

Praktische Informatik 3: Funktionale Programmierung  
Vorlesung 13 vom 24.01.17: Scala — Eine praktische Einführung

Christoph Lüth

Universität Bremen

Wintersemester 2016/17



## Fahrplan

- ▶ Teil I: Funktionale Programmierung im Kleinen
- ▶ Teil II: Funktionale Programmierung im Großen
- ▶ Teil III: Funktionale Programmierung im richtigen Leben
  - ▶ Aktionen und Zustände
  - ▶ Monaden als Berechnungsmuster
  - ▶ Domänenspezifische Sprachen (DSLs)
  - ▶ **Scala — Eine praktische Einführung**
  - ▶ Rückblick & Ausblick



## Organisatorisches

- ▶ Anmeldung zu den Fachgesprächen ab sofort möglich
  - ▶ Unter stud.ip, Reiter „Terminvergabe“
- ▶ Nächste Woche noch mehr zu den Fachgesprächen
- ▶ Es gibt eine Liste mit Übungsfragen (auf der Homepage, unter Übungsblätter)



## Heute: Scala

- ▶ A **scalable language**
- ▶ Rein objektorientiert
- ▶ Funktional
- ▶ Eine "JVM-Sprache"
- ▶ Seit 2004 von Martin Odersky, EPFL Lausanne (<http://www.scala-lang.org/>).
- ▶ Seit 2011 kommerziell durch Lightbend Inc. (formerly Typesafe)



## Scala am Beispiel: 01-GCD.scala

Was sehen wir hier?

```
def gcdLoop(x: Long, y: Long): Long = {
  var a = x
  var b = y
  while (a != 0) {
    val temp = a
    a = b % a
    b = temp
  }
  return b
}

def gcd(x: Long, y: Long): Long =
  if (y == 0) x else gcd(y, x % y)
```

- ▶ Variablen, veränderlich (**var**)
  - ▶ *Mit Vorsicht benutzen!*
- ▶ Werte, unveränderlich (**val**)
- ▶ **while**-Schleifen
  - ▶ *Unnötig!*
- ▶ Rekursion
  - ▶ Endrekursion wird optimiert
- ▶ Typinferenz
  - ▶ Mehr als Java, weniger als Haskell
- ▶ Interaktive Auswertung



## Scala am Beispiel: 02-Rational-1.scala

Was sehen wir hier?

```
class Rational(n: Int, d: Int) {
  require(d != 0)
  private val g = gcd(n.abs, d.abs)
  val number = n / g
  val denom = d / g

  def this(n: Int) = this(n, 1)

  def add(that: Rational): Rational =
    new Rational(
      number * that.denom + that.number *
        denom,
      denom * that.denom
    )

  override def toString = number + "/" + denom

  private def gcd(a: Int, b: Int): Int =
    if (b == 0) a else gcd(b, a % b)
}
```

- ▶ Klassenparameter
- ▶ Konstruktoren (**this**)
- ▶ Klassenvorbedingungen (**require**)
- ▶ private Werte und Methoden
- ▶ Methoden, Syntax für Methodenanwendung
- ▶ **override** (nicht optional)
- ▶ Overloading
- ▶ Operatoren
- ▶ Companion objects (**object**)



## Algebraische Datentypen: 03-Expr.scala

Was sehen wir hier?

```
abstract class Expr
case class Var(name: String) extends Expr
case class Number(num: Double) extends Expr
case class UnOp(operator: String, arg: Expr)
  extends Expr
case class BinOp(operator: String,
  left: Expr, right: Expr) extends Expr

def eval(expr: Expr): Double = expr match {
  case v: Var => 0 // Variables evaluate to 0
  case Number(x) => x
  case BinOp("+", e1, e2) => eval(e1) + eval(e2)
  case BinOp("*", e1, e2) => eval(e1) * eval(e2)
  case UnOp("-", e) => -eval(e)
}

val e = BinOp("+", Number(12),
  UnOp("-", BinOp("+", Number(2.3),
    Number(3.7))))
```

- ▶ **case class** erzeugt
  - ▶ Factory-Methode für Konstruktoren
  - ▶ Parameter als implizite **val**
  - ▶ abgeleitete Implementierung für toString, equals
  - ▶ ... und pattern matching (**match**)
- ▶ Pattern sind
  - ▶ **case 4** => Literale
  - ▶ **case C(4)** => Konstruktoren
  - ▶ **case C(x)** => Variablen
  - ▶ **case C(\_)** => Wildcards
  - ▶ **case x: C** => getypte pattern
  - ▶ **case C(D(x: T, y), 4)** => geschachtelt

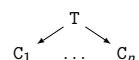


## Implementierung algebraischer Datentypen

Haskell:

```
data T = C1 | ... | Cn
```

Scala:



- ▶ Ein Typ T
- ▶ Konstruktoren erzeugen Datentyp

- ▶ Varianten als **Subtypen**
- ▶ Problem und Vorteil: **Erweiterbarkeit**
- ▶ **sealed** verhindert Erweiterung



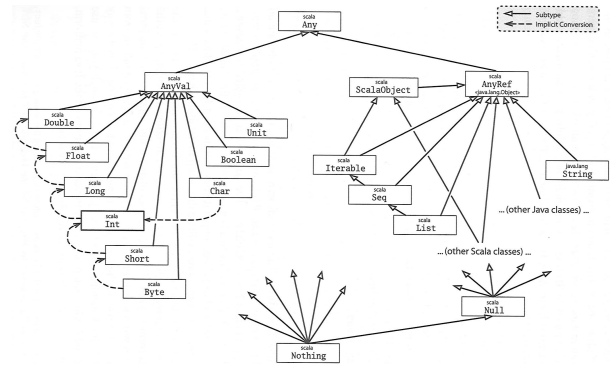
## Das Typsystem

Das Typsystem behebt mehrere Probleme von Java:

