Softwarekomponenten

– Softwaretechnik in funktionaler Sicht –

mit Beispielen in Scala 3

Thomas Letschert

Version 0.6
vom 31. März 2021

Inhaltsverzeichnis

Vorbemerkung

Softwarekomponenten sind Bestandteile eines Softwaresystems. Es gibt verschiedene Arten von Komponenten. *Plugins* beispielsweise sind Komponenten, die man zur Laufzeit austauschen kann. Wir betrachten hier *Entwicklungskomponenten*. Das sind Komponenten auf der Ebene des Quellcodes, die vor der Laufzeit, während der Entwicklung und vor dem *Deployment* gegen äquivalente Komponenten ausgetauscht werden können. Solche Entwicklungskomponenten werden auch Module genannt. Module sollen also leicht austauschbar sein. Noch besser sind vielseitig verwendbare, also *generische* Module, die sich ohne Änderung an verschiedene Anwendungssituationen anpassen können.

Es geht also um die Modularisierung von Software: das Kernthema der Softwaretechnik. Hier von einem funktionalen Blickwinkel aus betrachtet.

Das Konzept der objektorientierten Programmierung stammt aus den sechziger Jahren. Sie führte ein unscheinbares Dasein in der kleinen ökologischen Nische der Simulation. Ihr Siegeszug begann erst, als sie im größeren Umfang *gebraucht* wurde. Das war in den frühen 1980-ern, als Fortschritte in der Hardware-Entwicklung graphische Benutzeroberflächen möglich und bezahlbar machten. Die Entwicklung der Software graphischer Benutzeroberflächen wurde erst mit Objektorientierung beherrschbar. Diese "Killeranwendung" erweckte die Objektorientierung zum Leben. *Massentauglich* und weit verbreitet wurde sie, nachdem sie von anderen als von nur sehr obskuren Programmiersprachen unterstützt wurde und in den Softwareentwicklungsprozess als *Objektorientierter Entwurf* eingebracht und schließlich auch vom Status des allein selig machenden Dogmas befreit werden konnte.

Bei der funktionalen Programmierung ist Ähnliches zu beobachten. Das Konzept ist alt. Seine Existenz in einer elitären Nische ebenso. Erst die mit der modernen Prozessortechnik mögliche Allgegenwart der Nebenläufigkeit hat sie zu einer Notwendigkeit gemacht. Software für hochgradig nebenläufige und verteilte Anwendungen ist ohne den funktionalen Ansatz kaum beherrschbar. Das funktionale Konzept wird im größeren Umfang *gebraucht*.

Der Bedarf ist da. Die ersten weniger obskuren Programmiersprachen für den funktionalen Ansatz entstehen gerade oder werden entsprechend erweitert und ein funktionales Äquivalent zum objektorientierten Entwurfs wird erst die Massentauglichkeit der funktionalen Programmierung bringen. Dieses Äquivalent, also eine *funktionale Entwurfsmethodik* ist im Entstehen. Die softwaretechnischen Beherrschung des funktionalen Ansatzes, der funktionale Entwurf, sind Thema hier. Selbstverständlich kann hier nicht alles Wichtige besprochen werden. Solange es noch kein Standardwerk zum Thema *funktionaler Entwurf* gibt, bleibt auch noch offen, was allgemein akzeptierte Kernthemen sind. Mit der Auswahl hier sind wir aber sicher nicht zu weit davon entfernt.

Wir betrachten zwei Themenbereiche: In Teil eins betrachten Daten und Datenabstraktionen. Teil zwei ist den Monaden gewidmet, die wir als Entwurfsmuster betrachten. Das Thema im Hintergrund ist der algebraische Entwurf. Damit ist gemeint, dass die Entwicklung von Softwarekomponenten auf einem funktionalen Blick auf die Software geleitet sein sollte: Programme sind Ausdrücke eine anwendungsspezifischen Sprache als einer internen, eingebetteten DSL. Im Entwurf geht es darum, eine solche Sprache als "Algebra" zu konzipieren. Die Implementierung stellt dann eventuell variierende Interpretationen der Ausdrucksmittel zur Verfügung.

Als Programmiersprache wird Scala-3 verwendet. Eine moderne Feature-reiche Programmiersprache mit dem undogmatischen Blick auf den funktionalen Ansatz, der für einen gleitenden Übergang in funktionale Welt sehr hilfreich ist.

Die *Cats*-Bibliothek[1] ist in Software gegossene funktionale Entwurfs-Methodologie und spielt darum eine wichtige Rolle. Nicht direkt, sondern als eine wichtige themensetzende Instanz. Die hier präsentierten vereinfachten Lösungen sollen den Zugang zu einer Bibliothek wie Cats erleichtern. Der Text hier ist auch stark von [3] inspiriert. Die inzwischen klassische aber immer noch lesenswerte (Ok, eher, studierenswerte) Lektüre zum Thema ist [1]. Wer einen Einstieg in die Kategorientheorie sucht, ist mit [2] recht gut bedient.

[1] `https://typelevel.org/cats/`

Teil I

Daten- und Typ-Abstraktionen

Kapitel 1

Daten und Datenabstraktionen

1.1 Datenabstraktion: Struktur oder Schnittstelle

Typen sind Kategorien von Dingen. Bei der Zuordnung kann man einer eher äußeren und einer eher inneren Sicht folgen. Entweder wird der Typ eines Wertes von seinem *Verhalten* oder seiner *Struktur* bestimmt. Objektorientierte Programmiersprachen haben die äußere Verhaltenssicht populär gemacht: Was man ist, ergibt aus dem, wie man sich benimmt. In funktionalen Programmiersprachen ist dagegen eher die innere Struktursicht populär: Was man ist, ergibt daraus, wie man aus was zusammen gesetzt ist.

Nehmen wir den unvermeidlichen Stack. In einer objektorientierten Sicht wird das Wesen, der Typ, eines Stacks durch ihre Schnittstelle, ihr *Interface* dargestellt, in Scala als *Trait* (englisch für "Wesenszug"):

```
// Datenabstraktion OO-Stack
trait Stack[A] {
  def push(a: A): Unit
  def pop(): A
}

// eine Implementierung der Abstraktion
class ListStack[A] extends Stack[A] {
  import scala.collection.mutable.ListBuffer

  private val content = new ListBuffer[A]()

  override def push(a: A): Unit = content.prepend(a)
  override def pop(): A = content.remove(0)
}
```

Diese *verhaltensorientierte Sicht* auf Daten kann natürlich auch bei einer funktionalen Variante des Stacks angewendet werden:

```
// Datenabstraktion funktionaler Stack (unpaktikabl)
trait Stack[A] {
  def push(a: A): Stack[A]
  def pop(): (A, Stack[A]) // liefert Wert und kleineren Stack
}

// eine Implementierung der Abstraktion
// andere sind möglich, aber unterschiedliche Implementierungen
```

```
// können nicht kombiniert werden.
class ListStack[A](
  private val content: List[A] = List() ) extends Stack[A] {

  override def push(a: A): Stack[A] = ListStack(a :: content)

  override def pop(): (A, Stack[A]) = (content.head,
     ListStack(content.tail))
}
```

Die Angelegenheit wird dadurch allerdings unhandlicher und weniger praktikabel. Veränderliches gibt es in der funktionalen Welt nicht. `pop` liefert jetzt den obersten Wert im Stack und einen um diesen Wert gekürzten Stack. Das ist eine unvermeidliche Konsequenz der funktionalen Sicht der Dinge und hat nichts mit der Datenabstraktion zu tun.

Die verhaltensorientierte Abstraktion passt hier also nicht so glatt und natürlich wie bei der OO-Sicht der Dinge. Der Stack in der OO-Variante wird stets nur von außen und in Isolation angefasst. Man steckt etwas in einen Stack hinein und holt es dann später aus *dem selben* Stack wieder heraus. *push* und *pop* sind Operationen an der Schnittstelle eines Stacks. (Siehe Abbildung 1.1.)

