Reaktive Programmierung
Vorlesung 4 vom 13.05.14: Monads

Christoph Lüth & Martin Ring

Universität Bremen

Sommersemester 2014

Fahrplan

- Teil I: Grundlegende Konzepte
  - Was ist Reaktive Programmierung?
  - Einführung in Scala
  - Die Scala Collections
  - Monaden
  - ScalaCheck
- Teil II: Nebenläufigkeit
- Teil III: Fortgeschrittene Konzepte

Wozu Monaden?

- Monaden dienen zur expliziten Modellierung von Seiteneffekten.
- Beispiel: Entwicklung eines Flugkontrollsystems
- Möglichkeit 1: Java

```
class FlightControl where {
  public Control something(Data d) ...
```

  - Enthält mögl. Seiteneffekte (z.B. Ausnahmen)
- Möglichkeit 2: Haskell

```
something :: Data-> IO Control
```

  - Wir erkennen Seiteneffekte am Typ
- Scala: Kombination der beiden
  - Implizite Seiteneffekte möglich
  - Monaden erlauben Seiteneffekte explizit zu machen
  - Dadurch erhöhte Sicherheit, weiterhin alte Seiteneffekte möglich

Was sind Monaden? Mathematische Grundlagen

Definition (Monoid)

Ein Monoid ist gegeben durch $\mathcal{M} = \langle M, 0, \otimes \rangle$ so dass

$$0 \otimes m = m = m \otimes 0$$
$$(x \otimes y) \otimes z = x \otimes (y \otimes z)$$

- Beispiele für Monoiden: natürliche Zahlen, Listen, Mengen, Funktionen, ...
- Monoide erlauben Terme "flachzuklopfen":

$$\begin{aligned} a \otimes ((b \otimes 0) \otimes c) \otimes (0 \otimes d) = \; & ((a \otimes b) \otimes c) \otimes d \\ & a \otimes (b \otimes (c \otimes d)) \\ & a \otimes b \otimes c \otimes d \end{aligned}$$

- Monaden: Generalisierung mit M als "Typkonstruktoren mit map" (Funktoren)

Monaden in Haskell

- Beispiel für Konstruktorklasse (Typklasse für Typkonstruktoren)

```
class Monad m where
    (>>=) :: m a -> (a -> m b) -> m b
    return :: a -> m a
    (>>) :: m a -> m b -> m b

    p >> q = p >>= \_ -> q
```

- Instanzen sind IO, Option, List, ...

Die Monadengesetze

- Für Monaden müssen (sollen) folgende drei Gleichungen gelten:

```
m >>= (\x-> k x >>= h)   = (m >>= k)>>= h
         return a >>= k   = k a
           m >>= return   = m
```

- >>= ist assoziativ
- return ist rechtes und linkes neutrales Element
- Was bedeutet das?
  - Bei der Verkettung von Berechnung ist die Klammerung irrelevant
  - return berechnet nichts

Monaden in Scala

- In Scala kann man Monaden als Trait modellieren:

```
trait M[T] {
  def flatMap[U](f: T⇒ M[U]): M[U]
}
def unit[T](x: T): M[T]
```

- Erster Parameter von bind (>>=) durch das Objekt gegeben.
- Die Monadengesetze in Scala-Notation:

```
m flatMap f flatMap g == m flatMap (x ⇒ f(x) flatMap g)
    unit(x) flatMap f == f(x)
        m flatMap unit == m
```


Option als Monade

- Option[X] ist Berechnung vom Typ X, die entweder fehlschlägt (None) oder ein Ergebnis zurückliefert (Some(x))
- flatMap propagiert None
- def unit(x)= Some(x)

```
scala> Some(4).flatMap((x)⇒ None)
res16: Option[Nothing] = None

scala> Some(4).flatMap((x)⇒ Some(x+1))
res17: Option[Int] = Some(5)

scala> (None:Option[Int]).flatMap((x)⇒ Some(x+1))
res18: Option[Int] = None
```

- Modelliertes Berechnungsmodell: Explizit partielle Funktionen

Either als Monad

- Either[E, X] ist wie Option[X], aber E kann Fehlerinformationen beinhalten.
- flatMap propagiert Left
- def unit(x)= Right(x)
- Beispiel:

```
def inv(x: Double)= if (x== 0) Left("Div 0!") else Right(1/x)
val t: Either[String, Double]= Right(3)
val z: Either[String, Double]= Right(0)
t.right.flatMap(inv)
z.right.flatMap(inv)
(t.right.flatMap(inv)).right.flatMap((x)⇒ Right(x*2))
(z.right.flatMap(inv)).right.flatMap((x)⇒ Right(x*2))
```

- Modelliertes Berechnungsmodell: Ausnahmen

Listen als Monade

- List[X] ist eine Berechnung vom Typ X mit mehreren Ergebnissen
- flatMap berechnet alle möglichen Kombinationen
- def unit(x)= List(x)
- Beispiel:

```
def f(x:String)= List(x.reverse, x++ x)
List("a", "b", "c").flatMap(f)
List("xy", "ab").flatMap(f)
List("xy", "ab").flatMap(f).flatMap(f)
List(1,2,3).flatMap(_.toString)
List(21,22,23).flatMap(_.toString)
```

