TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

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 Argumente 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
  $\Rightarrow$ Funktionales Pendant zur Turing-Maschine

Syntax:

$$
\begin{array}{rcl}
\Gamma & = & (N, T, \Pi, E) \\
N & = & \{E\} \\
T & = & \{\lambda, \rightarrow, \ (\ ,\ )\ , id\} \\
\Pi & = & \left\{ \begin{array}{lcl} E & \rightarrow & id \\ E & \rightarrow & (\lambda\ id \rightarrow E) \\ E & \rightarrow & (E\ E) \end{array} \right\} \begin{array}{l} \textbf{Variable} \\ \textbf{Funktionsabstraktion} \\ \textbf{Funktionsanwendung} \end{array}
\end{array}
$$

Beispiele

Beispiel	Bedeutung
`(f x)`	Aufruf der Funktion `f` mit Argument `x`
`(f (g x))`	Aufruf der Funktion `f` mit Argument `(g x)`
`((f g) x)`	Aufruf der Funktion `(f g)` mit Argument `x`
	(`(f g)` liefert eine Funktion)
`(λx → (f (f x)))`	Funktion, die `x` nimmt und zweimal `f` anwendet
`((λx → (f (f x))) y)`	Aufruf der obigen Funktion mit Argument `y`

- Beachte Klammernsetzung!
- Funktionen sind **anonym**
- Konventionen:
  - `(λx y → t)` statt `(λx → (λy → t))`
  - `(f x y)` statt `((f x) y)`
  - `(λx → f x)` statt `(λx → (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(λx → e) = FV(e) \ {x}`
- `FV(f g) = FV(f) ∪ FV(g)`

Frage

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

```
(λf → (λx → f (x y))) ((λy → 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 `((λx → f x) x)[(g y)/x]` ergibt
- Die Substitution `((λx → (λy → x y z)) x)[y/z]` ergibt .

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 `((λx → f x) x)[(g y)/x]` ergibt `((x→ f x) (g y))`.
- Die Substitution `((λx → (λy → x y z)) x)[y/z]` 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])`
- `(λx → e)[t/x] = (λx → e)`
- `(λy → e)[t/x] = (λy → e[t/x])` falls $x \neq y$ und $y \notin FV(t)$

Semantik von Lambda-Ausdrücken I

α-Konversion: Umbenennen von Parametern.

- Term `λx → t` ist äquivalent zu `λy → t[y/x]`, wenn y nicht bereits als freie Variable im Term `t` vorkommt.
- Beispiel: `λa → f (f a)` ist äquivalent zu `λb → f (f b)`, aber nicht zu `λf → f (f f)`

β-Konversion: Auswertung von Funktionsaufrufen.

- Term `(λx → s) t` ist äquivalent zu `s[t/x]`, wenn diese Substitution erlaubt ist
- Beispiel: `(λx → plus x x) (f a)` wird zu `plus (f a) (f a)`
- Evtl. zuvor α-Konversion zum Umbenennen von Parametern
- Beispiel: `(λx → λy → x y) (f y)` erfordert α-Konversion

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 Argumente bereits zu Variablen oder Funktionsabstraktionen ausgewertet wurden

```
   (λx → x x) ((λw → w) (λa → a))
=  (λx → x x) (λa → a)
=  (λa → a) (λa → a)
=  (λa → a)
```

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

```
(λx → x x) ((λw → w) (λa → a))
=  ((λw → w) (λa → a)) ((λw → w) (λa → a))
=  (λa → a) ((λw → w) (λa → a))
=  (λw → w) (λa → a)
=  (λa → a)
```

Siehe Skript für schrittweise Auswertung!

Frage

Werten Sie den folgenden Term aus!

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

$$(\lambda x\ y \to x)\ (\lambda x \to x)\ ((\lambda x \to x\ x)(\lambda x \to x\ x))$$

Erweiterungen

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

```
1     = λf x → f x
3     = λf x → f (f (f x))
plus  = λm n f x → m f (n f x)
```

Weitere Beispiele I

- Beispiel 1: Auswertung einer Funktion mit zwei Parametern:

```
   (λx y → x + y) 3 4
=  (λy → 3 + y) 4
=  3 + 4
=  7
```

- Beispiel 2: α-Konversion notwendig

```
   (λx y → (x + y)) y z
=  (λx v → (x + v)) y z
=  (λv → (y + v)) z
=  y + z
```

Weitere Beispiele II

- Beispiel 3: Funktion als Argument

```
   (λf x → f (f x)) (λy → y*2) 3
=  (λx → (λy → y*2) ((λy → y*2) x)) 3
=  (λy → y*2) ((λy → y*2) 3)
=  (λy → y*2) (3*2)
=  (λy → y*2) 6
=  6*2
=  12
```

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 = (λx y → if x <= y then x else y) in
  let min3 = (λx y z → 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 `(λx → t) s`.

Rekursion

- Fixpunkt-Operator erlaubt Kodierung rekursiver Funktionen

```
fix = λf → (λx → f (x x)) (λx → f (x x))
```

- Man kann nachrechnen:

```
fix f = ... = f (fix f)
```

- Beispiel

```
facR = (λf x → if x == 0 then 1 else x * f (x - 1))
fac = fix facR

fac 5 = ... = 5 * fac (5-1) = ... = 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 (λx → x + 2) [1,2,3,4]   = [3,4,5,6]
map (λy → 3 * y) [1,2,3,4]   = [3,6,9,12]
```

Funktionen höherer Ordnung II

- Beispiel: Filtern von Elementen

```
filter (λx → x < 10)      [1,32,7]   = [1,7]
filter (λx → x % 2 == 0)  [1,32,8]   = [32,8]
```

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

```
foldl (λy x → 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