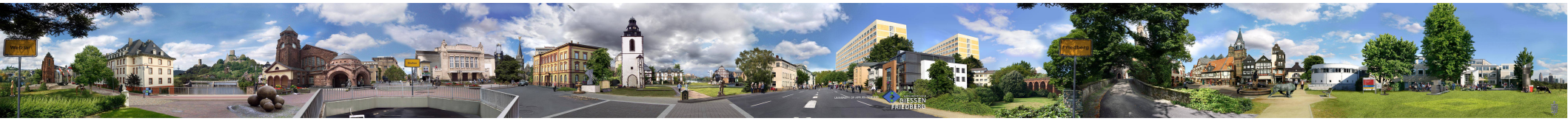

Funktionale Programmierung

Th. Letschert

TH Mittelhessen Gießen

University of Applied Sciences

Rechnen mit Funktionen

- Funktionsdefinitionen, Closures
- Funktionen höherer Ordnung
- Kombinatoren

Rechnen mit Funktionen = Funktionale Modularisierung – Teil I

Verkleben von Funktionen

In John Hughes *Why Functional Programming Matters*, wird das was hier „Rechnen mit Funktionen“ genannt wird, als erste von zwei Arten des *Verklebens* (to clue) bezeichnet.

„The first of the two new kinds of glue enables simple functions to be glued together to make more complex ones.“*

Modularisierung

Es um Modularisierung – das Kernthema des Software-Engineerings – Wie kann ein größeres System aus kleineren zusammengesetzt werden.

Funktionale Programmierung bietet gute „Kleber“, die die Module (also Funktionen) zu neuen Modulen (Funktionen) kombinieren („verkleben“)

„This is the key to functional programming's power - it allows greatly improved modularisation. It is also the goal for which functional programmers must strive - smaller and simpler and more general modules, glued together with the new glues we shall describe.“*

* *aus* *Why Functional Programming Matters* von John Hughes
http://www.cse.chalmers.se/~rjmh/Papers/whyfp.html

Was ist eine Funktion

Eine Funktion ist eine Paarmenge:

- Eine Menge ist ...
- Das kartesische Produkt $A \times B$ zweier Mengen A und B ist ...
- Eine Relation R zwischen zwei Mengen A und B ist eine Teilmenge des Kreuzprodukts $A \times B$.
- Eine Funktion $f : A \to B$ von einer Menge A nach einer Menge B ist eine rechts-eindeutige Relation aus $A \times B$. D.h. zu jedem $a \epsilon A$ gibt es genau ein (höchstens ein - bei partiellen Funktionen) Element $b \epsilon B$ mit $< a, b > \epsilon f$; dieses b wird mit $f(a)$ bezeichnet.

kurz: Eine Funktion ist eine i.A. unendliche Menge von Paaren bei der es verboten ist, dass ein Element mehrfach als linker Partner vorkommt.

Was ist eine Funktion

Eine Funktion ist ein Algorithmus zur Berechnung eines Wertes aus anderen Werten

Der Algorithmus wird durch eine Berechnungsformel angegeben

Beispiel:

$$f(x) = 2*x + 5$$

oder

$$f : Int \rightarrow Int \quad = \quad x \rightarrow 2*x + 5$$

„Closure“, „Lambda“: Funktionsausdruck

Ein Funktionsausdruck ist ein Ausdruck dessen Wert eine Funktion ist

Closure: **Ein Funktionsausdruck in dem freie Variablen vorkommen (dürfen), die in ihrem statischen Kontext gebunden werden.**

Beispiel:

```
val f = (x:Int) => 2*x + 5
```

ein Funktionsausdruck ohne Bezug zum Kontext

```
val g = (x:Int) => f(x) + 5
```

ein Funktionsausdruck mit Bezug zum Kontext: f ist in diesem Ausdruck nicht definiert, d.h. ungebunden oder frei. Der Bedeutung von f und damit von g wird vom Kontext (der Umgebung) beeinflusst.

```
val h = (x:Int) => f(x) + y
```

ein Funktionsausdruck mit Bezug zum Kontext: f und y frei. Der Bedeutung von f und y und damit von h wird vom Kontext (der Umgebung) beeinflusst.

