Reaktive Programmierung
Vorlesung 4 vom 23.04.15: A Practical Introduction to Scala

Christoph Lüth & Martin Ring

Universität Bremen

Sommersemester 2015

Fahrplan

- Teil I: Grundlegende Konzepte
  - Was ist Reaktive Programmierung?
  - Nebenläufigkeit und Monaden in Haskell
  - Funktional-Reaktive Programmierung
  - Einführung in Scala
  - Die Scala Collections
  - 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
- Klassenvorbedingunge (`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`
    - strukturelle Gleichheit
  - . . . 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

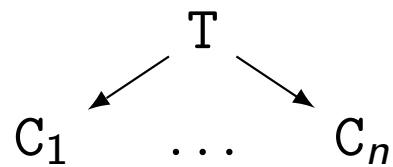
Implementierung algebraischer Datentypen

Haskell:

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

- Ein Typ `T`
- Konstruktoren erzeugen Datentyp

Scala:



- Varianten als Subtypen
- Problem und Vorteil:

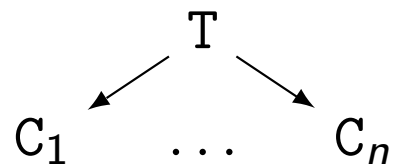
Implementierung algebraischer Datentypen

Haskell:

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

- Ein Typ `T`
- Konstruktoren erzeugen Datentyp

Scala:



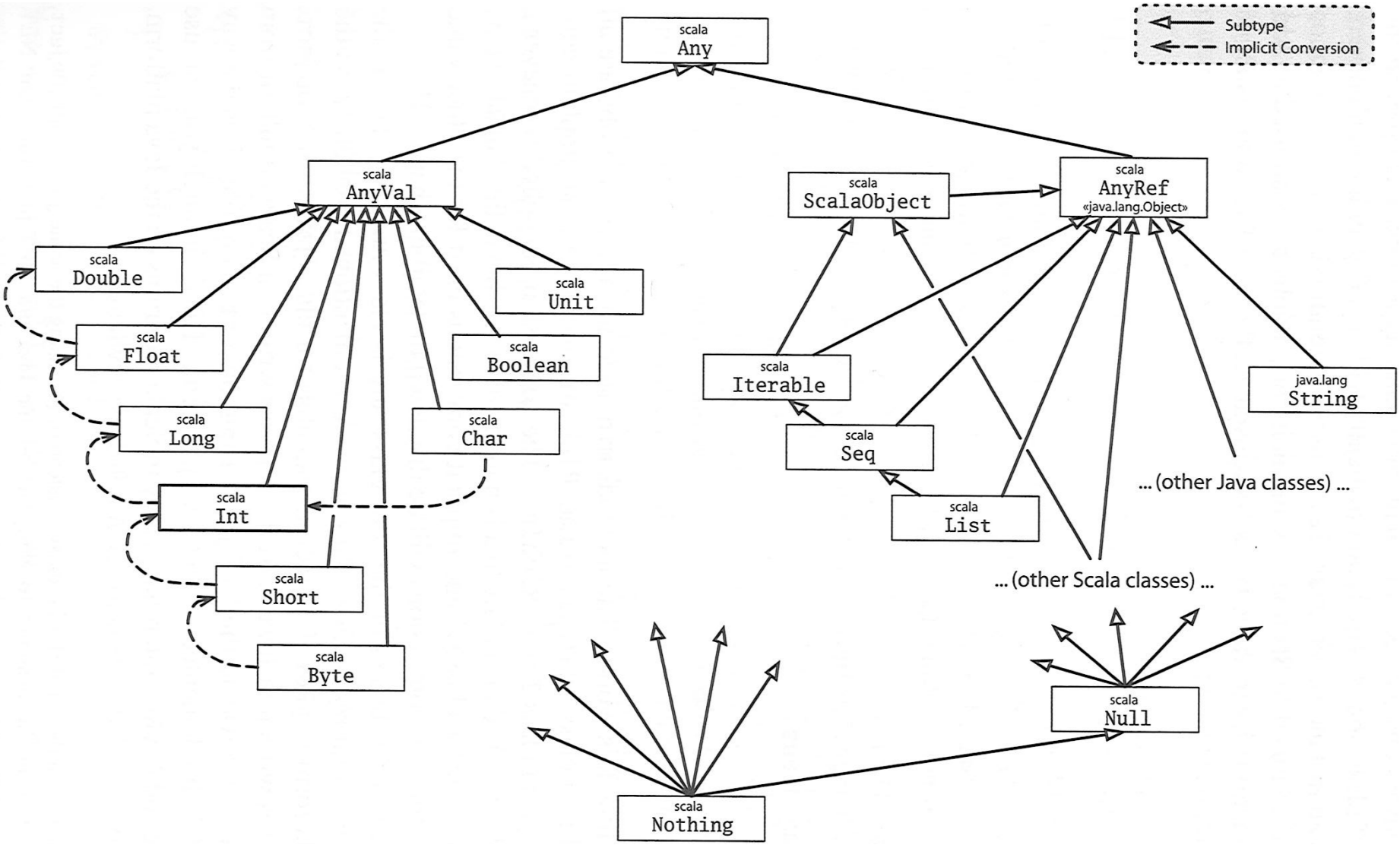
- Varianten als Subtypen
- Problem und Vorteil: Erweiterbarkeit
- `sealed` verhindert Erweiterung

