

Reaktive Programmierung
Vorlesung 2 vom 29.04.14: A Practical Introduction to Scala

Christoph Lüth & Martin Ring

Universität Bremen

Sommersemester 2014

Fahrplan

- ▶ Teil I: Grundlegende Konzepte
 - ▶ Was ist Reaktive Programmierung?
 - ▶ Einführung in Scala
 - ▶ Die Scala Collections
 - ▶ Monaden
 - ▶ ScalaCheck
- ▶ Teil II: Nebenläufigkeit
- ▶ Teil III: Fortgeschrittene Konzepte

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 Typesafe Inc.

Scala am Beispiel: 01-GCD.scala

Was sehen wir hier?

Scala am Beispiel: 01-GCD.scala

Was sehen wir hier?

- ▶ Variablen, veränderlich
- ▶ Werte, unveränderlich
- ▶ `while`-Schleifen
- ▶ Rekursion — einfache Endrekursion wird optimiert
- ▶ Typinferenz — mehr als Java, weniger als Haskell

Scala am Beispiel: 01-GCD.scala

Was sehen wir hier?

- ▶ Variablen, veränderlich — *Mit Vorsicht benutzen!*
- ▶ Werte, unveränderlich
- ▶ `while`-Schleifen
- ▶ Rekursion — einfache Endrekursion wird optimiert
- ▶ Typinferenz — mehr als Java, weniger als Haskell

Scala am Beispiel: 01-GCD.scala

Was sehen wir hier?

- ▶ Variablen, veränderlich — *Mit Vorsicht benutzen!*
- ▶ Werte, unveränderlich
- ▶ `while`-Schleifen — *Unnötig!*
- ▶ Rekursion — einfache Endrekursion wird optimiert
- ▶ Typinferenz — mehr als Java, weniger als Haskell

Scala am Beispiel: 02-Rational.scala

Was sehen wir hier?

Scala am Beispiel: 02-Rational.scala

Was sehen wir hier?

- ▶ Klassenparameter
- ▶ `this`
- ▶ Methoden, Syntax für Methodenanwendung
- ▶ `override` (nicht optional)
- ▶ `private` Werte und Methoden
- ▶ Klassenvorbedingung (`require`)
- ▶ Overloading
- ▶ Operatoren

Algebraische Datentypen: 03-Expr.scala

Was sehen wir hier?

Algebraische Datentypen: 03-Expr.scala

Was sehen wir hier?

- ▶ case class erzeugt
 - ▶ Factory-Methode für Konstruktoren
 - ▶ Parameter als implizite val
 - ▶ abgeleitete Implementierung für toString, equals
 - ▶ ... und pattern matching
- ▶ 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

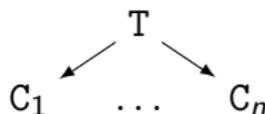
Implementierung algebraischer Datentypen

Haskell:

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

- ▶ Ein Typ T
- ▶ Konstruktoren erzeugen Datentyp

Scala:



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

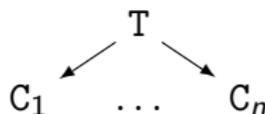
Implementierung algebraischer Datentypen

Haskell:

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

- ▶ Ein Typ T
- ▶ Konstruktoren erzeugen Datentyp

Scala:



