Software-Komponenten

Th. Letschert

THM

University of Applied Sciences

Monaden

- Listenartige Monaden,
- Fehlermanagement-Monaden
- Monaden-Definition
- Monaden-Gesetze

Übersicht

Funktoren, halbe, ganze und spezielle Monaden

Monade: Ein weiteres Muster zur Organisation von Verarbeitungsschritten

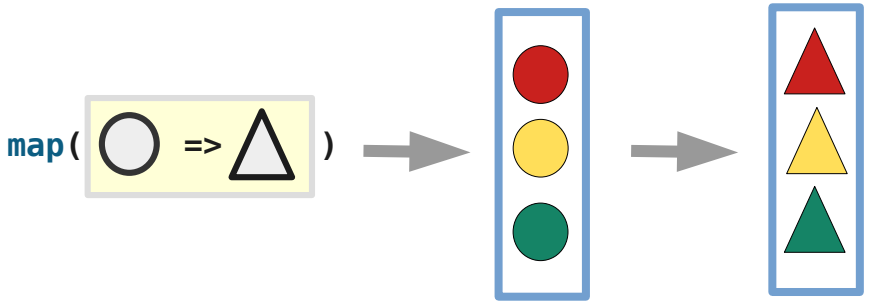
- **Funktor:**
  - Transparente Berechnungen in einem Kontext
  - die verkettbar sind
  - zentrale Operation: **map**
- **Semi-Monade: „Weiterentwickelter“ Funktor**
  - Transparente Berechnungen in einem Kontext
  - als geschachtete Iterationen verkettbar sind
  - (weitere) zentrale Operation: **flatMap**
- **Monade: „Weiterentwickelte“ Semi-Monade**
  - (weitere) zentrale („Konstruktor“) Operation: **pure**
- **Spezialisiere Monaden: Monaden mit zusätzlichen Fähigkeiten**
  - z.B.: Zur Fehlerbehandlung, Zur Verwaltung eines Zustands etc.
  - Also Basis-Komponenten von Bibliotheken wie Cats zur Verfügung gestellt
  - Basis für **Anwendungsentwicklung**

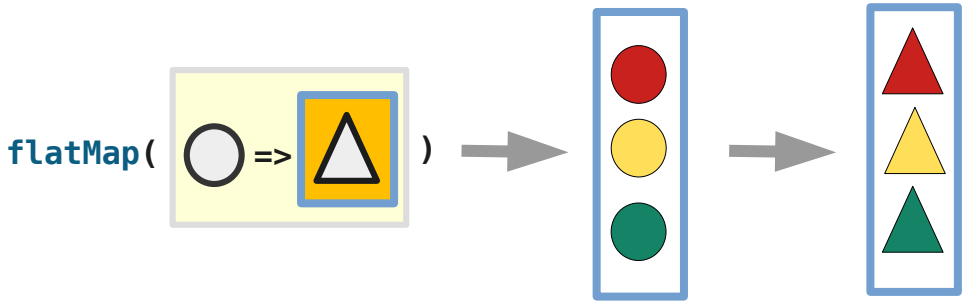
Semi-Monade: Funktor mit flatMap

map und flatMap

- map: hebt Funktionen auf dem Inhalt in einen Kontext
- flatMap: hebt Kontext-erzeugende Funktionen in einen Kontext

In Cats werden Semi-Monaden „*FlatMap*“ genannt.





Semi-Monade: Map und flatMap

FlatMap und geschachtelte Iterationen

flatmap hilft bei der Formulierung von geschachtelten Iterationen

Beispiel: alle Kombinationen der Elemente von zwei Listen

[a,b] , [1,2] => [(a,1), (a,2), (b,1), (b,2)]

```
val l1 = List("a", "b")
val l2 = List(1, 2)

val pairs =
  l1.map(x =>
    l2.map(y =>
      (x,y))).flatten
```

`List((a,1), (a,2), (b,1), (b,2))`

***map*: Verschachtelte Liste von Paaren muss flach geklopft werden**

```
val pairs =
  l1.flatMap(x =>
    l2.map(y =>
      (x,y)))
```

`List((a,1), (a,2), (b,1), (b,2))`

***flatMap*: übersichtlicher, Unverschachtelte flache Liste von Paaren**

```
val pairs =
  for (
    x <- l1;
    y <- l2
  ) yield (x, y)
```

`List((a,1), (a,2), (b,1), (b,2))`

***Mit for-Comprehension*: map/ flatMap in intuitiver Notation**

Semi-Monade: Map und flatMap

FlatMap und geschachtelte Iterationen

flatmap hilft bei der Formulierung von geschachtelten Iterationen

Beispiel: alle Permutationen einer Liste

```
def perms[A](lst: List[A]): List[List[A]] = lst match {
  case Nil => List(Nil)
  case head :: tail =>
    for (
      permsOfRest <- perms(lst.tail);
      headInpermsOfRest <- inserts(lst.head, permsOfRest)
    ) yield headInpermsOfRest
}


def inserts[A](x: A, lst: List[A]): List[List[A]] = lst match {
  case Nil => List(List(x))
  case head :: tail =>
    (x :: lst) :: (
        for(
          xInRest <- inserts[A](x, lst.tail)
        ) yield lst.head :: xInRest
      )
}

val ps = perms("abc".toList)                    "abc, bac, bca, acb, cab, cba"
    .map(_.mkString(""))
    .mkString(", ")
```

