Maeder@dfki.de>
Martin Ring <maring@informatik.uni-bremen.de>
Diedrich Wolter <dwolter@informatik.uni-bremen.de>
- ▶ **Fragestunde:**
Berthold Hoffmann <hof@informatik.uni-bremen.de>
- ▶ **Webseite:** www.informatik.uni-bremen.de/~cx1/lehre/pi3.ws12

2 [21]

Termine

- ▶ **Vorlesung:** Di 14 – 16, MZH 1380/1400
- ▶ **Tutorien:**
Mi 16 – 18 OAS 3000 Marcus Ermler
Do 8 – 10 MZH 1110 Marcus Ermler
Do 10 – 12 MZH 7260 Christian Maeder
Do 10 – 12 MZH 1470 Diedrich Wolter
Do 12 – 14 MZH 1450 Martin Ring
Do 12 – 14 MZH 1460 Diedrich Wolter
- ▶ **Fragestunde:** Fr 9 – 11 Berthold Hoffmann (Cartesium 2.048)
- ▶ **Anmeldung** zu den Übungsgruppen über stud.ip

3 [21]

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)

4 [21]

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

- ▶ **Fachgespräch** (Individualität der Leistung) am Ende

5 [21]

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.

6 [21]

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**

7 [21]

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
- ▶ Blick über den Tellerrand — Blick in die Zukunft
- ▶ Studium $\neq$ Programmierkurs — was kommt in 10 Jahren?

8 [21]

Warum Haskell?

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

9 [21]

Geschichtliches

- ▶ **Grundlagen** 1920/30
 - ▶ Kombinatorlogik und λ -Kalkül (Schönfinkel, Curry, Church)
- ▶ Erste **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, F#, Clojure

10 [21]

Programme als Funktionen

- ▶ Programme als Funktionen
$$P : \text{Eingabe} \rightarrow \text{Ausgabe}$$
- ▶ Keine veränderlichen **Variablen** — kein versteckter **Zustand**
- ▶ Rückgabewert hängt ausschließlich von Werten der Argumente ab, nicht vom Aufrufkontext (**referentielle Transparenz**)
- ▶ Alle **Abhängigkeiten** explizit

11 [21]

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)
      ~> 2 * fac (2-1)
      ~> 2 * fac 1
      ~> 2 * (if 1 == 0 then 1 else 1 * fac (1-1))
      ~> 2 * 1 * fac (1-1)
      ~> 2 * 1 * fac 0
      ~> 2 * 1 * (if 0 == 0 then 1 else 0 * fac (0-1))
      ~> 2 * 1 * 1 ~> 2
```

12 [21]

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

13 [21]

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 $E \left[\begin{matrix} t \\ x \end{matrix} \right]$ ersetzt

- ▶ **Auswertung** kann **divergieren!**

14 [21]

Ausdrücke und Werte

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

15 [21]

Bedeutung (Semantik) von Programmen

- ▶ **Operationale** Semantik:

- ▶ Durch den **Ausführungsbegriff**
- ▶ Ein Programm ist, was es tut

- ▶ **Denotationelle** Semantik:

- ▶ Programme werden auf **mathematische Objekte** abgebildet (Denotat)
- ▶ Für funktionale Programme: **rekursiv** definierte Funktionen

Äquivalenz von operationaler und denotationaler Semantik

Sei P ein funktionales Programm, $\rightsquigarrow_P$ die dadurch definierte Reduktion, und $\llbracket P \rrbracket$ das Denotat. Dann gilt für alle Ausdrücke t und Werte v

$$t \rightsquigarrow_P v \iff \llbracket P \rrbracket(t) = v$$

16 [21]

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

17 [21]

Signaturen

- ▶ Jede Funktion hat eine Signatur

```
fac :: Int → Int
```

```
repeat :: Int → String → String
```

- ▶ Typüberprüfung

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

18 [21]

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.

19 [21]

Imperativ vs. Funktional

- ▶ Imperative Programmierung:

- ▶ Zustandsübergang $\Sigma \rightarrow \Sigma$, Lesen/Schreiben von Variablen:
- ▶ Kontrollstrukturen: Fallunterscheidung if ... then ... else
Iteration while ...

- ▶ Funktionale Programmierung:

- ▶ Funktionen $f : E \rightarrow A$
- ▶ Kontrollstrukturen: Fallunterscheidung
Rekursion

20 [21]

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 $f :: a \rightarrow b$

21 [21]