Funktionen höherer Ordnung

Eine Funktion höherer Ordnung hat

- **Funktionen als Parameter und / oder**
- **ein Ergebnis**

das eine Funktion ist.

Beispiel: Erzeuge eine „addiere-drei“ Funktion

```
def mkPlus(x: Int) : Int => Int = (y: Int) => x+y

val plus3 = mkPlus(3)

val fuenf = plus3(2)

assert(fuenf == 5)
```

Funktionen höherer Ordnung

Beispiel: Erzeuge eine „wende n-mal an“-Funktion

```
def nMal(n: Int, f: Int => Int) : Int => Int =
  (x:Int) =>
      if (n == 0) x
      else nMal(n-1, f)(f(x))
```

wende f n-mal an

```
val plusEins = (x:Int) => x+1

val plusZehn: Int => Int = nMal(10, plusEins)

val zehn = plusZehn(0)

assert(zehn == 10)
```

plusZehn : wende plus 1 zehn-mal an.

Funktionen die Funktionen manipulieren bilden den Kern und die Essenz der funktionalen Programmierung.

Leider sind solche Funktionen nicht immer einfach zu verstehen.

Es ist darum unbedingt wichtig das Verständnis zu fördern indem kleine Funktionen mit sprechenden Namen definiert werden.

OO und der funktionale Stil

Objekt-Orientierung steht nicht im prinzipiellen Widerspruch zu funktionalen Stil

zustandslose Objekte sind erlaubte und ehrenwerte Mitglieder der funktionalen Welt

Beispiel: Erzeuge eine „wende n-mal an“-Funktion

```
class mkMal(n: Int) {
  def mal(f: Int => Int) : Int => Int =
    (x:Int) =>
      if (n == 0) x
      else new mkMal(n-1).mal(f)(f(x))
}

val plusEins = (x:Int) => x+1

val plusZehn: Int => Int = new mkMal(10).mal(plusEins)

val zehn = plusZehn(0)

assert(zehn == 10)
```

wende f n-mal an, OO-Variante

plusZehn : wende plus 1 zehn-mal an.

In der funktionalen Welt sind Zustandsänderungen verboten (dogmatische Sicht) oder auf ein absolutes Minimum zu beschränken (moderate Sicht).

Es ist aber nicht verboten Objektorientierung zu verwenden. – Objekte, die ihren Zustand nicht ändern, sind ehrenwerte Bewohner der funktionalen Welt.

Scala und der funktionale Stil

Scala erlaubt es Objekt-Orientierung und funktionale Programmierung zu kombinieren.

Natürlich auch auf eine schlechte Art. – Je mächtiger ein Werkzeug ist, um so mehr Schaden kann man mit ihm anrichten.

Mit impliziten Objekterzeugungen

kann die Lesbarkeit und Eleganz funktionaler Programme verbessert werden

Beispiel: Erzeuge eine „wende n-mal an“-Funktion

```
implicit class mkMal(n: Int) {
  def mal(f: Int => Int) : Int => Int =
    (x:Int) =>
      if (n == 0) x
      else new mkMal(n-1).mal(f)(f(x))
}

val plusEins = (x:Int) => x+1

val plusZehn = 10 mal plusEins

val zehn = plusZehn(0)

assert(zehn == 10)
```

Currying

Eine Funktion mit zwei Parametern ist äquivalent zu

- **einer Funktion mit einem Parameter, die eine Funktion erzeugt**
- **die den zweiten Parameter annimmt und das Ergebnis produziert**

Currying kann eingesetzt werden um Funktionen in eine „natürlichere Form“ zu bringen

