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.

Ein kleines Beispiel zum Einstieg, das bereits einige Konzepte enthält.

Das Beispiel ist zwar (noch) nicht aus der Versicherungsbranche, allerdings ist das ein einfacher Algorithmus, den eigentlich jeder kennen oder zumindest verstehen sollte…

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/

Die eigentliche Quicksort-Funktion ist ja noch sehr nahe an der Beschreibung des Algorithmus.

Die split-Funktion ist eher unübersichtlich. Dieses Beispiel ist aber immerhin bereits in zwei Funktionen unterteilt…

Erkennt man hier den Algorithmus? Ist der Code fehlerfrei?

…. sollte ich vielleicht absichtlich einen Fehler hier einbauen ???

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)
```


Funktional sieht der Code beinahe so aus wie die Beschreibung des Algorithmus.

Es ist sehr leicht, die Korrektheit der Implementierung zu erkennen.

Die rekursiven Aufrufe (und der Filter) können parallelisiert werden.

Hier schon ein bisschen was erklären (case classes/pattern matching/funktionale/rekursion)

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))
    }
}
```


Ach ja, das geht natürlich mit jedem Datentyp, der eine Ordnung implementiert…

Einige funktionale Programmiersprachen

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

- Funktionale Programmierung ist nicht neu!
- LISP: Hauptsächlich KI
- ML: statisch getypt / Typsysteme
- Scheme (minimalistisch) und Common Lisp (komplex): Verbreitung von LISP
- Haskell: Funktionale Programmierung “reine Lehre”
- Scala
- Clojure (LISP-Dialekt auf der JVM)

Warum funktionale Programmierung?

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

Letztendlich rühren die Vorteile mehr oder weniger alle aus dem Ansatz, Funktionen eher im mathematischen Sinn zu sehen und nicht zur Veränderung eines Zustands.

Funktionale Programmierung

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

Ein paar für funktionale Programmierung typische Eigenschaften und Patterns.

Scala

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

Backbone von Twitter ist in Scala geschrieben

Objekt-Funktional

Sprache hat alle möglichen Sachen integriert, z. B. XML, Reguläre Ausdrücke etc.

-> Umfangreich / Komplex (Odersky-Buch hat ca. 800 Seiten!)

Erweiterbarkeit z. B. für DSLs durch Higher Order Functions (Beispiel untill später)

Klassendefinition “Person” in Scala

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


Erst mal ein wenig Objektorientierung und Bezug zu Java.

Durch Angabe der Attribute bei der Klassen-Deklaration bekommt man die private Member + getter und setter sowie einen entsprechenden Konstruktor “geschenkt”

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; }
}
```


In Java ist da schon deutlich mehr Boilerplate code zu schreiben…

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)!
```


Klassisches Beispiel für Rekursion ist die Fakultät. Der Code sieht praktisch aus wie die mathematische Definition.

Sowas geht natürlich auch in Java, C, mit ein wenig mehr boilerplate code…

Allerdings trifft man rekursive Funktionen in funktionalem Code sehr viel häufiger an.

Die Funktionsdefintion in Scala kann ausserhalb jeder Klassendefinition erfolgen.

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)
}
```


Noch ein klassisches Beispiel, einfache Rekursion.

Diese Funktion ist tail-recursive, d. h. der Compiler kann daraus eine Schleife machen (Performance / Stackspace).

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)
}
```


Funktionen können auch innerhalb anderer Funktionen definiert werden (also nicht nur innerhalb von Klassen).

Auch hier sieht man sehr schön, dass die Implementierung sehr nahe an der fachlichen Lösungsbeschreibung ist.

Auch wieder tail-recursive und daher vom Compiler zur Schleife gemacht.

Im Beispielcode ist noch ein Printen drin, da sieht man wie (und wie schnell) die Annäherung an die Lösung erfolgt.

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)
}
```


Diese drei Funktionen haben das gleiche Rekursionsschema:

Bei 0 wird ein Wert zurückgegeben, sonst erfolgt der rekursive Aufruf mit n - 1, dessen Ergebnis dann in einem Berechnungsausdruck verarbeitet wird.

Durch Higher Order Functions ist es möglich, das Rekursionsschema zu verallgemeinern.

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)
```


Unsere drei Funktionen werden nun durch einen Aufruf von myRecScheme erzeugt. Erstes Argument ist der Initialwert (Terminierungsfall), zweites Argument ein Lamda-Ausdruck, der die Verarbeitung des Rekursiven Aufrufs und des aktuellen Wertes zum Endergebnis definiert.
Wenn Parameter in der angegebenen Reihenfolge und nur ein Mal verwendet werden, kann man auch “_” schreiben.