Monade: Semi-Monade + pure

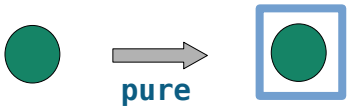
Monaden

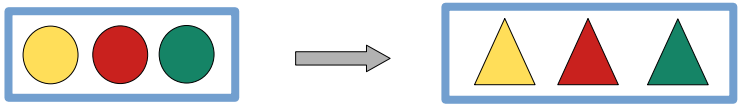
sind Semi-Monaden, also

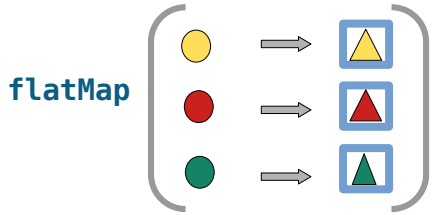
- Funktoren mit
- einer flatMap-Funktion

mit einer

- pure-Funktion

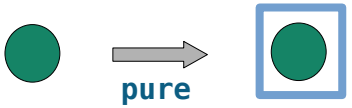




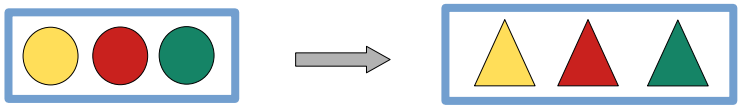


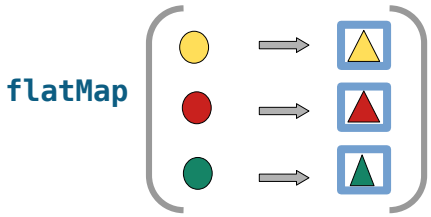
Monaden

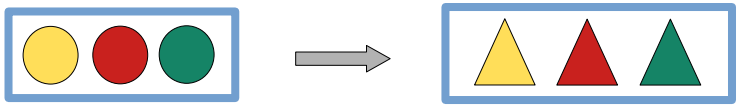
Monade als Typklasse

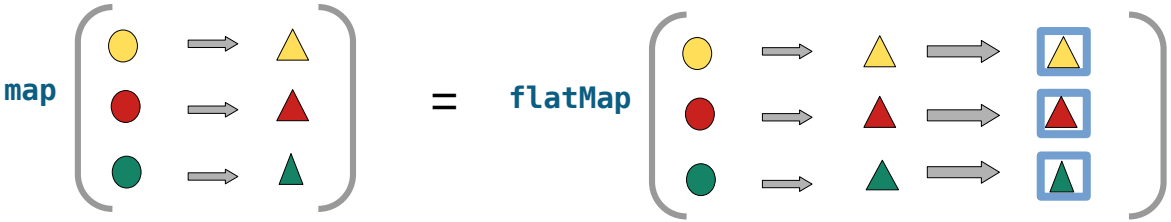



```
trait Functor[F[_]] {
  extension[A, B] (x: F[A]) def map(f: A => B): F[B]
}

trait Monad[F[_]] extends Functor[F] {

  def pure[A](x: A): F[A]

  extension[A, B] (x: F[A]) {

    def flatMap(f: A => F[B]): F[B]

    def map(f: A => B) = x.flatMap(f.andThen(pure))

  }
}
```









Listenartige Monaden

Listenartige Strukturen

Seq, List, LazyList, Array, Vector, ...

sind in Scala (und allen anderen (vernünftigen) Sprachen)

- filterbare Funktoren,
- mit flatMap
- und einem Konstruktor als pure-Funktion

Listenartige Monaden: geschachtelte Iterationen

flatMap und geschachtelte Iteration

ohne for-Comprehension

```
def pairs[F[_]: Monad, A, B](
      l1: F[A],
      l2: F[B]): F[(A, B)] =
  l1.flatMap(x =>
    l2.map(y =>
      (x,y)))
```

```
trait Functor[F[_]] {
  extension[A, B] (x: F[A]) def map(f: A => B): F[B]
}

trait Monad[F[_]] extends Functor[F] {
  def pure[A](x: A): F[A]
  extension[A, B] (x: F[A]) {
    def flatMap(f: A => F[B]): F[B]
    def map(f: A => B) = x.flatMap(f.andThen(pure))
  }
}

object Monad {
  def apply[F[_]: Monad] = summon[Monad[F]]
}
```

```
given Monad[List] with {
  def pure[A](x: A): List[A] =
    List(x)
  extension [A, B](xs: List[A]) {
    def flatMap(f: A => List[B]): List[B] =
      xs.flatMap(f)
    override def map(f: A => B) =
      xs.map(f)
  }
}

val l1 = List("a", "b")
val l2 = List(1, 2)

val pairs_l1_l2 = pairs(l1, l2)
```

flatMap der Klasse List

map der Klasse List, das vordefinierte map wird ignoriert

List((a,1), (a,2), (b,1), (b,2))

Listenartige Monaden: geschachtelte Iterationen

flatMap und geschachtelte Iteration

mit for-Comprehension

```
def pairs[F[_]: Monad, A, B](
      l1: F[A],
      l2: F[B]): F[(A, B)] =
  for (
    x <- l1;
    y <- l2
  ) yield (x, y)
```

```
given Monad[List] with {
  def pure[A](x: A): List[A] =
    List(x)
  extension [A, B](xs: List[A]) {
    def flatMap(f: A => List[B]): List[B] =
      xs.flatMap(f)
    override def map(f: A => B) =
      xs.map(f)
  }
}

val l1 = List("a", "b")
val l2 = List(1, 2)

val pairs_l1_l2 = pairs(l1, l2)
```

```
List((a,1), (a,2), (b,1), (b,2))
```

```
trait Functor[F[_]] {
  extension[A, B] (x: F[A]) def map(f: A => B): F[B]
}

trait Monad[F[_]] extends Functor[F] {
  def pure[A](x: A): F[A]
  extension[A, B] (x: F[A]) {
    def flatMap(f: A => F[B]): F[B]
    def map(f: A => B) = x.flatMap(f.andThen(pure))
  }
}

object Monad {
  def apply[F[_]: Monad] = summon[Monad[F]]
}
```