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 $Z \longrightarrow X \subseteq Z \longrightarrow Y$, aber $X \longrightarrow Z \not\subseteq Y \longrightarrow Z$

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:

$$\textit{thin interface} + \textit{trait} = \textit{rich interface}$$

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

Was sind Traits?

- Trait $\approx$ Abstrakte Klasse ohne Parameter:

```
trait Foo[T] {
  def foo: T
  def bar: String = "Hallo"
}
```

- Erlauben "Mehrfachvererbung":

```
class C extends Foo[Int] with Bar[String] { ... }
```

- Können auch als Mixins verwendet werden:

```
trait Funny {
  def laugh() = println("hahaha")
}

(new C with Funny).laugh() // hahaha
```

Implizite Parameter

- Implizite Parameter:

```
def laugh(implicit stream: PrintStream) =
  stream.println("hahaha")
```

Implizite Parameter

- Implizite Parameter:

```
def laugh(implicit stream: PrintStream) =
  stream.println("hahaha")
```

- Werden im Kontext des Aufrufs aufgelöst. (Durch den Typen)

Implizite Parameter

- Implizite Parameter:

```
def laugh(implicit stream: PrintStream) =
  stream.println("hahaha")
```

- Werden im Kontext des Aufrufs aufgelöst. (Durch den Typen)
- Implizite Parameter + Traits $\approx$ Typklassen:

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

def print[T](value: T)(implicit show: Show[T]) =
  println(show.show(value))

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

print(7)
```

Implizite Konversionen

- Implizite Konversionen:

```
implicit def stringToInt(string: String) = string.toInt

val x: Int = "3"
x * "5" == 15 // true
"5" % "4" == 1 // true
```

Implizite Konversionen

- Implizite Konversionen:

```
implicit def stringToInt(string: String) = string.toInt

val x: Int = "3"
x * "5" == 15 // true
"5" % "4" == 1 // true
```

- Mit großer Vorsicht zu genießen!
- “Extension Methods” / “Pimp-My-Library” allerdings sehr nützlich!

Implizite Konversionen

- Implizite Konversionen:

```
implicit def stringToInt(string: String) = string.toInt

val x: Int = "3"
x * "5" == 15 // true
"5" % "4" == 1 // true
```

- Mit großer Vorsicht zu genießen!
- “Extension Methods” / “Pimp-My-Library” allerdings sehr nützlich!
- Besser: Implizite Klassen

```
implicit class RichString(s: String) {
  def shuffle = Random.shuffle(s.toList)
                      .mkString
}

"Hallo".shuffle // "laoHl"
```

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