Beispiel: Erzeuge eine „wende n-mal an“-Funktion

```
def nMal(n: Int)(f: Int => Int) : Int => Int =
  (x:Int) =>
      if (n == 0) x
      else nMal(n-1)(f)(f(x))


val plusEins = (x:Int) => x+1

val zehnMal = nMal(10) _ // = nMal(10)(_)

val plusZehn = zehnMal(plusEins)

val zehn = plusZehn(0)

assert(zehn == 10)
```

Variante 1: mit Platzhalter „_“

```
def nMal(n: Int): (Int => Int) => Int => Int =
  (f:Int => Int) =>
    (x:Int) =>
      if (n == 0) x
      else nMal(n-1)(f)(f(x))


val plusEins = (x:Int) => x+1

val zehnMal = nMal(10)

val plusZehn = zehnMal(plusEins)

val zehn = plusZehn(0)

assert(zehn == 10)
```

Variante 2: „echt“ gecurryt

Funktionsdefinition mit Mustern

Die Lesbarkeit von Funktionsdefinitionen kann mit der Verwendung von Mustern verbessert werden

Beispiel:

```
def f(n: Int) : Int = n match {
  case 0 => 1
  case _ => f(n-1)*n
}

assert ( f(10) == 1*2*3*4*5*6*7*8*9*10 )
```

Verallgemeinerte Algorithmen

Die Lesbarkeit von Programmen kann verbessert werden, wenn Wiederholungen vermieden werden.
Ein Mittel dazu ist die Abstraktion als Trennung des Allgemeinen und des Besonderen
Beispiel Summe / Quadrat-Summe einer Zahlenfolge :

```
// Summe aller Zahlen von von bis bis
def sum(von: Int, bis: Int) : Int =
  if (von > bis) 0
  else von + sum(von+1, bis)
```

Das eine Besondere

```
// Summe der Quadrate der Zahlen von von bis bis
def qsum(von: Int, bis: Int) : Int =
  if (von > bis) 0
  else von*von + qsum(von+1, bis)
```

Das andere Besondere

```
def fsum(f: Int => Int)(von: Int, bis: Int): Int =
  if (von > bis) 0
  else f(von) + fsum(f)(von+1, bis)
```

Das Allgemeine

Das eine Besondere:

```
val sum = fsum( (x:Int) => x ) _
```

Das andere Besondere:

```
val qsum = fsum( (x:Int) => x*x ) _
```

Verallgemeinerte Algorithmen

Beispiel Variante von fsum:

```
def fsum(f: Int => Int) : (Int, Int) => Int = {
  def h(von: Int, bis:Int) : Int =
    if (von > bis) 0
    else f(von) + fsum(f)(von+1, bis)
  (von:Int, bis: Int) => h(von, bis)
}
```

Das Allgemeine etwas anders

Verallgemeinerte Algorithmen auf Datenstrukturen

Catamorphismen : Verallgemeinerte Funktionen auf Datenstrukturen

Viele Funktionen operieren in gleichartiger Weise auf strukturierten Daten,

- z.B. Durchlaufen von Listen oder Bäumen
  mit Aktionen auf jedem Element / jedem Knoten

Die Algorithmen folgen dabei der Struktur der Daten

- z.B. Rekursion über die Liste / den Baum

Catamorphismus von Daten und Funktionen: Die Funktionen sind „struktur-gleich“ mit den Daten*

- solche strukturgleichen Algorithmen sind ein wesentlicher Bestandteil der funktionalen Programmierung
- Man kann sie zu Funktionen höherer Ordnung verallgemeinern:
  map, filter, reduce, zip, …
- Man nennt diese Funktionen gelegentlich „Catamorphismen“
- Man nennt diese Funktionen oft „Kombinatoren“ denn sie kombinieren Funktionen zu neuen Funktionen

*theoretisch Interessierte mögen dazu Wikipedia konsultieren:

https://en.wikipedia.org/wiki/Catamorphism

Kombinatoren

Funktionen

- **höherer Ordnung mit Funktionsargumenten**
- **die Rekursions-Muster implementieren:**
- **map, filter, reduce, zip, …**

> One style of functional programming is based purely on recursive equations. Such equations are easy to explain, and adequate for any computational purpose, but hard to use well as programs get bigger and more complicated. In a sense, recursive equations are the 'assembly language' of functional programming, and direct recursion the goto.