Fehler-Management – Monaden

Fehler-Management-Monade: Typ mit guter und böser Variante

Manche Typen haben zwei Varianten, die man als
„böse“ / „leer“ bzw. „gut“ / „gefüllt“ klassifizieren kann

Z.B.:

„Böse“ / „Leer“

„Gut“

- **Listen: leer / nicht leer**
- **Option: None / Some**
- **Either: Left / Right**
- **Try: Failure / Success**
- **…**

Das sind Fehler-Management / Pass- / Failure-Monaden

Die Varianten des Kontexts geben flatMap (also jeder Iteration) die Möglichkeit

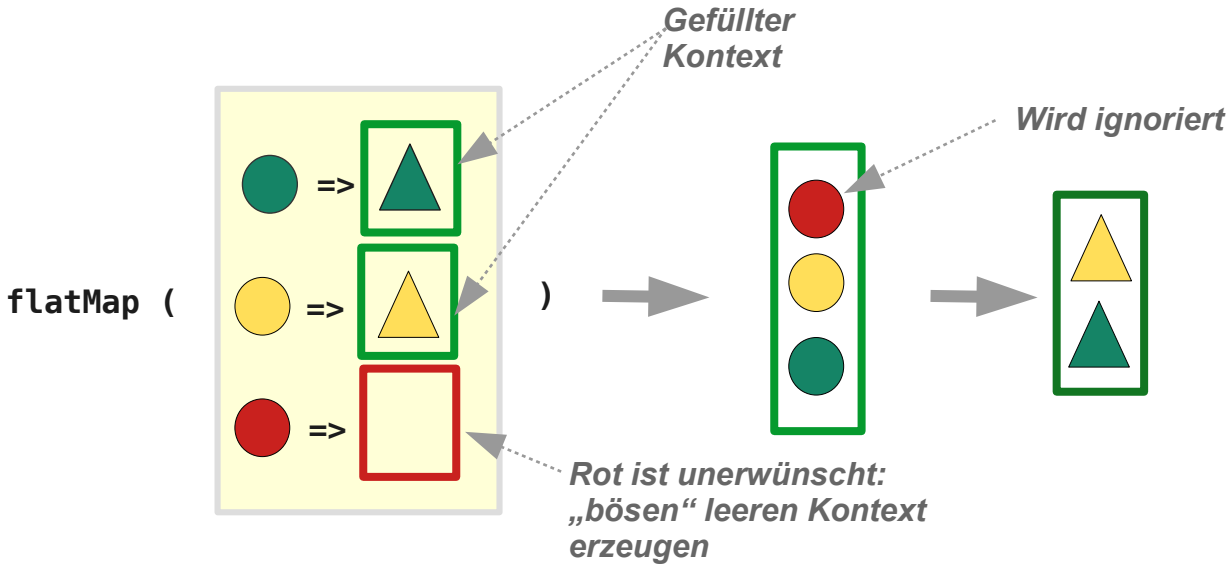
- **nicht nur eine neues Element erzeugen,**
  das dann in den Kontext eingefügt wird.
- **sie kann auch eine andere Art von Kontext (die „böse“ / „leere“ Variante) erzeugen,**
  die bei weiteren flatMaps ignoriert wird – ist ja leer!
- **Das gibt dem Konzept „Berechnung in einem Kontext“ erweiterte Ausdrucksmöglichkeiten**

List als Fehler-Management Monade

Die an flatMap übergebene Funktion kann

- ein **gutes** Ergebnis erzeugen: ein in eine Liste verpacktes, neues Element, oder
- ein **böses** Ergebnis erzeugen: die leere Liste,

Die leere Liste wird bei der Erzeugung des Gesamtergebnisses ignoriert

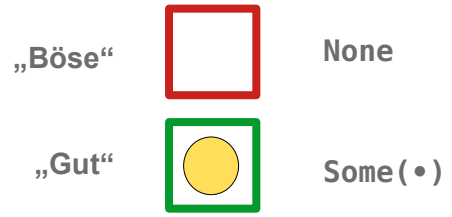


Fehler-Management – Monaden

Option

Option als Fehlermanagement-Monade

Beispiel:




```
val stringOpt1 = Some("1")
val stringOpt2 = Some("one")

val f: String => Option[Int] = str =>
  try {
    Some(str.toInt)
  } catch {
    case e: NumberFormatException => None
  }

val intOpt1 = stringOpt1.flatMap(f) // Some(1)
val intOpt2 = stringOpt2.flatMap(f) // None
```

Option

Beispiel Option

```scala
def readInt(): Option[Int] = try {
  val str = scala.io.StdIn.readLine()
  Some(str.toInt)
} catch {
  case _: java.lang.NumberFormatException =>
    None
}

def div42(divisor: Int): Option[Int] =
  if (divisor != 0)
    Some(42 / divisor)
  else
    None


def factors(n: Int): Option[List[Int]] = {
  def isPrime(n: Int): Boolean =
    (n == 2) || (2 to n / 2 + 1).count(n % _ == 0) == 0

  if (n < 2) Some(List())
  else {
    Some(
      (2 to n / 2).filter(i => {
        n % i == 0 && isPrime(i)
      }).toList
    )
  }
}
```

```scala
val fL =
  for ( i <- readInt();
        j <- div42(i);
        k <- factors(j))
    yield k
```

