Aktuelle Kapitel Sozialer Web – Technologien – Einführung Scala

Seminaristischer Unterricht

Prof. Dr.-Ing. Hendrik Gärtner

Gliederung

- Organisatorisches
- Programmierparadigmen
  - Kurze Wiederholung
  - ‚von Neumann‘-Flaschenhals
  - Imperativ/Deklarativer Ansatz
  - Funktionaler Ansatz
- Funktionale Programmierung
  - Basiskonzept
  - Ausdrücke und Auswertungsstrategien
  - Einführung in die funktionalen Elemente von Scala

Klausur

- Prüfungszeiträume
  - 01.02.2016 - 20.02.2016
  - 29.03.2016 - 09.04.2016
- Vorschlag Prüfungstermine
  - 03.02.2016 von 12:15-13:45 Uhr
  - 06.04.2015 von 15:45-17:15 Uhr
- 16 Veranstaltungen

Bitte überprüfen, ob es Terminkollisionen gibt!!!!

Imperative versus Deklarative (Oberbegriff)-/Funktionale Programmierung

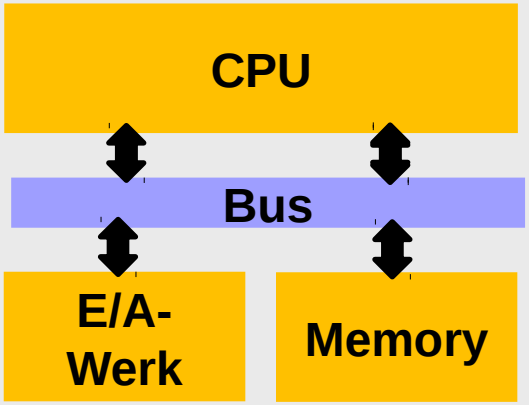
Imperative Programmierung

Beispiel: Ausgabe der Quadratzahlen 1-6 in der Sprache C

```
int i;
for (i=1; i<=6; i=i+1) {
    int q = i * i;
    printf("%d\n", q);}
```

- Folge von Anweisungen
- Veränderungen des Status des Programms
  - Speicherzellen, Prozessorregistern
  - Variablen
  - Programmzähler
  - Imperative Anweisungen: z.B. Schleife

Von Neumann-Rechner



Veränderbare Variablen ≈ Speicherzellen

Variablen referenzieren ≈ Lade-Operation

Variablen zuweisen ≈ Speicher-Operation

Kontrollstrukturen ≈ Jumps

…

Programmierung intuitiv – leicht vorstellbar und erlernbar

Problem: Scale up (Skalierung)! Konzept der Wort-für-Wort-Verarbeitung behindert die Skalierung von Operationen[1]

[1]Backus, John: Can Programming be Liberated from the von Neumann Style?, Turing Award Lecture, 1978.

Deklarative Programmierung

Beispiel: Ausgabe der Quadratzahlen 1-6 in der Sprache Scala

```
(for (i <- Range(0,6)) yield i*i).foreach(println)
```

- Bei der imperativen Programmierung steht das **„Wie“** im Vordergrund
- Bei der deklarativen Programmierung steht das **„Was“** im Vordergrund
- *Keine Beschreibung des Lösungswegs sondern eine Definition des Ergebnisses*
- Basieren deklarative Paradigmen auf mathematischen, rechnerunabhängigen Theorien

Funktionale Programmierung

Basiert auf dem **Lambda-Kalkül:**

- „Programmiersprache“ auf Basis von Funktionen:
  - Funktionsdefinitionen, Definieren formaler Parameter sowie
  - das Auswerten und Einsetzen aktueller Parameter

Funktionales Programmieren bedeutet:

- Erzeugen von Funktionen zur Lösung eines Problems
- Funktionen können dabei Unterfunktionen haben
- Computer wertet die Funktionen aus und gibt das Ergebnis aus
- Definierte Funktionsnamen werden nach den gegebenen Definitionen und Regeln ausgewertet

Moderne Programmiersprachen

- Vereinen die grundlegenden Paradigmen der Imperativen und Deklarativen Programmierung
- Übernehmen Konstrukte aus anderen – teilweise älteren Sprachen
- Sind sehr ausdrucksstark
- Einsatz von Scala – besteht aus Elementen von:
  - Java/Eiffel
  - Ruby
  - ML/Erlang
  - Prolog

Aber: Programmiersprachen sind nur Hilfsmittel zur Formulierung bzw. Umsetzung von **Algorithmen**

Motivation Funktionales Programmieren

Imperative Programmierung ist limitiert durch den sogenannten **von Neummann-Flaschenhals**:

Imperatives Programmieren verleitet dazu, Programme so zu gestalten, dass Wort für Wort Daten verarbeitet werden

Folgerung:

Es werden neue Techniken benötigt, mit denen Abstraktionen auf höherem Level durchgeführt werden können: z.B. Collections, Polynome, Geometrische Formen, etc.

