

Software Entwicklung 1

Annette Bieniusa

AG Softech
FB Informatik
TU Kaiserslautern

Überblick für heute

- Programmierparadigmen
 - Imperative vs. deklarative Programmierung
 - Beispiele
- Funktionale Programmierung
 - Wichtige Eigenschaften (Referentielle Transparenz)
 - Theoretische Grundlage: Lambda-Kalkül
 - Funktionen höherer Ordnung: `map`, `filter`, `fold`

Programmierparadigmen

Imperative Programmierung

- Programm = Folge von Anweisungen bzw. Befehlen, die in vorgegebener Reihenfolge abgearbeitet werden
- Entspricht der Arbeitsweise von Prozessoren
- Beispiele: C/C++, Java, Assembler, Pascal, Delphi, ...
- Strukturierte Programmierung: Kontrollfluss durch Verzweigungen und Schleifen
- Prozedurale Programmierung: Abstraktionsmechanismus durch Prozeduren/Methoden

Wie wird etwas berechnet?

Deklarative Programmierung

Was wird berechnet?

- Beschreibt die Programmlogik, ohne den Kontrollfluss zu beschreiben
- Besteht daher nicht aus Anweisungen
- Basiert auf mathematischen Theorien
- Erlaubt relativ einfache Beweise über die Eigenschaften von Programmen
- Beispiele: LISP, ML, OCaml, Oz, Haskell, Scheme, Erlang, Prolog, Curry, SQL, etc.

Beispiel für deklarative Programmierung: Funktionale Programmierung

- Beispiel: Programmiersprache Haskell
- Basiert auf dem Lambda-Kalkül
- Beschreibt Berechnungen als (math.) Funktionen

```
quicksort [] = []  
quicksort (x:xs) = quicksort [n | n <- xs, n < x]  
                  ++ [x]  
                  ++ quicksort [n | n <- xs, n >= x]
```

Beispiel für deklarative Programmierung: Anfragesprachen

- Beispiel: Datenbanksprache SQL (Structured Query Language)
- Basiert auf relationaler Algebra
- Definiert Struktur von Datenbanktabellen, ermittelt und modifiziert Einträge

```
UPDATE salary
SET income = income + 0.1 * income
WHERE status = 'employed'
```

Beispiel für deklarative Programmierung: Logische Programmierung

- Beispiel: Prolog
- Basiert auf Logik-Kalkül
- Verwendet Fakten und Regeln, um Anfragen an eine Datenbasis zu beantworten

```
father(abraham, isaac).  
mother(sarah, isaac).  
male(isaac).  
son(X, Y) :- father(Y, X), male(X).
```

Die Anfrage, wer der Sohn von Isaac ist, wird wie folgt aufgelöst:

```
?- son(X, abraham).  
X = isaac
```


Bemerkung

- Verschiedene Ausprägungen von deklarativer und imperativer Programmierung
 - Objekt-orientiert
 - Aspekt-orientiert
 - etc.
- Sprachen sind meist nicht streng an ein Paradigma gebunden, sondern entwickeln sich

Funktionale Programmierung

Funktionale Programmierung I

- Programme bestehen im Kern aus Funktionen und Funktionsanwendungen
- Funktion bildet Eingabewerte auf Ausgaben ab
- Bei gleicher Eingabe wird bei jedem Funktionsaufruf die gleiche Ausgabe geliefert

```
int a = 2;  
int x = f(a);  
int y = x + x;
```

```
int a = 2;  
int x = f(a);  
int y = f(a) + f(a);
```

Funktionale Programmierung II

```
int a = 2;  
int x = f(a);  
int y = x + x;
```

```
int a = 2;  
int x = f(a);  
int y = f(a) + f(a);
```

Frage

Welche Werte nehmen **x** und **y** für das linke und das rechte Beispiel mit der gegebenen Java-Implementierung an?

```
static int y = 0;  
static int f(a) {  
    y++;  
    return a + y;  
}
```

Eigenschaften funktionaler Programmierung

- Referentielle Transparenz
 - Ausdrücke haben einen eindeutigen Wert
 - Wert hängt ausschließlich von den Werten der Teilausdrücke ab
- Funktionen als Werte
 - können an Variablen gebunden werden
 - können als Parameter übergeben werden
 - auch als Rückgabe einer Funktion verwendet werden

Der Lambda-Kalkül

- Formale Sprache mit Regeln zur Auswertung von Funktionen
- Alle berechenbaren Funktionen können in ihm kodiert werden
⇒ Funktionales Pendant zur Turing-Maschine

Syntax:

$$\Gamma = (N, T, \Pi, T)$$

$$N = \{E\}$$

$$T = \{\lambda, \rightarrow, (,), id\}$$

$$\Pi = \left\{ \begin{array}{l} E \rightarrow id \\ E \rightarrow (\lambda id \rightarrow E) \\ E \rightarrow (E E) \end{array} \right\} \begin{array}{l} \mathbf{Variable} \\ \mathbf{Funktionsabstraktion} \\ \mathbf{Funktionsaufruf} \end{array}$$