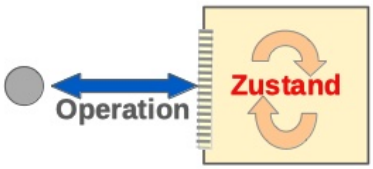

Abbildung 1.1: OO Datenabstraktion: Interaktion mit der Schnittstelle eines Objekts.

In der funktionalen Sicht sind Stacks selbst Werte, die von den Funktionen *push* und *pop* konsumiert und produziert werden. Eine Funktion muss den Stack verarbeiten, den eine andere Funktion produziert hat. Das Wissen über den inneren Aufbau eines Stapels muss jeder Funktion bekannt sein und es muss fix und für alle gleich sein. Es kann nicht sein, dass *push* eine Repräsentation erzeugt, die *pop* unbekannt ist.

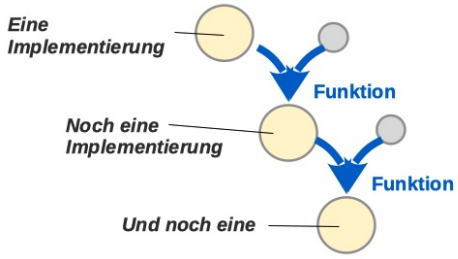

Abbildung 1.2: Funktionale Datenabstraktion im Schnittstellen-Stil: Angemessen?

Das eingängige und nützliche Schnittstellenkonzept kann also nicht einfach auf die funktionale Programmierung übertragen werden, mit der Konsequenz, dass dort sehr oft einfach keinerlei Datenabstraktion betrieben wird. Ein Stack wird nicht über sein Verhalten, sondern über seine Struktur definiert, denn die gemeinsame Struktur aller Stacks ist das, was die Arbeit der Funktionen bestimmt.

Das wird etwas deutlicher, wenn es Funktionen gibt, die mehr als ein Exemplar mit eventuell unterschiedlicher Implementierung verarbeiten. Nehmen wir komplexe Zahlen als Datenabstraktion:

```
trait Complex {
  def add (other: Complex): Complex
}

case class CartComplex(val x: Double, y: Double) extends Complex {
  def add (other: Complex): Complex =
  if (other.isInstanceOf[CartComplex])
       CartComplex(
       x +
         other.asInstanceOf[CartComplex].x,
       y +
         other.asInstanceOf[CartComplex].y)
  else ??? // hmm, was jetzt ???
}

case class PolarComplex(val r: Double, phi: Double) extends Complex {
  def add (other: Complex): Complex = ???
}
```

Es gebt jetzt keinen eindeutig identifizierten Platz mehr, an dem das Wissen über die Implementierung konzentriert werden könnte. Jede Funktion muss jede mögliche Implementierung kennen. Das ist nicht praktikabel. Im funktionalen Umfeld ist es darum naheliegend ganz auf eine verhaltensorientierte Abstraktion zu verzichten und die Daten-Typen mit Daten-Strukturen zu identifizieren. Die fehlende Abstraktion ist akzeptabel – solange man sich im Bereich der Hobby-Programmierung bewegt.

1.2 Datenabstraktion mit existenziellen Typen

Das Konzept der Datenabstraktion kann also nicht einfach und unbesehen von der OO-Welt in die funktionale Welt übertragen werden und diesem Grund wird es in der Regel dort auch nicht eingesetzt. Das heißt aber nicht, dass es gar nicht möglich ist eine funktionale Datenabstraktion zu betreiben. Sie muss nur an den funktionalen Kontext angepasst werden. Daten und Operationen auf den Daten müssen dazu auf eine andere Art zusammengefasst werden, als in der objektorientierten Programmierung. Dort wird *eine veränderliche Datenstruktur* mit den Operationen gekapselt. In der funktionalen Programmierung kann man statt dessen alle Exemplare des verhaltensorientierten Datentyps zusammen mit den darauf anwendbaren Funktionen kapseln.

Nehmen wir wieder unseren Stack in verhaltensorienterter funktionaler Abstraktion. Zur Vereinfachung ersetzen wir den generischen Parameter `A` durch den festen Typ `Int`:

```
trait Stack {
  def push(a: Int): Stack
  def pop(): (Int, Stack)
}
```

Dieser Stack soll unterschiedliche Implementierungen haben können, die den Nutzer nichts angehen und die er nicht kennen muss oder kennen darf. Die Implementierung ist geheim, aber natürlich eindeutig und fest. Ein Nutzer kann nicht mit unterschiedlichen Implementierungen

gleichzeitig umgehen. Das kann man erreichen, indem `Stack` in einen Trait gepackt wird, den die Nutzer erweitern:

```
trait Stacks {

  trait Stack {
    def push(a: Int): Stack
    def pop: (Int, Stack)
  }

  def emptyStack: Stack

}

trait StackUser extends Stacks {

  def useStack(): Unit = {
    val s0: Stack = emptyStack
    val s1 = s0.push(1)
    val s2 = s1.push(2)
    val (v1, s3) = s2.pop
    val (v2, s4) = s3.pop

    // keine Chance zum Zugriff
    //auf die Implementierung des Stacks

    println(v1)
    println(v2)
  }

}
```

Jetzt fehlt natürlich noch die Offenheit in der Implementierung. Also die Möglichkeit beispielsweise `ListStacks` als Erweiterung von `Stacks` definieren zu können. Die gewünschte Offenheit in der Implementierung kann uns ein generischer Trait StackF liefern:

```
trait Stacks {

  // zur Vermeidung von Namenskollisionen umbenannt
  trait StackF[StackRep] {
    def push(a: Int): StackF[StackRep]
    def pop: (Int, StackF[StackRep])
  }

  // Existenzieller Typ SomeStackRep
  type Stack = StackF[SomeStackRep] forSome { type SomeStackRep }

  def emptyStack: Stack

}

class ListStacks extends Stacks {

  private class ListStack(private val rep: List[Int]) extends
      StackF[List[Int]] {
    def emptyStack: StackF[List[Int]] =
```

```
      new ListStack(List())

    def push(x: Int): StackF[List[Int]] =
      new ListStack(x :: this.rep)

    def pop: (Int, StackF[List[Int]]) =
      (rep.head, new ListStack(this.rep.tail))
  }

  override def emptyStack: Stack = new ListStack(List())

}
```

`SomeStackRep` ist ein existenziell quantifizierter Typ: Irgendein beliebiger aber fester Typ. Existenziell quantifizierte Typen bringen genau die gewünschte Datenabstraktion zum Ausdruck: Etwas von dem die Nutzer nichts wissen wissen, außer dem, dass es es sie gibt. Dummerweise gibt es dieses Feature in Scala 3 nicht mehr. Der Compiler schlägt vor:

> *Existential types are no longer supported -*
> *use a wildcard or dependent type instead*

Existenzieller Typ als abstrakter Typ

Bevor wir uns darauf einlassen, bleiben wir erst einmal bei etwas konventionelleren Mitteln, einem abstrakten Typ:

```
trait Stacks {

  trait StackF[StackRep] {
    def push(a: Int): StackF[StackRep]
    def pop: (Int, StackF[StackRep])
  }

  protected type SomeStackRep // abstrakter Typ
  type Stack = StackF[SomeStackRep]

  def emptyStack: Stack
}
```

In einer Implementierung wird der abstrakte Typ einfach durch einen konkreten ersetzt:

```
class ListStacks extends Stacks {

  override type SomeStackRep = List[Int]

  private class ListStack(private val rep: SomeStackRep) extends
      StackF[SomeStackRep] {

    def emptyStack: StackF[SomeStackRep] =
      new ListStack(List())

    def push(x: Int): StackF[SomeStackRep] =
      new ListStack(x :: this.rep)

    def pop: (Int, StackF[SomeStackRep]) =
```

```
      (rep.head, new ListStack(this.rep.tail))
  }

  override def emptyStack: Stack = new ListStack(List())
}
```

Der abstrakte Typ ist in diesem Fall ein ausreichender Ersatz für den existenziellen Typ.