J. Gibbons, in *Origami programming*
https://www.cs.ox.ac.uk/jeremy.gibbons/publications/origami.pdf

Verallgemeinerte Algorithmen auf Datenstrukturen

Map: Eine Funktion auf alle Elemente einer Liste anwenden

```
def doubleAll(lst: List[Int]) : List[Int] =
  lst match {
    case Nil => Nil
    case h :: t => 2*h :: doubleAll(t)
  }
```

```
def tripleAll(lst: List[Int]) : List[Int] =
  lst match {
    case Nil => Nil
    case h :: t => 3*h :: tripleAll(t)
  }
```

Verallgemeinern

```
def map(f:Int => Int)(lst: List[Int]) : List[Int] = lst match {
  case Nil => Nil
  case h :: t => f(h) :: map(f)(t)
}
```

Spezialisieren

```
val doubleAll = map(x => 2*x) _
```

```
val tripleAll = map(x => 3*x) _
```

Verallgemeinerte Algorithmen auf Datenstrukturen

Map in Scala: Eine Methode die von allen Kollektionen angeboten wird

```
def map(f:Int => Int) : List[Int] => List[Int] = lst => lst match {
  case Nil => Nil
  case h :: t => f(h) :: map(f)(t)
}

val doubleAll = map(x => 2*x)
val tripleAll = map(x => 3*x)

assert( List(1,2,3).map(x => 2*x) == doubleAll(List(1,2,3)) )
assert( List(1,2,3).map(_*3)      == tripleAll(List(1,2,3)) )
```

```
List(1,2,3).map(_*3) == List(1,2,3).map( x => x*3 )
```

Verallgemeinerte Algorithmen auf Datenstrukturen

Reduce : Eine Struktur (z.B. eine Liste) zu einem Wert reduzieren

```
def sumAll(lst: List[Int]) : Int = lst match {
  case Nil => throw new IllegalArgumentException
  case h :: Nil => h
  case h :: t => h + sumAll(t)
}
```

```
def multAll(lst: List[Int]) : Int = lst match {
  case Nil => throw new IllegalArgumentException
  case h :: Nil => h
  case h :: t => h * multAll(t)
}
```

Verallgemeinern

```
def reduce(f:(Int, Int) => Int) : List[Int] => Int =
  lst => lst match {
    case Nil      => throw new IllegalArgumentException
    case h :: Nil => h
    case h :: t   => f(h, reduce(f)(t))
}
```

Spezialisieren

```
val sumAll = reduce( _ + _ )
```

```
val multAll = reduce( _ * _ )
```

Verallgemeinerte Algorithmen auf Datenstrukturen

Fold : Eine Struktur (z.B. eine Liste) zu einem Wert zusammen-falten
(reduce mit Startwert, neutrales Element der Operation)

```
def sumAll(lst: List[Int]) : Int = lst match {
  case Nil => 0
  case h :: t => h + sumAll(t)
}
```

```
def multAll(lst: List[Int]) : Int = lst match {
  case Nil => 1
  case h :: t => h * multAll(t)
}
```

Verallgemeinern

```
def fold(start: Int)(f:(Int, Int) => Int) : List[Int] => Int =
  lst => lst match {
    case Nil      => start
    case h :: t   => f(h, fold(start)(f)(t))
  }
```

Spezialisieren

```
val sumAll = fold(0)( _ + _ )
```

```
val multAll = fold(1)( _ * _ )
```

