



Praktische Informatik 3: Funktionale Programmierung

Vorlesung 13 (24.01.23): Eine praktische Einführung in Scala

Christoph Lüth



Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH



Universität
Bremen

Wintersemester 2022/23

Organisatorisches

- ▶ Erinnerung: elektronische Klausur am **13.03.2023 um 14:00/15:45**
- ▶ Zwei Slots zu 90 Minuten
- ▶ Elektronische Registrierung ab morgen
- ▶ Alte Klausuren werden auf der Webseite zur Verfügung gestellt
- ▶ Nächste Woche mehr

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
 - ▶ Funktionale Webanwendungen
 - ▶ **Scala — Eine praktische Einführung**
 - ▶ Rückblick & Ausblick

Heute: Scala

- ▶ A **scal**able **lan**guage
- ▶ 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)

I. Scala am Beispiel

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)
```

Scala am Beispiel: 01-GCD.scala

Was sehen wir hier?

► Variablen, veränderlich (**var**)

```
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)
```

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!***

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**)

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

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!*

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

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

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 numer = n / g  
  val denom = d / g  
  
  def this(n: Int) = this(n, 1)  
  
  def add(that: Rational): Rational =  
    new Rational(  
      numer * that.denom + that.numer * denom,  
      denom * that.denom  
    )  
  
  override def toString = numer + "/" + denom  
  
  private def gcd(a: Int, b: Int): Int =  
    if (b == 0) a else gcd(b, a % b)  
}
```

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 numer = n / g  
  val denom = d / g  
  
  def this(n: Int) = this(n, 1)  
  
  def add(that: Rational): Rational =  
    new Rational(  
      numer * that.denom + that.numer * denom,  
      denom * that.denom  
    )  
  
  override def toString = numer + "/" + denom  
  
  private def gcd(a: Int, b: Int): Int =  
    if (b == 0) a else gcd(b, a % b)  
}
```

► Klassenparameter

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 numer = n / g  
  val denom = d / g  
  
  def this(n: Int) = this(n, 1)  
  
  def add(that: Rational): Rational =  
    new Rational(  
      numer * that.denom + that.numer * denom,  
      denom * that.denom  
    )  
  
  override def toString = numer + "/" + denom  
  
  private def gcd(a: Int, b: Int): Int =  
    if (b == 0) a else gcd(b, a % b)  
}
```

- ▶ Klassenparameter
- ▶ Konstruktoren (**this**)

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 numer = n / g  
  val denom = d / g  
  
  def this(n: Int) = this(n, 1)  
  
  def add(that: Rational): Rational =  
    new Rational(  
      numer * that.denom + that.numer * denom,  
      denom * that.denom  
    )  
  
  override def toString = numer + "/" + denom  
  
  private def gcd(a: Int, b: Int): Int =  
    if (b == 0) a else gcd(b, a % b)  
}
```

- ▶ Klassenparameter
- ▶ Konstruktoren (**this**)
- ▶ Klassenvorbedingungen (**require**)

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 numer = n / g  
  val denom = d / g  
  
  def this(n: Int) = this(n, 1)  
  
  def add(that: Rational): Rational =  
    new Rational(  
      numer * that.denom + that.numer * denom,  
      denom * that.denom  
    )  
  
  override def toString = numer + "/" + 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

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 numer = n / g  
  val denom = d / g  
  
  def this(n: Int) = this(n, 1)  
  
  def add(that: Rational): Rational =  
    new Rational(  
      numer * that.denom + that.numer * denom,  
      denom * that.denom  
    )  
  
  override def toString = numer + "/" + 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

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 numer = n / g  
  val denom = d / g  
  
  def this(n: Int) = this(n, 1)  
  
  def add(that: Rational): Rational =  
    new Rational(  
      numer * that.denom + that.numer * denom,  
      denom * that.denom  
    )  
  
  override def toString = numer + "/" + 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)

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 numer = n / g  
  val denom = d / g  
  
  def this(n: Int) = this(n, 1)  
  
  def add(that: Rational): Rational =  
    new Rational(  
      numer * that.denom + that.numer * denom,  
      denom * that.denom  
    )  
  
  override def toString = numer + "/" + 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

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 numer = n / g  
  val denom = d / g  
  
  def this(n: Int) = this(n, 1)  
  
  def add(that: Rational): Rational =  
    new Rational(  
      numer * that.denom + that.numer * denom,  
      denom * that.denom  
    )  
  
  override def toString = numer + "/" + 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

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 numer = n / g  
  val denom = d / g  
  
  def this(n: Int) = this(n, 1)  
  
  def add(that: Rational): Rational =  
    new Rational(  
      numer * that.denom + that.numer * denom,  
      denom * that.denom  
    )  
  
  override def toString = numer + "/" + 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
- ▶ Singleton objects (**object**)

II. Das Typsystem

Algebraische Datentypen: 03-Expr.scala

Was sehen wir hier?

