

# Funktionale Programmierung

## Einführung

Prof. Dr. Oliver Braun

Fakultät für Informatik und Mathematik  
Hochschule München

Letzte Änderung: 09.10.2018 06:54

### Inhaltsverzeichnis

Was ist imperative Programmierung . . . . .	1
Was ist funktionale Programmierung . . . . .	2
Die Idee ist schon lange bekannt . . . . .	2
First-class functions . . . . .	2
$\lambda$ -Kalkül als Basis für Haskell . . . . .	3
Was ist eine Funktion? . . . . .	3
$\lambda$ -Ausdrücke . . . . .	3
$\alpha$ -Konversion . . . . .	4
$\beta$ -Reduktion . . . . .	4
Übung . . . . .	5
Freie Variablen . . . . .	5
Mehrere Argumente . . . . .	5
$\eta$ -Konversion . . . . .	6
Auswertung . . . . .	6
Kombinatoren . . . . .	6

### Was ist imperative Programmierung

- zentrale Idee: Programmieren mit Zustandsänderungen, z.B. in Java

```
int sum = 0;
for (int i = 0; i < a.length(); i++) {
```

```
    sum += a[i];  
}
```

- die **Kernidee** ist: `sum` wird so lange geändert, bis der richtige Wert darin steht
- implementiert wird das **wie**

## Was ist funktionale Programmierung

- Paradigma das auf (mathematischen) Funktionen aufbaut, z.B.

$$f(x, y) = x + 3y$$

- das Ergebnis einer mathematischen Funktion ist nur abhängig von ihren Parametern
  - $f(2, 1) = 2 + 3 \cdot 1 = 5$
- **keine** Seiteneffekte! **referentiell transparent**
- zentrale Idee: Programmieren mit (**unveränderbaren**) Werten
- deklarativ: implementiert wird nur das **was**

## Die Idee ist schon lange bekannt

- 1955: LISP = erste funktionale Programmiersprache
- John Backus
  - 1978: [Can Programming Be Liberated from the von Neumann Style?](#)
  - entwickelte Fortran und später Algol
- John Hughes
  - 1990: [Why Functional Programming Matters](#)

## First-class functions

- Funktionen sind *first-class citizens*
  - Funktionen sind Ausdrücke, die wie alle anderen Werte behandelt werden können
  - z.B.
    - \* Liste von Funktionen
    - \* Funktionen als Parameter für Funktionen

\* Funktionen als Ergebnis von Funktionen

- Funktionen werden auf Argumente *angewendet* und *berechnet*
  - engl. *evaluated, reduced*

## $\lambda$ -Kalkül als Basis für Haskell

- die Grundlage für eine funktionale Sprache ist der  $\lambda$ -Kalkül
  - im Gegensatz dazu ist die Grundlage für eine imperative Sprache die Turingmaschine
- **Haskell** basiert ausschließlich auf dem  $\lambda$ -Kalkül
  - Haskell ist **pure**
  - hoher Grad an Abstraktionen und Komponierbarkeit
- Haskell-Programme werden aus unabhängigen Funktionen zusammengesetzt
  - ähnlich wie Lego
- minimalistische Syntax

## Was ist eine Funktion?

- Relation zwischen
  - einer Menge von Eingaben (Definitionsbereich, *domain*) und
  - einer Menge von Ausgaben (Wertebereich, *codomain*)
- Beispiel

$$f(1) = A$$

$$f(2) = B$$

$$f(3) = C$$

- Definitionsbereich  $\{1, 2, 3\}$
- Wertebereich  $\{A, B, C\}$

## $\lambda$ -Ausdrücke

- eine  $\lambda$ -**Abstraktion** entspricht einer Funktion ohne Namen
  - anonyme Funktion
- Beispiel Identitätsfunktion

$$\lambda x.x$$

- das  $\lambda$  gefolgt von der **Variablen**  $x$  heißt **Kopf** (engl. *head*)
- der **Rumpf** (engl. *body*) ist der ( $\lambda$ -)Ausdruck nach dem Punkt
- das  $x$  im Kopf heißt **Parameter** und **bindet** alle Vorkommen von  $x$  im Rumpf