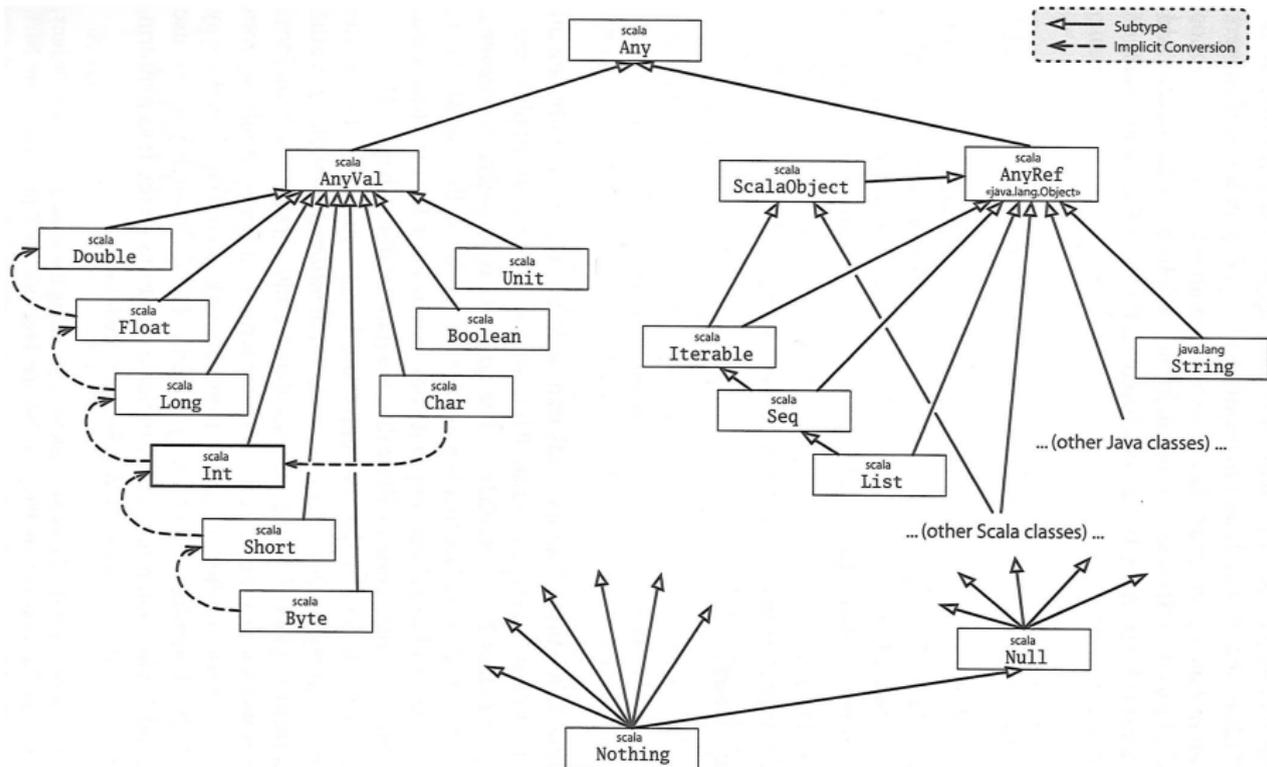
- ▶ Varianten als **Subtypen**
- ▶ Problem und Vorteil:
Erweiterbarkeit
- ▶ `sealed` verhindert Erweiterung

Das Typsystem

Behandelt:

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

Vererbungshierarchie



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

Parametrische Polymorphie

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

Typvarianz

class C[+T]

- ▶ **Kovariant**
- ▶ $S < T$, dann $C[S] < C[T]$
- ▶ Parameter T nicht in Def.bereich

class C[T]

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

class C[-T]

- ▶ **Kontravariant**
- ▶ $S < T$, dann $C[T] < C[S]$
- ▶ Parameter T nicht in Wertebereich

Beispiel:

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

Traits: 04-Funny.scala

Was sehen wir hier?

- ▶ Traits (Mix-ins): abstrakte Klassen, Interfaces; Haskell: Typklassen
- ▶ Unterschied zu Klassen:
 - ▶ Keine Parameter
 - ▶ Keine feste Oberklasse (super dynamisch gebunden)
- ▶ Nützlich zur Strukturierung:

thin interface + trait = rich interface

Beispiel: 04-Ordered.scala, 04-Rational.scala

Was wir ausgelassen haben...

- ▶ Komprehension (nicht nur für Listen)
- ▶ Gleichheit (`==`, `equals`)
- ▶ Implizite Parameter und Typkonversionen
- ▶ Nebenläufigkeit (Aktoren)

Scala — Die Sprache

- ▶ Objekt-orientiert:
 - ▶ Veränderlicher, gekapselter **Zustand**
 - ▶ Subtypen und Vererbung
 - ▶ Klassen und Objekte
- ▶ Funktional:
 - ▶ Unveränderliche **Werte**
 - ▶ Polymorphie
 - ▶ Funktionen höherer Ordnung

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