Software-Komponenten

Th. Letschert

THM

University of Applied Sciences

Monadentransformer

- Monaden kombinieren
- Monaden-Transformer: TryT, OptionT, ReaderT, StateT

Monaden mischen

Monaden sind nicht (einfach) mischbar

Beispiel **Monaden wie List und Option können nicht einfach (in einer for-Comprehension) „gemischt“ werden. Man bleibt besser „sortenrein“:**

```
def factors(l: Long): List[Long] = {
  def isPrime(n: Long): Boolean =
    Range.Long(2L, n / 2 + 1, 1).count(n % _ == 0) == 0

  Range.Long(2L, l / 2 + 1, 1)
    .filter((i: Long) =>
      l % i == 0 && isPrime(i)).toList
}

def StringToLong(str: String): Option[Long] =
  str.toLongOption

def stringToFactorsString(str: String): List[String] =
  for (x <- StringToLong(str);
       f <- factors(x))
    yield f.toString
```

Found: List[String]
Required: Option[Any]

Nicht OK
List und Option werden in einer for-Comprehension gemischt

```
def factors(l: Long): List[Long] = {
  def isPrime(n: Long): Boolean =
    Range.Long(2L, n / 2 + 1, 1).count(n % _ == 0) == 0

  Range.Long(2L, l / 2 + 1, 1)
    .filter((i: Long) =>
      l % i == 0 && isPrime(i)).toList
}

def StringToLong(str: String): List[Long] =
  str.toLongOption match {
    case None => Nil
    case Some(l) => l :: Nil
  }

def stringToFactorsString(str: String): List[String] =
  for (x <- StringToLong(str);
       f <- factors(x))
    yield f.toString
```

OK
Keine Monaden-Mischung

Monaden mischen

Monaden sind nicht (einfach) mischbar

Beispiel List und Option – Manchmal will / muss man aber 2 Monaden kombinieren:

```
def factors(l: Long): List[Long] = ...

def StringToLong(str: String): Option[Long] = ...


def stringToFactorsString(str: String): Option[List[String]] = ???
```

Monaden mischen

Monaden sind (nicht einfach) mischbar

Beispiel List und Option – Manchmal will / muss man aber 2 Monaden kombinieren:

```
def factors(l: Long): List[Long] = ...

def StringToLong(str: String): Option[Long] = ...

def stringToFactorsString(str: String): Option[List[String]] =
  for (x <- StringToLong(str);
       f <- factors(x).match {
         case Nil => None;
         case l => Some(l.map( _.toString))
       })
    yield f
```

Die „Monaden-Mischung“ ist möglich. Wenn auch nicht so einfach wie vielleicht gedacht.

```
stringToFactorsString("100000") // ~> Some(List(2, 5))

stringToFactorsString("hugo") // ~> None
```

Monaden kombinieren – Future / Option

Beispiel: Systematische Kombination von Future und Option

Schritt 1: Erkenne: Nur Gleiches kombiniert werden kann

```
// Future
def isPrime(n: Long): Future[Boolean] =
  Future { Range.Long(2L, n / 2 + 1, 1).count(n % _ == 0) == 0 }

// Option
def toLong(str: String): Option[Long] =
  str.toLongOption

// so geht's nicht: Future / Option gemischt
def f(str: String) =
  for (l <- toLong(str);
       b <- isPrime(l))
    yield s"$l is ${if(!b) "not" else ""} prime"
```

Found: scala.concurrent.Future[String]
Required: Option[Any]

Monaden kombinieren – Future / Option

Schritt 2: Definiere einen Typ der beide Monaden (hier Option und Future) kombiniert.

Entweder:

- **Future[Option[•]] oder**
- **Option[Future[•]]**

Mit einem (monadischen) Hüll-Typ

- **FutureOption[•] oder**
- **OptionFuture[•]**

die dann noch mit angepassten

- **map- / flatMap-Methoden (auf FutureOption / OptionFuture)**

ausgestattet werden können.

Monaden kombinieren – Future / Option

Schritt 3:
Entscheidung für eine Variante, z, B.: Future[Option[•]]

```
def isPrime(n: Long): Future[Option[Boolean]] =
  Future {
    Some(Range.Long(2L, n / 2 + 1, 1)
      .count(n % _ == 0) == 0)
  }

def toLong(str: String): Future[Option[Long]] =
  Future.successful(str.toLongOption)


def f(str: String): Future[Option[String]] = {
  for (lo <- toLong(str);
       bo <- lo match {
         case None => Future.successful(None)
         case Some(l) => isPrime(l)
       })
    yield bo match {
      case None => Some(s"$str is not a number")
      case Some(b) =>
        if (b) Some(s"$str is prime")
        else Some(s"$str is not prime")
    }
}
```

So: mit Anpassung der Basisoperationen

```
def isPrime(n: Long): Future[Boolean] =
  Future {
    Range.Long(2L, n / 2 + 1, 1)
      .count(n % _ == 0) == 0
  }

def toLong(str: String): Option[Long] =
  str.toLongOption


def f(str: String): Future[Option[String]] = {
  for (lo <- Future { toLong(str) };
       bo <- lo match {
         case None => Future.successful(None)
         case Some(l) => isPrime(l).map(Some(_))
       })
    yield bo match {
      case None => Some(s"$str is not a number")
      case Some(b) =>
        if (b) Some(s"$str is prime")
        else Some(s"$str is not prime")
    }
}
```

Oder so: mit Anpassung der Verwender

Monaden kombinieren – Future / Option

Schritt 4a:

Future[Option[•]] => FutureOption[•]

```
f("2946901").wrapped.onComplete {
  case Success(value) =>
    value match {
      case Some(str) => println(str)
      case None => println("Some Error occured")
    }
  case Failure(e) => println(e)
}
```

Anwendungsbeispiel

```
case class FutureOption[A](wrapped: Future[Option[A]]) {
  def map[B](f: A => B): FutureOption[B] =
    FutureOption(wrapped.map(_.map(f)))

  def flatMap[B](f: A => FutureOption[B]): FutureOption[B] =
    FutureOption(wrapped.flatMap {
      case None => Future.successful(None)
      case Some(x) => f(x).wrapped })
}
```

```
def isPrime(n: Long): FutureOption[Boolean] =
  FutureOption(Future {
    Some(Range.Long(2L, n / 2 + 1, 1)
      .count(n % _ == 0) == 0)
  })

def toLong(str: String): FutureOption[Long] =
  FutureOption(Future.successful(str.toLongOption))
```

```
def f(str: String): FutureOption[String] = {
  for (l <- toLong(str);
       b <- isPrime(l))
    yield
      if (b) s"$str is prime"
      else s"$str is not prime"
}
```

Mit Anpassung der Basisoperationen

Monaden kombinieren – Future / Option

Schritt 4b: mit Helfer-Funktion lift als Extension

Future[Option[•]] => FutureOption[•]

```
case class FutureOption[A](wrapped: Future[Option[A]]) {
  def map[B](f: A => B): FutureOption[B] =
    FutureOption(wrapped.map(_.map(f)))
  def flatMap[B](f: A => FutureOption[B]): FutureOption[B] =
    FutureOption(wrapped.flatMap {
      case None => Future.successful(None)
      case Some(x) => f(x).wrapped })
}
```

```
extension[A] (opt_a: Option[A]) {
  def lift: FutureOption[A] = FutureOption(
     Future.successful(opt_a))
}
```

```
extension[A] (fut_a: Future[A]) {
  def lift: FutureOption[A] =
    FutureOption(fut_a.map(Some(_)))
}
```