Existenzieller Typ als Typ-Wildcard

Ein Wildcard steht für einen bestimmten, aber unbekannten beliebigen Typ. Er ist damit ein passender Ersatz für einen existenziellen Typ:

```
trait Stacks {

  trait StackF[StackRep] {
    def push(a: Int): StackF[StackRep]
    def pop: (Int, StackF[StackRep])
  }

  type Stack = StackF[?] // ?: wildcard type

  def emptyStack: Stack
}
```

In einer Implementierung wird der Typ-Wildcard einfach durch einen bestimmten Typ (hier List[Int]) ersetzt:

```
class ListStacks extends Stacks {

  private class ListStack(private val rep: List[Int]) extends
      StackF[List[Int]] {

    def emptyStack: StackF[List[Int]] =
      new ListStack(List())

    def push(x: Int): StackF[List[Int]] =
      new ListStack(x :: this.rep)

    def pop: (Int, StackF[List[Int]]) =
      (rep.head, new ListStack(this.rep.tail))
  }

  override def emptyStack: Stack = new ListStack(List())
}
```

Existenzieller Typ als Dependent Type

Ein *dependent type* ist ein von einem Wert abhängiger Typ. In unserem Beispiel lassen wir den Repräsentanten des Stacks, die Implementierung, den Typ bestimmen:

```
trait Stacks {

  trait StackF { // Schnittstelle
```

```
    type StackRep
    val stackRep: StackRep
    def push(a: Int): StackF
    def pop: (Int, StackF)
  }

  // dependent type: der Wert rep bestimmt den Typ Rep
  abstract class StackBase[Rep](rep: Rep) extends StackF {
    type StackRep = Rep
    val stackRep = rep
  }

  def emptyStack: StackF
}
```

Eine Listenimplementierungs des Stacks ist jetzt:

```
class ListStacks extends Stacks {

  case class Stack(rep: List[Int]) extends StackBase(rep) {

    def push(x: Int): StackF =
      new Stack(x :: rep)

    def pop: (Int, Stack) =
      (rep.head, new Stack(rep.tail))
  }

  override def emptyStack = Stack(List())
}
```

Die Nutzung bleibt unverändert:

```
trait StackUser extends Stacks {
  def useStack(): Unit = {
    val s0: StackF = emptyStack
    val s1 = s0.push(1)
    val s2 = s1.push(2)
    val (v1, s3) = s2.pop
    val (v2, s4) = s3.pop
  }
}

...

object ListStackUser extends ListStacks with StackUser
ListStackUser.useStack()
```

1.3 Algebraische und verallgemeinerte algebraische Datentypen

1.3.1 ADTs

Die unter objektorientierten Programmierern so beliebte verhaltensorientierte Datenabstraktion spielte in der funktionalen Welt lange Zeit kaum eine Rolle. Sie ist, wie wir gesehen haben, nicht

direkt aus der Objektorientierung übertragbar und tendiert, wie wir ebenfalls gerade gesehen haben, zur Unhandlichkeit. Etwas weiter unten werden wir sehen, dass es mit den *Typklassen* auch ein handlicheres Format für verhaltensorientierte funktionale Typabstraktionen gibt. Beschäftigen wir uns aber zunächst einmal mit der Struktur von Daten.

Ein *algebraischer Datentyp* (ADT, *algebraic datatype*, nicht zu verwechseln mit und eigentlich das Gegenteil von *abstract datatype*, ADT) ist das Kernkonzept einer strukturellen Sicht auf Daten in der funktionalen Programmierung. ADTs sind Datenstrukturen die ausschließlich als

- Typ-Polynome:
  - Konstante: vorgegebene Typen
  - Summe: Auswahl unter alternativen Strukturen
  - Produkt: Kombination von Strukturen

  und mit
- Rekursion: Strukturen als Lösung von Strukturgleichungen

definiert werden. Beispielsweise ist

$$L = Int + Int * L$$

eine Gleichung, mit einer Unbekannten L. Die Lösung der Gleichung beschreibt den Typ der beliebig langen Integer-Listen.

ADTs sind rein funktionale Beschreibungen möglicher Werte als baumförmige strikt hierarchische Strukturen. Besonders interessant ist das, was nicht geht: Zyklische Strukturen und gemeinsame Unterstrukturen sind nicht erlaubt.

Jede vernünftige Programmiersprache erlaubt die Definition von Typen in dieser Form – in mehr oder weniger umständlicher Art. In Scala können ADTs mit Vererbung als Alternative definiert werden, beispielsweise arithmetische Ausdrücke:

```
sealed trait Exp
case class Const(v: Int) extends Exp
case class Add(e1: Exp, e2: Exp) extends Exp
case class Mult(e1: Exp, e2: Exp) extends Exp
```

oder generisch:

```
sealed trait Exp[+A]
case class Const(v: Int) extends Exp[A]
case class Add(e1: Exp, e2: Exp) extends Exp[A]
case class Mult(e1: Exp, e2: Exp) extends Exp[A]
```

Als Enum wird das aber etwas kompakter und übersichtlicher:

```
enum Exp {
  case Const(v: Int)
  case Add(e1: Exp, e2: Exp)
  case Mult(e1: Exp, e2: Exp)
}
```

beziehungsweise:

```
enum Exp[+A] {
  case Const(v: A)
  case Add(e1: Exp[A], e2: Exp[A])
  case Mult(e1: Exp[A], e2: Exp[A])
}
```

Funktionen auf solchen Strukturen lassen sich einfach mit Pattern-Match und Rekursion definieren:

```
import Exp._

def eval(exp: Exp[Int]): Int =
  exp match {
    case Const(v) => v
    case Add(e1, e2) => eval(e1) + eval(e2)
    case Mult(e1, e2) => eval(e1) * eval(e2)
  }
```

Klassen definieren Typprodukte und gelegentlich auch Typgleichungen: Im Körper einer solchen Definition kann der zu definierte Typ ja verwendet werden.

```
class Pair(x: Int, y: String) // ein Typprodukt

class X {a: Int, x:X} // Eine Typgleichung: X ~Int * X
```

1.3.2 GADTs

Bei *verallgemeinerten algebraischen Datentypen* (GADT, *generalized Algebraic Datatype*) können die Varianten des Typs sich im Typargument unterscheiden. Ein Beispiel macht das schnell klar:

```
enum Exp[A] {
  case IntConst(i: Int) extends Exp[Int]
  case BoolConst(b: Boolean) extends Exp[Boolean]
  case Add(e1: Exp[Int], e2: Exp[Int]) extends Exp[Int]
  case Or(e1: Exp[Boolean], e2: Exp[Boolean]) extends Exp[Boolean]
}

import Exp._

def eval[A](e: Exp[A]): A = e match {
  case IntConst(i) => i
  case BoolConst(b) => b
  case Add(e1: Exp[Int], e2: Exp[Int]) => eval(e1) + eval(e2)
  case Or(e1: Exp[Boolean], e2: Exp[Boolean]) => eval(e1) || eval(e2)
}

val eI = Add(Add(IntConst(1), IntConst(1)), IntConst(1))
val eB = Or(Or(BoolConst(true), BoolConst(true)), BoolConst(false))

val vI = eval(eI) // 3
val vB = eval(eB) // true
```

1.4 Typklassen

Bei *Typklassen* geht es um die Einteilung von Typen in Klassen. Typen sind eine Klassifikation von Werten nach ihrer Struktur. Typklassen sind eine verhaltensorientierte Klassifikation von Typen nach den Fähigkeiten / Verarbeitungsmöglichkeiten, die Werte haben müssen, die zu einem Typ der Klasse gehören. Eine Typklasse verhält sich zu einem Typ darum so wie ein Interface zu einer Klasse in objektorientierten Sicht der Welt (siehe Abb. 1.3).

