Funktionale Programmierung und mehr mit Scala

SwissLife IT-Weiterbildung, 10. Dezember 2014

Thomas Herrmann, 42ways UG

42WAYS

from good ideas to great software

Prozedural vs. Funktional - Beispiel Quicksort

1. Wähle ein beliebiges Element (pivot) aus der Liste aus, z. B. das Element am Anfang der Liste.

2. Teile den Rest der Liste in zwei Listen. Alle Elemente kleiner oder gleich pivot kommen in die erste, alle Elemente größer oder gleich pivot in die zweite Liste. Nun ist pivot an der richtigen Position.

3. Führe den Algorithmus rekursiv für die beiden Teillisten aus.

Prozedural vs. Funktional - Quicksort in C

```
void quicksort(int a[], int lower, int upper) {
     int i;
     if ( upper > lower ) {
          i = split(a, lower, upper);
          quicksort(a, lower, i - 1);
          quicksort(a, i + 1, upper);
     }
}

int split(int a[], int lower, int upper) {
     int i, p, q, t;

     p = lower + 1; q = upper;
     pivot = a[lower];

     while ( q >= p ) {
          while ( a[p] < pivot )
               p++ ;

          while ( a[q] > pivot )
               q-- ;

          if ( q > p ) {
               t = a[p]; a[p] = a[q]; a[q] = t;
          }
     }

     t = a[lower]; a[lower] = a[q]; a[q] = t;

     return q;
}
```

Adapted from http://programminggeeks.com/c-code-for-quick-sort/

Prozedural vs. Funktional - Quicksort in Scala

```
def quicksort(list: List[Int]): List[Int] = {
    list match {
        case Nil     => Nil
        case x :: xs => quicksort(xs.filter(  <= x)) ++
                        List(x) ++
                        quicksort(xs.filter(_ > x))
    }
}

def l = List(2,8,3,1,7,0,9,5,4,6)
quicksort(l)
```

Prozedural vs. Funktional - Quicksort in Scala

```
def quicksort[T <% Ordered[T]](list: List[T]): List[T] = {
    list match {
        case Nil     => Nil
        case x :: xs => quicksort(xs.filter(  <= x)) ++
                        List(x) ++
                        quicksort(xs.filter(_ > x))
    }
}
```

Einige funktionale Programmiersprachen

- LISP (1958)
- ML (1973)
- Scheme (1975)
- Common Lisp (1984)
- Haskell (1987)
- Scala (2003)
- Clojure (2007)

Warum funktionale Programmierung?

- Vermeidung von Seiteneffekten
- Referenzielle Transparenz
  - Lesbarkeit / Verifizierbarkeit
  - Testbarkeit
  - Wiederverwendbarkeit
  - Optimierbarkeit
  - Parallelisierbarkeit

Funktionale Programmierung

- Funktion ist “first class citizen"
- “Pure Functions” ohne Seiteneffekte / stateless
- Rekursion
- Higher order functions
- Typsysteme / Funktionale Datenstrukturen
- Strict vs. Lazy evaluation

Scala

- Entstanden 2003
- Sprache auf der Java Virtual Machine (JVM)
- Integration mit Java
- Objektorientiert und Funktional
- Sehr umfangreich / komplex
- Erweiterbar

Klassendefinition “Person” in Scala

```
class Person(val name: String, val birth: Date,
             val sex: Sex, val profession: Profession)
```

Klassendefinition “Person” in Java

```
public class Person {
    private String name;
    private Date birth;
    private Sex sex;
    private Profession profession;

    public Person(String name, Date birth, Sex sex,
                  Profession profession) {
        this.name = name;
        this.birth = birth;
        this.sex = sex;
        this.profession = profession;
    }

    public String getName { return name; }
    public Date getBirth { return birth; }
    public Sex getSex { return sex; }
    public Profession getProfession { return profession; }
}
```

Rekursion in Scala

```
def fac(n: Int): Int = {
    if (n == 0) 1
    else n * fac(n-1)
}
```

Mathematische Definition:

$$0! = 1$$
$$n! = n * (n-1)!$$

Rekursion in Scala

```
// Der Euklidische Algorithmus zur Berechnung des größten
// gemeinsamen Teilers zweier natürlicher Zahlen m, n
// (m ≥ 0, n > 0).

def ggt(m: Int, n: Int): Int = {
    if (n == 0) m
    else ggt(n, m % n)
}
```