Lift-Operationen

```
def isPrime(n: Long): Future[Boolean] =
  Future {
    Range.Long(2L, n / 2 + 1, 1)
      .count(n % _ == 0) == 0
  }

def toLong(str: String): Option[Long] =
  str.toLongOption

def f(str: String): FutureOption[String] = {
  for (l <- toLong(str).lift;
       b <- isPrime(l).lift)
    yield
      if (b) s"$str is prime"
      else s"$str is not prime"
}
```

Ohne Anpassung der Basisoperationen
Statt dessen Lift-Operationen

Monaden kombinieren – Future / List

Beispiel List und Future kombinieren

So wäre es schön, geht aber nicht:

```
def countFactors(l: Long): Future[Long] = {
  def isPrime(n: Long): Boolean = Range.Long(2L, n / 2 + 1, 1).count(n % _ == 0) == 0

  Future {
    Range.Long(2L, l / 2 + 1, 1)
      .filter((i: Long) =>
        l % i == 0 && isPrime(i))
      .length
  }
}

// so nicht:
val res =
  for (lng <- List[Long](2946901, 29469010, 294690100);
       count <- countFactors(lng))   // Found: scala.concurrent.Future[String] Required: IterableOnce[Any]
    yield s"$lng has ${count} prime factors"
```

Monaden kombinieren – Future / List

Beispiel Future[List[•]] ~> FutureList[•]

Konzept:

```
def countFactors(l: Long): Future[Long] = ...

case class FutureList[A](wrapped: Future[List[A]]) {
  def map[B](f: A => B): FutureList[B] = ???
  def flatMap[B](f: A => FutureList[B]): FutureList[B] = ???
}

extension[A] (lst: List[A]) {
  def liftL: FutureList[A] = ???
}

extension[A] (fut: Future[A]) {
  def liftF: FutureList[A] = ???
}

val res =
  for (lng <- List(2946901, 29469010, 294690100).liftL;
       count <- countFactors(lng).liftF)
    yield s"$lng has ${count} prime factors"
```

Die Typen passen!
Können die Methoden entsprechend definiert werden?

Monaden kombinieren – Future / List

Beispiel **Future[List[•]] ~> FutureList[•]**

Die lift-Operationen und map sind offensichtlich:

```
case class FutureList[A](wrapped: Future[List[A]]) {
  def map[B](f: A => B): FutureList[B] =
     FutureList(wrapped.map(_.map(f)))
  def flatMap[B](f: A => FutureList[B]): FutureList[B] = ???
}

extension[A] (lst_a: List[A]) {
  def liftFL: FutureList[A] = FutureList(Future.successful(lst_a))
}

extension[A] (fut_a: Future[A]) {
  def liftFL: FutureList[A] = FutureList(fut_a.map(List(_)))
}
```

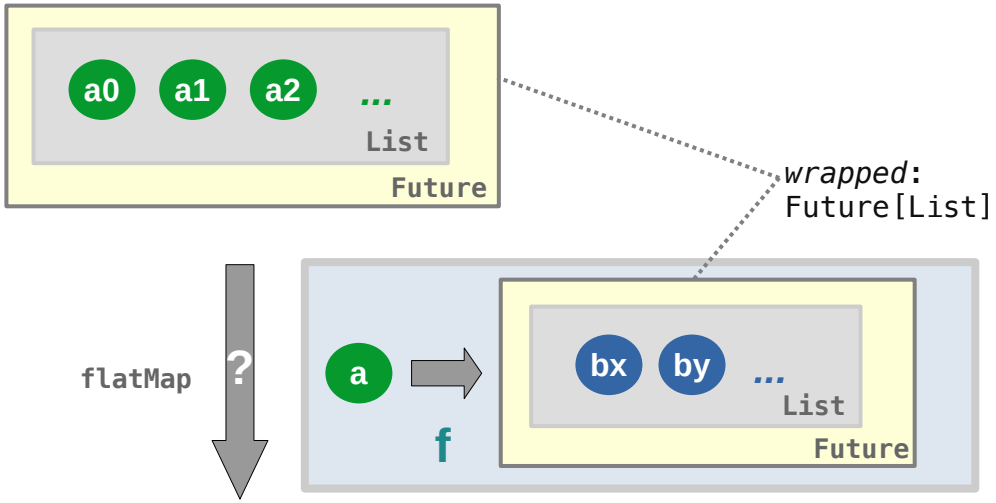
alles entsprechend zu FutureOption – ausser flatMap

Monaden kombinieren – Future / List

Beispiel Future[List[•]] ~> FutureList[•]

flatMap:

```
case class FutureList[A](wrapped: Future[List[A]]) {
  def map[B](f: A => B): FutureList[B] = FutureList(wrapped.map(_.map(f)))
  def flatMap[B](f: A => FutureList[B]): FutureList[B] = ???
}
```



Signatur von flatMap:
– das fordern die Typen --

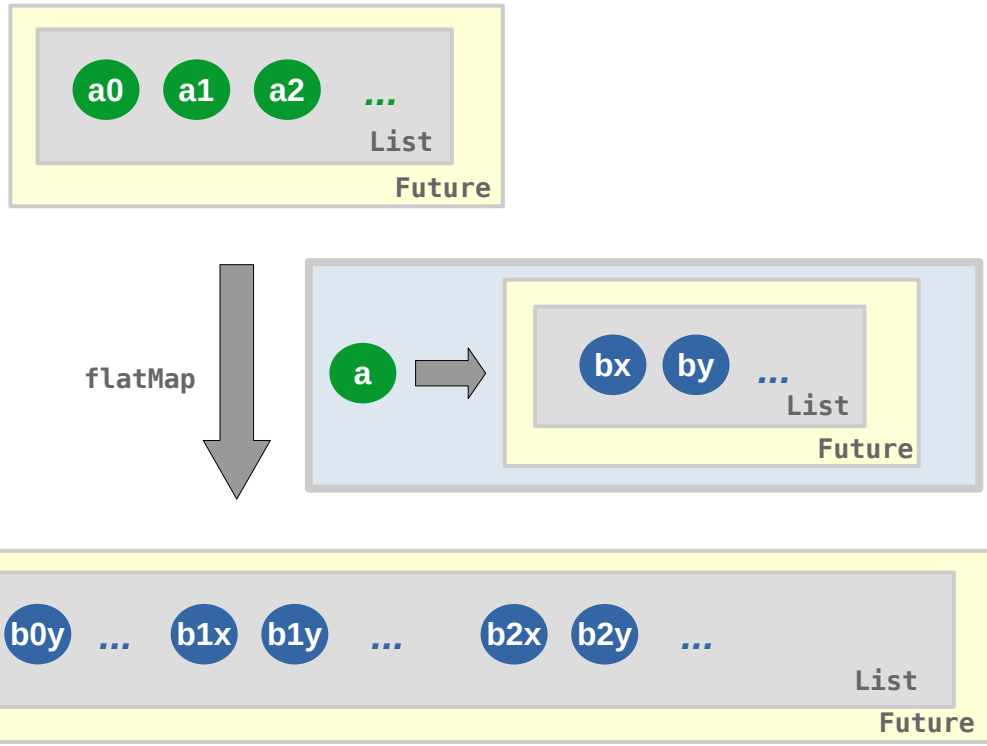
f *liefert ein* Future[List[•]]
flatMap(f) *muss auch ein* Future[List[•]] liefern
(flatMap: map + flatten)

