Praktische Informatik 3

Einführung in die Funktionale Programmierung

Vorlesung vom 31.10.2006: Einführung

Christoph Lüth

WS 06/07

Personal

- *Vorlesung*: Christoph Lüth `<cxl>`,
  Cartesium 2.046, Tel. 64223
- Tutoren: Matthias Berger `<tokio>`
  Klaus Hartke `<hartke>`
  Cui Jian `<ken>`
  Friederike Jolk `<rikej>`
  Christian Maeder `<maeder>`
  Diedrich Wolter `<dwolter>`
- Fragestunde: Berthold Hoffmann `<hof>`
- Website:
  `www.informatik.uni-bremen.de/~cxl/lehre/pi3.ws06`.
- Newsgroup: `fb3.lv.pi3`.

Termine

- *Vorlesung*:
  Di 8 – 10, NW1 H2 (W0020)
- *Tutorien*:

Di	17 – 19	MZH 1380	Diedrich Wolter
Mi	8 – 10	MZH 7250	Friederike Jolk
		MZH 7260	Cui Jian
	13 – 15	MZH 7210	Klaus Hartke
		MZH 8090	Christian Maeder
Mi	17 – 19	MZH 4194	Matthias Berger

- *Fragestunde* (FAQ):
  Di 10 – 12 Berthold Hoffmann (Cartesium 2.048)

Übungsbetrieb

- Ausgabe der Übungsblätter über die Webseite *Dienstag nachmittag*
- Besprechung der Übungsblätter in den Tutorien
- *Bearbeitungszeit* zwei Wochen
- *Abgabe* vor der Vorlesung
- *Sechs* Übungsblätter (und ein Bonusblatt)
- Übungsgruppen: max. *drei Teilnehmer* (nur in Ausnahmefällen vier)

Inhalt der Veranstaltung

- *Deklarative* und *funktionale* Programmierung
  - Betonung auf *Konzepten* und *Methodik*
- Bis Weihnachten: *Grundlagen*
  - *Funktionen*, *Typen*, *Funktionen höherer Ordnung*, *Polymorphie*
- Nach Weihnachten: *Ausweitung* und *Anwendung*
  - z.B: formale *Korrektheit*; Nebenläufigkeit; *Grafik* und *Animation*;
- *Lektüre*:
  Simon Thompson: *Haskell — The Craft of Functional Programming* (Addison-Wesley, 1999)

Scheinkriterien — Vorschlag:

- *Bearbeitung* eines Übungsblattes: $\geq 40\%$
- *Alle Übungsblätter* sind zu bearbeiten.
- *Insgesamt* mind. 50% aller Punkte
- Es gibt ein *Bonusübungsblatt*, um Ausfälle zu kompensieren.
- *Individualität der Leistung* wird sichergestellt durch:
  - *Prüfungsgespräch* (auch auf Wunsch)
  - *Vorstellung* einer *Lösung* im Tutorium
  - *Beteiligung* im Tutorium

Spielregeln

- *Quellen angeben* bei
  - *Gruppenübergreifender* Zusammenarbeit;
  - Internetrecherche, Literatur, etc.
- Erster *Täuschungsversuch*:
  - *Null* Punkte
  - Fachgespräch.
- Zweiter *Täuschungsversuch*: *Kein Schein*.
- *Deadline verpaßt?*
  - *Vorher* ankündigen, sonst *null* Punkte.

Einführung in die Funktionale Progammierung

Warum funktionale Programmierung (FP) lernen?

- *Abstraktion*
  - Denken in *Algorithmen*, nicht in *Programmiersprachen*
- FP konzentriert sich auf *wesentlichen* Elemente moderner Programmierung:
  - *Datenabstraktion*
  - *Modularisierung* und *Dekomposition*
  - *Typisierung* und *Spezifikation*
- Blick über den Tellerrand — Blick in die Zukunft
- Studium $\neq$ Programmierkurs — was kommt in 10 Jahren?