Funktionen in Funktionen

```
// Quadratwurzel (Methode von Newton)

def sqrt(x: Double): Double = {

    def sqrtIter(guess: Double, x: Double) : Double = {
        if (isGoodEnough(guess, x)) guess
        else sqrtIter(improve(guess, x), x)
    }

    def isGoodEnough(guess: Double, x: Double) : Boolean =
        Math.abs(guess * guess - x) / x < 0.001

    def improve(guess: Double, x: Double) : Double =
        (guess + x / guess) / 2

    sqrtIter(1.0, x)
}
```

Funktionale Abstraktion

```
def fac(n: Int): Int = {
    if (n == 0) 1
    else n * fac(n-1)
}
```

```
def sumUpTo(n: Int): Int = {
    if (n == 0) 0
    else n + sumUpTo(n-1)
}

def squareSumUpTo(n: Int): Int = {
    if (n == 0) 0
    else n * n + squareSumUpTo(n-1)
}
```

Funktionale Abstraktion

```
def myRecScheme(initval: Int, func: (Int, Int) => Int):
                                                  Int => Int = {
    n => if (n == 0) initval
         else func(n, myRecScheme(initval, func)(n-1))
}

def fac = myRecScheme(1, _*_)

def sumUpTo = myRecScheme(0, _+_)

def squareSumUpTo = myRecScheme(0, (a, b) => a * a + b)
```

“Spracherweiterungen” durch Funktionen

```
def until(condition: => Boolean)(block: => Unit) {
    if (!condition) {
        block
        until(condition)(block)
    }
}

var x = 10
until (x == 0) {
    x -= 1
    println (x)
}
```

Listen, Filter und Funktionen

```
1 to 10

1 to 10 filter (_ % 2 == 0)

1 to 10 map (_ * 2)

(1 to 10) zip (1 to 10 map (_ * 2))

List("Peter", "Paul", "Mary") map (_.toUpperCase)

def fac(n: Int) = (1 to n) reduce(_*_)

def squareSum(n: Int) = (1 to n) map(x => x*x ) reduce(_+_)
```

Typisches Pattern: filter map reduce

```
// 10% Rabatt auf alle Artikel, die teurer als 20 € sind

val itemPrices = Vector(10.00, 58.20, 8.99, 36.75, 47.90, 7.50)

def calculateDiscount(prices: Seq[Double]): Double = {
    prices filter (price => price >= 20.0) map
        (price => price * 0.10) reduce
            ((total, price) => total + price)
}
```

Generatoren und Filter

```
// aus "Programming in Scala", p. 484 ff.
def queens(n: Int): List[List[(Int, Int)]] = {
    def placeQueens(k: Int): List[List[(Int, Int)]] =
        if (k == 0)
            List(List())
        else
            for {
                queens <- placeQueens(k-1)
                column <- 1 to n
                queen = (k, column)
                if isSafe(queen, queens)
            } yield queen :: queens

    def isSafe(queen: (Int, Int), queens: List[(Int, Int)]) =
        queens forall (q => !inCheck(queen, q))

    def inCheck(q1: (Int, Int), q2: (Int, Int)) =
        q1._1 == q2._1 || q1._2 == q2._2 ||
                    (q1._1 - q2._1).abs == (q1._2 - q2._2).abs

    placeQueens(n)
}
```

Currying - Funktionen zum Teil auswerten

```
// Diskontierung mit Zinssatz als Parameter
def diskont(zinsSatz: Double, wert: Double) =
                                        wert * 1 / (1 + zinsSatz)

// Diskontierung mit festgelegten Zinssätzen
def diskont_0175: Double => Double = diskont(0.0175, _)
def diskont_0125: Double => Double = diskont(0.0125, _)
```

Allgemeine Higher Order Functions

```
def partial1[A,B,C](a: A, f: (A,B) => C): B => C =
    b => f(a, b)

def curry[A,B,C](f: (A,B) => C): A => (B => C) =
    a => b => f(a,b)
```

```
def uncurry[A,B,C](f: A => B => C): (A, B) => C =
    (a, b) => f(a)(b)

def compose[A,B,C](f: B => C, g: A => B): A => C =
    a => f(g(a))
```

Currying zur Vereinheitlichung von Schnittstellen

```
def klassische_sterbetafel(tafel: String, sex: String,
                           age: Int): Double = {
    // .....
}

def unisex_sterbetafel(tafel: String, age: Int): Double = {
    // .....
}

def bmi_sterbetafel(tafel: String, sex: String,
                    groesse: Int, gewicht: Double,
                    age: Int): Double = {
    // ....
}
```

