

Reaktive Programmierung
Vorlesung 6 vom 24.05.2022
Meta-Programmierung

Christoph Lüth, Martin Ring

Universität Bremen

Sommersemester 2022

Fahrplan

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

Was ist Meta-Programmierung?

"Programme höherer Ordnung" / Makros



Was sehen wir heute?

- ▶ Anwendungsbeispiel: JSON Serialisierung
- ▶ Meta-Programmierung in Scala:
 - ▶ Inlining
 - ▶ Compile-Time Operations
 - ▶ Typeclass Derivation
- ▶ 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 $\xleftarrow{\text{JSON}}$ Haskell

JSON: Erster Versuch

JSON1.scala

- ▶ Unpraktisch: Für jeden Typ muss manuell eine Instanz erzeugt werden
- ▶ Idee: Typklassenableitung 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

I. Metaprogrammierung in Haskell

Was brauchen wir?

- ▶ Idee: Funktion $AST \rightarrow AST$ zur Compilezeit ausführen
- ① Repräsentation des AST in der Sprache \rightarrow Template Haskell
- ② Konversion von und in diese Repräsentation \rightarrow Template Haskell, Reification und Splicing
- ③ Büchereien für häufige Use-Cases (e.g. JSON) \rightarrow Generics

Template Haskell I

- ▶ **Gettype** Repräsentation des AST in Haskell
 - ▶ Vgl. Reflektion in Java
 - ▶ Datentyp `Exp` für Ausdrücke, `Dec` für Deklarationen, ...
 - ▶ <https://hackage.haskell.org/package/template-haskell-2.10.0.0/docs/Language-Haskell-TH-Syntax.html>
 - ▶ Beispiel: Repräsentation von `("foo", x)`:

```
r = TupE [Just (LitE (StringL "foo")), Just (VarE (mkName "x"))]
```

Template Haskell II

- ▶ Reifikation: vom Code zur Repräsentation (Quotation via "Oxford brackets")

```
r2 = runQ [| ("foo", x) |]
```

- ▶ Splicing: von der Repräsentation zum Code

```
$(return r)$
```

- ▶ Q: The "quotation monad"
 - ▶ Erlaubt e.g. auch IO-Aktionen einzubetten (`runIO`).
- ▶ Beispiele: generische Selektion aus Tupeln, generisches Currying

Was ist mit Jason?

- ▶ Wir könnten Serialisiererin JSON mit Template Haskell schreiben.
- ▶ Aber: müsste für jeden Datentyp spezifisch sein.
- ▶ Besser: **generisch** für alle Datentypen
 - ▶ Konstante Konstruktoren werden zu Konstanten
 - ▶ Mehrstellige Konstruktoren werden zu Produkten
 - ▶ Namen durch Konstruktornamen gegeben

Generische Programmierung: Polynomiale Datentypen

- ▶ **Polynomiale** Datentypen sind gegeben durch
 - ▶ Produkte (d.h. konstante oder mehrstellige Konstruktoren)
 - ▶ Summen (mehrere Konstruktoren)
- ▶ Beispiele: Listen, Maybe, Bäume, ...
- ▶ Gegenbeispiel: alles mit Funktionsräumen, Sequenzen, Arrays
- ▶ Polynomiale Datentypen können **generisch** repräsentiert werden: **Generics**

Generische Programmierung: Generics

- ▶ Generische Repräsentation von Datentypen:
https://wiki.haskell.org/GHC.Generics#Representation_types
- ▶ Konversion von und nach **Generic**:
<https://hackage.haskell.org/package/base-4.16.1.0/docs/GHC-Generics.html#g:24>
- ▶ Beispiel:

```
data T = Null | Two T T deriving Generic
t = Two Null (Two Null Null)

from t
```

- ▶ Damit: **generische** Konversion von und nach JSON

II. Metaprogrammierung in Scala

Metaprogrammierung in Scala: Scalameta

- ▶ Idee: Typsichere Operationen zur Compilezeit

```
import scala.compiletime.constValue
import scala.compiletime.ops.int.*

type A = 3
type B = 4
type C = A + B
def sevenToFourteen(c: C) = constValue[C * 2]
```

