Neues Kapitel

Neues Kapitel

Überblick

Funktionen

Es gibt verschiedene Notationen für Funktionen:

- $f : \mathbf{N} \to \mathbf{N},\ n \mapsto n \cdot n$
- $f : \mathbf{N} \to \mathbf{N},\ f(n) = n \cdot n$
- $\lambda n.n \cdot n$ (bedeutet $n \mapsto n \cdot n$)

Die ersten beiden Definitionen geben den Definitions- und Wertebereich an (die „Signatur“ der Funktion).

Sie geben der Funktion auch den Namen f.

Die dritte Definition entstammt dem λ-Kalkül. Sie gibt der Funktion keinen Namen.

Referentielle Integrität

```
class X {
  int n = 0;
  int f(int k) {
    return k * k;
  }
  int g(int k) {
    n = n + 1;
    return n * k;
  }
}
```

Was ergeben $f(5)$ und $g(5)$?

Referentielle Integrität: Das Ergebnis eines Funktionsaufrufs hängt nur von den Parametern ab.

Java hat sie nicht.

Referentielle Integrität

Die referentielle Integrität wird durch das Vorhandensein von Variablen verhindert.

Auch Ein- und Ausgaben verhindern sie:

Die Funktion **double** *readDouble*() liest in Java eine Zahl aus einer Datei.

Diese ist *nicht* immer die gleiche.

Ein weiteres Gegenbeispiel ist eine Methode, die uns eine Zufallszahl gibt.

Referentielle Integrität

Referentielle Integrität hat den Vorteil, daß die Reihenfolge, in der Parameter ausgewertet werden, keine Rolle spielt (Parallelität).

Beispiel:

- $f : n \mapsto n \cdot n$
- $g : (n, m) \mapsto n + m$

Was ist $g(f(3), g(f(2), f(1)))$?

$$\begin{aligned} g(f(3), g(f(2), f(1))) &= g(9, g(f(2), f(1))) \\ &= g(9, g(4, f(1))) \\ &= g(9, g(4, 1)) \\ &= g(9, 5) \\ &= 14 \end{aligned}$$

Andere Auswertungsreihenfolgen ergeben auch 14.

Gilt dies auch für Java-Programme?

```
class Reihenfolge {
  static int n = 0;
  static int f(int x) {
    n = n + x;
    return n;
  }
  static int g(int x, int y) {
    return x - y;
  }
  public static void main(String args[ ]) {
    int result = g(f(20), f(50));
    System.out.println(result);
  }
}
```

Was ist die Ausgabe?

Aus der *Java Language Specification*:

Siehe hier: docs.oracle.com

> *The argument expressions, if any, are evaluated in order, from left to right. If the evaluation of any argument expression completes abruptly, then no part of any argument expression to its right appears to have been evaluated, and the method invocation completes abruptly for the same reason. The result of evaluating the j'th argument expression is the j'th argument value, for* $1 \leq j \leq n$. *Evaluation then continues, using the argument values, as described below.*

Die Parameter werden *von links nach rechts evaluiert.*

Ein Programm, das sich darauf verläßt, ist nicht ideal.

Die Programmiersprache C gibt keine Garantie für die Reihenfolge, in der Parameter einer Funktion evaluiert werden.

```
#include ⟨stdio.h⟩
int n = 0;
int f(int x) {
  n = n + x;
  return n;
}
int g(int x, int y) {
  return x − y;
}
void main() {
  printf("%d\n", g(f(20), f(50)));
}
```

Ergebnis unterscheidet sich für SUN Workstations und Intel PCs.

Call by Value

Diese Art der Evaluierung nennen wir *Call by Value*:

$$\begin{aligned} g(f(3), g(f(2), f(1))) &= g(9, g(f(2), f(1))) \\ &= g(9, g(4, f(1))) \\ &= g(9, g(4, 1)) \\ &= g(9, 5) \\ &= 14 \end{aligned}$$

Bevor eine Funktion evaluiert wird, werden rekursiv alle ihre Parameter evaluiert.

Normale Evaluierung: Call by Value mit Reihenfolge von links nach rechts.

Call by Value

Call by Value kann ineffizient sein:

```
int h(int x, int y) {
  if(x < 0) {
    return 0;
  }
  else {
    return x * y;
  }
}
...
h(-3, sehrTeuer(25));
```

Um $h(-3, x)$ zu berechnen, ist der Wert von x irrelevant.

Trotztdem wird $sehrTeuer(25)$ aufgerufen.

Wir können eine Funktion auch so weit wie möglich außen evaluieren:

$$\begin{aligned}
g(f(3), g(f(2), f(1))) &= f(3) + g(f(2), f(1))) \\
&= (3 * 3) + g(f(2), f(1)) \\
&= 9 + g(f(2), f(1)) \\
&= 9 + (f(2) + f(1)) \\
&= 9 + ((2 * 2) + f(1)) \\
&= 9 + (4 + f(1)) \\
&= 9 + (4 + (1 * 1)) \\
&= 9 + (4 + 1) \\
&= 9 + 5 \\
&= 14
\end{aligned}$$

Dies nennen wir *Call by Name*.