Option

Beispiel Option map vs flatMap 2

flatMap ist geeignet um eine „weitere Dimension / Variation“ in die Verarbeitungskette zu bringen,

Typischerweise die Möglichkeit des Fehlschlagens in jeder Verarbeitungsstufe

```
val factorListMap =
  readInt()
    .map( x => if (x != 0) Some(div42Int(x)) else None )
    .map( y => if (y != None) factors(y.get) else None )
```

~

```
val factorListFlatMap =
  readInt()
    .flatMap( div42Opt )
    .map( factors )
```

Modularer, eleganter, übersichtlicher

```
def readInt(): Option[Int] = try {
  val str = scala.io.StdIn.readLine()
  Some(str.toInt)
} catch {
  case _: java.lang.NumberFormatException => None
}
```

```
def div42Int(divisor: Int): Int =
  42 / divisor
```

```
def div42Opt(divisor: Int): Option[Int] =
  if (divisor != 0) Some(42 / divisor) else None
```

```
def factors(n: Int): List[Int] = {
  def isPrime(n: Int): Boolean =
    (2 to n / 2 + 1).count(n % _ == 0) == 0

  if (n < 2) List()
  else
    (2 to n / 2).filter(i => {
      n % i == 0 && isPrime(i)
    }).toList
}
```

Fehler-Management – Monaden

Option

For-Comprehension

```
val factorListMap_1 =
  readInt()
    .map( x => if (x != 0) Some(div42Int(x)) else None )
    .map( y => if (y != None) factors(y.get) else None )
```

Verarbeitungskette mit map

```
val factorListMap_2 =
  for ( x <- readInt();
        y = if (x != 0) Some(div42Int(x)) else None
   ) yield if (y != None) factors(y.get) else None
```

Äquivalenter for-Ausdruck

```
val factorListFlatMap_1 =
  readInt()
    .flatMap( div42Opt )
    .map( factors )
```

Verarbeitungskette mit flatMap

```
val factorListFlatMap_2 =
  for (x <- readInt();
       y <- div42Opt(x)
  ) yield factors(y)
```

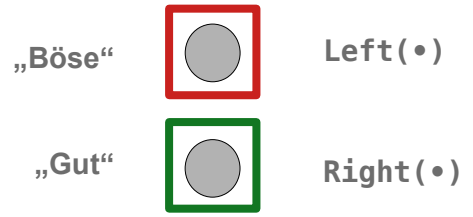
Äquivalenter for-Ausdruck

Fehler-Management – Monaden

Either

Either hat zwei Varianten: Left und Right

- **Die Left-Variante spielt die Sonderrolle „Böse“**
- **Die Right-Variante spielt die Rolle „Gut“**




```
val stringEither1 = Right("1")
val stringEither2 = Right("one")

val f: String => Either[String, Int] = str =>
  try {
    Right(str.toInt)
  } catch {
    case e: NumberFormatException => Left("Not a number")
  }

val intEither1 = stringEither1.flatMap(f) // Right(1)
val intEither2 = stringEither2.flatMap(f) // Left(Not a number)
```

Fehler-Management – Monaden

Either

Wie Option erleichtert auch Either die Verkettung von Operationen, die fehlschlagen können

Beispiel: Ausdrucksauswertung

```
import Exp._

type ErrorMsg = String

def eval(e: Exp): Either[ErrorMsg, Int] = e match {

    case Const(v: Int) => Right(v)

    case Add(l, r) =>
      for (
        v1 <- eval(l);
        v2 <- eval(r)
      ) yield v1+v2

    case Sub(l, r) =>
      for (
        v1 <- eval(l);
        v2 <- eval(r)
      ) yield v1+v2

    case Mult(l, r) =>
       for (
         v1 <- eval(l);
         v2 <- eval(r)
       ) yield v1+v2

     case Div(l, r) =>
       eval(l).flatMap( (v1:Int) =>
         eval(r).flatMap( (v2:Int) =>
           if (v2 !=0) Right(v1 / v2) else Left("Divide by zero")
         )
       )

}
```

```
enum Exp {
  case Const(v: Int)
  case Add(t1: Exp, t2: Exp)
  case Sub(t1: Exp, t2: Exp)
  case Mult(t1: Exp, t2: Exp)
  case Div(t1: Exp, t2: Exp)
}
```

Either ist nicht filterbar, darum kann bei der Division keine for-Comprehension mit if verwendet werden.

```
val exp_1: Exp = Add(Const(18), Div(Mult(Const(12), Const(4)), Const(2)))
val exp_2: Exp = Add(Const(18), Div(Mult(Const(12), Const(4)), Const(0)))

val v_1 = eval(exp_1) // Right(26)
val v_2 = eval(exp_2) // Left(Divide by zero)
```

Fehler-Management – Monaden

Try

- Ein Objekt vom Typ Try[T] ist entweder
  - ein Success(x) wobei x vom Typ T ist, oder
  - ein Failure(t) wobei t vom Typ Throwable ist
- Typische Verwendung von for-Ausdrücken mit Try:
  Kombination von Aktionen, bei denen Fehler auftreten können
- Beispiel 1:

„Böse“  Failure(•)