Monaden kombinieren – Future / List

Beispiel **Future[List[•]] ~> FutureList[•]**

flatMap:

```
case class FutureList[A](wrapped: Future[List[A]]) {
  def map[B](f: A => B): FutureList[B] = FutureList(wrapped.map(_.map(f)))
  def flatMap[B](f: A => FutureList[B]): FutureList[B] = ???
}
```



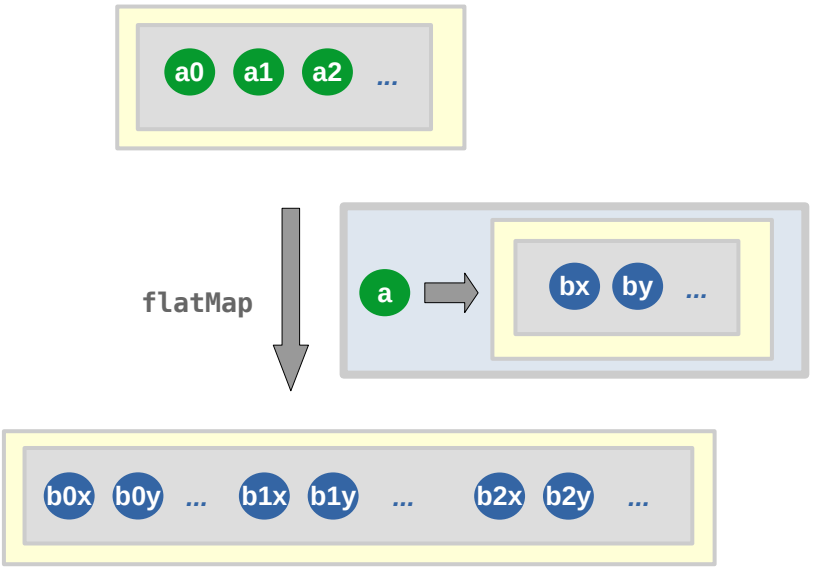
Nahe liegende Implementierung:
Die verschachtelten Werte von f werden („innen“) verkettet

Monaden kombinieren – Future / List

Beispiel **Future[List[•]] ~> FutureList[•]**

FlatMap: Die verschachtelten Werte von f werden verkettet

```
def flatMap[B](f: A => FutureList[B]): FutureList[B] = {

  def concat[A](fl1: Future[List[A]], fl2: Future[List[A]]): Future[List[A]] =
    for (x <- fl1;
         y <- fl2)
      yield x ::: y

  FutureList (
    wrapped.flatMap( (lst: List[A]) =>
      lst.foldLeft(
        Future.successful(Nil: List[B])
      )(
        (acc: Future[List[B]], a: A) =>
          concat(f(a).wrapped, acc)
      )
    )
  )
}
```



Monaden kombinieren – Future / List

Beispiel Future[List[•]] ~> FutureList[•]

Insgesamt:

```
case class FutureList[A](wrapped: Future[List[A]]) {
  def map[B](f: A => B): FutureList[B] = FutureList(wrapped.map(_.map(f)))
  def flatMap[B](f: A => FutureList[B]): FutureList[B] = {
    def concat[A](fl1:  Future[List[A]], fl2: Future[List[A]]): Future[List[A]] =
      for (x <- fl1;
           y <- fl2)
        yield x ::: y
    FutureList (
      wrapped.flatMap( (lst: List[A]) =>
        lst.foldLeft(
          Future.successful(Nil: List[B])
        )(
          (acc: Future[List[B]], a: A) =>
            concat(f(a).wrapped, acc)
        )
      )
    )
  }
}

extension[A] (lst_a: List[A]) {
  def liftFL: FutureList[A] = FutureList(Future.successful(lst_a))
}

extension[A] (fut_a: Future[A]) {
  def liftFL: FutureList[A] = FutureList(fut_a.map(List(_)))
}
```

Monaden kombinieren – Future / List

Beispiel Future[List[•]] ~> FutureList[•]

Anwendungsbeispiel:

```
val res =
  for (lng <- List(2946901L, 29469010L, 294690100L).liftL;
       count <- countFactors(lng).liftF)
    yield s"$lng has ${count} prime factors"

res.wrapped.onComplete {
    case Success(value) => println(value.mkString("\n"))
    case Failure(e) => println(e)
  }


Thread.sleep(10000)
```

```
294690100 has 3 prime factors
29469010 has 3 prime factors
2946901 has 0 prime factors
```

Monaden kombinieren – Reader / Option

Beispiel A: Termauswertung mit Option zur Fehlerbehandlung

```
given Monad[Option] {
  def pure[A](a: A): Option[A] = Some(a)

  extension[A, B](oa: Option[A]) {
    def flatMap(f: A => Option[B]): Option[B] =
      oa.flatMap(f)

    override def map(f: A => B): Option[B] =
      oa.map(f)
  }
}

// Termauswertung mit der Option-Monade
def eval(term: Term): Option[Int] = term match {
  case Literal(v)   => summon[Monad[Option]].pure(v)
  case Add(t1, t2)  => for (v1 <- eval(t1); v2 <- eval(t2)) yield v1 + v2
  case Sub(t1, t2)  => for (v1 <- eval(t1); v2 <- eval(t2)) yield v1 - v2
  case Mult(t1, t2) => for (v1 <- eval(t1); v2 <- eval(t2)) yield v1 * v2
  case Div(t1, t2)  =>
    for (v1 <- eval(t1);
         v2 <- eval(t2);
         if (v2 != 0))
      yield v1 / v2
}

val term1: Term = Add(Literal(18), Div(Mult(Literal(12), Literal(4)), Literal(2)))
val term2: Term = Add(Literal(18), Div(Mult(Literal(12), Literal(4)), Literal(0)))
val termValue1 = eval(term1) // Some(42)
val termValue2 = eval(term2) // None
```

```
enum Term {
  case Literal(v: Int)
  // kein case Const(name: String)
  case Add(t1: Term, t2: Term)
  case Sub(t1: Term, t2: Term)
  case Mult(t1: Term, t2: Term)
  case Div(t1: Term, t2: Term)
}

import Term._
```

Monaden kombinieren – Reader / Option

Beispiel B : Termauswertung mit Reader für definierte Konstanten

```scala
case class Reader[Z, A](run: Z => A) {
  def apply(z: Z): A = run(z)
}

type Env = Map[String, Int]
type EnvReader[A] = Reader[Env, A]

given Monad[EnvReader] with {
  def pure[A](a: A): EnvReader[A] = Reader(z => a)

  extension[A, B](x: EnvReader[A]) {
    def flatMap(f: A => EnvReader[B]): EnvReader[B] =
      Reader(z => f(x(z))(z))

    override def map(f: A => B): EnvReader[B] =
      Reader(x.run andThen f)
  }
}
```

```scala
def eval(term: Term): EnvReader[Int] = term match {
  case Literal(v)   =>
     Monad[EnvReader].pure(v)
  case Const(name)  =>
     Reader(env => env(name))
  case Add(t1, t2)  =>
     for (v1 <- eval(t1); v2 <- eval(t2)) yield v1 + v2
  case Sub(t1, t2)  =>
     for (v1 <- eval(t1); v2 <- eval(t2)) yield v1 - v2
  case Mult(t1, t2) =>
     for (v1 <- eval(t1); v2 <- eval(t2)) yield v1 * v2
}
```

```scala
enum Term {
  case Literal(v: Int)
  case Const(name: String)
  case Add(t1: Term, t2: Term)
  case Sub(t1: Term, t2: Term)
  case Mult(t1: Term, t2: Term)
  // kein case Div(t1: Term, t2: Term)
}
import Term._
```

```scala
val term: Term = Add(
  Mult(Literal(18), Const("two")),
  Add(Const("four"), Const("two")))

val termValueInEnv = eval(term)

val res = termValueInEnv(
           Map("two" -> 2,
               "four" -> 4)) // 42
```