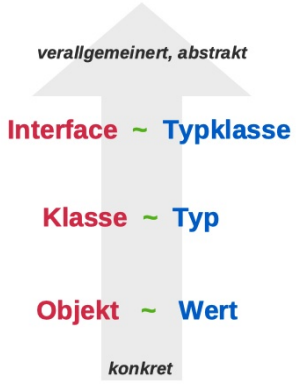

Abbildung 1.3: Objektorientierte und funktionale Daten-Abstraktion

Bei einer Typklasse wird eine *Signatur* definiert: Welche Funktionen mit welchen Parametern gibt es. Dabei kommt eine ungebundene Typvariable zum Einsatz, die alle möglichen Instanzen repräsentiert. Beispielsweise die Typklasse *Monoid*:

```
trait Monoid[M] {
  def combine(x: M, y: M): M
  def unit: M
}
```

Hier ist `M` die freie Typvariable. Bei einer generischen Komponente kann auf die Typklasse als Anforderung an den Typparameter Bezug genommen werden:

```
def sumList[M: Monoid](lst: List[M]): M =
  lst.foldLeft(summon[Monoid[M]].unit)( (acc, a) =>
    summon[Monoid[M]].combine(acc, a))
```

Mit `summon` wird auf die Eigenschaft eines Typarguments für den Parameter `M` Bezug genommen: Es muss eine Instanz von `M` sein.

Eine *Instanz* ist ein Typ der die in der Klasse geforderten Funktionen bietet. Instanzen werden wie im folgenden Beispiel definiert:

```
given Monoid[Int] with {
  def combine(x: Int, y: Int): Int = x + y
  def unit: Int = 0
}
```

Das etwas lästige `summon` im Anwendungscode kann mit einem Begleiter-Objekt eliminiert werden: Mit

```
object Monoid {
  def apply[M:Monoid] = summon[Monoid[M]]
}
```

oder

```
object Monoid {
  def apply[M](using m: Monoid[M]) = m
}
```

vereinfacht sich die Summe einer Liste von Monoid-Werten zu:

```
def sumList[M: Monoid](lst: List[M]): M =
  lst.foldLeft(Monoid[M].unit)( (acc, a) =>
    Monoid[M].combine(acc, a))
```

1.4.1 Typklassen mit Extension-Methoden definieren

Oft ist es angenehmer und übersichtlicher, wenn die Funktionsanwendung als Methodenaufruf infix verwendet werden kann. Wenn man also `x.combine(y)` oder `x combine y` schreiben kann statt `combine(x, y)`. Dazu muss die Funktion als *Extension* definiert werden:

```
trait Monoid[M] {
  def combine(x: M, y: M): M
  def unit: M
}

object Monoid {
  def apply[M:Monoid] = summon[Monoid[M]]
}

def sumList[M: Monoid](lst: List[M]): M =
  lst.foldLeft(Monoid[M].unit)( (acc, a) =>
    Monoid[M].combine(acc, a))

given Monoid[Int] with {
  def combine(x: Int, y: Int): Int = x + y
  def unit: Int = 0
}
```

oder auch mit einem Operator:

```
trait Monoid[M] {
  extension (x: M) def * (y: M): M
  def unit: M
}

object Monoid {
 def apply[M: Monoid] = summon[Monoid[M]]
}

def sumList[M: Monoid](lst: List[M]): M =
  lst.foldLeft(Monoid[M].unit)( (acc, a) =>
    acc * a)
```

```
given Monoid[List[Int]] with {
  extension (x: List[Int]) def * (y: List[Int]): List[Int] = x ::: y
  def unit: List[Int] = Nil
}
```

Der Operator ist `*`. Bei Listen als Monoid wird er als Listen-Verkettung interpretiert.

1.4.2 Typklassen für Typen höherer Art

Typkonstuktoren wie `List` haben Typen als Argument und erzeugen daraus neue Typen. `List` macht beispielsweise aus `Int` den Typ `List[Int]`. `List` darum kein einfacher Typ, sondern ein Typ höherer Art (ein *higher kinded type*).

Solche Typen höherer Art können auch in Klassen eingeteilt werden. Da Typen höherer Art oft verwendet werden, um Strukturen unabhängig von ihren Inhalt zu beschreiben, klassifizieren sie oft Strukturtypen. Ein einfaches Beispiel ist `Swapable` als Charakterisierung aller Strukturen, deren Inhalt "geswappt" werden kann:

```
trait Swapable[S[_,_]] {
  def swap[A,B](s: S[A, B]): S[B, A]
}

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

def swapAll[S[_, _]: Swapable, A, B](lst: List[S[A, B]]): List[S[B, A]] =
  lst.map(Swapable[S].swap(_))

case class Pair[A, B](a: A, b: B)

given Swapable[Pair] with { // Paare sind Swapable
  def swap[A, B](p: Pair[A,B]): Pair[B, A] = p match {
    case Pair(a, b) => Pair(b,a)
  }
}

val lst_1: List[Pair[Int, String]] = List(Pair(1, "a"), Pair(2, "b"))
val lst_2 = swapAll(lst_1) // List(Pair("a",1), Pair("b",2))
```

Für diejenigen, die eine Methodensyntax via `extension` bevorzugen, ein weiteres Beispiel mit der Typklasse `Flatable`, die Strukturen repräsentiert, die flach geklopft werden können:

```
trait Flatable[F[_]] {
  extension[A] (fa: F[A]) def flatten: List[A]
}

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

def flatList[F[_]: Flatable, A](lst: List[F[A]]): List[A] =
  lst.map( _.flatten ).flatten

case class Pair[A](x: A, y: A)
```

```
given Flatable[Pair] with {
  extension[A] (p: Pair[A]) def flatten : List[A] = p match {
    case Pair(a, b) => List(a,b)
  }
}

val lst_1: List[Pair[String]] = List(Pair("a", "b"), Pair("c", "d"))
val lst_2: List[String] = flatList(lst_1) // List("a", "b", "c", "d")
```

1.4.3 Typrelationen und Phantomtypen

Typklassen mit mehr als einem Typparameter nennt man auch *Typrelationen*. Mit ihnen kann die Beziehung zwischen zwei Typen beschrieben werden. Als Beispiel nehmen wir die Typrelation "konvertierbar". Zwei Typen sind in dieser Relation, wenn Werte des einen in Werte des anderen Typs konvertiert werden können.

Als Beispiel nehmen wir physikalische Werte. Das sind *Double*-Werte, die einer Einheit zugeordnet sind und ein Maß haben (z.B: eine Länge *(Einheit)* von 12,5 *(Wert)* Meter *(Maß)*). Das Typsystem soll dabei garantieren, dass Werte unterschiedlicher Einheiten nicht versehentlich vermischt werden, aber korrekte Konversionen von einem Maß (Meter) in ein anderes (Inch) möglich sind.

Ein Wert, der einer bestimmten Einheit zugeordnet ist, wird durch den Typ `Quantity` repräsentiert:

```
final case class Quantity[U](value: Double)
```

Der Typparameter `U` repräsentiert dabei die Einheit. `U` enthält nichts vom Typ `U`. `U` kommt ausschließlich als Typparameter vor. So etwas nennt man zu recht einen *Phantomtyp*. Der Vorteil des Phantomtyps ist, dass er ausschließlich zur Übersetzungszeit verwendet wird und im übersetzten Code vollständig verschwunden ist.

Die Konversion ist eine Typrelation (2-stellige Typklasse) zwischen Werten mit einer Einheit:

```
trait Convertible[U1, U2] {
  def convert(u1: Double): Double
}
```

Werte, die zu konvertierbaren Einheiten gehören kann man addieren:

```
def add[U1, U2](x: Quantity[U1], y: Quantity[U2])
            (using ev: Convertible[U1, U2]): Quantity[U2] =
  Quantity(ev.convert(x.value) + y.value)
```

Werte in Meilen und Kilometern kann man konvertieren:

```
given Convertible[Mile, Km] with { // mile => km
  def convert(mile: Double): Double = mile * 1.60934
}
given Convertible[Km, Mile] with { // km => mile
  def convert(km: Double): Double = km / 1.60934
}
```

Natürlich kann man auch Werte einer Einheit in sich selbst konvertieren:

```
given Convertible[Km, Km] with {
```

```
  def convert(km: Double): Double = km
}
given Convertible[Mile, Mile] with {
  def convert(mile: Double): Double = mile
}
```

Jetzt kann typsicher unter Beachtung der Einheiten gerechnet werden:

```
val v1: Quantity[Km] = Quantity[Km](2.0)
val v2: Quantity[Km] = Quantity[Km](1.0)
val v3: Quantity[Mile] = Quantity[Mile](1.0)

val v1Plus2 = add(v1, v2) // Quantity(3.0)
val v1Plus3 = add(v1, v3) // Quantity(2.2427454732996135)
val v3Plus1 = add(v3, v1) // Quantity(3.60934)
val v3Plus3 = add(v3, v3) // Quantity(2.0)
```