Inlining

```
inline def assert(inline debug: Boolean, assertion: => Boolean): Unit =
  if debug then
  else if !debug then
  else error("debug mode must be a known value")
```

RP SS 2022

17 [26]



Heterogene Listen

- Generische Tupel

```
import compiletime._

type Concat[L <: Tuple, +R <: Tuple] <: Tuple = L match
  case EmptyTuple => R
  case h *: t => h *: Concat[t, R]

def concat[L <: Tuple, R <: Tuple](l: L, r: R): Concat[L,R] =
  runtime.Tuples.concat(l, r).asInstanceOf[Concat[L,R]]

val example: (Int, String, Boolean, Double) = concat((1, "Hallo"), (false, 4.2))
```

- Viele Operationen normaler Listen vorhanden:
- Was ist der parameter für map? => Polymorphe Funktionen

RP SS 2022

18 [26]



Polymorphe Funktionen und Match Types

- Match Types werden zur Compilezeit reduziert

```
type F[X] = X match
  case Int => Boolean
  case String => Int
  case Boolean => String
  case Double => Double
```

- Damit können Polymorphe Funktionen implementiert werden:

```
def f[X](x: X): F[X] = x match
  case i: Int => i > 0
  case s: String => s.length
  case b: Boolean => b.toString
  case d: Double => -d
```

```
val example2: (Boolean, Int, String, Double) = example.map(f)
```

RP SS 2022

19 [26]



Typklassenableitung

- Typklassen können abgeleitet werden:

```
enum Tree[T] derives Eq, Ordering, Show:
  case Branch(left: Tree[T], right: Tree[T])
  case Leaf(elem: T)
```

- Wird übersetzt zu

```
given [T: Eq] : Eq[Tree[T]] = Eq.derived
given [T: Ordering] : Ordering[Tree] = Ordering.derived
given [T: Show] : Show[Tree] = Show.derived
```

- Was macht Eq.derived?

RP SS 2022

20 [26]



Typklassenableitung

- Implementierung der Ableitung:

```
import scala.deriving.Mirror

object Eq:
  def derived[T](using Mirror.Of[T]): Eq[T] = ...
```

- Damit können wir für Alle Algebraischen Datentypen Typklassen ableiten, aber...
- Wie Sieht der Mirror für Tree aus?

RP SS 2022

21 [26]



Typklassenableitung: Mirrors

- Mirror für Tree

```
new Mirror.Sum:
  type MirroredType = Tree
  type MirroredElemTypes[T] = (Branch[T], Leaf[T])
  type MirroredMonoType = Tree[_]
  type MirroredLabel = "Tree"
  type MirroredElemLabels = ("Branch", "Leaf")

  def ordinal(x: MirroredMonoType): Int = x match
    case _: Branch[_] => 0
    case _: Leaf[_] => 1
```

RP SS 2022

22 [26]



Typklassenableitung: Mirrors

- Mirror für Branch

```
new Mirror.Product:
  type MirroredType = Branch
  type MirroredElemTypes[T] = (Tree[T], Tree[T])
  type MirroredMonoType = Branch[_]
  type MirroredLabel = "Branch"
  type MirroredElemLabels = ("left", "right")

  def fromProduct(p: Product): MirroredMonoType =
    new Branch(...)
```

RP SS 2022

23 [26]



Typklassenableitung: Mirrors

- Mirror für Leaf

```
new Mirror.Product:
  type MirroredType = Leaf
  type MirroredElemTypes[T] = Tuple1[T]
  type MirroredMonoType = Leaf[_]
  type MirroredLabel = "Leaf"
  type MirroredElemLabels = Tuple1["elem"]

  def fromProduct(p: Product): MirroredMonoType =
    new Leaf(...)
```

RP SS 2022

24 [26]



Json Zweiter Versuch

JSON2.scala

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

Zusammenfassung

- ▶ **Meta-Programmierung:** Programme, die Programme erzeugen
- ▶ Typsichere Manipulation des AST:
 - ▶ Template Haskell, Scalameta
- ▶ Damit **generische Programmierung**
 - ▶ Generic Haskell, Scala