Monaden kombinieren – Reader / Option

Beispiel C : Termauswertung mit Reader und Option für Terme mit Konstanten und Fehlerbehandlung

Definition einer monadischen Reader/Option-Klasse

```
case class Reader[Z, A](run: Z => A) {
  def apply(z: Z): A = run(z)
}
```

```
type Env = Map[String, Int]
type EnvReaderOpt[A] = Reader[Env, Option[A]]
```

Option in Reader nicht Reader in Option! – So sieht es irgendwie „natürlicher“ aus, als umgekehrt.

```
given Monad[EnvReaderOpt] with {
  def pure[A](a: A): EnvReaderOpt[A] = Reader(z => Some(a))

  extension[A, B](x: EnvReaderOpt[A]) {
    def flatMap(f: A => EnvReaderOpt[B]): EnvReaderOpt[B] =
      Reader(z => x.run(z) match {
        case Some(a) => f(a).run(z)
        case None => None
      })

    override def map(f: A => B): EnvReaderOpt[B] =
      Reader(z => x.run(z).map(f))
   }
}
```

None muss abgefangen werden. In flatMap mit expliziter Fallunterscheidung, in map reicht dazu map.

Monaden kombinieren – Reader / Option

Beispiel C : Termauswertung mit Reader und Option für definierte Konstanten und Fehlerbehandlung

Geben wir ihr auch noch einen Filter:

```
given MWithFilter[EnvReaderOpt] with {
  def pure[A](a: A): EnvReaderOpt[A] = Reader(z => Some(a))

  extension[A, B](x: EnvReaderOpt[A]) {
    def flatMap(f: A => EnvReaderOpt[B]): EnvReaderOpt[B] =
      Reader(z => x.run(z) match {
        case Some(a) => f(a).run(z)
        case None => None
      })
    override def map(f: A => B): EnvReaderOpt[B] =
      Reader(z => x.run(z).map(f))

    def withFilter(p: A => Boolean): EnvReaderOpt[A] =
      Reader( z => x.run(z).filter(p) )
  }
}
```

Typklassen-Instanz:
EnvReaderOpt ist eine Monade
mit Filter

```
trait MWithFilter[F[_]] extends Monad[F] {
  extension[A, B] (x: F[A]) {
    def withFilter(f: A => Boolean): F[A]
  }
}

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

Typklasse: Monade mit Filter

Monaden kombinieren – Reader / Option

Beispiel C : Termauswertung mit Reader und Option für definierte Konstanten und Fehlerbehandlung

```
def eval(term: Term): EnvReaderOpt[Int] =
  term match {
    case Literal(v)   =>
      MWithFilter[EnvReaderOpt].pure(v)
    case Const(n)     =>
      Reader(env => Some(env(n)))
    case Add(t1, t2)  =>
      for (v1 <- eval(t1);
           v2 <- eval(t2))
        yield v1 + v2
    case Sub(t1, t2)  =>
      for (v1 <- eval(t1);
           v2 <- eval(t2))
        yield v1 - v2
    case Mult(t1, t2) =>
      for (v1 <- eval(t1);
           v2 <- eval(t2))
        yield v1 * v2
    case Div(t1, t2)  =>
      for (v1 <- eval(t1);
           v2 <- eval(t2);
           if v2 != 0)
        yield v1 / v2
  }
```

```
enum Term {
  case Literal(v: Int)
  case Const(name: String)
  case Add(t1: Term, t2: Term)
  case Sub(t1: Term, t2: Term)
  case Mult(t1: Term, t2: Term)
  case Div(t1: Term, t2: Term)
}
import Term._
```

```
val term1: Term = Add(
    Mult(Literal(18), Const("two")),
    Add(Const("four"), Const("two")))

val term2 = Add(
  Mult(Literal(18), Const("two")),
  Div(Const("four"), (Sub(Const("two"), Literal(2)))))

val termValueInEnv1 = eval(term1)
val res1 = termValueInEnv1(Map("two" -> 2, "four" -> 4)) // Some(42)

val termValueInEnv2 = eval(term2)
val res2 = termValueInEnv2(Map("two" -> 2, "four" -> 4)) // None
```

Kombination von Monaden: Möglichkeit und Grenzen

Monaden kombinieren

Mit viel „mühsamer Bastelarbeit“ ist vieles möglich

Was ist grundsätzlich alles möglich?

Kombination von Monaden

Gegeben seien zwei Monaden M_1 und M_2

Können sie (immer ?) wie in den vorhergehenden Beispielen

- **Future + Option => FutureOption(Future[Option[•]])**
- **Future + List => FutureList(Future[List[•]])**
- **Reader + Option => ReaderOption(Reader[Option[•]])**

zu einer Monade vereinigt werden:

- **Ist $M_1[M_2[•]]$ eine Monade wenn M_1 und M_2 Monaden sind?**
- **Sind $M_1[M_2[•]]$ und $M_2[M_1[•]]$ äquivalent?**

Kombination von Monaden

Beispiel: ListTry vs TryList

```
def toInt(str: String): Try[Int] = Try { str.toInt }
val lst = List("1", "12", "two", "3")
```

```
case class ListTry[A](wrapped: List[Try[A]])

extension[A] (lst: List[A]) {
  def liftL: ListTry[A] = ???
}

extension[A] (tr: Try[A]) {
  def liftT: ListTry[A] = ???
}

given Monad[ListTry] with { ??? }


val res =
  for (s <- lst.liftL;
       i <- toInt(s).liftT)
    yield i
```

TryList

```
case class TryList[A](wrapped: Try[List[A]])

extension[A] (lst: List[A]) {
  def liftL: TryList[A] = ???
}

extension[A] (tr: Try[A]) {
  def liftT: TryList[A] = ???
}

given Monad[TryList] with {  ???  }


val res =
  for (s <- lst.liftL;
       i <- toInt(s).liftT)
    yield i
```

ListTry

Kombination von Monaden

Beispiel: ListTry vs TryList

```
case class ListTry[A](wrapped: List[Try[A]])

given Monad[ListTry] with {

  def pure[A](a: A): ListTry[A] = ListTry(Success(a)::Nil)

  extension[A, B](x: ListTry[A]) {
    def flatMap(f: A => ListTry[B]): ListTry[B] =
      ListTry(
        x.wrapped.flatMap {
          case Failure(t) => List(Failure(t))
          case Success(a) => f(a).wrapped }
      )

    override def map(f: A => B): ListTry[B] =
      ListTry(x.wrapped.map(_.map(f)))
  }
}

extension[A] (lst: List[A]) {
  def liftL: ListTry[A] = ListTry(lst.map(Success(_)))
}

extension[A] (tr: Try[A]) {
  def liftT: ListTry[A] = ListTry(List(tr))
}
```

flatMap und map werden an das geschachtelte Objekt delegiert.

Bei flatMap muss dazu mit einer expliziten Fallunterscheidung gearbeitet werden. Bei map ist das nicht notwendig.

Kombination von Monaden

Beispiel: ListTry vs TryList