„Gut“  Success(•)

```
def div(x: Int, y: Int): Try[Int] =
  try {
    Success(x / y)
  } catch {
    case t : ArithmeticException => Failure(t)
  }
```

~

```
def div(x: Int, y: Int): Try[Int] =
  Try { x / y }
```

Try

Beispiel 2

```
import scala.util.Try

def even(x: Int): Boolean =
  if (x < 0) throw new IllegalArgumentException else x%2 == 0

val checkEven =
  for (v1 <- Try(scala.io.StdIn.readLine().toInt);
       v2 <- Try(even(v1))
   ) yield("" + v1 + " is even = " + v2)

println(checkEven)
```

Fehler-Management – Monaden

Try

Beispiel 3:

```scala
import scala.util.{Failure, Success, Try}


def div(x: Int, y: Int): Try[Double] =
  if (y != 0)
    Success(x.toDouble / y.toDouble)
  else {
    Failure(new Throwable("divide by zero"))
  }


val xDiVy =
  for (a <- Try(scala.io.StdIn.readLine().toInt);
       b <- Try(scala.io.StdIn.readLine().toInt);
       c <- div(a, b)
    ) yield (s" $a / $b = $c ")
```

~

```scala
val xDiVy =
  Try(scala.io.StdIn.readLine().toInt)
  .flatMap(a =>
    Try(scala.io.StdIn.readLine().toInt)
    .flatMap( b =>
      div(a, b).map(c =>
        s" $a / $b = $c "
      )
    )
  )
```

Fehler-Management – Monaden

Future

Future: Hülle für einen Wert, der

- **irgendwann einmal – vielleicht – verfügbar sein wird, oder es schon ist, oder**
- **eine Exception, die bei der asynchronen Berechnung des Wertes aufgetreten ist**

```
def futureLong(): Future[Long] = Future {
  val str = scala.io.StdIn.readLine()
  str.toLong // may fail
}

def futureFactors(l: Long): Future[List[Long]] = Future {
  factors(l)
}

val futureInputFactors = futureLong.flatMap( futureFactors )

futureInputFactors.onComplete {
  case Success(v) => println(v)
  case Failure(t) =>println(t)
}

Thread.sleep(2000) // wait for completion
```

„Böse“  Failure(•)

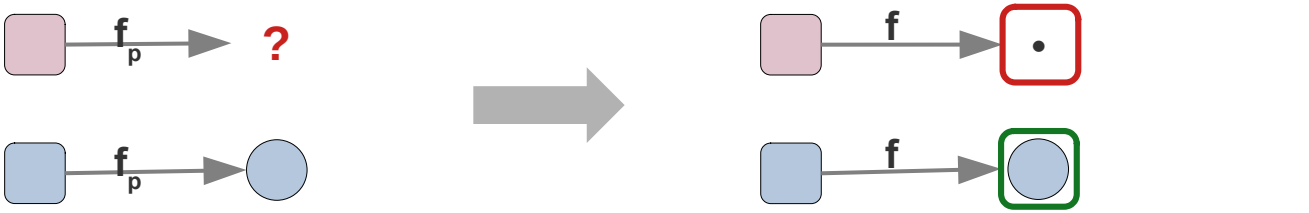
„Gut“  Success(•)

Fehlermanagement-Monade: Allgemein

Monade: Verkettungen von Operationen auf Containern

Fehlermangement-Monade: Verkettung von partiellen Funktionen auf Container-Elementen

- **Partielle Funktionen f_p mit**
  - **f_p: A => B**
  - **wobei $f_p(x)$ undefiniert ist für manche Argumente x**
- **können als totale Funktionen f modelliert werden mit**
  - **f: A => C[B] und**
  - **$f(x) = C_{Good}(f_p(x))$ „guter/gefüllter Container“ falls $f_p(x)$ definiert ist**
  - **$f(x) = C_{Bad}(\bullet)$ „böser/leerer Container“ falls $f_p(x)$ undefiniert ist**



Partielle Funktion
Wert => Wert

Totale Funktion
Wert => verpackter Wert oder „leere“ Verpackung

Fehlermanagement-Monade: Allgemein

Fehlermangement-Monade:

Monade mit zwei Varianten

- **Gut / Erfolg**
- **Böse / Leer / Misserfolg**

Funktionalität:

- **flatMap ist ohne Wirkung bei der bösen / leeren Variante**
- **error: Es gibt eine Möglichkeit die böse / leere / Misserfolgs-Variante zu erzeugen**

Fehlermanagement-Monade: Allgemein

Typklasse

```
trait Functor[F[_]] {
  extension[A, B] (x: F[A]) def map(f: A => B): F[B]
}
```

Ein Funktor

```
trait Monad[F[_]] extends Functor[F] {
  def pure[A](x: A): F[A]
  extension[A, B] (x: F[A]) {
    def flatMap(f: A => F[B]): F[B]
    def map(f: A => B) = x.flatMap(f.andThen(pure))
  }
}
object Monad {
  def apply[F[_]: Monad] = summon[Monad[F]]
}
```

Eine Monade

```
trait ErrorMonad[F[_]] extends Monad[F] {
  def error[A](): F[A]
}
object ErrorMonad {
  def apply[F[_]: ErrorMonad] = summon[ErrorMonad[F]]
}
```