- ▶ Werte vs. Objekte
- ▶ Scala vs. Java
- ▶ NULL references



## Vererbungshierarchie



Quelle: Odersky, Spoon, Venners: *Programming in Scala*



## Parametrische Polymorphie

- ▶ Typparameter (wie in Haskell, Generics in Java), Bsp. `List [T]`
- ▶ Problem: Vererbung und Polymorphie
- ▶ Ziel: wenn  $S < T$ , dann  $List [S] < List [T]$
- ▶ **Does not work** — 04-Ref .hs
- ▶ Warum?
  - ▶ Funktionsraum nicht monoton im ersten Argument
  - ▶ Sei  $X \subseteq Y$ , dann  $Z \rightarrow X \subseteq Z \rightarrow Y$ , aber  $X \rightarrow Z \not\subseteq Y \rightarrow Z$
  - ▶ Sondern  $Y \rightarrow Z \subseteq X \rightarrow Z$



## Typvarianz

- |   |   |   |
|---|---|---|
| <p><b>class C[+T]</b></p> <ul style="list-style-type: none"> <li>▶ <b>Kovariant</b></li> <li>▶ Wenn <math>S &lt; T</math>, dann <math>C[S] &lt; C[T]</math></li> <li>▶ Parametertyp T nur im Wertebereich von Methoden</li> </ul> | <p><b>class C[T]</b></p> <ul style="list-style-type: none"> <li>▶ <b>Rigide</b></li> <li>▶ Kein Subtyping</li> <li>▶ Parametertyp T kann beliebig verwendet werden</li> </ul> | <p><b>class C[-T]</b></p> <ul style="list-style-type: none"> <li>▶ <b>Kontravariant</b></li> <li>▶ Wenn <math>S &lt; T</math>, dann <math>C[T] &lt; C[S]</math></li> <li>▶ Parametertyp T nur im Definitionsbereich von Methoden</li> </ul> |
|---|---|---|

Beispiel:

```
class Funktion[-S, +T] {
  def apply(x:S) : T
}
```



## Traits: 05-Funny.scala

Was sehen wir hier?

- ▶ Trait (Mix-ins): abstrakte Klassen, Interfaces; Haskell: Typklassen
- ▶ „Abstrakte Klassen ohne Konstruktor“
- ▶ Unterschied zu Klassen:
  - ▶ Mehrfachvererbung möglich
  - ▶ Keine feste Oberklasse (`super` dynamisch gebunden)
  - ▶ Nützlich zur Strukturierung (Aspektororientierung)
- ▶ Nützlich zur Strukturierung:

*thin interface + trait = rich interface*

Beispiel: 05-Ordered.scala, 05-Rational.scala



## More Traits

- ▶ Ad-Hoc Polymorphie mit Traits
- ▶ Typklasse:

```
trait Show[T] {
  def show(value: T): String
}
```

- ▶ Instanz:

```
implicit object ShowInt extends Show[Int] {
  def show(value: Int) = value.toString
}
```

- ▶ In Aktion:

```
def print[T](value: T)(implicit show: Show[T]) = {
  println(show.show(value));
}
```



## Was wir ausgelassen haben...

- ▶ **Komprehension** (nicht nur für Listen)
- ▶ **Gleichheit**: `==` (final), `equals` (nicht final), `eq` (Referenzen)
- ▶ **Implizite** Parameter und Typkonversionen
- ▶ **Nebenläufigkeit** (Aktoren, Futures)
- ▶ Typsichere **Metaprogrammierung**
- ▶ Das *simple build tool* sbt
- ▶ Der JavaScript-Compiler `scala.js`



## Schlamm Schlacht der Programmiersprachen

	Haskell	Scala	Java
Klassen und Objekte	-	+	+
Funktionen höherer Ordnung	+	+	-
Typinferenz	+	(+)	-
Parametrische Polymorphie	+	+	+
Ad-hoc-Polymorphie	+	+	-
Typsichere Metaprogrammierung	+	+	-

Alle: Nebenläufigkeit, Garbage Collection, FFI



## Scala — Die Sprache

- ▶ Objekt-orientiert:
  - ▶ Veränderlicher, gekapselter **Zustand**
  - ▶ **Subtypen** und Vererbung
  - ▶ **Klassen** und **Objekte**
- ▶ Funktional:
  - ▶ Unveränderliche **Werte**
  - ▶ Parametrische und Ad-hoc **Polymorphie**
  - ▶ Funktionen höherer Ordnung
  - ▶ Hindley-Milner **Typinferenz**



## Beurteilung

- ▶ **Vorteile:**
  - ▶ Funktional programmieren, in der Java-Welt leben
  - ▶ Gelungene Integration funktionaler und OO-Konzepte
  - ▶ Sauberer Sprachentwurf, effiziente Implementierung, reiche Büchereien
- ▶ **Nachteile:**
  - ▶ Manchmal etwas **zu** viel
  - ▶ Entwickelt sich ständig weiter
  - ▶ One-Compiler-Language, vergleichsweise langsam
- ▶ Mehr Scala?
  - ▶ Besuchen Sie auch **Reaktive Programmierung** (SoSe 2017)