Idealerweise sollten neue Theorien entwickelt werden für Collections, Shapes, Strings, etc.

Was ist eine Theorie?

Eine Theorie in der Mathematik besteht aus:

- Einem oder mehreren Datentypen
- Operationen auf den Typen
- Gesetze für den Umgang mit den Werten und den Operationen

Normalerweise beschreiben Theorien keine Veränderung (mutations) von Werten

(Identität bleibt erhalten, während die Werte innerhalb verändert werden)

Theorie ohne Mutationen: Polynome

Die Theorie über Polynome beinhaltet Regeln zur Addition von zwei Polynomen:

(a * x + b) + (c * x + d) = ((a + c) * x) + (b + d)

Die Theorie beinhaltet aber keinen Operator für die Veränderung des Polynoms (z.B. Änderung des Koeffizienten) während die Identität des Polynoms gleich bleibt

In der Imperativen Programmierung wäre das möglich:

```
Class Polynomial(double[ ] coefficient){
Polynomial P=….
p.coefficient[0]= 565
…}
```

Theorie ohne Mutationen: Strings

Die String-Theorie definiert einen Operator ++ zur Verbindung zweier Strings, der sich assoziativ verhält:

(a ++ b) ++ c = a ++ (b ++ c)

Aber die String-Theorie definiert keinen Operator zur Veränderung eines einzelnen Buchstaben innerhalb eines Strings, während die Identität des Strings dieselbe bleibt

(Viele Programmiersprachen wie Java folgen der Theorie und bieten dazu ebenfalls keinen Operator an.)

Mit Mutationen

val a:Array= new Array[Int](20)

- Erzeugung eines Arrays mit 20 Plätzen

a(5)= 10

- Entspricht Mutation
- Das Objekt a ist noch dasselbe
- Inhalt wurde geändert

Konsequenz für die Programmierung

Programmieren auf Basis Mathematischer Gesetze bedeutet:

- Theorien für Operatoren zu entwickeln und diese auf Basis von Funktionen umzusetzen
- Mutationen zu vermeiden
- auf Funktionen zu abstrahieren und
- diese beliebig zusammen zu setzen.

Funktionales Programmieren

- Funktionales Programmieren in einem eingeschränkten Sinn bedeutet Programmieren ohne veränderbare Variablen, Zuweisungen, Schleifen und anderen imperativen Strukturen
- Funktionales Programmieren in einem weiteren Sinn bedeutet die Fokussierung auf Funktionen
- Funktionen sind Werte die erzeugt, verwendet und zusammengesetzt werden können

→ Funktionale Programmiersprachen sind dafür ausgelegt

Einstieg in die funktionalen Elemente von Scala

Elemente von Programmiersprachen

Funktionales Programmieren:

- Kann mit der Verwendung eines Taschenrechners verglichen werden
- Kann meist über eine interaktive Shell (REPL – Read Eval Print Loop) angewendet werden:
  - Eingabe von Ausdrücken
  - Evaluation der Ausdrücke
  - Ausgabe des Ergebnisses

Die interaktive Shell von Scala wird über „scala“ aufgerufen.

Jede nicht primitive Sprache bietet:

- Primitive Ausdrücke für die Repräsentation der einfachsten Elemente
- Wege um die Ausdrücke zu kombinieren
- Wege um die Ausdrücke zu abstrahieren und dann darauf zu referenzieren

Evaluation von einfachen Ausdrücken

Ein Ausdruck wird mit den folgenden Regeln ausgewertet:

1. Nimm den Operator, der am weitesten links steht
2. Werte die Operanden des Operators aus
3. Wende den Operator an

Ein Name wird ausgewertet in dem er durch die rechte Seite der Definition ersetzt wird.

Die Auswertung ist beendet sobald ein Wert erreicht worden ist.