Hat dies irgendeinen Vorteil???

Call by Name

„Evaluierung“ von h mittels Call by Name:

$h(-3, sehrTeuer(25))$
if$(-3 < 0)$ {**return** 0;} **else** {**return** $(-3) * sehrTeuer(25)$};
if$(true)$ {**return** 0;} **else** {**return** $(-3) * sehrTeuer(25)$;}
return 0;
0

Es stellt sich heraus, daß $sehrTeuer(25)$ *gar nicht* evaluiert wird.

Call by Name und Call by Value sind unter referentieller Integrität *fast* äquivalent.

Wenn $sehrTeuer(25)$ nicht definiert ist (Fehler oder Endlosschleife) dann sind die Ergebnisse verschieden.

$$list(n) = n : list(n+1)$$

Es sei [] eine leere Liste und $x : l$ möge die Liste sein, deren erstes Element x ist gefolgt von der Liste l.

Dann ist $list(4) = [4, 5, 6, 7, 8, 9, \ldots]$ eine unendliche Liste!

$$take(n, x : rest) = \begin{cases} x & \text{falls } n = 1, \\ take(n-1, rest) & \text{sonst.} \end{cases}$$

$take(n, l)$ gibt uns das $n$te Element der Liste l.

Beispiele:

$take(3, [1, 4, 9, 16, 25]) = 9$

$take(6, [1, 2, 3, 4, 5, 6, 7, \ldots]) = 6$

$take(3, list(4)) = 6$

Lazy Evaluation

$$
\begin{aligned}
take(3, list(4)) &= take(3, 4 : list(4+1)) \\
&= take(3-1, list(4+1)) \\
&= take(2, list(4+1)) \\
&= take(2, (4+1) : list((4+1)+1)) \\
&= take(2-1, list((4+1)+1)) \\
&= take(1, list((4+1)+1)) \\
&= take(1, ((4+1)+1) : list(((4+1)+1)+1) \\
&= (4+1)+1 \\
&= 5+1 \\
&= 6
\end{aligned}
$$

Wir nennen dies auch *Lazy Evaluation*, denn es wird nur das evaluiert, was wir wirklich brauchen.

Überblick

Funktionale Programmierung

Vereinfacht gesagt:

Wir definieren einige Funktionen.

Wir lassen eine von ihnen z.B. mit Lazy Evaluation auswerten und erhalten das Ergebnis.

Das ist funktionale Programmierung.

Ein Programm besteht aus *Funktionsdeklarationen*:

$$
\begin{aligned}
&\mathit{list}(n) \;=\; n \;:\; \mathit{list}(n+1)\\
&\mathit{take}(n, x : \mathit{rest}) \;=\\
&\quad \textbf{if } n = 1\\
&\quad \textbf{then } x\\
&\quad \textbf{else } \mathit{take}(n-1, \mathit{rest})
\end{aligned}
$$

Haskell

Wir verwenden die funktionale Programmiersprache *Haskell*.

Ein Programm besteht aus einer Folge von Deklarationen, die Funktionen definieren.

$$\begin{aligned} \mathit{factorial}(1) &= 1 \\ \mathit{factorial}(n) &= n * \mathit{factorial}(n-1) \end{aligned}$$

Gibt es mehrere Deklarationen für *factorial*, dann wird die erste verwendet, welche „paßt“.

- Für $\mathit{factorial}(1)$ wird die erste Zeile verwendet.
- Für $\mathit{factorial}(5)$ wird die zweite Zeile verwendet.

Evaluierung

$$factorial(1) = 1$$
$$factorial(n) = n * factorial(n-1)$$

Wie wird $factorial(3)$ evaluiert?

$factorial(3)$
$3 * factorial(3-1)$
$3 * factorial(2)$
$3 * (2 * factorial(2-1))$
$3 * (2 * factorial(1))$
$3 * (2 * 1)$
$3 * 2$
6

Es wird die erste passende linke Seite, durch die rechte Seite ersetzt.

Dabei passt n zu jeder natürlichen Zahl.

Typdeklarationen

Eine „anständige“ Definition einer Funktion umfaßt auch ihren Typ:

$$factorial : \mathbf{N} \rightarrow \mathbf{N},\ factorial(n) = ...$$

In Haskell können wir neben Funktionsdeklarationen auch *Typdeklarationen* erstellen:

$$\begin{aligned} &factorial :: Integer \rightarrow Integer \\ &factorial(1) = 1 \\ &factorial(n) = n * factorial(n-1) \end{aligned}$$

Typdeklarationen sind nicht immer nötig, aber wir sollten sie immer erstellen. Dadurch wird die Fehlersuche viel leichter.