

Einführung

Funktionale Programmierung

Prof. Dr. Oliver Braun

Letzte Änderung: 09.10.2018 06:54

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**

- 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**

- 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](#)

- 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*

- die Grundlage für eine funktionale Sprache ist der λ -Kalkül
 - im Gegensatz dazu ist die Grundlage für eine imperative Sprache die Turingmaschine
- **Haskell** basiert ausschließlich auf dem λ -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 $\{A, B, C\}$

- eine λ -**Abstraktion** entspricht einer Funktion ohne Namen
 - anonyme Funktion
- Beispiel Identitätsfunktion

$$\lambda x.x$$

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

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

$$\lambda x.x$$

$$\lambda y.y$$

$$\lambda m.m$$

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

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

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

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

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

$$\rightsquigarrow 42 + 1$$

$$\rightsquigarrow 43$$

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

- Variablen in λ -Ausdrücken, die nicht gebunden sind, heißen **freie Variablen**
- Beispiel:

$$\lambda x. xy$$

- x ist gebunden
- y ist frei
- Beispiel: β -Reduktion

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

$$\rightsquigarrow zy$$

Mehrere Argumente

- jedes λ 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

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

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

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

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

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

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

$$\lambda x.x$$

$$\lambda x y.x$$

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