```
def toInt(str: String): Try[Int] = Try { str.toInt }

val lst = List("1", "12", "two", "3")
```

```
val res =
  for (s <- lst.liftL;
       i <- toInt(s).liftT)
    yield i
```

liefert das erwartete Ergebnis:

```
res.wrapped.mkString("\n")
```

```
Success(1)
Success(12)
Failure(java.lang.NumberFormatException: For input string: "two")
Success(3)
```

Kombination von Monaden

Beispiel: ListTry vs TryList

```
case class TryList[A](wrapped: Try[List[A]])

extension[A] (lst: List[A]) {
  def liftL: TryList[A] = TryList(Success(lst))
}

extension[A] (tr: Try[A]) {
  def liftT: TryList[A] = TryList(tr.map(List(_)))
}

given Monad[TryList] with {

  def pure[A](a: A): TryList[A] = TryList( Success(List(a)) )

  extension[A, B](x: TryList[A]) {
    def flatMap(f: A => TryList[B]): TryList[B] = ???

    override def map(f: A => B): TryList[B] =
      TryList( x.wrapped.map( _.map(f) ) )
  }

}
```

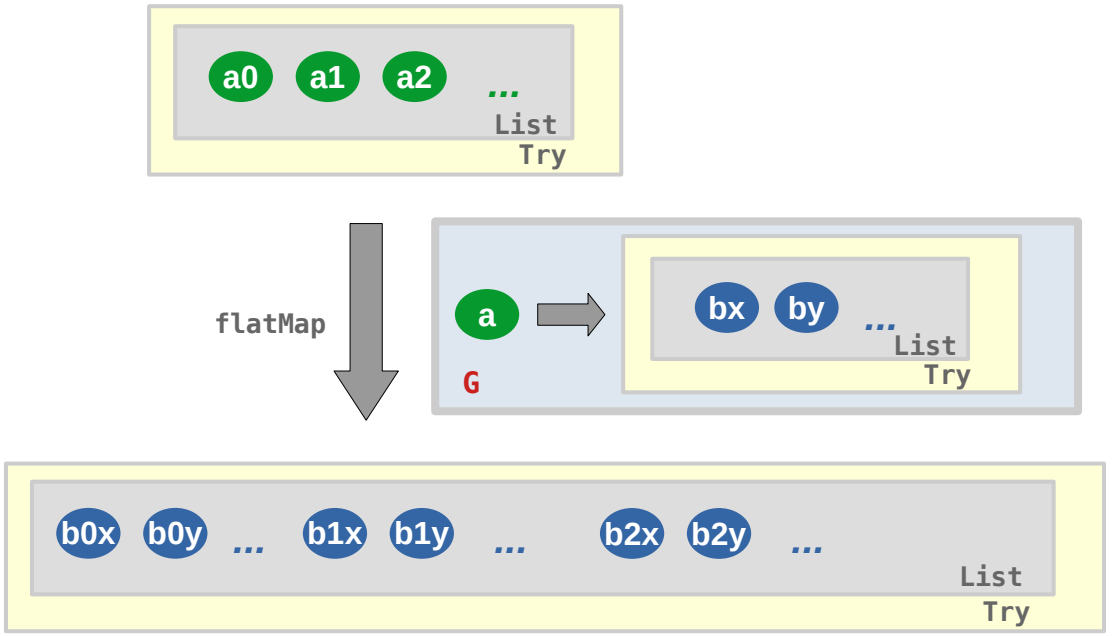
TryList: Die Definition der Methoden hat eine naheliegende Form ….

… ausser für flatMap

Kombination von Monaden

Beispiel: ListTry vs TryList / flatMap

```
def flatMap(f: A => TryList[B]): TryList[B] = {

  def G(lst: List[A]): Try[List[B]] = ???

  TryList (
    x.wrapped.flatMap(
      (lst: List[A]) => G(lst))
  )

}
```



Die Funktion f kann uns viele Try[List[B]] liefern

Naheliegende Implementierung (siehe FutureList oben):

Die verschachtelten Werte von f werden verkettet

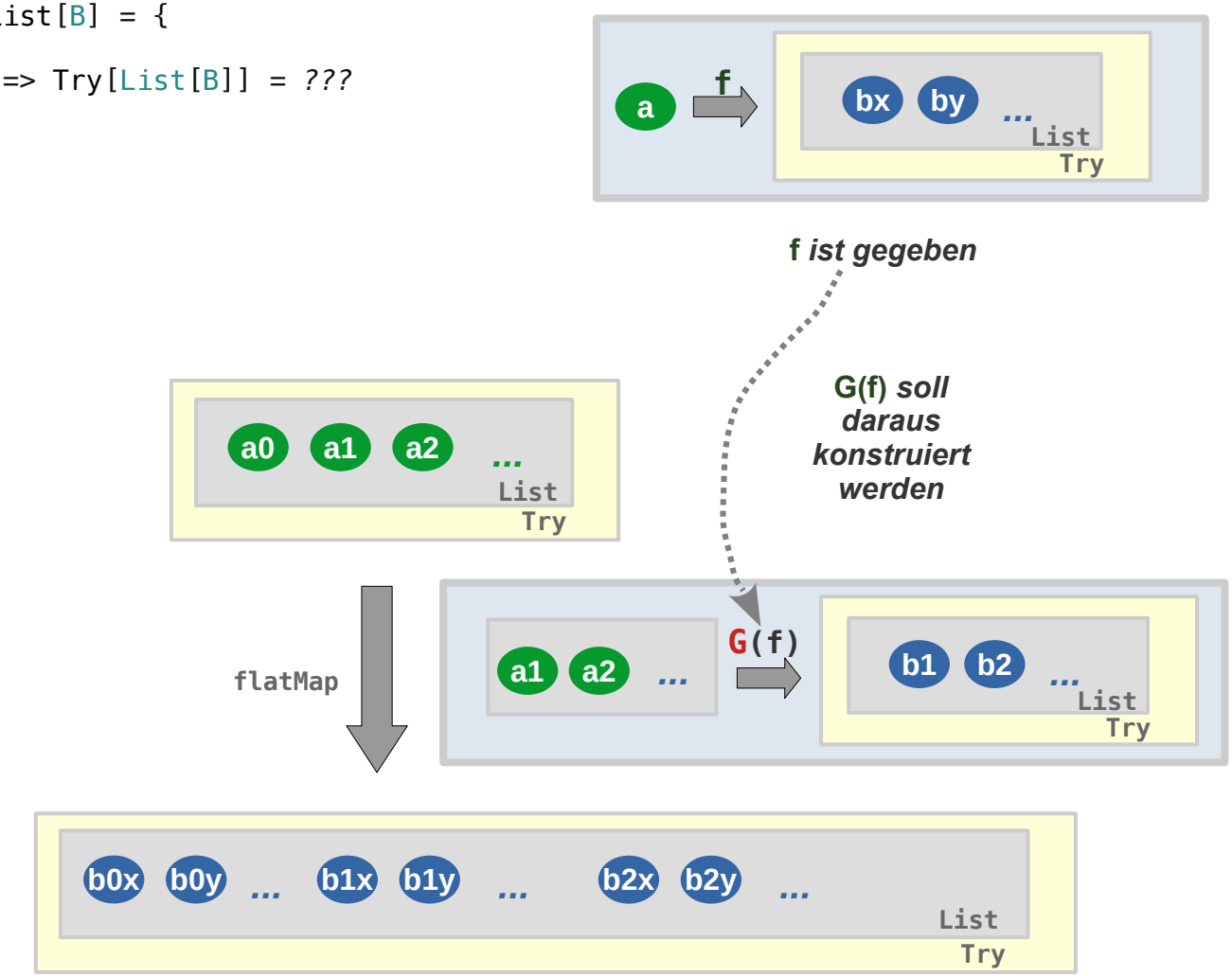
Kombination von Monaden

Beispiel: ListTry vs TryList / flatMap

```
def flatMap(f: A => TryList[B]): TryList[B] = {

  def G(f: A => TryList[B]): List[A] => Try[List[B]] = ???