Ein Wert ist bspw. eine Zahl (später auch anderes)

Beispiel

Evaluierung des Audrucks:

radius= 10

(2 * pi) * radius

(2 * 3,142) * radius

6,284 * radius

6,284 * 10

62,84

Parameter

Definitionen können Parameter haben:

```
scala> def square(x:Double) = x * x
square:(x:Double)Double

scala>square(2)
4

scala>square(2+4)
36

scala>square(square(4))
256

def sumOfSquares(x:Double,y:Double)= square(x)+square(y)
sumOfSquares(Double,Double) Double
```

Parameter und Return Types

Funktionsparameter werden hinter den Namen geschrieben und erfordern einen Typen (mit ‚:‘ getrennt):

def sumOfSquares(x:Double,y:Double):Double = …

Wenn es einen Rückgabetyp gibt, wird dieser hinter die Parameterliste geschrieben

Primitive Typen sind aus Java übernommen, werden aber große geschrieben (sind eigentlich Objekte):

Int – 32bit Integer, Double – 64bit Floating Point, Boolean – values true und false

Evaluation von Funktionsanwendungen

Funktionen werden ähnlich wie Ausdrücke evaluiert:

1. Auswertung der Funktionsparameter von links nach rechts
2. Ersetze die Funktionsanwendung durch die rechte Seite der Definition und
3. Ersetze gleichzeitig die formalen Parameter mit den aktuell übergebenden Werten

Beispiel

sumOfSquares(3, 2+2)

sumOfSquares(3,4)

square(3) + square(4)

3 * 3 + square(4)

9 + square(4)

9 + 4 * 4

9 + 16

25

Substitution Model

- Das Evaluationsschema wird „Substitution Model“ genannt
- Die Idee dabei ist, jeden Ausdruck zu einem Wert zu evaluieren
- Kann für jeden beliebigen Ausdruck angewendet werden – sofern er keinen Seiteneffekt hat
- Das Modell ist im γ-Kalkül formalisiert worden – es ist die mathematische Basis der funktionalen Programmierung

Ergebnis

Evaluiert jeder Ausdruck zu einem Wert?

Nein, Beispiel

```
def loop:Int = loop
```

Endlosschleife

loop → loop → loop → …

Änderung der Evaluationsstrategie

- Der Interpreter wertet normalerweise erst die Parameter aus und wendet dann die Funktionsdefinition an
- Als Alternative könnte auch die Funktion auf nicht reduzierte Parameter angewendet werden, z.B.:

sumOfSquares(3, 2+2)

square(3) + square(2+2)

3 * 3 + square(2 + 2)

9 + square(2+2)

9 + (2+2) * (2+2)

9 + 4 * (2+2)

9 + 4 * 4

9 + 16

25

Call By Value/Call By Name

Die erste Evaluationsstrategie wird Call By Value genannt, die zweite Call By Name

Beide Strategien reduzieren zum selben Wert, sofern:

- Der reduzierte Ausdruck aus reinen Funktionen besteht
- Die Evaluation beider Ausdrücke terminieren

- Call By Value hat den Vorteil, dass jedes Funktionsargument nur einmal ausgewertet wird.

- Call By Name hat den Vorteil, dass ein Funktionsparameter nicht ausgewertet wird, wenn er nicht verwendet wird

Auswertungsreihenfolge

```
def test(x:Int, y:Int):Int = x*x
```

Welche Auswertungsart benötigt die wenigsten Schritte?

```
test(2,3)
test (3+4,8)
test (3,2*4)
test (3+4,2*4)
```

Unterschied CBN/CBV

Gibt es einen Ausdruck, der mit Call By Name terminiert und unter Call By Value nicht?

Definition der Funktion first und loop:

```
def first(x:Int, y:Int) = x
def loop:Int = loop
```

Aufruf unter Call By Name:

```
first(4,loop)        ???
```

Aufruf mit Call By Value

```
first (4, loop)      ???
```

Standardmäßig erfolgt der Aufruf Call By Value, da meistens weniger Reduktionsschritte erforderlich sind!

Veränderung der Evaluationsstrategie

Soll die Evaluationsstrategie auf Call By Name geändert werden, so ist dies über den => Operator möglich.

Beispiel:

```
def constOne(x:Int, y: =>Int) = 1
```

Aufrufe:

```
constOne (4, loop)   ???
constOne (loop, 4)   ???
```

Auswertungsregeln für Boolesche Ausdrücke

```
!true   -> false
!false  -> true
true && e -> e
false && e ->     false
true || e ->    true
false || e  -> e
```

(e ist ein beliebiger Boolescher Ausdruck)

- || und && benötigen nicht immer ihren rechten Operanden
- Verhalten wird „short-circuit evaluation“ genannt (Lazy Evaluation für Boolesche Ausdrücke)

Aufgabe

Schreiben Sie eine Funktion *and*, die das logische und repräsentiert. Achten Sie dabei darauf, dass bei der Angabe eines nicht terminierenden zweiten Operanden die Funktion nicht in eine Endlosschleife gerät!

```
def loop:Boolean = loop
def and(x:Boolean, y:Boolean): Boolean = if (x) y else false
and(false,loop)
```

```
def loop:Boolean = loop
def and(x:Boolean, y: => Boolean): Boolean =
        if (x) y else false
and(false,loop)
```