Beispiele

Beispiel

 $(f\ x)$
 $(f\ (g\ x))$
 $((f\ g)\ x)$
 $(\lambda x \rightarrow (f\ (f\ x)))$
 $((\lambda x \rightarrow (f\ (f\ x)))\ y)$

Bedeutung

Aufruf der Funktion f mit Parameter x

Aufruf der Funktion f mit Parameter $(g\ x)$

Aufruf der Funktion $(f\ g)$ mit Parameter x
 $((f\ g)$ liefert eine Funktion)

Funktion, die x nimmt und zweimal f anwendet

Aufruf der obigen Funktion mit Parameter y

- Beachte Klammersetzung!
- Funktionen sind **anonym**
- Konventionen:
 - $(\lambda x\ y \rightarrow t)$ statt $(\lambda x \rightarrow (\lambda y \rightarrow t))$
 - $(f\ x\ y)$ statt $((f\ x)\ y)$
 - $(\lambda x \rightarrow f\ x)$ statt $(\lambda x \rightarrow (f\ x))$

Variablenbindung

- Verwendung einer Variable bezieht sich immer auf den gleichnamigen Parameter der nächsten umschließenden Funktionsabstraktion
- Falls keine solche Parameterdeklaration existiert, ist die Variable **frei**.

Formale Definition

- $FV(x) = \{x\}$ für eine Variable x
- $FV(\lambda x \rightarrow e) = FV(e) \setminus \{x\}$
- $FV(f \ g) = FV(f) \cup FV(g)$

Frage

Welche der Variablen sind im folgenden Term frei bzw. gebunden?

$$(\lambda f \rightarrow (\lambda x \rightarrow f \ (x \ y))) \ ((\lambda y \rightarrow y) \ f)$$

Substitution

- $s[t/x]$ bezeichnet den Term, der entsteht, wenn alle **freien** Variablen x in s durch den Term t ersetzt werden
- Substitution ist **nicht erlaubt**, wenn eine freie Variable in t nach der Substitution gebunden wäre

Frage

- Die Substitution $((\lambda x \rightarrow f\ x)\ x)[(g\ y)/x]$ ergibt
- Die Substitution $((\lambda x \rightarrow (\lambda y \rightarrow x\ y))\ x)[y/x]$

Substitution

- $s[t/x]$ bezeichnet den Term, der entsteht, wenn alle **freien** Variablen x in s durch den Term t ersetzt werden
- Substitution ist **nicht erlaubt**, wenn eine freie Variable in t nach der Substitution gebunden wäre

Frage

- Die Substitution $((\lambda x \rightarrow f\ x)\ x)[(g\ y)/x]$ ergibt $((x \rightarrow f\ x)\ (g\ y))$.
- Die Substitution $((\lambda x \rightarrow (\lambda y \rightarrow x\ y))\ x)[y/x]$ ist nicht erlaubt!

Formale Definition:

- $x[t/x] = t$
- $y[t/x] = y$ falls $x \neq y$
- $(f\ g)[t/x] = (f[t/x]\ g[t/x])$
- $(\lambda x \rightarrow e)[t/x] = (\lambda x \rightarrow e)$
- $(\lambda y \rightarrow e)[t/x] = (\lambda y \rightarrow e[t/x])$ falls $x \neq y$ und $y \notin FV(t)$

Semantik von Lambda-Ausdrücken I

α -Konversion: Umbenennen von Parametern.

- Term $\lambda x \rightarrow t$ ist äquivalent zu $\lambda y \rightarrow t[y/x]$, wenn y nicht bereits als freie Variable im Term t vorkommt.
- Beispiel: $\lambda a \rightarrow f (f a)$ ist äquivalent zu $\lambda b \rightarrow f (f b)$, aber nicht zu $\lambda f \rightarrow f (f f)$

β -Konversion: Auswertung von Funktionsaufrufen.

- Term $(\lambda x \rightarrow s) t$ ist äquivalent zu $s[t/x]$, wenn diese Substitution erlaubt ist
- Evtl. zuvor α -Konversion zum Umbenennen von Parametern Beispiel:
 $(\lambda x \rightarrow \text{plus } x x) (f a)$ wird zu $\text{plus } (f a) (f a)$

Semantik von Lambda-Ausdrücken II

Ein Term ist in **β -Normalform**, wenn keine β -Umformungen mehr möglich sind (auch nicht durch weitere Umbenennungen).