  TryList (
    x.wrapped.flatMap(
      (lst: List[A]) => G(f)(lst))
  )
}
```

Nahe liegende Implementierung von G (siehe FutureList oben):

Die verschachtelten Werte von f werden verkettet



Kombination von Monaden: Möglichkeit und Grenzen

Kombination von Monaden

Beispiel: ListTry vs TryList / flatMap

Die verschachtelten Werte von f werden verkettet:

```
extension[A, B](x: TryList[A]) {

  def flatMap(f: A => TryList[B]): TryList[B] = {
    TryList (
      x.wrapped.flatMap(
        (lst: List[A]) => G(f)(lst))
    )
  }

  ...
}
```

```
def G[A, B](f: A => TryList[B]):  List[A] => Try[List[B]] = {

  def concat[A](fl1:  Try[List[A]], fl2: Try[List[A]]): Try[List[A]] =
    for (x <- fl1;
         y <- fl2)
      yield x ::: y

  lst => lst.foldLeft(
      Success(Nil: List[B])
    )((acc: Try[List[B]], a: A) =>
      concat(f(a).wrapped, acc)
    )
}
```

Zwei verknüpfen

Alle verknüpfen, zusammenfalten

Kombination von Monaden

Beispiel: ListTry vs TryList / flatMap

Liefert diese Definition das Erwartete ?

```
def toInt(str: String): Try[Int] = Try { str.toInt }
val lst = List("1", "12", "two", "3")
```

```
val resTL =
  for (s <- lst.liftL;
       i <- toInt(s).liftT)
    yield i
```

```
resTL.wrapped match {
  case Success(lst)
    => lst.mkString("\n")
  case Failure(e)
    => println(e)
}
```

java.lang.NumberFormatException: For input string: "two"

Kombination von Monaden: Möglichkeit und Grenzen

Kombination von Monaden

Beispiel: ListTry vs TryList / Beobachtung:

$M_1[M_2[\bullet]]$ und $M_2[M_1[\bullet]]$ sind nicht unbedingt äquivalent

```
def toInt(str: String): Try[Int] = Try { str.toInt }
val lst = List("1", "12", "two", "3")
```

```
val resLT =
  for (s <- lst.liftL;
       i <- toInt(s).liftL)
    yield i
```

```
resLT.wrapped.mkString("\n")
```

Success(1)
Success(12)
Failure(java.lang.NumberFormatException: For input string: "two")
Success(3)

ListTry

```
val resTL =
  for (s <- lst.liftL;
       i <- toInt(s).liftT)
    yield i
```

```
resTL.wrapped match {
  case Success(lst)
    => lst.mkString("\n")
  case Failure(e)
    => println(e)
}
```

java.lang.NumberFormatException: For input string: "two"

TryList

Kombination von Monaden

Beispiel: ListTry vs TryList

TryList ?

Try ist wie Option und Either eine Pass / Fail – Monade:

Die „Iteration“ ist eine Verkettung von Operationen, die mit mit dem ersten Fehlschlag beendet wird.

Konsequenz:

Try, Option, Either sind nur als „innere Monaden“ sinnvoll

Kombination von Monaden

Mühsam und nicht immer möglich

Mit Bastelarbeit können

- etliche, aber nicht alle Monaden
- in der einen, der anderen oder in beiden Reihenfolgen

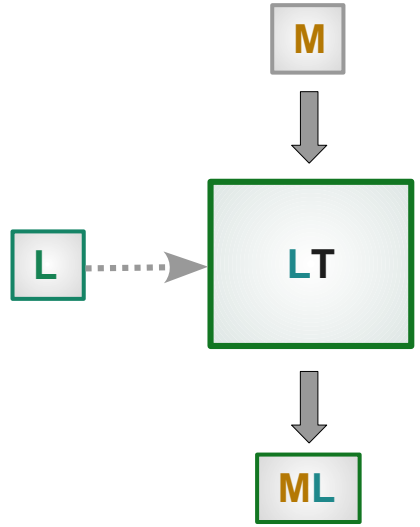
kombiniert werden.

Monadentransformer zur Vereinfachung der Kombination!

Monadentransformer

Die Idee

- Seien L und M zwei Monaden die kombiniert werden sollen
- Eine der der beiden – L – sei die Basis
- Zur Basis L wird ein Transformer LT[•] definiert
- L*T* wird auf M angewendet L*T*[M]
- das liefert die Kombination von L und M: LT[M] ~> ML[•]
- Beispiel
  - L = Try
  - M = List
  - TryT[List] ~> ListTry[•]



Monadentransformer: TryT

TryT[List]

Für TryT haben wir oben eine Vorlage

```
case class ListTry[A](wrapped: List[Try[A]])

given Monad[ListTry] with {

  def pure[A](a: A): ListTry[A] = ListTry(Success(a)::Nil)

  extension[A, B](x: ListTry[A]) {
    def flatMap(f: A => ListTry[B]): ListTry[B] =
      ListTry(
        x.wrapped.flatMap { // Bearbeite erfolgreiche Listenelemente mit f
          case Failure(t) => List(Failure(t))
          case Success(a) => f(a).wrapped }
      )

    override def map(f: A => B): ListTry[B] =
      ListTry(x.wrapped.map(_.map(f)))
  }
}
```

Sie muss nur verallgemeinert werden.

Monadentransformer: TryT

TryT[List]

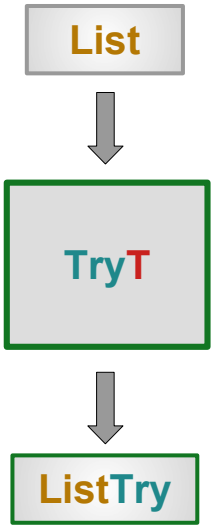
ListTry mit `List => M[_]: Monad` verallgemeinert, als Klassendefinition

```
case class TryT[M[_] : Monad, A](wrapped: M[Try[A]])  {

  def flatMap[B](f: A => TryT[M, B]): TryT[M, B] =
    TryT(
      wrapped.flatMap {
        case Failure(t) => summon[Monad[M]].pure(Failure(t))
        case Success(a) => f(a).wrapped }
    )

  def map[B](f: A => B): TryT[M, B] =
    TryT(wrapped.map(_.map(f)))
}

def pure[M[_]: Monad, A](a: A): TryT[M, A] =
  TryT(summon[Monad[M]].pure(Success(a)))
```

```
extension[M[_]: Monad, A] (ma: M[A]) {
  def liftM: TryT[M, A] = TryT(ma.map(Success(_)))
}

extension[M[_]: Monad, A] (tr: Try[A]) {
  def liftT: TryT[M, A] = TryT(summon[Monad[M]].pure(tr))
}
```

Monadentransformer: TryT

TryT[List]



Ein kleiner Test zeigt, dass der Transformer sich so wie erwartet verhält:

```
def toInt(str: String): Try[Int] = Try { str.toInt }
val lst = List("1", "12", "two", "3")