Mit *Extensions* wird das noch hübscher:

```
final case class Quantity[U](value: Double)

trait Convertible[U1, U2] {
  def convert(u1: Double): Double
}

def add[U1, U2](x: Quantity[U1], y: Quantity[U2])
            (using ev: Convertible[U1, U2]): Quantity[U2] =
  Quantity(ev.convert(x.value) + y.value)

trait Km
trait Mile

import scala.language.postfixOps

object UnitOps {
  // km und mile als Postfix-Operatoren,
  // add als Infix-Operator +
  extension (x: Double) def km : Quantity[Km] = Quantity[Km](x)
  extension (x: Double) def mile : Quantity[Mile] = Quantity[Mile](x)
  extension[U1, U2] (x: Quantity[U1]) def + (y: Quantity[U2])(using ev:
      Convertible[U1, U2]) = add(x, y)
}
import UnitOps._


given Convertible[Km, Km] with {
  def convert(km: Double): Double = km
}
given Convertible[Mile, Mile] with {
  def convert(mile: Double): Double = mile
}
given Convertible[Mile, Km] with { // mile => km
  def convert(mile: Double): Double = mile * 1.60934
}
given Convertible[Km, Mile] with { // km => mile
  def convert(km: Double): Double = km / 1.60934
}
```

```
val v1 = 2.0 km
val v2 = 1.0 km
val v3 = 1.0 mile

val v1Plus2 = v1 + v2 // Quantity(3.0)
val v1Plus3 = v1 + v3 // Quantity(2.2427454732996135)
val v3Plus1 = v3 + v1 // Quantity(3.60934)
val v3Plus3 = v3 + v3 // Quantity(2.0)
```

1.4.4 Typklassen-Erweiterung

Typklassen können zur neuen Typklassen erweitert werden. Die Basisklasse und die erweiterte Klasse stehen dann in einer hierarchischen Beziehung. Halbgruppen und Monoide sind das mathematisch inspirierte Standardbeispiel. Eine Halbgruppe ist eine Struktur mit einer zweistelligen Operation, ein Monoid ist eine Halbgruppe mit einem neutralen Element.

Die Beziehung zwischen Halbgruppe (Basis) und Monoid (Erweiterung) kann auf zwei Arten ausgedrückt werden:

- als OO-Vererbung
- als Erweiterung der Anforderungen

Im Beispiel, zuerst die *OO-Erweiterung* einer Typklasse:

```
trait Semigroup[A] {
  def combine(x: A, y: A): A
}

trait Monoid[A] extends Semigroup[A] {
  def empty: A
}

given Semigroup[Int] with {
  def combine(x: Int, y: Int): Int = x+y
}

given Monoid[Int] with {
  def empty: Int = 0
  def combine(x: Int, y: Int): Int = x+y // muss definiert werden!
}
```

Eine entsprechende *Anforderungserweiterung* ist:

```
trait Semigroup[A] {
  def combine(x: A, y: A): A
}

trait Monoid[A: Semigroup] {
  def empty: A
}

given Semigroup[Int] with {
  def combine(x: Int, y: Int): Int = x+y
```

```
}

given Monoid[Int] with {
  def empty: Int = 0
  //nicht notwendig: def combine(x: Int, y: Int): Int = x+y
}
```

Bei *OO-Erweiterung* müssen in einer Instanz alle Methodendefinitionen wiederholt werden. Bei der einer Erweiterung der Anforderungen ist das nicht der Fall.

Kapitel 2

Algorithmische Abstraktionen

2.1 Generische (Ent-)Faltungen

2.1.1 Katamorphismen

Ein *Katamorphismus* ist eine "zerstörerische" (*Kata*strophe) Um-*Gestalt*ung (*Morph*ismus) einer Datenstruktur. Eine Struktur wird dabei auf einen Wert reduziert. Katamorphismen sind auch als *Faltungen* oder *Reduktionen* bekannt. Eine Liste kann beispielsweise durch Addition ihrer Elemente auf ihre Summe reduziert / zusammen gefaltet werden.

```
enum IntList {
  case Empty
  case Cons(head: Int, tail: IntList)
}

import IntList.{Empty, Cons}

def sumList(lst: IntList): Int = lst match {
  case Empty => 0
  case Cons(h, t) => h + sumList(t)
}
```

Faltungen folgen stets gleichen Muster: Für jede der Variante der Datenstruktur gibt man eine Transformation an. Ist die Liste ein `Empty` dann wird sie zu einer 0. Ist sie ein `Cons` dann wird mit `+` gefaltet. Der Spezialfall der Summenbildung kann darum zu einem generischen Falten verallgemeinert werden, in dem das Muster aller Faltungen von Listen definiert wird:

```
def foldList[A](
     f_empty: A, // Wert für Empty
     f_cons: (Int, A) => A // Funktion für Cons
   ) : IntList => A = {
  case Empty => f_empty
  case Cons(h, t) => f_cons(h, foldList(f_empty, f_cons)(t))
}
```

Die Summenbildung ist dann ein Spezialfall:

```
val sumF = foldList[Int](
    0, // Wert für Empty
    {case (i, a) => i + a}) // Funktion für Cons
```

```
val lst = Cons(1, Cons(2, Cons(3, Empty)))
val sum = sumF(lst) // 6
```

Das Prinzip gilt natürlich auch für generische Datenstrukturen. Nehmen wir als Beispiel "generische" (d.h. parametrisch polymorphe) Binärbäume

```
enum BTree[+A] {
  case Leaf(v: A)
  case Node(left: BTree[A], right: BTree[A])
}
import BTree.{Leaf, Node}
```

mit ihrer "generischen" (d.h. algorithmisch verallgemeinerten) Faltung:

```
def foldTree[A, B](f_Leaf: A => B,
                   f_Node: (B, B) => B) : BTree[A] => B = {
  case Leaf(v) =>
    f_Leaf(v)
  case Node(l, r) =>
    f_Node(
      foldTree(f_Leaf, f_Node)(l),
      foldTree(f_Leaf, f_Node)(r))
}
```

Die Summe aller Blätter eines Baums ist dann wieder ein Spezialfall der allgemeinen Baumfaltung:

```
val treeSumF: BTree[Int] => Int =
  foldTree[Int, Int](
    v => v,
    _ + _
  )

val tree: BTree[Int] =
  Node(Node(Leaf(1), Leaf(2)), Node(Leaf(3), Leaf(4)))
val treeSum = treeSumF(tree) // 10
```

2.1.2 Anamorphismen

"Katamorphismus" leitet sich vom Präfix *kata-* (hinab) ab. In "Anamorphismus" steckt der Präfix *ana-*, der für "hinauf" steht. Ein Katamorphismus reduziert eine Struktur zu einem Wert. Ein *Anamorphismus* entfaltet einen Wert, bzw. baut aus einem Wert eine Struktur auf.

Der Aufbau kann in unterschiedlicher Art realisiert werden. Man gibt typischerweise ein rekursives Verfahren an. Dazu benötigt man

- einen Startwert
- eine Funktion, die
  - die nächste Stufe der Struktur (die nächste Entfaltung) und
  - den nächsten Startwert erzeugt

  sowie

- ein Kriterium, mit dem das Ende des Erzeugungsprozesses bestimmt wird.

Wenn eine Liste zu erzeugen ist, dann kann das folgendermaßen konkret umgesetzt werden:

- Nimm einen Startwert (*seed*) `s: S`,
- eine Generator-Funktion (*generator*) `g: S => Option[(T, S)]`
  die ein Paar aus nächstem Wert und neuem Startwert liefert, oder mit `None` das Ende des Erzeugungsprozesses anzeigt
- und erzeuge damit aus einem `s: S` eine Folge oder einen Strom von T-Werten.

