Praktische Informatik 3: Funktionale Programmierung
Vorlesung 13 vom 20.01.15: Scala — Eine praktische Einführung

Christoph Lüth

Universität Bremen

Wintersemester 2014/15

Fahrplan

- Teil I: Funktionale Programmierung im Kleinen
- Teil II: Funktionale Programmierung im Großen
- Teil III: Funktionale Programmierung im richtigen Leben
  - Aktionen und Zustände
  - Effizienzaspekte
  - Scala — Eine praktische Einführung
  - Rückblich & Ausblick

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?

- Interaktive Auswertung
- 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?

- Interaktive Auswertung
- 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?

- Interaktive Auswertung
- 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`
  - ...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

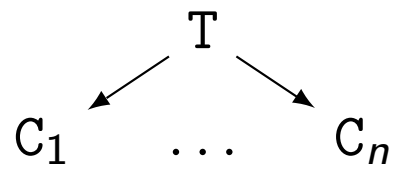
Implementierung algebraischer Datentypen

Haskell:

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

- Ein Typ `T`
- Konstruktoren erzeugen Datentyp

Scala:



- Varianten als Subtypen
- Problem und Vorteil:

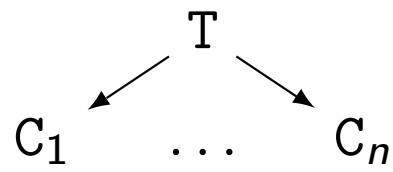
Implementierung algebraischer Datentypen

Haskell:

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

- Ein Typ `T`
- Konstruktoren erzeugen Datentyp

Scala:



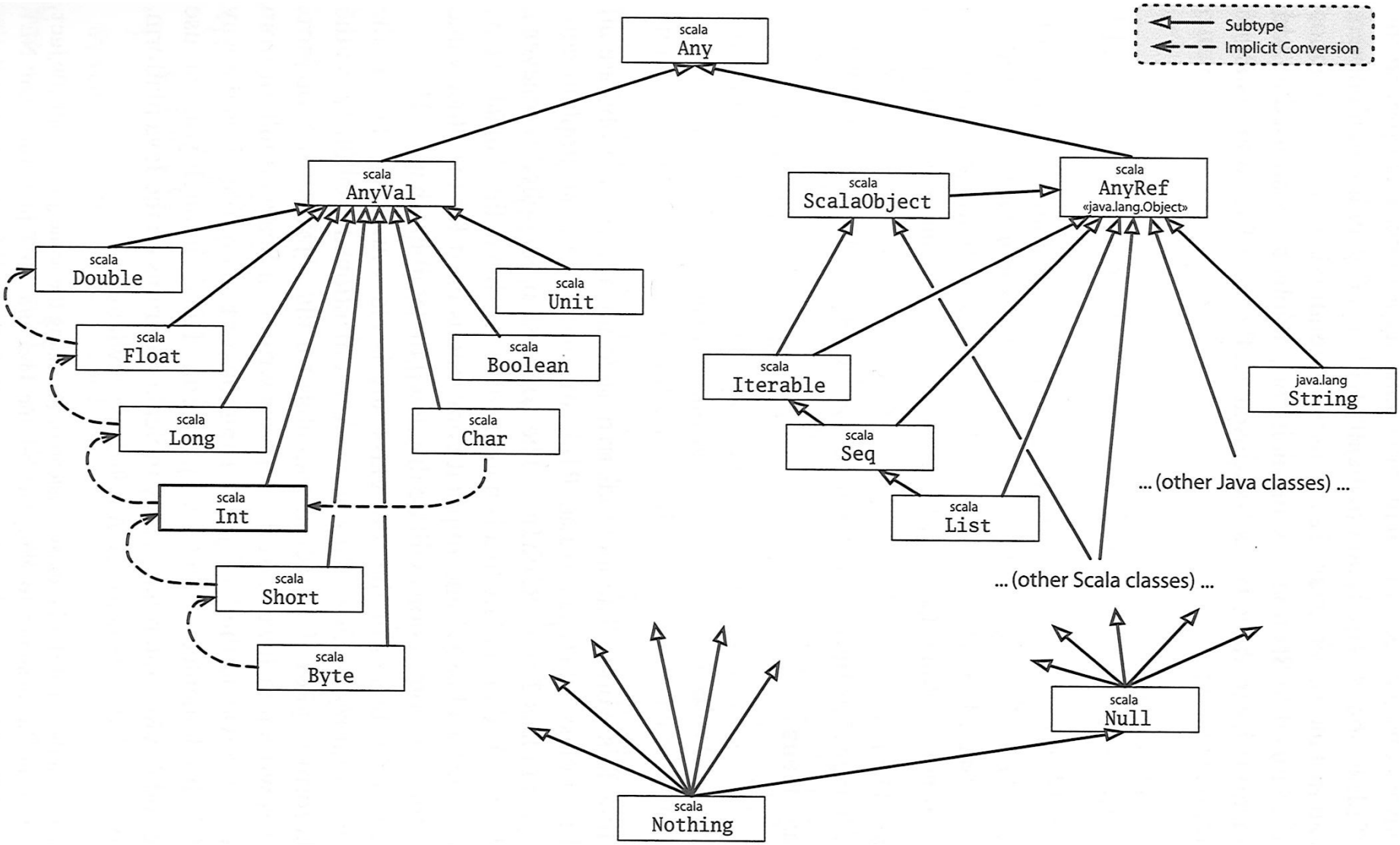
- Varianten als Subtypen
- Problem und Vorteil: Erweiterbarkeit
- `sealed` verhindert Erweiterung

Das Typsystem

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`

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

Typvarianz

`class C[+T]`

- Kovariant
- `S` $<$ `T`, dann `C[S]` $<$ `C[T]`
- Parameter `T` nur im Wertebereich von Methoden

`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` nur im Definitionsbereich von Methoden

Beispiel:

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

Traits: `05-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: `05-Ordered.scala`, `05-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
- Mehr Scala?
  - Besuchen Sie auch unsere Veranstaltung Reaktive Programmierung