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) // flatMap der Klasse List
    override def map(f: A => B) =
      xs.map(f) // map der Klasse List, ignorieren das vordefinierte map
  }
}

val res =
  for (s <- lst.liftM;
       i <- toInt(s).liftT)
    yield i

println(res.wrapped.mkString("\n"))
```

```
Success(1)
Success(12)
Failure(java.lang.NumberFormatException: For input string: "two")
Success(3)
```

Monadentransformer: OptionT

OptionT[•]

Der Monaden-Transformer für Option kann ebenfalls als Verallgemeinerung definiert werden. Beispielsweise als Verallgemeinerung von FutureOption

```
case class OptionT[M[_]: Monad, A](wrapped: M[Option[A]]) {

  def map[B](f: A => B): OptionT[M, B] =
    OptionT(wrapped.map(_.map(f)))

  def flatMap[B](f: A => OptionT[M, B]): OptionT[M, B] =
    OptionT(wrapped.flatMap {
      case None => summon[Monad[M]].pure(None)
      case Some(x) => f(x).wrapped })
}

extension[M[_]: Monad, A] (opt_a: Option[A]) {
  def liftOT: OptionT[M, A] =
    OptionT(summon[Monad[M]].pure(opt_a))
}

extension[M[_]: Monad, A] (m_a: M[A]) {
  def liftOT: OptionT[M, A] =
    OptionT(m_a.map(Some(_)))
}
```

Monadentransformer: OptionT

OptionT[•]

Anwendungsbeispiel:

```scala
import scala.concurrent.ExecutionContext

implicit val ec: ExecutionContext =
     ExecutionContext.global

def isPrime(n: Long): Future[Boolean] =
  Future (
    Range.Long(2L, n / 2 + 1, 1)
      .count(n % _ == 0) == 0
    )

def toLong(str: String): Option[Long] =
  str.toLongOption

def f(str: String): OptionT[Future, String] = {
  for (l <- toLong(str).liftOT;
       b <- isPrime(l).liftOT)
    yield
      if (b) s"$str is prime"
      else s"$str is not prime"
}
```

```scala
given Monad[Future] with {
  def pure[A](x: A): Future[A] = Future.successful(x)

  extension[A, B](fa: Future[A]) {
    def flatMap(f: A => Future[B]): Future[B] =
      fa.flatMap(f)

    override def map(f: A => B): Future[B] = fa.map(f)
  }
}


f("2946901").wrapped.onComplete {
  case Success(value) =>
    value match {
      case Some(str) => println(str)
      case None => println("Some Error occured")
    }
  case Failure(e) => println(e)
}

Thread.sleep(3000)
```

2946901 is prime

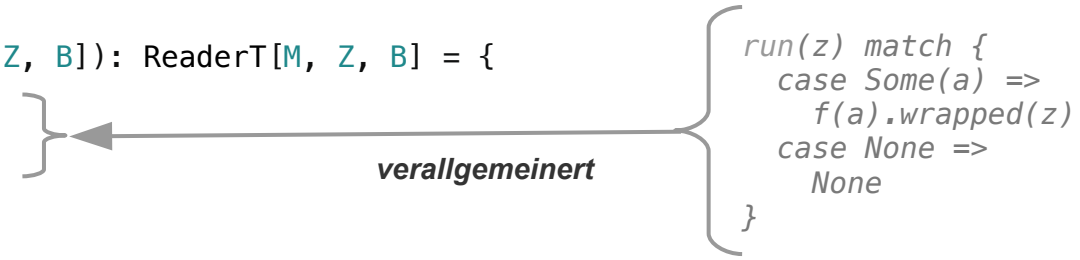
Monadentransformer: ReaderT

ReaderT[•]

Definition als Verallgemeinerung der Kombination OptionReader

```scala
trait MWithFilter[F[_]] extends Monad[F] {
  extension[A, B] (x: F[A]) {
    def withFilter(f: A => Boolean): F[A]
  }
}

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

case class ReaderT[M[_]: MWithFilter, Z, A](run: Z => M[A]) {

  def map[B](f: A => B): ReaderT[M, Z, B] =
    ReaderT(z => run(z).map(f))

  def flatMap[B](f: A => ReaderT[M, Z, B]): ReaderT[M, Z, B] = {
    ReaderT ( z => {
      val ma: M[A] = run(z)
      ma.flatMap(a => f(a).run(z))
    })
  }

  def withFilter(p: A => Boolean): ReaderT[M, Z, A] =
    ReaderT(z => run(z).withFilter(p) )
}
```

Wenn M filterbar ist, dann ist auch ReaderT[M] filterbar



Monadentransformer: ReaderT

ReaderT[•]

Beispiel Ausdrucksauswertung: Terme

```
enum Term {
  case Literal(v: Int)
  case Const(name: String)
  case Add(t1: Term, t2: Term)
  case Sub(t1: Term, t2: Term)
  case Mult(t1: Term, t2: Term)
  case Div(t1: Term, t2: Term)
}
import Term._
```

Monadentransformer: ReaderT

ReaderT[•]

Beispiel Ausdrucksauswertung: generische Auswertungsfunktion

```
type Env = PartialFunction[String, Int]

def eval[M[_]: MWithFilterZero](term: Term): ReaderT[M, Env, Int] = term match {
  case Literal(v)   => ReaderT(env =>
    MWithFilter[M].pure(v))

  case Const(n) => ReaderT(
    (env: Env) =>
      if (env.isDefinedAt(n) ) MWithFilterZero[M].pure(env(n))
      else MWithFilter[M].zero
  )
  case Add(t1, t2)  =>
    for (v1 <- eval(t1);
         v2 <- eval(t2))
      yield v1 + v2
  case Sub(t1, t2)  =>
    for (v1 <- eval(t1);
         v2 <- eval(t2))
      yield v1 - v2
  case Mult(t1, t2) =>
    for (v1 <- eval(t1);
         v2 <- eval(t2))
      yield v1 * v2
  case Div(t1, t2)  =>
    for (v1 <- eval(t1);
         v2 <- eval(t2);
         if v2 != 0 )
      yield v1 / v2
}
```

Env ist irgendeine partielle Funktion (Namen können ja undefiniert sein)

Weitere Fehlerquelle:

Wenn der Name n undefiniert ist, dann sollte die „leere“ Monade geliefert werden.

M sollte darum ein zero haben.

Monadentransformer: ReaderT

ReaderT[•]

Beispiel Ausdrucksauswertung: Monade mit filter und zero

```
trait MWithFilterZero[F[_]] extends Monad[F] {
  def zero[A]: F[A]
  extension[A, B] (x: F[A]) {
    def withFilter(f: A => Boolean): F[A]
  }
}

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

case class ReaderT[M[_]: MWithFilterZero, Z, A](run: Z => M[A]) {

  def map[B](f: A => B): ReaderT[M, Z, B] =
    ReaderT(z => run(z).map(f))

  def flatMap[B](f: A => ReaderT[M, Z, B]): ReaderT[M, Z, B] = {
    ReaderT ( z => {
      val ma: M[A] = run(z)
      ma.flatMap(a => f(a).run(z))
    })
  }

  def withFilter(p: A => Boolean): ReaderT[M, Z, A] =
    ReaderT(z => run(z).withFilter(p) )
}
```

ReaderT[•]

Beispiel Ausdrucksauswertung: Test