Als Code (mit S = T = Int):

```
def unfold(s: Int)(g: Int => Option[(Int, Int)]): List[Int] =
  g(s) match {
    case None => Nil
    case Some((t, s1)) => t :: unfold(s1)(g)
  }
```

Oder generisch (parametrisch polymorph) (mit S = T):

```
def unfold[T](s: T)(g: T => Option[(T, T)]): List[T] =
  g(s) match {
    case None => Nil
    case Some((t, s1)) => t :: unfold(s1)(g)
  }
```

Damit kann eine Folge von natürlichen Zahlen erzeugt werden:

```
def natsTo(n: Int) =
  unfold(1)(x =>
    if (x<n) Some((x, x+1))
    else None)

val natsTo10 = natsTo(10) // List(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

oder auch eine Folge von Fakultäten

```
def facsTo(n: Int) = unfold[(Int,Int)]((1,0))(
  { case (r, x) =>
    if (x<=n)
      Some(((r, x), (r*(x+1), x+1) ))
    else
      None
  }
)

val facsTo10 = facsTo(10) // List((1,0), (1,1), (2,2), ... (3628800,10))
```

Weitere Varianten dieses Entfaltens sind möglich. Beispielsweise eine, bei der rekursive Aufruf durch die Definition einer Hilfsfunktion `h` vereinfacht und beschleunigt wird:

```
def unfold[A,B](p: B => Boolean, g: B => (A, B)): B => List[A] = {

  def h(b:B): List[A] =
    if (p(b)) Nil
```

```
    else {
      val (a, b1) = g(b)
      a :: h(b1)
    }

  h
}
```

Mit den Anwendungsbeispielen

```
def numbersDownFrom(n: Int) =
  unfold[Int, Int]( (x => x < 0), (x => (x, x-1)) )(n)

def numbersUpTo(n:Int) =
  unfold[Int, Int]((x => x > n), (x => (x, x+1)) )(0)

val downFrom5 = numbersDownFrom(5) // List(5, 4, 3, 2, 1, 0)
val upTo5 = numbersUpTo(5) // List(0, 1, 2, 3, 4, 5)
```

2.1.3 Bäume aufspannen

Anamorphismen können auch komplexere Strukturen erzeugen als Listen. Nehmen wir Binärbäume:

```
enum Tree[+A] {
  case Leaf(v: A)
  case Node(v: A, l: Tree[A], r: Tree[A])
}
```

Ein Binärbaum ist ein Blatt oder ein Knoten mit zwei Unterbäumen. Bei der rekursiven Erzeugung aus einem Startwert muss

- mit einem Prädikat `p` entschieden werden, ob aus dem Startwert (Saat) `s` ein Blatt oder ein Knoten zu erzeugen ist. Nach der Entscheidung kann der nächste Schritt ausgeführt werden.
- Dazu benötigen wir weiter drei weitere Funktionen:
  - `f`:eine Funktion die Blatt und Knotenwerte bestimmt
  - `g1`: eine Funktion die den Startwert für den linke Unterbaum und
  - `g2`: eine Funktion die den Startwert für den linke Unterbaum bestimmt.

Natürlich sind auch wieder andere Varianten der Baumerzeugung möglich.

In Codeform mit einer Hilfsfunktion `h` zur Vereinfachung und Beschleunigung der Rekursion:

```
import Tree.{Leaf, Node}

def unfold[A,B]( p: B => Boolean, // stop?
                 f: B => A,     // determine node value
                 g1: B => B,    // seed for left branch
                 g2: B => B     // seed for right branch
               ): B => Tree[A] = {

  def h(b:B): Tree[A] =
```

```
    if (p(b)) Leaf(f(b))
    else {
      val b1 = g1(b)
      val b2 = g2(b)
      Node(f(b), h(b1), h(b2))
    }

  h
}
```

Mit

```
def fibCallTree(n: Int) = unfold[Int, Int](
  x => x <= 1,
  x => x,
  x => x-1,
  x => x-2)(n)
```

kann beispielsweise der Aufrufbaum der Fibonacci-Funktion erzeugt werden. Etwa der von `fib(4)` (siehe Abb. 2.1):

```
val fibCallTree4 = fibCallTree(4)
```

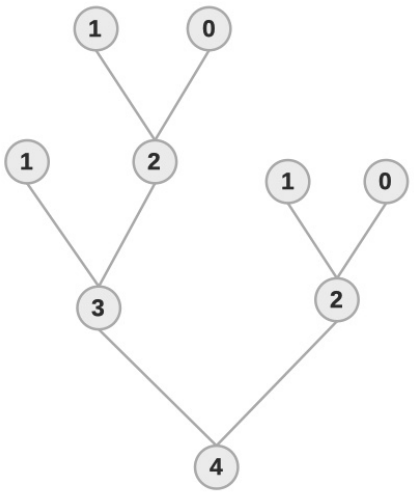

Abbildung 2.1: Aufrufbaum von fib(4).

2.2 Hylomorphismen und der MinMax-Algorithmus

2.2.1 Hylomorphismen

Ein *Hylomorphismus* ist die Kombination eines Ana- und eines Katamorphismus'. Etliche Algorithmen können als eine solche Kombinanation von Entfaltung und Faltung verstanden werden. Die Fakultätsfunktion ist beispielsweise mit

$$n! = 1 * 2 * \cdots * n$$

als Entfaltung von n zu $1, 2, 3, \cdots n$ und anschließender Faltung mit der Multiplikation definiert. Das lässt sich explizit formulieren als (siehe Abb. 2.2

```
def unfold[A,B](p: B => Boolean, g: B => (A, B)): B => List[A] = {
  def h(b:B): List[A] =
    if (p(b)) Nil
    else {
      val (a, b1) = g(b)
      a :: h(b1)
    }
  h
}

def natsTo(n: Int): List[Int] =
  unfold[Int, Int](
      x => x > n,
      x => (x, x+1)
    )(1)


val f5 = natsTo(5).foldLeft(1)( _ * _) // 120
```

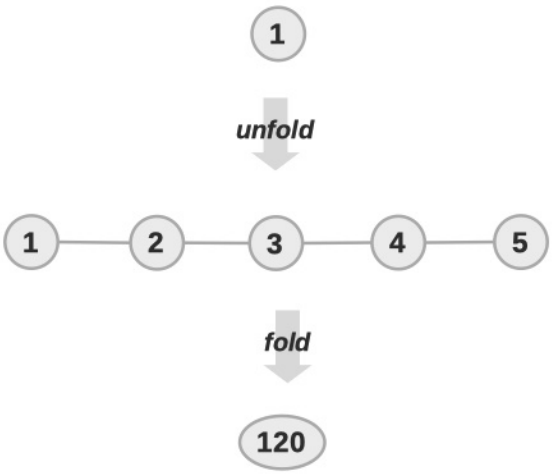

Abbildung 2.2: Fakultätsberechnung als Hylomorphismus.

Auch die Fibonacci-Funktion kann als Hylomorphismus verstanden werden: In einem ersten Schritt, dem Anamorphismus, wird der Aufrufbaum erzeugt. Im zweiten Schritt, dem Katamorphismus, wird dieser dann mit der Addition zusammengefaltet (siehe Abb. 2.3).

```
def fib(n: Int): Int = {

  val tree = unfold[Int, Int](x => x <= 1, x => x,
    x => x-1, x => x-2)(n)

  fold(
    (v: Int) => v,
    (v:Int, x:Int, y:Int) => x + y)(tree)

}
```

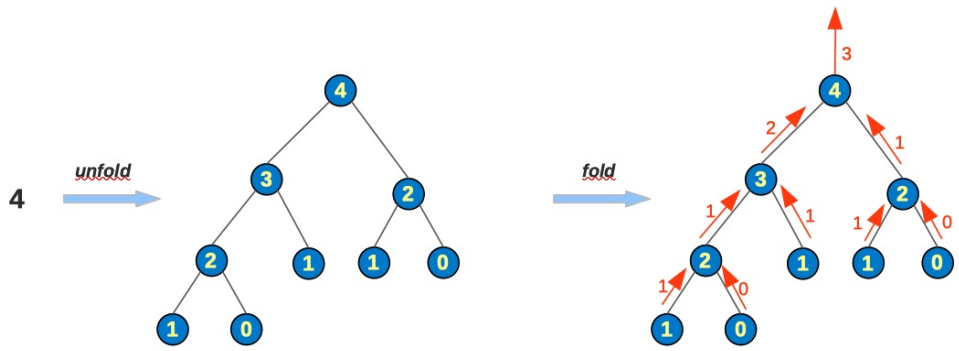

Abbildung 2.3: Fibonacciberechnung als Hylomorphismus.