Eine Monade mit Fehlermanagement

Fehlermanagement-Monade: Allgemein

Beispiel: generische Fehlerbehandlung

```
def readInt[F[_]: ErrorMonad](): F[Int] = try {
  val str = scala.io.StdIn.readLine()
  ErrorMonad[F].pure(str.toInt)
} catch {
  case _: java.lang.NumberFormatException =>
    ErrorMonad[F].error()
}

def div42[F[_]: ErrorMonad](divisor: Int): F[Int] =
  if (divisor != 0)
    ErrorMonad[F].pure(42 / divisor)
  else
    ErrorMonad[F].error()


def factors[F[_]: ErrorMonad](n: Int): F[List[Int]] = {
  def isPrime(n: Int): Boolean =
    (n == 2) || (2 to n / 2 + 1).count(n % _ == 0) == 0

  if (n < 2) ErrorMonad[F].pure(List())
  else {
    ErrorMonad[F].pure(
      (2 to n / 2).filter(i => {
        n % i == 0 && isPrime(i)
      }).toList
    )
  }
}
```

Fehlermanagement-Monade: Allgemein

Beispiel: generische Fehlerbehandlung / Instanzierung:

List als Fehlermangement-Monade

```
given ErrorMonad[List] with {
  def pure[A](x: A): List[A] = List(x)
  def error[A](): List[A] = Nil
  extension [A, B](xs: List[A]) {
    def flatMap(f: A => List[B]): List[B] =
      xs.flatMap(f)
    override def map(f: A => B) =
      xs.map(f)
  }
}
```

```
val fL =
  readInt()
    .flatMap( div42 )
    .flatMap( factors )
```

oder

```
val fL =
  for ( i <- readInt();
        j <- div42(i);
        k <- factors(j)
  ) yield k
```

```
0     => List()
Hallo => List()
1     => List(List(2,3,7))
```

Fehlermanagement-Monade: Cats

Cats

bietet eine wesentlich ausgefeiltere Variante der Fehlermanagement-Monade:

die Typklasse MonadError*

**siehe https://typelevel.org/cats/api/cats/MonadError.html

Definition Monade

Monade: Funktor mit flatMap und pure

Monaden sind ein Muster zur Gestaltung von Typ-Abstraktionen, also ein SWT-Konzept.

Sie können auf unterschiedliche Arten definiert werden und zu unterschiedliche Zwecken eingesetzt werden.

Die (gebräuchlichen Namen für die) Funktionen einer Monade sind:

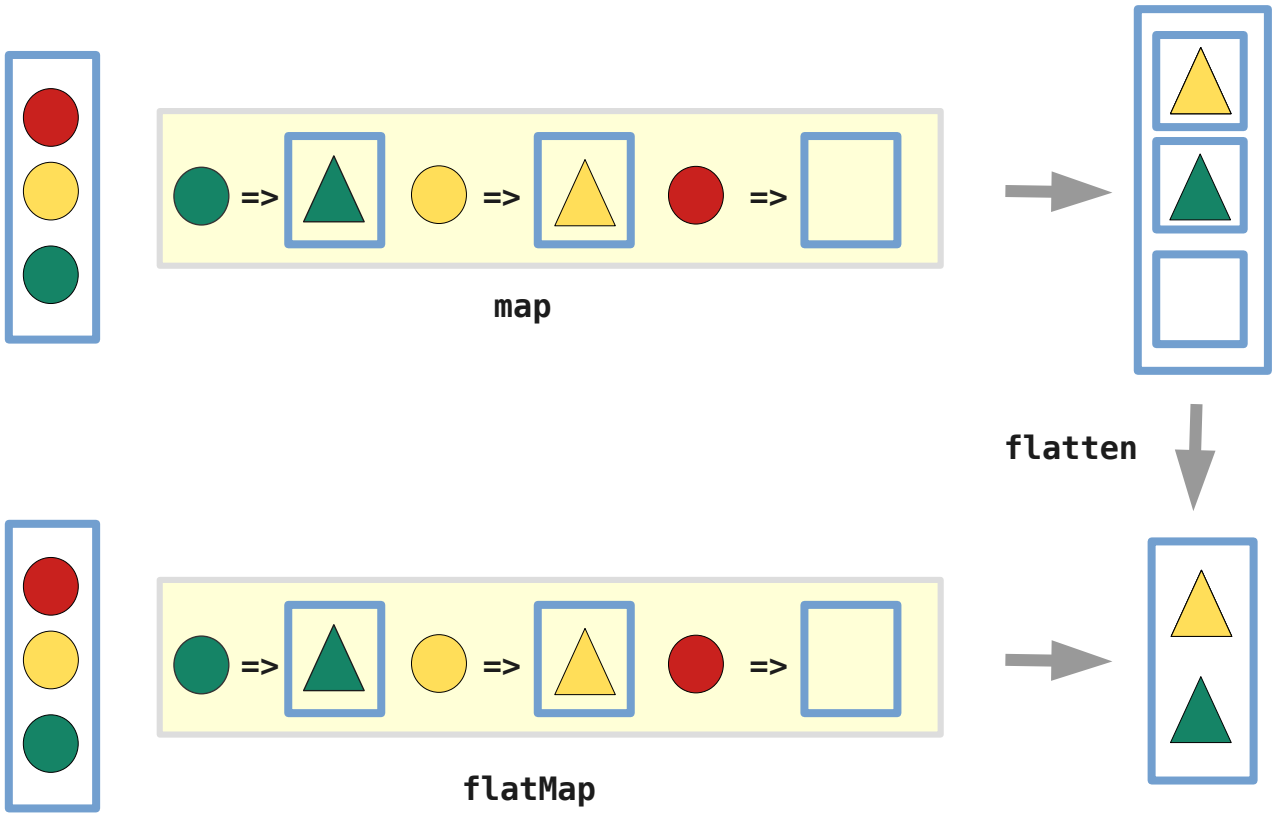
- **pure** ~ unit, return, lift, point, η (*eta*)
- **flatMap** ~ bind, >>=
- **flatten** ~ multiplication, join, μ (*my*) flatten *kann mit flatMap definiert werden*
- **map** map *kann mit pure und flatMap definiert werden*