- Modelliertes Berechnungsmodell: Mehrdeutige Berechnungen

Set als Monade

- Set[X] ist eine Berechnung vom Typ X mit mehreren Ergebnissen (Reihenfolge irrelevant)
- flatMap berechnet alle möglichen Kombinationen
- def unit(x)= Set(x)

```
scala> Set("a", "b", "c").flatMap((x)⇒ Set(x, x+ x, x+ x+
    x))
res35: scala.collection.immutable.Set[String] = Set(a, ccc,
    b, bbb, cc, c, aa, bb, aaa)
scala> Set(1,2,3).flatMap(_.toString)
res36: scala.collection.immutable.Set[Char] = Set(1, 2, 3)
scala> Set(21,22,23).flatMap(_.toString)
res37: scala.collection.immutable.Set[Char] = Set(2, 1, 3)
```

- Modelliertes Berechnungsmodell: Mehrdeutige Berechnungen ohne Berücksichtigung der Reihenfolge der Ergebnisse

Syntaktischer Zucker für Monaden

- Ein Generator:
  `for { x← e } yield f` steht für `e.map(x ⇒ f)`
- Mehrere Generatoren:
  `for {x← e; gens } yield f` steht für
  `x.flatMap (x⇒ for { gens } yield f)`

Fallbeispiel: Ein Interpreter

- Ein einfacher Datentyp für arithmetische Ausdrücke:

```
abstract class Expr
case class BinOp (op:String, left:Expr, right:Expr) extends
    Expr
case class Number (number:Int) extends Expr
```

- Eine einfache rekursive Auswertungsfunktion dazu:

```
def evalOp1(o:String, l:Int, r:Int) : Int = o match {
  case "*" ⇒ l * r
  case "-" ⇒ l - r
  case "+" ⇒ l + r }

def eval1(e:Expr) : Int = e match {
  case Number(n) ⇒ n
  case BinOp(o,l,r) ⇒ evalOp1(o,eval1(l),eval1(r)) }
```


Problem: Partialität

- Problem: Division durch 0 erzeugt Ausnahme.
- Lösung: explizite Modellierung durch Option

```
def evalOp2(o:String, l:Int, r:Int) : Option[Int] = o match {
  case "*" ⇒ Some(l * r)
  case "-" ⇒ Some(l - r)
  case "+" ⇒ Some(l + r)
  case "/" ⇒ if (r == 0) None else Some(l / r)
  }
```


Explizite Partialität durch Option

- Bei der Auswertung werden die Teilergebnisse mit for verkettet:

```
def eval2(e:Expr) : Option[Int] = e match {
  case Number(n) ⇒ Some(n)
  case BinOp(o,l,r) ⇒
  for {
    x ← eval2(l)
    y ← eval2(r)
    z ← evalOp2(o,x,y)
    } yield z }
```


Erweiterung: Mehrdeutige Ergebnisse

- Der Operator ? gibt zwei Ergebnisse zurück:

```
def evalOp3(o:String, l:Int, r:Int) : List[Int] = o match {
  case "*" ⇒ List(l * r)
  case "-" ⇒ List(l - r)
  case "+" ⇒ List(l + r)
  case "/" ⇒ if (r == 0) List() else List(l / r)
  case "?" ⇒ List(l, r) }

def eval3(e:Expr) : List[Int] = e match {
  case Number(n) ⇒ List(n)
  case BinOp(o,l,r) ⇒
  for {
    x ← eval3(l)
    y ← eval3(r)
    z ← evalOp3(o,x,y)
    } yield z }
```

Vergleich: Option und List

```
def evalOp2(o:String, l:Int, r:Int) : Option[Int] = o match {
  case "*" ⇒ Some(l * r)
  case "-" ⇒ Some(l - r)
  case "+" ⇒ Some(l + r)
  case "/" ⇒ if (r == 0) None else Some(l / r)
  }

def eval2(e:Expr) : Option[Int] = e match {
  case Number(n) ⇒ Some(n)
  case BinOp(o,l,r) ⇒
  for {
    x ← eval2(l)
    y ← eval2(r)
    z ← evalOp2(o,x,y)
    } yield z }


def evalOp3(o:String, l:Int, r:Int) : List[Int] = o match {
  case "*" ⇒ List(l * r)
  case "-" ⇒ List(l - r)
  case "+" ⇒ List(l + r)
  case "/" ⇒ if (r == 0) List() else List(l / r)
  case "?" ⇒ List(l, r) }

def eval3(e:Expr) : List[Int] = e match {
  case Number(n) ⇒ List(n)
  case BinOp(o,l,r) ⇒
  for {
    x ← eval3(l)
    y ← eval3(r)
    z ← evalOp3(o,x,y)
    } yield z }
```


Zusammenfassung

- Monaden dienen zur expliziten Modellierung von Seiteneffekten.
- Monaden sind ein Datentyp mit einem generischen Typ und zwei Operationen, die assoziativ und unitär sind (Monadengesetze)
- In Scala werden Monaden durch die Klasse Iterable und die for-Notation unterstützt.
- Fallbeispiel: Interpreter, Modellierung von Partialität und Mehrdeutigkeit
- Anmerkung: Kombination von Monaden nicht ganz einfach
  - Insbesondere ist M[M[T]] nicht notwendigerweise eine Monade