```
val fib10 = fib(10) // 55
```

Die Faltungs- und Entfaltungs-Funktionen sind dabei wie oben definiert.

Die Kata-, Ana- und Hylomorphismen müssen nicht zwingend auf Binärbäume beschränkt werden. Sie können auch auf für Bäume mit einem beliebigen Verzweigungsgrad definiert werden. Nehmen wir als Baumdefinition:

```
enum Tree[+A] {
  case Empty extends Tree[Nothing]
  case Node(v: A, succs: Seq[Tree[A]])
}
```

Auf diesen Bäumen definieren wir jetzt unsere Morphismen etwas übersichtlicher als Klassen:

```
// Katamorphismus
case class TreeKata[IN, OUT](
   v_empty: OUT, // value for empty node
   f_node: (IN, Seq[OUT]) => OUT) // value for node
     extends Function[Tree[IN], OUT] {
  override def apply(tree: Tree[IN]): OUT = tree match {
    case Empty => v_empty
    case Node(v, succs) => f_node(v, succs.map( t => apply(t)) )
  }
}

// Anamorphismus
case class TreeAna[IN, OUT](
   p: IN => Boolean, // stop if true
   f: IN => OUT, // compute node value
   g: IN => Seq[IN]) // generate new seeds
     extends Function[IN, Tree[OUT]] {
  override def apply(b: IN): Tree[OUT] =
    if (p(b))
      Empty
    else {
      val seeds = g(b)
      Node (f(b), seeds.map(s => apply(s)))
    }
```

```
}

// Hylomorphismus
case class TreeHylo[IN, M, OUT](
   kata: TreeKata[M, OUT],
   ana: TreeAna[IN, M]) extends Function[IN, OUT]{
  override def apply(x: IN): OUT =
   ana.andThen(kata)(x)
}
```

Die Fibonacci-Funktion mit den Faltungsfunktionen in dieser Formulierung ist:

```
def fib(n: Int): Int = {

  val ana = TreeAna[Int, Int](
    x => x < 0, // stop if < 0
    x => if (x < 2) x else 0, // label outermost nodes with 0 or 1
    x => Seq(x-1, x-2)) // generate new seeds

  val kata = TreeKata[Int, Int](
    0,
    (v, seq) => v + seq(0) + seq(1)
  )

  TreeHylo(kata, ana)(n)
}
```

2.2.2 Der MinMax-Algorithmus als Hylomorhismus

Der bekannte *MinMax-Algorithmus*[1] kann als Hylomophismus interpretiert werden. Ausgehend von der aktuellen Konfiguration (Spielsituation, Zustand des Spielbretts) wird der Spielbaum durch einen Anamorphismus aufgebaut. Zu dessen Analyse wird dann ein Katamorphismus eingesetzt.

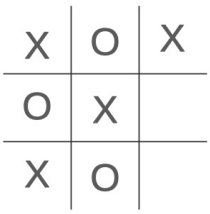

Abbildung 2.4: Tic-Tac-Toe: Weiß (Spielstein X) hat gewonnen.

Nehmen wir als einfaches Beispiel das *Tic-Tac-Toe*-Spiel bei dem eine 3×3 Spielfeld abwechselnd von einem weißen und einem schwarzen Spieler mit ihren Spielsteinen (Weiß setzt X, Schwarz setzt O) belegt werden. Gewinner ist, wer als erster eine Zeile, Spalte oder Diagonale mit seinen Steinen besetzt hat. [2]

Mit dem MinMax-Algorithmus kann der optimale nächste Spielzug berechnet werden. Ausgehend von der aktuellen Situation (Konfiguration) auf dem Spielfeld werden die Stellungen (Konfigura-

[1] https://de.wikipedia.org/wiki/Minimax-Algorithmus
[2] https://de.wikipedia.org/wiki/Tic-Tac-Toe

tionen) berechnet, die sich aus den möglichen Zügen und Folgezügen der Spieler ergeben. (Siehe Abb. 2.5)

Daraus ergibt sich eine Bewertung der nächsten Züge als Maximum (Weiß ist am Zug) der Minima (Schwarz wird den für Schwarz besten Zug wählen) der Bewertungen der Unterbäume. Endknoten (finale Stellungen) werden mit 1 / -1 / 0 bewertet, je nach dem, ob die Stellung einen Sieg für Weiß, einen Sieg für Schwarz oder ein Patt bedeutet.

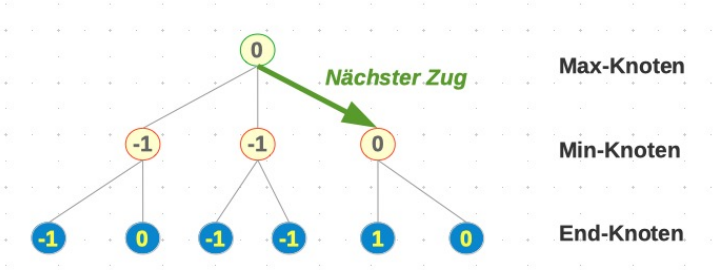

Abbildung 2.5: MinMax-Baum.

Der Algorithmus ist einfach:

$$minMax(s) = \begin{cases} score(s) & s \text{ ist Endstellung} \\ max\{minMax(s') \mid s' \text{ ist Folgekonfiguration von } s\} & s \text{ ist Max-Knoten} \\ min\{minMax(s') \mid s' \text{ ist Folgekonfiguration von } s\} & s \text{ ist Min-Knoten} \end{cases}$$

In jeder Stufe der Rekursion werden die Folgekonfigurationen berechnet und verarbeitet. Das kann man in zwei Phasen aufteilen:

- Entfalten: MinMax-Knoten mit allen Nachfolgern und deren Nachfolger.
- Falten: Berechnung der Maxima und Minima.

Dazu benötigt man eine Baumdefinition:

```
enum MinMaxNode(val config: TicTacToeConfig) {
  case MaxNode(override val config: TicTacToeConfig) extends
      MinMaxNode(config)
  case MinNode(override val config: TicTacToeConfig) extends
      MinMaxNode(config)
}

import MinMaxNode._

final case class MinMaxTree(
  node: MinMaxNode,
  successors: Seq[MinMaxTree])
```

Ein MinMax-Baum besteht aus Knoten die jeweils Min- oder Max-Knoten sind und eine Konfiguration enthalten.

Eine Konfiguration (Spielsituation, Spielzustand) bei *Tic-Tac-Toe* gibt an, wie 3×3 Felder mit einem Spielstein (*Token*) für Weiß (X), oder einem Spielstein für Schwarz (O) belegt oder frei sind.

Eine Konfiguration kann auf unterschiedliche Art als Datenstruktur repräsentiert werden. Eine naheliegende Darstellung des Spielfelds mit seiner aktuellen Belegung wäre eine 3×3 Matrix mit optionalen Tokens:

```
enum Player {
 case WhitePlayer
 case BlackPlayer
}
import Player._

enum TicTacToeToken(val owner: Player) {
 case X extends TicTacToeToken(WhitePlayer)
 case O extends TicTacToeToken(BlackPlayer)
}

type TicTacToe_Config = Array[Array[Option[TicTacToeToken]]]
```

Für die Logik eines Spiels, also die Definition der möglichen Züge oder die graphische Darstellung wäre das sicher eine recht geeignete Datenstruktur. Für einen MinMax-Baum, der eventuell tausende an Konfigurationen enthält, ist eine kompaktere Darstellung aber eher angebracht. Wir nehmen einen Vektor der Länge 9, der das 3×3 Feld zeilenweise darstellt. (Siele Abb. 2.6.) Vector ist die geeignete funktionale Datenstruktur zur Speicherung von Informationen, die sich häufig ändern.

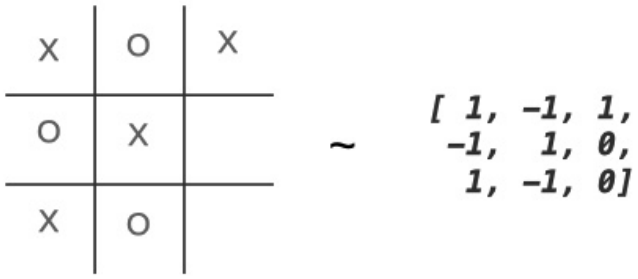

Abbildung 2.6: Konfigurationen in Vektor-Kodierung.

Die Konfigurationen in dieser Kodierung packen wir mit den benötigten Operationen in bewährter OO-Art in eine Klasse:

