

Praktische Informatik 3: Einführung in die Funktionale Programmierung
Vorlesung vom 02.02.2011: The Next Big Thing — Scala

Christoph Lüth & Dennis Walter

Universität Bremen

Wintersemester 2010/11

Rev. 1368

1 [21]

Fahrplan

- ▶ Teil I: Funktionale Programmierung im Kleinen
- ▶ Teil II: Funktionale Programmierung im Großen
- ▶ Teil III: Funktionale Programmierung im richtigen Leben
 - ▶ Effizient Funktional Programmieren
 - ▶ Fallstudie: Kombinatoren
 - ▶ Eine Einführung in Scala
 - ▶ Rückblick & Ausblick

2 [21]

Scala

- ▶ A **scalable language**
- ▶ Multi-paradigm language: funktional + oobjektorientiert
- ▶ „Lebt im Java-Ökosystem“
- ▶ Martin Odersky, ETH Lausanne
- ▶ <http://www.scala-lang.org/>

3 [21]

Scala — Die Sprache

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

4 [21]

Scala ist skalierbar

- ▶ „A language that grows on you.“
- ▶ Statt fertiger Sprache mit vielen Konstrukten **Rahmenwerk** zur Implementation eigener Konstrukte:
 - ▶ Einfache Basis
 - ▶ Flexible Syntax
 - ▶ Flexibles Typsystem
- ▶ Nachteil: Easy to learn but hard to master.
- ▶ Einfach zu benutzen:
 - ▶ Leichtgewichtig durch Typinferenz und Interpreter

5 [21]

Durchgängige Objektorientierung

- ▶ **Alles** in Scala ist ein Objekt
 - ▶ Keine primitiven Typen
- ▶ Operatoren sind Methoden
 - ▶ Beliebig überladbar
- ▶ Kein **static**, sondern Singleton-Objekte (**object**)
- ▶ Beispiel: `Rational.scala`

6 [21]

Werte

- ▶ Veränderliche Variablen **var**, unveränderliche Werte **val**
 - ▶ Zuweisung an Werte nicht erlaubt
- ▶ Dadurch unveränderliche Objekte → referentielle Transparenz
- ▶ “Unreine” Sprache
- ▶ **lazy val**: wird nach Bedarf ausgewertet.
- ▶ Sonderbehandlung von **Endrekursion** für bessere Effizienz
 - ▶ Damit **effiziente** funktionale Programmierung möglich
- ▶ Beispiel: `Gcd.scala`

7 [21]

Funktionale Aspekte

- ▶ Listen mit pattern matching
- ▶ Funktionen höherer Ordnung
- ▶ Listenkomprehension
- ▶ Beispiel: `Queens.scala`

8 [21]

Algebraische Datentypen

- ▶ Case Classes
 - ▶ Konzise Syntax für Konstruktoren: **factory methods**, kein **new**
 - ▶ Parameter werden zu **val**-Feldern
 - ▶ Pattern Match mit Selektoren
- ▶ Disjunkte Vereinigung durch Vererbung
- ▶ Beispiel: Expr.scala

9 [21]

Algebraische Datentypen und Vererbung

- ▶ Algebraische Datentypen können **erweitert** werden.
- ▶ Beispiel Expr:

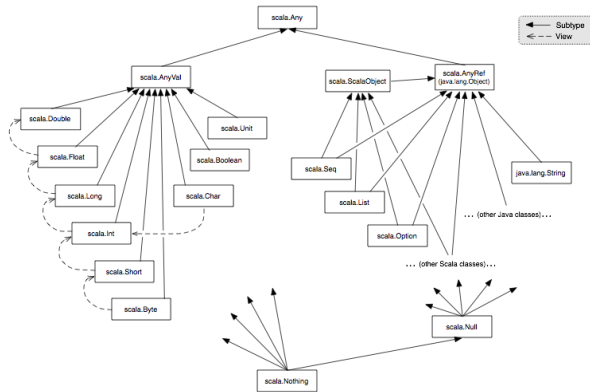

```
case class UnOp(op: String, e: Expr) extends Expr
```

- ▶ Vorteil: **flexibel**
- ▶ Nachteil: Fehleranfällig
- ▶ Verbot der Erweiterung: **sealed classes**

```
sealed abstract class Expr
...
```

10 [21]

Scala: Klassenhierarchie



11 [21]

Polymorphie und Subtypen

- ▶ Generische Typen (Scala) \cong Parametrische Polymorphie (Haskell)
 - ▶ In Scala besser als in Java wegen Typinferenz
- ▶ Problem mit generischen Typen und Polymorphie: **Varianz**
 - ▶ Gegeben List [T]
 - ▶ Ist List [String] < List [AnyRef]?
 - ▶ Gilt das immer? **Nein!**