- Beide Arten von Konversion (α und β) können auf beliebige Teilausdrücke angewandt werden.
- Reihenfolge der Auswertung hat dabei keinen Einfluss auf das Ergebnis.
- In manchen Fällen gibt es jedoch Reihenfolgen, welche zu unendlich langen Umformungen führen können, ohne dass eine β -Normalform erreicht wird.
- Es gibt auch Terme, die für keine Reihenfolge eine β -Normalform erreicht.

Auswertungsstrategien

- **call-by-value**: Reduziere den äußersten Aufruf; aber nur, wenn Parameter bereits zu Variablen oder Funktionsabstraktionen ausgewertet wurden

$$\begin{aligned}
 & (\lambda x \rightarrow x \ x) \ ((\lambda w \rightarrow w) \ (\lambda a \rightarrow a)) \\
 = & (\lambda x \rightarrow x \ x) \ (\lambda a \rightarrow a) \\
 = & (\lambda a \rightarrow a) \ (\lambda a \rightarrow a) \\
 = & (\lambda a \rightarrow a)
 \end{aligned}$$

- **call-by-name**: Es wird jeweils der äußerste linkest mögliche Aufruf ausgewertet; keine Reduktion innerhalb von Funktionsabstraktionen

$$\begin{aligned}
 & (\lambda x \rightarrow x \ x) \ ((\lambda w \rightarrow w) \ (\lambda a \rightarrow a)) \\
 = & ((\lambda w \rightarrow w) \ (\lambda a \rightarrow a)) \ ((\lambda w \rightarrow w) \ (\lambda a \rightarrow a)) \\
 = & (\lambda a \rightarrow a) \ ((\lambda w \rightarrow w) \ (\lambda a \rightarrow a)) \\
 = & (\lambda w \rightarrow w) \ (\lambda a \rightarrow a) \\
 = & (\lambda a \rightarrow a)
 \end{aligned}$$

Frage

Werten Sie den folgenden Term aus!

- mittels call-by-name
- mittels call-by-value

$$(\lambda x y \rightarrow x) (\lambda x \rightarrow x) ((\lambda x \rightarrow x x) (\lambda x \rightarrow x x))$$

Erweiterungen

- Im Lambda-Kalkül lassen sich auch Zahlen, boolesche Werte, `if`-Ausdrücke (`if x then y else z`) kodieren
- Beispiel:

`1` = $\lambda f x \rightarrow f x$

`3` = $\lambda f x \rightarrow f (f (f x))$

`plus` = $\lambda m n f x \rightarrow m f (n f x)$

Weitere Beispiele I

- Beispiel 1: Auswertung einer Funktion mit zwei Parametern:

$$\begin{aligned}
 & (\lambda x y \rightarrow x + y) 3 4 \\
 = & (\lambda y \rightarrow 3 + y) 4 \\
 = & 3 + 4 \\
 = & 7
 \end{aligned}$$

- Beispiel 2: α -Konversion notwendig

$$\begin{aligned}
 & (\lambda x y \rightarrow (x + y)) y z \\
 = & (\lambda x v \rightarrow (x + v)) y z \\
 = & (\lambda v \rightarrow (y + v)) z \\
 = & y + z
 \end{aligned}$$

Weitere Beispiele II

■ Beispiel 3: Funktion als Parameter

$$\begin{aligned}
 & (\lambda f \ x \rightarrow f \ (f \ x)) \ (\lambda y \rightarrow y*2) \ 3 \\
 = & (\lambda x \rightarrow (\lambda y \rightarrow y*2) \ ((\lambda y \rightarrow y*2) \ x)) \ 3 \\
 = & (\lambda y \rightarrow y*2) \ ((\lambda y \rightarrow y*2) \ 3) \\
 = & (\lambda y \rightarrow y*2) \ (3*2) \\
 = & (\lambda y \rightarrow y*2) \ 6 \\
 = & 6*2 \\
 = & 12
 \end{aligned}$$

Abstraktion über Ausdrücke

- Idee: Abstraktion von Ausdrücken
(vergleichbar zur Abstraktion von Anweisungen bei Prozeduren)

```
(λx y z →  
  if x <= y then  
    if x <= z then x else z  
  else  
    if y <= z then y else z)
```

Mit Abstraktion über eine Minimumfunktion:

```
(λmin → (λx y z → min x (min y z)))  
  (λx y → if x <= y then x else y)
```

Wertvereinbarung

```
let min = ( $\lambda x y \rightarrow$  if  $x \leq y$  then  $x$  else  $y$ ) in
  let min3 = ( $\lambda x y z \rightarrow$  min  $x$  (min  $y z$ )) in
    min3 6 3 12
```

- Eine **Wertvereinbarung** weist einem Bezeichner einen Wert zu
- `let $x = s$ in t` steht für den Term $(\lambda x \rightarrow t) s$.