Flatten

flatMap und flatten

flatten : Flachklopfen: Eine Ebene der Verschachtelung entfernen

faltMap ist eine Kombination aus map und flatten: fa.flatMap(f) = fa.map(f).flatten



Definition Monade

Monade als Typklasse

Monade: Funktor mit flatMap und pure

```
trait Monad[F[_]] extends Functor[F] {
  def pure[A](a: A): F[A]
  extension[A, B] (fa: F[A]) {
    def flatMap(f: A => F[B]): F[B]
    override def map(f: A => B) = fa.flatMap(f.andThen(pure))
  }
  extension[A] (ffa: F[F[A]]) {
    def flatten():F[A] = ffa.flatMap( fa => fa)
  }
}
```

map und flatten können mit flatMap und pure definiert werden

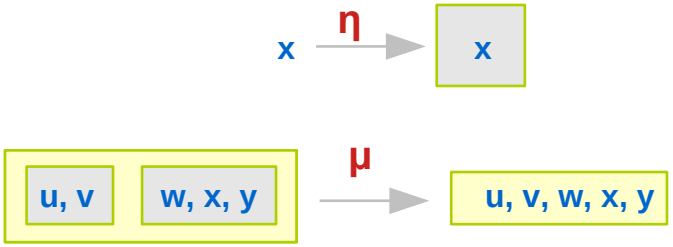
Definition Monade

Monade = Funktor + 2 nat. Transformationen

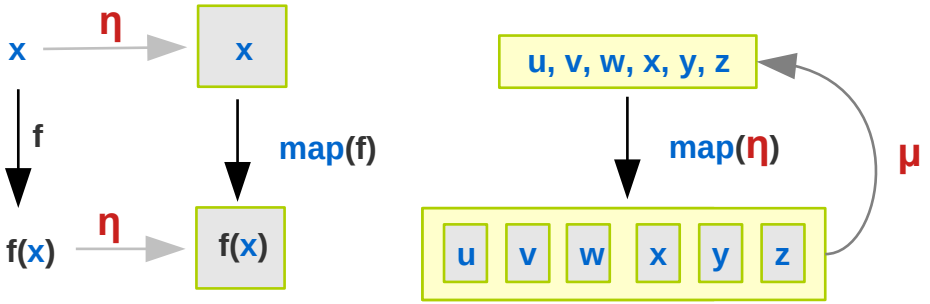
Alternative Definition einer Monade:

Eine Monade M besteht aus

- **einem Funktor M[_] und**
- **zwei natürlichen Transformationen:**
  - **η: *T* → M[*T*]** **(übliche Namen: *pure*, *unit*, *return*, *lift*)**
  - **μ: M[M[_]] → M[_]** **(übliche Namen: *flatten*, *multiplication*, *join*)**
- **wobei μ und η vernünftig und mit der map-Funktion von M verträglich sind**



Die Transformationen η und μ



Vernünftig und verträglich mit map

Definition Monade

Monade = Funktor + 2 nat. Transformationen

```
trait NT[F[_], G[_]] {
  def tau[T](v: F[T]): G[T]
}
```

Natürliche Transformation

```
trait Functor[F[_]] {
  extension[A, B] (x: F[A]) def map(f: A => B): F[B]
}
```

Funktor

```
trait Monad[F[_]] extends Functor[F] {
  type Id[T] = T
  type FF[T] = F[F[T]]
  def pure: NT[Id, F]
  def flatten:  NT[FF, F]
}
```

Eine Monade ist ein Funktor mit pure und flatten, pure und flatten sind natürliche Transformationen

```
given Monad[Option] with {

  def pure: NT[Id, Option] = new NT[Id, Option] {
    def tau[T](v: Id[T]): Option[T] = Some(v)
  }

  def flatten: NT[FF, Option] = new NT[FF, Option] {
    override def tau[T](v: FF[T]): Option[T] = v match {
      case None => None
      case Some(o) => o
    }
  }

  extension[A, B] (fa: Option[A]) def map(f: A => B): Option[B] =
    fa match {
      case None => None
      case Some(x) => Some(f(x))
    }
}
```

Option als eine (solche) Monade

Definition Monade

Monade = Funktor + 2 nat. Transformationen

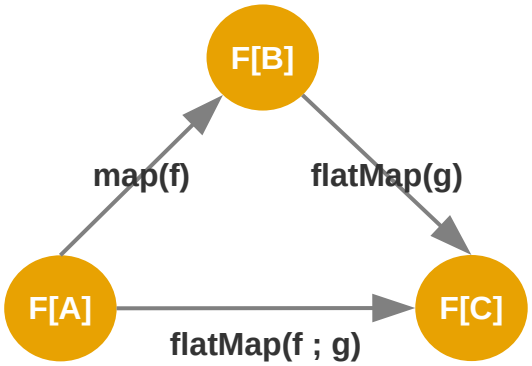
Verwendungsbeispiel:

```
def createMonad[A, F[_]: Monad](a: A): F[A] =
  implicitly[Monad[F]].pure.tau(a)

def intToString[F[_]: Monad]: Int => F[String] =
  (x: Int) => implicitly[Monad[F]].pure.tau(x.toString)


val m1 = createMonad(42)
val m2 = m1 flatMap(intToString) // Some("42")
```

