

Praktische Informatik 3: Funktionale Programmierung
Vorlesung 13 vom 22.01.13: Scala — When Java meets Functional

Christoph Lüth

Universität Bremen

Wintersemester 2012/13

Organisatorisches

- ▶ **Fachgespräche:**

- ▶ Vorschlag Termine: 01.02., 11.02.
- ▶ Dritter Termin ggf. 12.02. (nach Bedarf)

- ▶ **Evaluation:**

- ▶ Es wird einen **Online-Fragebogen** geben (unter Stud.IP)
- ▶ Bitte **ausfüllen!**

Fahrplan

- ▶ Teil I: Funktionale Programmierung im Kleinen
- ▶ Teil II: Funktionale Programmierung im Großen
- ▶ Teil III: Funktionale Programmierung im richtigen Leben
 - ▶ Effizienzaspekte
 - ▶ Eine Einführung in Scala
 - ▶ Rückblick & Ausblick

Heute: Scala

- ▶ A **scalable language**
- ▶ Multi-paradigma-Sprache
- ▶ “Lebt im Java-Ökosystem”
- ▶ Martin Odersky, ETH Lausanne
- ▶ <http://www.scala-lang.org/>

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: 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`
 - ▶ ... und pattern matching
- ▶ Pattern sind
 - ▶ Konstanten
 - ▶ Konstruktoren
 - ▶ Variablen
 - ▶ Wildcards
 - ▶ getypte pattern

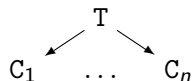
Implementierung algebraischer Datentypen

Haskell:

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

- ▶ Ein Typ T
- ▶ Konstruktoren erzeugen Datentyp

Scala:



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

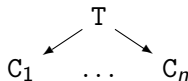
Implementierung algebraischer Datentypen

Haskell:

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

- ▶ Ein Typ T
- ▶ Konstruktoren erzeugen Datentyp

Scala:



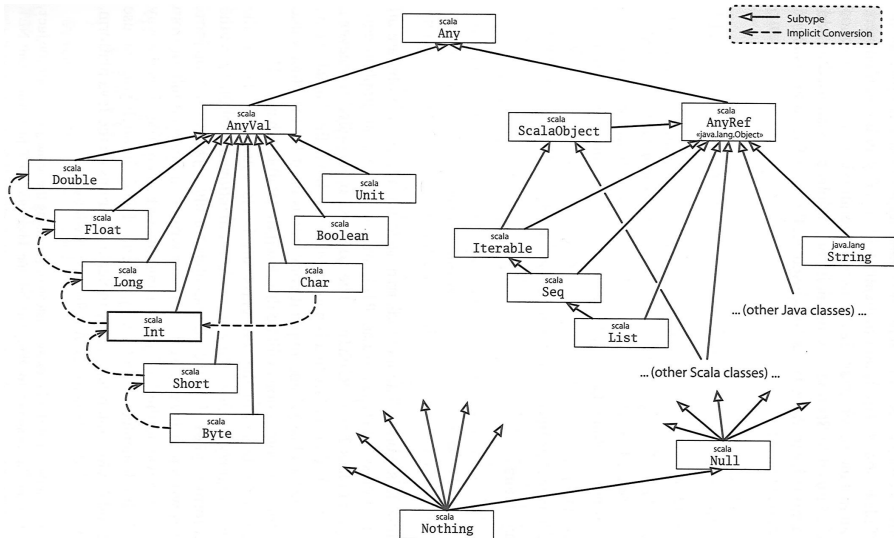
- ▶ Varianten als **Subtypen**
- ▶ Problem und Vorteil:
Erweiterbarkeit
- ▶ `sealed` verhindert Erweiterung

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
- ▶ Nativer XML Support
- ▶ 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