Verallgemeinerte Algorithmen auf Datenstrukturen

Fold : Von rechts oder links falten

```
def foldRight[T](start: T)(f:(T, T) => T) : List[T] => T =
  lst => lst match {
    case Nil      => start
    case h :: t   => f(h, foldRight(start)(f)(t))
  }

val concatR = foldRight("\"\"")("(" + _ + "+" + _ + ")")

println(concatR(List("a", "b", "c")))
```

```
(a+(b+(c+"")))
```

```
def foldLeft[T](start: T)(f:(T, T) => T) : List[T] => T =
  lst => lst match {
    case Nil      => start
    case h :: t   => foldLeft(f(start, h))(f)(t)
  }

val concatL = foldLeft("\"\"")( "(" + _ + "+" + _ + ")")

println(concatL(List("a", "b", "c")))
```

```
(((""+a)+b)+c)
```

Verallgemeinerte Algorithmen auf Datenstrukturen

***Fold* und *reduce* in Scala : Methoden von Kollektionstypen**

```
println(
    List("a", "b", "c").
      foldRight("\"\"")("(" + _ + "+" + _ + ")") )
```

```
(a+(b+(c+"")))
```

```
println(
    List("a", "b", "c").
      foldLeft("\"\"")("(" + _ + "+" + _ + ")") )
```

```
(((""+a)+b)+c)
```

Verallgemeinerte Algorithmen auf Datenstrukturen

Flatmap : Eine Struktur „mappen“ und dann „flach klopfen“

```
val lst: List[List[Int]] = List(List(1), List(2,3), List(4, 5, 6))
```

```
println(
        lst.map { x:List[Int] => x.map(_*2) }
      )
```

```
List(List(2), List(4, 6), List(8, 10, 12))
```

```
println(
        lst.flatMap { x:List[Int] => x.map(_*2) }
      )
```

```
List(2, 4, 6, 8, 10, 12)
```

Verallgemeinerte Algorithmen auf Datenstrukturen

Weitere Catamorphismen:

- **zip** — **Zwei Listen zu einer Liste von Paaren zippen**
- **filter** — **Eine Liste mit einem Prädikat filtern**
- **...**

```
val l1 = List(1, 2, 3, 4, 5)
val l2 = List ("a", "b", "c", "d", "e")

println( l1.zip(l2) )        // ~> List((1,a), (2,b), (3,c), (4,d), (5,e))
println( l1.zip(l2).unzip )  // ~> (List(1, 2, 3, 4, 5),List(a, b, c, d, e))
println( l2.zipWithIndex )   // ~> List((a,0), (b,1), (c,2), (d,3), (e,4))

println( l1.reverse )        // ~> List(5, 4, 3, 2, 1)

println( l1.filter( _ % 2 == 0) ) // ~> List(2,4)
```

Beispiel

Quicksort – funktional

```
def quicksort(lst: List[Int]) : List[Int] =
  if (lst.length <= 1) lst
  else {
    val pivot = lst(0)
    val small = lst.filter { _ < pivot }
    val mid   = lst.filter { _ == pivot }
    val large = lst.filter { _ > pivot }
    quicksort(small) ::: mid :::  quicksort(large)
  }
```

Beispiel

Permutationen – funktional

```
def perms[T](l: List[T]) : List[List[T]] = l match {
  case Nil   => List(Nil)
  case x::ll => (perms(ll) flatMap( ins(x, _) ))
}


def ins[T](x: T, l: List[T]) : List[List[T]] = l match {
  case Nil => List(List(x))
  case a::rl => (x::l) :: (ins(x, rl) map(a :: _))
}

println( perms(List("a", "b", "c") ) map ( _ reduce(_+_) ) )
```

```
List(abc, bac, bca, acb, cab, cba)
```

Rechnen mit Funktionen

Hinweis

Dieser Foliensatz basiert im Wesentlichen auf Abschnitt 1, 2 und 3, von
John Hughes: *Why Functional Programming Matters*
Computer Journal 32, 2 (1989)
online: http://www.cse.chalmers.se/~rjmh/Papers/whyfp.pdf