Monaden-Gesetze

Gesetz 1: Links-Natürlichkeit von flatMap

```
m.flatMap(f ; g) = m.map(f).flatMap(g)
```



Beispiel:

```
val m: Option[Int] = Some(40)

val f: Int => Int         = x => x+1
val g: Int => Option[Int] = x => if (x%2 == 0) Some(x/2) else None

val v1 = m.flatMap( f andThen g )
val v1 = m.map(f).flatMap(g)
```

Die Version mit For-Comprehension statt map / flatMap zeigt die zu erwartende Äquivalenz noch deutlicher:

```
for (x <- m;
     y = f(x);
     z <- g(y))
  yield z
```

~ `m.map(f).flatMap(g)`

≡

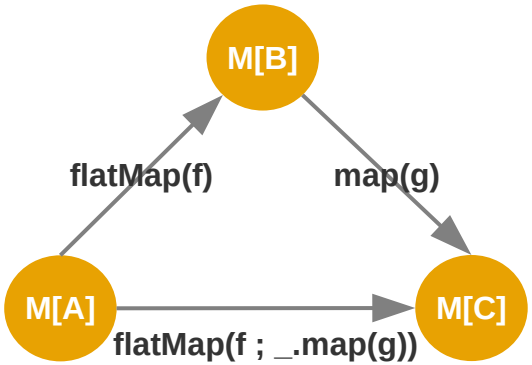
```
for (x <- m;
     y <- (f andThen g) (x) )
  yield y
```

~ `m.flatMap( f andThen g )`

Monaden-Gesetze

Gesetz 2: Rechts-Natürlichkeit von flatMap

```
m.flatMap(f).map(g) = m.flatMap(f ; m => m.map(g))
```



Beispiel:

```
val m: Option[Int] = Some(40)

val f: Int => Option[Int] = x => if (x%2 == 0) Some(x/2) else None
val g: Int => Int         = x => x+1

val v1 = m.flatMap(f).map(g)
val v2 = m.flatMap(f andThen (m => m.map(g)) )
```

Die Version mit For-Comprehension statt map / flatMap zeigt die zu erwartende Äquivalenz noch deutlicher:

```
for (x <- m;
     y <- f(x);
     z = g(y))                ~    m.flatMap(f).map(g)
  yield z
```

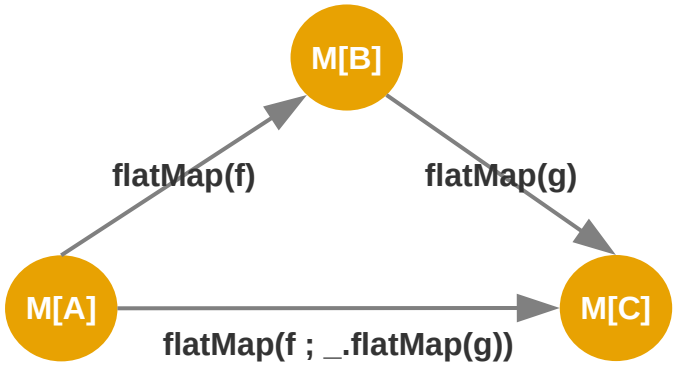
≡

```
for (x <- m1;
     y <- f(x).map(g) )       ~    m.flatMap(f andThen (m => m.map(g)) )
  yield y
```

Monaden-Gesetze

Gesetz 3: Assoziativität von flatMap

```
m.flatMap(f).flatMap(g) = m.flatMap(f ; m => m.flatMap(g))
```



Beispiel:

```
val m: Option[Int] = Some(40)

val f: Int => Option[Int] = x => if (x%2 == 0) Some(x/2) else None
val g: Int => Option[Int] = x => Some(x+1)

val v1 = m.flatMap(f).flatMap(g)
val v1 = m.flatMap(f andThen (m => m.flatMap(g)) )
```

Die Version mit For-Comprehension statt map / flatMap zeigt die zu erwartende Äquivalenz noch deutlicher:

```
for (x <- m;
     y <- f(x);              ~    m.flatMap(f).flatMap(g)
     z <- g(y))
  yield z
```

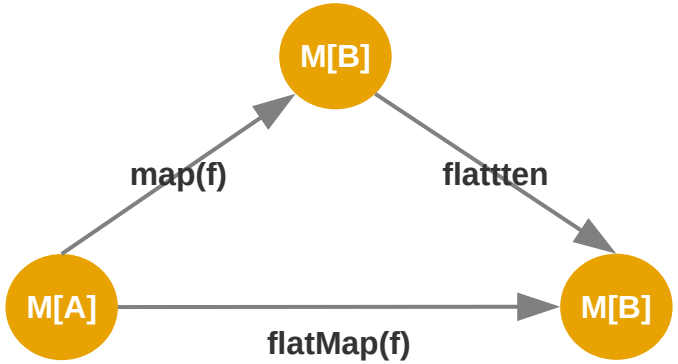
≡

```
for (x <- m;
     y <- f(x).flatMap(g) )  ~    m.flatMap(f andThen (m => m.flatMap(g)) )
  yield y
```

Monaden-Gesetze

flatMap = map + flatten

```
m.flatMap(f) = m.map(f).flatten
```



Monaden-Gesetze

Links- und Rechts-Neutralität von pure

pure ist links- und rechts-neutral für flatMap

- `pure(a).flatMap(f) = f(a)` **links-neutral**
- `m.flatMap(pure) = m` **rechts-neutral**