```
class TicTacToeConfig(val configVector: Vector[Byte]) {

 val X: Byte = 1
 val O: Byte = -1

 def score(): Int =
  isWinning() match {
   case Some(WhitePlayer) => 1
   case Some(BlackPlayer) => -1
   case None => 0
  }

 def successors(player: Player): Seq[TicTacToeConfig] = {
  val tok = if (player == WhitePlayer) X else O
  if (isFinal()) Nil
```

```
    else
      for (i <- 0 until 9
          if configVector(i) == 0
          ) yield TicTacToeConfig(configVector.updated(i, tok))
  }

  def isWinning(): Option[Player] = {
    val threeInARow: Option[(Int, Int, Int)] = Seq(
      (0, 1, 2), (3, 4, 5), (6, 7, 8),
      (0, 3, 6), (1, 4, 7), (2, 5, 8),
      (0, 4, 8), (2, 4, 6)).find {
      case (i, j, k) =>
        (configVector(j) == configVector(i)) & (configVector(k) ==
           configVector(i))
    }
    threeInARow match {
      case Some((i, _, _)) if configVector(i) == X =>
        Some(WhitePlayer)
      case Some((i, _, _)) if configVector(i) == O =>
        Some(BlackPlayer)
      case _ =>
        None
    }
  }

  def isFinal(): Boolean =
    isFull() || isWinning().isDefined

  def isFull(): Boolean = !configVector.contains(0)
}
```

Gewinnstellungen werden auf sehr einfache Art festgestellt. Wenn in einer Zeile, einer Spalte, oder einer Diagonale drei gleiche Spielsteine zu finden sind, dann hat der Spieler gewonnen, dem sie gehören.

- Die Zeilenpositionen sind $(0, 1, 2), (3, 4, 5)$ und $(6, 7, 8)$.
- Die Spaltenpositionen sind $(0, 3, 6), (1, 4, 7)$, und $(2, 5, 8)$.
- Die diagonalen Positionen sind $(0, 4, 8)$ und $(2, 4, 6)$

Der MinMax-Algorithmus ist dann schließlich

```
def bestNextConfigByMinMax(config: TicTacToeConfig, player: Player):
    TicTacToeConfig = {

  val ana: TreeAna[MinMaxNode, MinMaxNode] =
    TreeAna(
      node => node.config.isFinal(),
      node => node,
      node => node match {
        case MaxNode(c) => c.successors(WhitePlayer).map(c => MinNode(c))
        case MinNode(c) => c.successors(BlackPlayer).map(c => MaxNode(c))
      }
    )

  val kata: TreeKata[MinMaxNode, Int] =
```

```
    TreeKata[MinMaxNode, Int](0, (node, ys) =>
      node match {
        case MaxNode(c) => ys.max
        case MinNode(c) => ys.min
      }
    )

  def treeHylo: TreeHylo[MinMaxNode, MinMaxNode, Int] =
    TreeHylo(kata, ana)

  player match {
    case WhitePlayer =>
      config.successors(WhitePlayer).maxBy(c => treeHylo(MaxNode(c)))
    case BlackPlayer =>
      config.successors(BlackPlayer).minBy(c => treeHylo(MinNode(c)))
  }

}
```

TreeAna, TreeKata und TreeHylo sind die weiter oben definierten Baum-Ana-, -Kata- und -Hylo-Morphismen.

Kapitel 3

Algebraischer Entwurf, initial- und final-codierte Daten

3.1 Datenabstraktionen: Objektorientiert und Funktional

3.1.1 OO-Datenabstraktion: Interface

In der objektorientierten Programmierung lernt man recht früh die Bedeutung von *Interfaces*. Der Code einzelner Komponenten ist leichter wiederverwendbar, wenn die Verwender nur deren Interface kennen. Die Komponente sei in unserem Fall der MinMax-Algorithmus. Es ist klar, dass es ungeschickt ist, ihn von einer ganz bestimmten Codierung der Konfigurationen (Spielstellungen) abhängig zu machen. Genau das tun wir aber wenn in der Schnittstelle eine *Klasse* vorkommt:

```
class TicTacToeConfig ...

def bestNextConfigByMinMax(config: TicTacToeConfig, player: Player):
    TicTacToeConfig = ...
```

Die Lösung ist offensichtlich: Ersetze die *Klasse* durch ein *Interface* und mache die MinMax-Komponente damit unabhängig von der konkreten Implementierung Konfiguration:

```
trait MinMax {

  import Trees._
  import Player._

  // Interface:
  // Gemeinsame Schnittstelle zu Objekten unterschiedlicher Klassen
  // mit unterschiedlichen Implementierungen der vorgegebenen Methoden
  trait Config { // Trait als Interface
    def score(): Int
    def successors(player: Player): Seq[Config]
    def isWinning(): Option[Player]
    def isFinal(): Boolean =
      isFull() || isWinning().isDefined
    def isFull(): Boolean
  }

  private enum MinMaxNode(val config: Config) {
```

```
    case MaxNode(override val config: Config) extends MinMaxNode(config)
    case MinNode(override val config: Config) extends MinMaxNode(config)
  }

  import MinMaxNode._

  private final case class MinMaxTree(node: MinMaxNode, successors:
      Seq[MinMaxTree])

  def bestNextConfigByMinMax(config: Config, player: Player): Config = ...
      // unverändert

}
```

`bestNextConfigByMinMax` basierte bereits vorher nur auf den Intefrace-Methoden und war darum völlig unabhängig von deren Implementierung mit einer konkreten Datenstruktur. Der Code kann darum unverändert übernommen werden. Ein kleiner Test könnte jetzt folgendermaßen aussehen:

```
object Main {

  object MinMaxWithVectorConfig extends MinMax {

    class VectorConfig (val configVector: Vector[Byte]) extends Config {
      import Player._

      val X: Byte = 1
      val O: Byte = -1

      def score(): Int =
        isWinning() match {
          case Some(WhitePlayer) => 1
          case Some(BlackPlayer) => -1
          case None => 0
        }

      def successors(player: Player): Seq[VectorConfig] = {
        val tok = if (player == WhitePlayer) X else O
        if (isFinal()) Nil
        else
          for (i <- 0 until 9
              if configVector(i) == 0
              ) yield VectorConfig(configVector.updated(i, tok))
      }

      def isWinning(): Option[Player] = {
        val threeInARow: Option[(Int, Int, Int)] = Seq(
          (0, 1, 2), (3, 4, 5), (6, 7, 8),
          (0, 3, 6), (1, 4, 7), (2, 5, 8),
          (0, 4, 8), (2, 4, 6)).find {
          case (i, j, k) =>
            (configVector(j) == configVector(i)) & (configVector(k) ==
                configVector(i))
        }
        threeInARow match {
          case Some((i, _, _)) if configVector(i) == X =>
```

```
          Some(WhitePlayer)
        case Some((i, _, _)) if configVector(i) == 0 =>
          Some(BlackPlayer)
        case _ =>
          None
      }
    }

    def isFull(): Boolean = !configVector.contains(0)

    override def toString: String = configVector.toString
  }

}

import MinMaxWithVectorConfig.VectorConfig
import Player._

val actConfig: VectorConfig = VectorConfig( Vector(
  1, -1, 0,
  -1, 1, 0,
  1, -1, 0))

val expected: VectorConfig = VectorConfig( