

Reaktive Programmierung  
Vorlesung 8 vom 15.05.19  
Meta-Programmierung

Christoph Lüth, Martin Ring

Universität Bremen

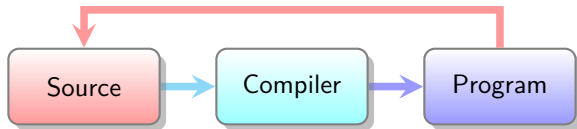
Sommersemester 2019

# Fahrplan

- ▶ Einführung
- ▶ Monaden und Monadentransformer
- ▶ Nebenläufigkeit: Futures and Promises
- ▶ Aktoren I: Grundlagen
- ▶ Aktoren II: Implementation
- ▶ **Meta-Programmierung**
- ▶ Bidirektionale Programmierung
- ▶ Reaktive Ströme I
- ▶ Reaktive Ströme II
- ▶ Funktional-Reaktive Programmierung
- ▶ Software Transactional Memory
- ▶ Eventual Consistency
- ▶ Robustheit und Entwurfsmuster
- ▶ Theorie der Nebenläufigkeit, Abschluss

# Was ist Meta-Programmierung?

“Programme höherer Ordnung” / Makros



# Was sehen wir heute?

- ▶ Anwendungsbeispiel: JSON Serialisierung
- ▶ Meta-Programmierung in Scala:
  - ▶ Scala Meta
- ▶ Meta-Programmierung in Haskell:
  - ▶ Template Haskell
- ▶ Generische Programmierung in Scala und Haskell

# Beispiel: JSON Serialisierung

Scala

```
case class Person(  
  names: List[String],  
  age: Int  
)
```

Haskell

```
data Person = Person {  
  names :: [String],  
  age :: Int  
}
```

Ziel: Scala  $\xleftrightarrow{\text{JSON}}$  Haskell

# JSON: Erster Versuch

JSON1.scala

# JSON: Erster Versuch

JSON1.scala

- ▶ Unpraktisch: Für jeden Typ muss manuell eine Instanz erzeugt werden
- ▶ Idee: Makros for the win

# Klassische Metaprogrammierung (Beispiel C)

```
#define square(n) ((n)*(n))  
#define UpTo(i, n) for((i) = 0; (i) < (n); (i)++)
```

```
UpTo(i,10) {  
    printf("i squared is: %d\n", square(i));  
}
```

- ▶ Eigene Sprache: C Präprozessor
- ▶ Keine Typsicherheit: einfache String Ersetzungen



# Metaprogrammierung in Scala: Scalameta

- ▶ Idee: Der Compiler ist im Programm verfügbar

```
> "x + 2 * 7".parse[Term].get.structure
```

```
Term.ApplyInfix(Term.Name("x"), Term.Name("+"), Nil,  
  Seq(Term.ApplyInfix(Lit.Int(2), Term.Name("*"), Nil,  
    Seq(Lit.Int(7))))))
```

- ▶ Abstrakter syntaxbaum (AST) als algebraischer Datentyp → typsicher
- ▶ Sehr komplexer Datentyp ...

# Quasiquotations

- ▶ Idee: Programmcode statt AST
- ▶ Zur Konstruktion ...

```
> val p = q"case class Person(name: String)"  
p: meta.Defn.Class = case class Person(name: String)
```

- ▶ ... und zur Extraktion

```
> val q"case class $name($param)" = p  
name: meta.Type.Name = Person  
param: scala.meta.Term.Param = name: String
```

# Makro Annotationen

- ▶ Idee: Funktion  $AST \rightarrow AST$  zur Compilezeit ausführen
- ▶ Werkzeug: Annotationen

```
class hello extends StaticAnnotation {  
  inline def apply(defn: Any): Any = meta { defn match {  
    case q"object $name { ..$members }" =>  
      q"""object $name {  
        ..$members  
        def hello: Unit = println("Hello")  
      }"""  
    case _ => abort("@hello must annotate an object")  
  } }  
}
```

@hello object Test

# JSON: Zweiter Versuch

JSON2.scala

# JSON: Zweiter Versuch

JSON2.scala

- ▶ Generische Ableitungen für **case classes**
- ▶ Funktioniert das für alle algebraischen Datentypen?

# Generische Programmierung

- ▶ Beispiel: YAML statt JSON erzeugen
- ▶ Idee: Abstraktion über die Struktur von Definitionen
- ▶ Erster Versuch: `ToMap.scala`

# Generische Programmierung

- ▶ Beispiel: YAML statt JSON erzeugen
- ▶ Idee: Abstraktion über die Struktur von Definitionen
- ▶ Erster Versuch: `ToMap.scala`
  - ▶ Das klappt so nicht ...
  - ▶ Keine geeignete Repräsentation!

# Heterogene Listen

- ▶ Generische Abstraktion von Tupeln

```
> val l = 42 :: "foo" :: 4.3 :: HNil  
l: Int :: String :: Double :: HNil = ...
```

- ▶ Viele Operationen normaler Listen vorhanden:
- ▶ Was ist der parameter für flatMap?



# Heterogene Listen

- ▶ Generische Abstraktion von Tupeln

```
> val l = 42 :: "foo" :: 4.3 :: HNil  
l: Int :: String :: Double :: HNil = ...
```

- ▶ Viele Operationen normaler Listen vorhanden:
- ▶ Was ist der parameter für flatMap?  
⇒ Polymorphe Funktionen

# Records

- ▶ Uns fehlen namen
- ▶ Dafür: Records

```
> import shapeless._; record._; import syntax.singleton._  
> val person = ("name" →> "Donald") :: ("age" →> 70) ::  
  HNil
```

```
person: String with KeyTag[String("name"),String] :: Int  
  with KeyTag[String("age"),Int] :: HNil = Donald :: 70  
  :: HNil
```

```
> person("name")
```

```
res1: String = Donald
```

# Die Typklasse Generic

- ▶ Typklasse Generic[T]

```
trait Generic[T] {  
  type Repr  
  def from(r: Repr): T  
  def to(t: T): Repr  
}
```

- ▶ kann magisch abgeleitet werden:

```
> case class Person(name: String, age: Int)  
> val gen = Generic[Person]  
gen: shapeless.Generic[Person]{type Repr = String :: Int  
  :: shapeless.HNil} = ...
```

- ▶ → Makro Magie
- ▶ Funktioniert allgemein für algebraische Datentypen

# JSON Serialisierung: Teil 3

JSON3.scala

# Automatische Linsen

```
case class Address(street: String, city: String, zip: Int)
case class Person(name: String, age: Int, address: Address)

val streetLens = lens[Person] >> 'address >> 'street
```

# Zusammenfassung

- ▶ Meta-Programmierung: “Programme Höherer Ordnung”
- ▶ Scalameta: Scala in Scala manipulieren
- ▶ Quasiquotations: Reify and Splice
- ▶ Macros mit Scalameta:  $AST \rightarrow AST$  zur Compilezeit
- ▶ Äquivalent in Haskell: TemplateHaskell
- ▶ Generische Programmierung in Shapeless
- ▶ Äquivalent in Haskell: GHC.Generic