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), später *SML* und *CAML*; *Hope* (Burstall); *Miranda* (Turner)
- 1990: *Haskell* als *Standardsprache*

Referentielle Transparenz

- Programme als Funktionen

$$P : Eingabe \to Ausgabe$$

- Keine *Variablen* — keine *Zustände*
- Alle *Abhängigkeiten explizit*:
- Rückgabewert hängt ausschließlich von Werten der Argumente ab, nicht vom Aufrufkontext:

Referentielle Transparenz

Funktionen als Programme

Programmieren durch Rechnen mit Symbolen:

$$\begin{aligned} 5 * (7 - 3) + 4 * 3 &= 5 * 4 + 12 \\ &= 20 + 12 \\ &= 32 \end{aligned}$$

Benutzt *Gleichheiten* ($7 - 3 = 4$ etc.), die durch Definition von $+$, $*$, $-$, ... gelten.

Programmieren mit Funktionen

- *Programme* werden durch *Gleichungen* definiert:

```
inc x = x+ 1
addDouble x y = 2*(x+ y)
```

- Auswertung durch *Reduktion* von *Ausdrücken*:

`addDouble 6 4` $\leadsto$`2*(6+ 4)` $\leadsto$`20`

- Nichtreduzierbare Ausdrücke sind *Werte*
  - *Zahlen*, *Zeichenketten*, *Wahrheitswerte*, ...

Auswertungsstrategie

- Von *außen* nach *innen*, *links* nach *rechts*.

```
inc (addDouble (inc 3) 4)
```
$\leadsto$`(addDouble (inc 3) 4)+ 1`
$\leadsto$`2*(inc 3+ 4)+ 1`
$\leadsto$`2*(3+ 1+ 4)+ 1`
$\leadsto$`2*8+1` $\leadsto$`17`

- Entspricht *call-by-need* (verzögerte Auswertung)
  - Argumentwerte werden erst ausgewertet, wenn sie benötigt werden.

Nichtnumerische Werte

Rechnen mit Zeichenketten:

```
  repeater s = s ++ s
repeater (repeater "hallo ")
```
$\leadsto$`repeater "hallo"++ repeater "hello"`
$\leadsto$`("hallo "++ "hallo ")++ repeater "hallo "`
$\leadsto$`"hallo hallo "++ repeater "hallo "`
$\leadsto$`"hallo hallo "++ ("hallo "++ "hallo ")`
$\leadsto$`"hallo hallo hallo hallo"`

Typisierung

Typen unterscheiden Arten von Ausdrücken

- *Basistypen* (Zahlen, Zeichen)
- *strukturierte Typen* (Listen, Tupel, etc)

Wozu Typen?

- *Typüberprüfung* während *Übersetzung* erspart *Laufzeitfehler*
- *Programmsicherheit*

Übersicht: Typen in Haskell

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`
Listen	`[a]`	`[6, 9, 20]` `["oh", "dear"]`
Tupel	`(a, b)`	`(1, 'a') ('a', 4)`
Funktionen	`a-> b`	

Definition von Funktionen

- Zwei wesentliche *Konstrukte*:
  - Fallunterscheidung
  - Rekursion
- Jede Funktion hat *Signatur*
- *Beispiel*:

```
fac :: Int-> Int
fac n = if n == 0 then 1
        else n * (fac (n-1))
```

- Auswertung kann *divergieren*!

Haskell in Aktion: `hugs`

- `hugs` ist ein Haskell-Interpreter
  - Klein, schnelle Übersetzung, gemächliche Ausführung.
- `ghc` ist ein Haskell-*Compiler*:
  - Groß, langsame Übersetzung, schnelle Ausführung.
- Funktionsweise:
  - `hugs` und `ghc` lesen *Definitionen* (Programme, Typen, ...) aus Datei (*Skript*)
  - *Kommandozeilenmodus*: Reduktion von Ausdrücken
  - *Keine Definitionen* in der Kommandozeile
  - Haskell in Aktion.

Zusammenfassung

- Haskell ist eine *funktionale Programmiersprache*
- *Programme* sind *Funktionen*, definiert durch *Gleichungen*
  - Referentielle Transparenz — keine Zustände oder 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`