```
given MWithFilterZero[Option] with {

  def pure[A](a: A): Option[A] = Some(a)

  def zero[A]: Option[A] = None

  extension[A, B](o: Option[A]) {

    def flatMap(f: A => Option[B]): Option[B] =
      o.flatMap(f)

    override def map(f: A => B): Option[B] =
        o.map(f)

    def withFilter(p: A => Boolean): Option[A] =
      o.filter(p)
  }
}

val term: Term = Add(Literal(18), Div(Mult(Literal(12), Literal(4)), Const("two")))

val termValue1 = eval(term).run(Map("two" -> 2)) // Some(42)
val termValue2 = eval(term).run(Map("two" -> 0)) // None  (Division durch 0)

val termValue3 = eval(term).run(Map("zwei" -> 2)) // None (Undefinierter Name)
```

Monadentransformer: StateT

State-Monade

`State[S, A] ≈ S => (A, S)`

State = Reader + Writer

Berechnungen in einem veränderlichen Kontext

Beispiel: Auswertung von Termen mit inc-Operator

siehe Foliensatz 8: Auswertung ohne Fehlerbehandlung mit State-Monade

```
enum Term {
  case Literal(v: Int)
  case Variable(name: String)
  case Inc(name: String)
  case Add(t1: Term, t2: Term)
  case Sub(t1: Term, t2: Term)
  case Mult(t1: Term, t2: Term)
  case Div(t1: Term, t2: Term)
}

def eval(exp: Term): State[Int, Env] = exp match {
  case Literal(v)   => State.pure(v)
  case Variable(n)  => State(env => (env(n), env))
  case Inc(name)    => State(env => {
    val v: Int = env.getOrElse(name, 0)
    (v, env + (name -> (v+1)))
  })
  case Add(t1, t2)  =>
    for (v1 <- eval(t1);
         v2 <- eval(t2))
      yield v1 + v2

  case Sub(t1, t2)  => ...

  ...
}
```

```
type Env = Map[String, Int]

case class State[A, S](ma: S => (A, S)) {
  def apply(s: S) = ma(s)
  def map[B](f: A => B): State[B, S] =
    State(s => {
      val (a, newState) = ma(s)
      (f(a), newState)
    })
  def flatMap[B](f: A => State[B, S]): State[B, S] =
    State(s => {
      val (a, s1) = ma(s)
      f(a)(s1)
    })
}

object State {
  def pure[A, S](a: A): State[A, S] = State(s => (a, s))
}
```

Keine Fehlerbehandlung:
Absturz bei Division durch 0!

State-Monade

Beispiel: Auswertung von Termen mit inc-Operator

```
val term =
  Div(
    Add(
      Add(
        Mult( Add(Inc("x"), Literal(10)),
              Add(Inc("x"), Inc("x"))),
      Mult(
        Add(Inc("y"), Inc("x")),
        Inc("y"))
      ),
      Add(
        Inc("x"),
        Inc("x"))
    ),
    Variable("z")
  )
```

```
val result = eval(term)(Map("z" -> 1))._1 // 42
```

```
val result = eval(term)(Map("z" -> 0))._1 // java.lang.ArithmeticException: / by zero
```

Fehlerbehandlung, z.B. mit Option, kann das Problem lösen. Dazu muss State mit Option kombiniert werden.

Monadentransformer: StateT

StateT

Kombination: State + Option (siehe Foliensatz 8)

```
case class StateOpt[S, A](run: S => Option[(A, S)]) {
  def map[B](f: A => B): StateOpt[S, B] =
    StateOpt(s1 => run(s1).map {
      case (a, s2) => (f(a), s2)
  })
  def flatMap[B](f: A => StateOpt[S, B]): StateOpt[S, B] =
    StateOpt(s1 => run(s1).flatMap {
      case (a, s2) => f(a).ma(s2)
    })
  }

object StateOpt {
  def pure[A, S](a: A): StateOpt[S, A] =
     StateOpt(s => Some(a, s))
}
```

Monadentransformer: StateT

StateT

Verallgemeinerung von Option zu M: MWithFilterZero

in StateOption

```scala
trait MWithFilterZero[F[_]] extends Monad[F] {
  def zero[A]: F[A]
  extension[A, B] (x: F[A]) {
    def withFilter(f: A => Boolean): F[A]
  }
}

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

```scala
case class StateT[M[_]: MWithFilterZero, S, A](run: S => M[(A, S)]) {

  def map[B](f: A => B): StateT[M, S, B] =
    StateT(s1 => run(s1).map {
      case (a, s2) => (f(a), s2)
    })

  def flatMap[B](f: A => StateT[M, S, B]): StateT[M, S, B] =
    StateT(s1 => run(s1).flatMap {
      case (a, s2) => f(a).run(s2)
    })

  def withFilter(p: A => Boolean): StateT[M, S, A] =
    StateT(z => run(z).withFilter(x => p(x._1) ) )
}
```

Monadentransformer: StateT

StateT

Generische Auswertungsfunktion

```
enum Term {
  case Literal(v: Int)
  case Variable(name: String)
  case Inc(name: String)
  case Add(t1: Term, t2: Term)
  case Sub(t1: Term, t2: Term)
  case Mult(t1: Term, t2: Term)
  case Div(t1: Term, t2: Term)
}
import Term._
```

```
type Env = Map[String, Int]

def eval[M[_]:  MWithFilterZero](exp: Term): StateT[M, Env, Int] = exp match {
  case Literal(v)   => StateT(env => summon[Monad[M]].pure(v, env))
  case Variable(name)  => StateT(
      env =>
        if (env.isDefinedAt(name) ) {
          MWithFilterZero[M].pure(env(name), env)
        } else {
          MWithFilterZero[M].zero
        }
    )
  case Inc(name)  => StateT(
      (env: Env) =>
        if (env.isDefinedAt(name) ) {
          val v = env(name)
          MWithFilterZero[M].pure(
            env(name),
            env + (name -> (v+1)))
        } else {
          MWithFilterZero[M].zero
        }
    )
```

```
  case Add(t1, t2)  =>
    for (v1 <- eval(t1);
         v2 <- eval(t2))
      yield v1 + v2
  case Sub(t1, t2)  =>
    for (v1 <- eval(t1);
         v2 <- eval(t2))
      yield v1 - v2
  case Mult(t1, t2) =>
    for (v1 <- eval(t1);
         v2 <- eval(t2))
      yield v1 * v2

  case Div(t1, t2)  =>
    for (v1 <- eval(t1);
         v2 <- eval(t2);
         if v2 != 0 )
      yield v1 / v2
```

```
}
```

StateT

Generische Auswertungsfunktion: Anwendungsbeispiel

```
val term =
  Div(
    Add(
      Add(
        Mult( Add(Inc("x"), Literal(10)),
          Add(Inc("x"), Inc("x"))),
        Mult(
          Add(Inc("y"), Inc("x")),
          Inc("y"))
      ),
      Add(
        Inc("x"),
        Inc("x"))
    ),
    Variable("z")
  )
```

```
given MWithFilterZero[Option] with {

  def pure[A](a: A): Option[A] = Some(a)

  def zero[A]: Option[A] = None

  extension[A, B](o: Option[A]) {

    def flatMap(f: A => Option[B]): Option[B] =
      o.flatMap(f)

    override def map(f: A => B): Option[B] =
      o.map(f)

    def withFilter(p: A => Boolean): Option[A] =
      o.filter(p)
  }
}

val resultInState = eval(term)

val res = resultInState.run( Map("x" -> 1, "y" -> 0, "z" -> 1) )
```