### $\alpha$ -Konversion

- die  $\alpha$ -Konversion formalisiert die Regel, dass Namen “Schall und Rauch” sind
- d.h. die folgenden  $\lambda$ -Ausdrücke beschreiben alle die selbe Funktion:

$$\lambda x.x$$

$$\lambda y.y$$

$$\lambda m.m$$

- Achtung: Es können auch freie (nicht gebundene) Variablen in  $\lambda$ -Ausdrücken vorkommen (siehe später). Diese dürfen durch  $\alpha$ -Konversion nicht gebunden werden.

### $\beta$ -Reduktion

- formalisiert die Anwendung einer Funktion auf ein Argument
- Beispiel: Anwendung der Identitätsfunktion auf die Zahl 42

$$(\lambda x.x) 42$$

- Achtung:  $\lambda$ -Ausdruck muss geklammert werden, da sonst die 42 mit zum Rumpf gehören würde
- alle gebundenen Vorkommen des Parameters werden bei der  $\beta$ -Reduktion nun durch 42 ersetzt  $\Rightarrow$  Ergebnis 42
- Beispiel: Inkrementierungsfunktion und Argument 42

$$(\lambda x.x + 1) 42$$

$$\rightsquigarrow 42 + 1$$

$$\rightsquigarrow 43$$

## Übung

- Reduzieren Sie folgende Ausdrücke so weit wie möglich:
  - $\lambda x.x\ 42$
  - $(\lambda x.x)\ (\lambda y.y)$
  - $(\lambda x.x)\ (\lambda y.y)\ z$ 
    - \* Applikation ist linksassoziativ
  - $(\lambda x.x\ 42)\ \lambda x.x$

## Freie Variablen

- Variablen in  $\lambda$ -Ausdrücken, die nicht gebunden sind, heißen **freie Variablen**
- Beispiel:

$$\lambda x.xy$$

- $x$  ist gebunden
- $y$  ist frei

- Beispiel:  $\beta$ -Reduktion

$$(\lambda x.xy)\ z$$

$$\rightsquigarrow zy$$

## Mehrere Argumente

- jedes  $\lambda$  kann nur einen Parameter binden
- wie schreibt man jetzt eine Funktion mit mehr als einem Argument?
- Beispiel: Addition

$$\lambda x.\lambda y.x + y$$

- Anwendung:

$$(\lambda x.\lambda y.x + y)\ 4\ 3$$

$$\rightsquigarrow (\lambda y.4 + y)\ 3$$

$$\rightsquigarrow 4 + 3$$

$$\rightsquigarrow 7$$

- als (syntaktische) Abkürzung (engl. *syntactic sugar*) können wir schreiben

$$\lambda x\ y.x + y$$

## $\eta$ -Konversion

- formalisiert das Konzept der Extensionalität, d.h.
  - zwei Funktionen sind genau dann gleich, wenn sie für alle Argumente das selbe Resultat liefern
- Formal beschrieben

$$\lambda x. f x \equiv f$$

## Auswertung

- um einen **Term** auszuwerten (auszurechnen) wird die  $\beta$ -Reduktion so oft wie möglich hintereinander angewendet
- wenn keine weitere  $\beta$ -Reduktion mehr anwendbar ist, ist der Term in  **$\beta$ -Normalform**
- Achtung: Nicht jeder Term hat eine  $\beta$ -Normalform
  - es gibt  $\lambda$ -Ausdrücke die **divergieren**
  - Beispiel

$$(\lambda x. xx)(\lambda x. xx)$$

## Kombinatoren

- **Kombinatoren** sind  $\lambda$ -Terme, die keine freien Variablen enthalten
- sie können nur ihre Argumente **kombinieren**
- Beispiele:

$$\lambda x. x$$

$$\lambda x y. x$$

$$\lambda x y z. xz(yz)$$