Rekursion

- Fixpunkt-Operator erlaubt Kodierung rekursiver Funktionen

$$\text{fix} = \lambda f \rightarrow (\lambda x \rightarrow f (x x)) (\lambda x \rightarrow f (x x))$$

- Man kann nachrechnen:

$$\text{fix } f = \dots = f (\text{fix } f)$$

- Beispiel

$$\text{facR} = (\lambda f \ x \rightarrow \text{if } x == 0 \text{ then } 1 \text{ else } x * f (x - 1))$$

$$\text{fac} = \text{fix facR}$$

$$\text{fac } 5 = \dots = 5 * \text{fac } (5-1) = \dots = 120$$

Funktionen höherer Ordnung I

- Funktion, die eine (oder mehrere) Funktionen als Argument nimmt oder als Ergebnis liefert
- Parametrisierung über eine Berechnung möglich!
- Beispiel: Anwendung von Funktion auf alle Elemente einer Liste

```
map ( $\lambda x \rightarrow x + 2$ ) [1,2,3,4] = [3,4,5,6]
```

```
map ( $\lambda y \rightarrow 3 * y$ ) [1,2,3,4] = [3,6,9,12]
```

Funktionen höherer Ordnung II

- Beispiel: Filtern von Elementen

```
filter ( $\lambda x \rightarrow x < 10$ ) [1,32,7] = [1,7]  
filter ( $\lambda x \rightarrow x \% 2 == 0$ ) [1,32,8] = [32,8]
```

- Beispiel: Faltung von Listelementen, ausgehend von Initialwert, mittels einer Funktion zu einem Wert

```
foldl ( $\lambda y x \rightarrow y + x$ ) 0 [1,2,3]  
= (((0 + 1) + 2) + 3) = 6
```

Frage

Was ist das Ergebnis der folgenden Aufrufe?

■ `map cook [🐮 , 🥔 , 🐔 , 🌽] =`

Frage

Was ist das Ergebnis der folgenden Aufrufe?

■ `map cook` [🐮 , 🥔 , 🐔 , 🌽] = [🍔 , 🍟 , 🍗 , 🍿]

Frage

Was ist das Ergebnis der folgenden Aufrufe?

■ `map cook` [🐮 , 🥔 , 🐔 , 🌽] = [🍔 , 🍟 , 🍗 , 🍿]

■ `filter isVegetarian` [🍔 , 🍟 , 🍗 , 🍿]

Frage

Was ist das Ergebnis der folgenden Aufrufe?

■ `map cook` [🐮 , 🥔 , 🐔 , 🌽] = [🍔 , 🍟 , 🍗 , 🍿]

■ `filter isVegetarian` [🍔 , 🍟 , 🍗 , 🍿] [🍟 , 🍿]

Frage

Was ist das Ergebnis der folgenden Aufrufe?

■ `map cook` [🐮 , 🥔 , 🐔 , 🌽] = [🍔 , 🍟 , 🍗 , 🍿]

■ `filter isVegetarian` [🍔 , 🍟 , 🍗 , 🍿] [🍟 , 🍿]

■ `foldl feed` 😊 [🍔 , 🍟 , 🍗 , 🍿] =

Frage

Was ist das Ergebnis der folgenden Aufrufe?

■ `map cook` [🐮 , 🥔 , 🐔 , 🌽] = [🍔 , 🍟 , 🍗 , 🍿]

■ `filter isVegetarian` [🍔 , 🍟 , 🍗 , 🍿] [🍟 , 🍿]

■ `foldl feed` 😊 [🍔 , 🍟 , 🍗 , 🍿] = 🍌

Funktionspointer in C I

- Gleiche Idee wie bei Funktionen höherer Ordnung
- Beispiel: Sortierfunktionen der Standardbibliothek sind über Vergleichsfunktion parametrisiert

```
7 typedef int intfunction(int);
8
9 void map_array(intfunction f, int *ar, int size)
10 {
11     for (int i = 0; i < size; i ++)
```

```
12     {
13         ar[i] = f(ar[i]);
14     }
15 }
```

Funktionspointer in C II

```
1     int increment(int i) {
2         return i + 1;
3     }
4     int print(int i) {
5         printf("%i ", i);
6         return 0;
7     }
8     int main(void) {
9         int n = 10;
10        int a[n];
11        // Initialisiere a
12        for(int i = 0; i < n; i++) { a[i] = i; }
13        // Inkrementiere jeden Eintrag
14        map_array(increment, a, n);
15        // Gebe jeden Eintrag auf Konsole aus
16        map_array(print, a, n);
17        return 0;
18    }
19
```

Zusammenfassung

- Deklaratives Programmierparadigma: Funktionale Programmierung
 - Referentielle Transparenz und Funktionen als Werte
 - Theoretische Grundlage: Lambda-Kalkül
 - Elemente funktionaler Programmierung finden sich in vielen, auch imperativen Programmiersprachen
- Nächste Vorlesung: Lambda-Ausdrücke in Java