“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)
}
```


Mit Hilfe von Higher Order Functions können auch Funktionen definiert werden, die aussehen und sich anfühlen wie native Sprachsyntax.

Das funktioniert, weil die “=> T” - Parameter call-by-name sind.

Stichwort: DSLs

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(_+_)
```


Listen natürlicher Zahlen (Ranges) können sehr einfach erzeugt werden.

Für die Listen gibt es viele Funktionen, wie z. B. filter, contains, foreach, map, …

Mit zip können zwei Listen kombiniert werden (wie beim “Reißverschluss” == “zipper”).

map wendet eine Funktion auf jedes Listenelement an,

reduce erzeugt ein Ergebnis aus der Liste mit Hilfe einer Aggregatfunktion.

map und reduce werden oft kombiniert, um Daten erst aufzubereiten (map) und dann ein Ergebnis zu generieren (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)
}
```


Ein ganz typisches Pattern ist, eine Liste zu filtern, dann eine Funktion auf die Elemente anzuwenden (map) und deren Ergebnis dann zu aggregieren (reduce).

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)
}
```


Die “for”-Expression (es gibt auch for-Schleifen, also “for (x <- xs) block”) in Scala ist sehr viel mächtiger als for-Schleifen in anderen Sprachen, da sie mehrere Sequenzen (also quasi geschachtelte Schleifen) gleichzeitig generieren und dabei auch direkt Bedingungen auswerten kann.
Das n-queens-Beispiel zeigt ganz gut, wie “einfach” man ein doch relativ komplexes Suchproblem in Scala implementieren kann. Wäre sicher interessant, das mal prozedural (in Java oder gar in C) zu implementieren….

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, _)
```


Currying am Beispiel der Diskontierung, also mal ein Beispiel das auch bei Versicherungen relevant wäre…

Es ist zwar ziemlich trivial, zeigt aber das Prinzip (hoffentlich verständlich).

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))
```


Hier ein paar allgemeine Higher Order Functions.

Interessant hierbei ist, dass die Implementierung sich direkt aus den Parametertypen ergibt, welche in der Instanziierung beliebig komplexe Datentypen sein können….

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 = {
    // ....
}
```


Die Sterbetafeln können von verschiedenen Parametern abhängen, die z. T. durch den Tarif, z. T. durch das versicherte Risiko (die VP) definiert sind. Letztlich interessiert aber in der Barwertformel (und an vielen anderen Stellen) nur die Abhängigkeit vom Alter.

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);
}
```


Letztlich interessiert aber in der Barwertformel (und an vielen anderen Stellen) nur die Abhängigkeit vom Alter.

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, _)
```


Daher erzeugen wir uns aus den komplexeren Sterbetafeln jeweils nur die altersabhängige qx-Funktion, die wir der Barwertfunktion dann als Parameter übergeben können.

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)
```


Nachdem die qx-Funktion Tarif- und VP-abhängig erstellt wurde, kann der Barwert immer identisch aufgerufen werden.
Variationen / Abhängigkeiten wurden also “nach aussen” gezogen.
NB: Der aufmerksame Beobachter hat vielleicht gemerkt, dass die barwert-Funktion zwei Parameterlisten hat. Das liegt daran, dass die Abhängigkeit von der Zins-, qx- und spektren-Funktion ebenfalls per Currying aufgelöst werden kann.

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)
    }
}
```


Definition eines Binärbaumes mit Knotenwerten eines beliebigen Typs.

inOrder wandelt den Baum in eine Liste um, depth-first-Durchlauf.

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))
```


Ein Stream erzeugt eine “unendliche” Liste von Werten. Berechnet werden immer nur so viele Werte, wie aus dem Stream abgefragt werden.

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)
```


Um das kleinste Element einer Liste zu finden können wir mit Hilfe eines Streams einfach “alle” natürlichen Zahlen durchprobieren, bis wir die richtige gefunden haben.
Allerdings ist diese Implementierung natürlich sehr ineffizient…

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
```


Mit Hilfe von Streams können daher auch Algorithmen wie das Sieb des Eratosthenes direkt umgesetzt werden. Die Größe der Ergebnisliste ergibt sich ausschließlich durch das Abrufen der ersten n Primzahlen.

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)

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/