Programmieren auf Basis von Rekursionen

Definition von Rekursionen (1/2)

Beispiel der Berechnung einer Summe von 1..n

Ergebnis = $\sum_{i=0}^{n} i = 1+2+3+\ldots+n$

<u>Iterative Berechnung:</u>

```
def sum_iter(n:Integer):Integer = {
    var summe:Integer = 0
    for (i <- 1 to n) {summe+=i}
    summe
 }
```

Welche Elemente kommen aus der imperativen und welche aus der Funktionalen Progeammierung?

Definition von Rekursionen (2/2)

Rekursive Definition der Funktion:

$$sum(n) = \begin{cases} 0 \text{ wenn } n=0 & \text{Rekursionsanfang} \\ sum(n-1) + n & \text{Rekursionsschritt} \end{cases}$$

Ersetzung der Terme:

sum(3) = sum(2)+3 = sum (1)+2+3 =
sum(0)+1+2+3 = 0+1+2+3

Rekursive Berechnung:

sum (0) -> 0;

sum (N) -> N + sum(N-1).

Rekursion Vorgehensweise

1. Finden einer Abbruchbedingung / eines Basisfalls
2. Formulierung des Basisfalls als erste Regel
3. Finden des Rekursionsschritts (kann auch aus mehreren Regeln bestehen)
4. Formulierung der einzelnen Regeln bei Einhaltung der erforderlichen Reihenfolge

Die häufigste Rekursionsform ist die **lineare Rekursion**, bei der in jedem Fall der rekursiven Definition höchstens **ein rekursiver Aufruf** vorkommen darf. Die Berechnung verläuft dann entlang einer Kette von Aufrufen.

Die **primitive Rekursion** ist ein Spezialfall der linearen Rekursion. Hier definiert man Funktionen auf den natürlichen Zahlen, wobei in jedem rekursiven Aufruf dessen erster Parameter um Eins ab- oder zunimmt. **Jede primitiv-rekursive Definition kann unter Zuhilfenahme eines Stapels durch eine Schleife (Programmierung) (z.B. For-Schleife oder While-Schleife) ersetzt werden.**

Nachbildung von Schleifen

Schreiben Sie eine Funktion, die mittels Rekursion die Zahlen von x..y aufsummiert. Die Funktion soll zwei Eingabewerte X und Y haben und als Ergebnis die Summe liefern. Benutzen Sie für die Fallunterscheidung die Guards.

```
def summe(i:Integer,max:Integer):Integer =
    if (i>max) 0
        else i+ summe(i+1,max)
```

Übungsaufgabe

Wenn wir alle natürlich Zahlen unter 10 auflisten, die ein Vielfaches von 3 und 5 auflisten, bekommen 3,5,6,9. Die Summe dieser Vielfachen ist 23.

Schreiben Sie eine Funktion multiple, die für einen gegebenen Wert X alle Vielfachen von 3 und 5 findet und diese aufsummiert.

```
def multiple(x:Integer):Integer={
    if (x<=0) 0
      else if ((x % 3==0) || (x % 5 ==0)) x+multiple(x-1)
           else multiple(x-1)
}
```

Übungsaufgaben

Aufgabe 1:

Berechnung der 10001. Primzahl! Vorgehensweise?

1. Schreiben eines Primzahlentests
2. Entwerfen einer Funktion, die den Primzahlentest so lange aufruft, bis er 10001 mal true geliefert hat.

Aufgabe 2:

Schreiben Sie eine Funktion, die überprüft, ob innerhalb eines Ausdrucks (eine Liste von Character) eine valide Klammerung existiert.
So soll bspw.:
Dies ist ein (kleiner) ((()Test)) - true zurückgeben und
Dies)(() ist falsch - false.

Beispiel Rekursionen

Berechnung der 10001. Primzahl! Vorgehensweise?

1. Schreiben eines Primzahlentests
2. Entwerfen einer Funktion, die den Primzahlentest so lange aufruft, bis er 10001 mal true geliefert hat

```
def is_prim(X:Int)= calcPrim(X, 2, math.sqrt(X).toInt+1)

def calcPrim(X:Int, i:Int, Max:Int):Boolean =
   if (i>=Max) true
      else if (X % i == 0) false
          else calcPrim(X,i+1,Max)
```

Vielen Dank für

Ihre Aufmerksamkeit

Prof. Dr.-Ing. Hendrik Gärtner