Currying zur Vereinheitlichung von Schnittstellen

```
def  barwert(zins: Int => Double,
             qx: Int => Double, spektrum: Int => Double)
            (von: Int, bis: Int, ea: Int): Double = {
    if ( von > bis)
        0.0;
    else
        spektrum(von) + (1.0 - qx(von + eintrittsAlter)) *
         v(zins(von)) *
         barwert(zins, qx, spektrum)(von + 1, bis, sex, ea);
}
```

Currying zur Vereinheitlichung von Schnittstellen

```
def klassisches_qx(tafel: String, sex: String): Int => Double =
        klassische_sterbetafel(tafel, sex, _)

def unisex_qx(tafel: String): Int => Double =
        unisex_sterbetafel(tafel, _)
```

```
def bmi_qx(tafel: String, sex: String,
           groesse: Int, gewicht: Double): Int => Double =
        bmi_sterbetafel(tafel, sex, groesse, gewicht, _)
```

Currying zur Vereinheitlichung von Schnittstellen

```
qx = klassisches_qx(tarif.tafel, vp.sex)

barwert(zins, qx, spektrum)(von, bis, vp.ea)


qx = unisex_qx(tarif.tafel)

barwert(zins, qx, spektrum)(von, bis, vp.ea)


qx = bmi_qx(tarif.tafel, vp.sex, vp.groesse, vp.gewicht)

barwert(zins, qx, spektrum)(von, bis, vp.ea)
```

Funktionale Datenstrukturen / Case Classes

```
abstract class BinTree[+A]
case object EmptyTree extends BinTree
case class Node[A](element: A,
                   left:    BinTree[A],
                   right:   BinTree[A]) extends BinTree[A]

def inOrder[A](t: BinTree[A]): List[A] = {
    t match {
        case EmptyTree      => List()
        case Node(e, l, r) => inOrder(l):::List(e):::inOrder(r)
    }
}
```

Lazy evaluation / Streams

```
val ones: Stream[Int] = Stream.cons(1, ones)

def constant[A](n: A): Stream[A] = Stream.cons(n, constant(n))

def from(n: Int): Stream[Int] = Stream.cons(n, from(n+1))

def fibsFrom(a: Int, b: Int): Stream[Int] =
                                Stream.cons(a, fibsFrom(b, a + b))
```

Kleinste Zahl, die nicht in einer Liste vorkommt

```
// Inspired by "Pearls of functional algorithm design”
// by Richard Bird, Cambridge University Press, Chapter 1

def minfree(list: List[Int]): Int = {
    Stream.from(1).dropWhile(list contains).head
}

def list1 = List(26, 9, 22, 3, 17, 13, 21, 25, 14, 15, 20, 7, 2,
24, 12, 8, 19, 11, 27, 6, 5, 23, 18, 1, 16, 4, 10)
minfree(list1)

def list2 = List(26, 9, 22, 3, 17, 13, 21, 25, 14, 15, 20, 7, 2,
24, 8, 19, 11, 27, 6, 5, 23, 18, 1, 16, 4, 10)
minfree(list2)
```

Sieb des Eratosthenes - Eine “unendliche” Liste

```
import scala.language.postfixOps

// Konzeptionell:
// 1. Schreibe alle natürlichen Zahlen ab 2 hintereinander auf.
// 2. Die kleinste nicht gestrichene Zahl in dieser Folge ist
//    eine Primzahl. Streiche alle Vielfachen dieser Zahl.
// 3. Wiederhole Schritt 2 mit der kleinsten noch nicht
//    gestrichenen Zahl.

def sieve(natSeq: Stream[Int]): Stream[Int] =
        Stream.cons(natSeq.head,
                    sieve ((natSeq tail) filter
                                    (n => n % natSeq.head != 0)) )

def primes = sieve(Stream from 2)

def firstNprimes(n: Int) = primes take n toList
```

Fazit

- Funktionale Programmierung ermöglicht einfachen, wartbaren und parallelisierbaren Code

- Funktionale Programmierung bringt Flexibilität und neue Abstraktionslevel

- Scala ermöglicht die Kombination objektorientierter und funktionaler Programmierung auf der JVM

Literatur

Martin Odersky, Lex Spoon, Bill Venners:
Programming in Scala (artima)

Cay S. Horstmann:
Scala for the Impatient (Addison-Wesley)

Paul Chiusano, Rúnar Bjarnason:
Functional Programming in Scala (Manning)

Bruce A. Tate:
Seven Languages in Seven Weeks (Pragmatic Programmers)

Online-Resourcen

- http://www.scala-lang.org/
- http://twitter.github.io/scala_school/
- http://twitter.github.io/effectivescala/
- https://class.coursera.org/progfun-005
- https://class.coursera.org/reactive-001
- https://github.com/42ways/scala_vortrag_sl/