```
abstract class Expr
case class Var(name: String) extends Expr
case class Num (num: Double) extends Expr
case class Plus (left: Expr, right: Expr) extends Expr
case class Minus (left: Expr, right: Expr) extends Expr
case class Times (left: Expr, right: Expr) extends Expr
case class Div (left: Expr, right: Expr) extends Expr
```

```
// Evaluating an expression
def eval(expr: Expr): Double = expr match
  case v: Var => 0      // Variables evaluate to 0
  case Num(x) => x
  case Plus(e1, e2) => eval(e1) + eval(e2)
  case Minus(e1, e2) => eval(e1) - eval(e2)
  case Times(e1, e2) => eval(e1) * eval(e2)
  case Div(e1, e2) => eval(e1) / eval(e2)

val e = Times(Num(12), Plus(Num(2.3), Num(3.7)))
```

Algebraische Datentypen: 03-Expr.scala

Was sehen wir hier?

```
abstract class Expr
case class Var(name: String) extends Expr
case class Num(num: Double) extends Expr
case class Plus(left: Expr, right: Expr) extends Expr
case class Minus(left: Expr, right: Expr) extends Expr
case class Times(left: Expr, right: Expr) extends Expr
case class Div(left: Expr, right: Expr) extends Expr
```

```
// Evaluating an expression
def eval(expr: Expr): Double = expr match
  case v: Var => 0      // Variables evaluate to 0
  case Num(x) => x
  case Plus(e1, e2) => eval(e1) + eval(e2)
  case Minus(e1, e2) => eval(e1) - eval(e2)
  case Times(e1, e2) => eval(e1) * eval(e2)
  case Div(e1, e2) => eval(e1) / eval(e2)

val e = Times(Num(12), Plus(Num(2.3), Num(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

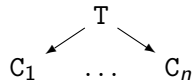
Implementierung algebraischer Datentypen

Haskell:

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

- ▶ Ein Typ T
- ▶ Konstruktoren erzeugen Datentyp

Scala:



- ▶ Varianten als **Subtypen**
- ▶ Problem und Vorteil:

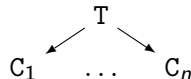
Implementierung algebraischer Datentypen

Haskell:

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

- ▶ Ein Typ T
- ▶ Konstruktoren erzeugen Datentyp

Scala:



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

☞ Siehe Übung 13.2

Das Typsystem

Das Typsystem behebt mehrere Probleme von Java:

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

The diagram illustrates the Scala type hierarchy. At the top is `scala.Any`. It branches into `scala.AnyVal` and `scala.AnyRef`. `scala.AnyVal` includes `scala.Double`, `scala.Float`, `scala.Long`, `scala.Int`, `scala.Short`, `scala.Byte`, `scala.Boolean`, `scala.Char`, and `scala.Unit`. `scala.AnyRef` includes `scala.Object` (which is `ScalaObject`), `scala.Iterable`, `scala.Seq`, `scala.List`, `scala.String` (labeled as `java.lang.String`), and other Java and Scala classes. `scala.Nothing` is a subtype of `scala.AnyVal` and `scala.AnyRef`. `scala.Null` is a subtype of `scala.Nothing`. A legend indicates that solid arrows represent subtypes and dashed arrows represent implicit conversions.

PI3 WS 22/23

III. Polymorphie und Vererbung

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.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.scala`
- ▶ 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

`class C[+T]`

- ▶ **Kovariant**
- ▶ Wenn $S < T$, dann $C[S] < C[T]$
- ▶ Parametertyp T nur im Wertebereich von Methoden

`class C[T]`

- ▶ **Rigide**
- ▶ Kein Subtyping
- ▶ Parametertyp T kann beliebig verwendet werden

`class C[-T]`

- ▶ **Kontravariant**
- ▶ Wenn $S < T$, dann $C[T] < C[S]$
- ▶ Parametertyp T nur im Definitionsbereich von Methoden
 - ☞ Siehe Übung 13.3

IV. Strukturierung mit Traits

Traits: 05-Funny.scala

Was sehen wir hier?

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

thin interface + trait = rich interface

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

Was wir ausgelassen haben...

- ▶ **Komprehension** (nicht nur für Listen)
- ▶ **Gleichheit**: `==` (final), `equals` (nicht final), `eq` (Referenzen)
- ▶ *string interpolation*
- ▶ **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

- ▶ 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 die Veranstaltung **Reaktive Programmierung** (soweit verfügbar)