12 [21]

Typvarianz

Gegeben folgende Klasse:

```
class Cell[T](init: T) {
  private var curr = init
  def get = curr
  def set (x: T) = { curr = x }
}
```

Problem: Ist Cell [String] < Cell [Any]?

```
val c1 = new Cell[String]("abc")
val c2 : Cell[Any] = c1
c2.set(1)
val s : String = c1.get
```

Also: Cell [String] **kein** Untertyp von Cell [Any]

13 [21]

Java: das gleiche Problem

Gegeben:

```
String [] a1 = { "abc" };
Object [] a2 = a1;
a2[0] = new Integer(1);
String s = a1[0];
```

Bei Ausführung **Laufzeitfehler**:

```
# javac Covariance.java
# java Covariance
Exception in thread "main" java.lang.ArrayStoreException: java.lang.
    at Covariance.main(Covariance.java:8)
```

14 [21]

Das Problem: Kontravarianz vs. Kovarianz

- ▶ Problem ist Typ von set : T => Cell [T] => ()
 - ▶ Nicht **var** und Zuweisung
- ▶ **Kovariant**:
 - ▶ Rechts des Funktionspfeils (Resultat)
 - ▶ Erhält Subtypenbeziehung
- ▶ **Kontravariant**:
 - ▶ Links des Funktionspfeils (Argument)
 - ▶ Kehrt Subtypenbeziehung
- ▶ **Position** der Typvariablen bestimmt **Varianz**:
 - ▶ Gegeben Mengen A, B, X mit A \subseteq B
 - ▶ Dann ist X \rightarrow A \subseteq X \rightarrow B
 - ▶ Aber **nicht** A \rightarrow X \subseteq B \rightarrow X
- ▶ Annotation der Varianz: Set [+T], Map [-T]

15 [21]

Beschränkte Polymorphie

Gegeben Listen:

```
abstract class List [+T]
case object Nil extends List [Nothing]
case class :: [T](hd: T, tl: List [T]) extends List [T]
```

- ▶ Problem: An List [T] kann nur T gehängt werden
- ▶ Wünschenswert: beliebiger Untertyp von T
- ▶ Lösung: **bounded polymorphism**

```
case class :: [U >: T](hd: U, tl: List [T])
    extends List [T]
```

16 [21]

Traits

```
trait Ordered[A] {  
  def cmp(a: A): Int  
  
  def < (a: A): Boolean = (this cmp a) < 0  
  def > (a: A): Boolean = (this cmp a) > 0  
  def ≤ (a: A): Boolean = (this cmp a) ≤ 0  
  def ≥ (a: A): Boolean = (this cmp a) ≥ 0  
  def cmpTo(that: A): Int = cmp(that) }  
  
class Rational extends Ordered[Rational] {  
  ...  
  def cmp(r: Rational) =  
    (this.numer * r.denom) - (r.numer * this.denom)}
```

- ▶ Mächtiger als Interfaces (Java): kann Implementierung enthalten
- ▶ Mächtiger als abstrakte Klassen: Mehrfachvererbung
- ▶ Mächtiger als Typklassen (Haskell): mehr als ein Typ-Parameter

17 [21]

Weitere Besonderheiten: apply

- ▶ apply erlaubt Definition von Factory-Methoden und mehr:
- ▶ $f(i)$ wird syntaktisch zu $f.apply(i)$
- ▶ Anderes Beispiel: Selektion aus array, Funktionen

18 [21]

Weitere Besonderheiten: Extraktoren

```
object EMail {  
  def apply(user: String, domain: String) =  
    user + "@" + domain  
  def unapply(str: String): Option[(String, String)] =  
    { val ps = str split "@"  
      if (ps.length == 2) Some(ps(0), ps(1)) else None  
    }  
}
```

- ▶ Extraktoren erlauben **erweitertes** pattern matching:

```
val s = EMail("cxl@dfki.de")  
s match { case EMail(user, domain) => .. }
```

- ▶ **Typgesteuertes** pattern matching:

```
val x : Any  
x match { case EMail(user, domain) => .. }
```

19 [21]

Weitere Besonderheiten

- ▶ Native XML support, Beispiel: `CCTerm.scala`
- ▶ Implizite Konversionen und Parameter

20 [21]

Zusammenfassung

- ▶ Haskell + Java = Scala (?)
- ▶ Skalierbare Sprache:
 - ▶ mächtige Basiskonstrukte
 - ▶ plus flexibles Typsystem
 - ▶ plus flexible Syntax ("syntaktischer Zucker")
- ▶ Die Zukunft von Java?

21 [21]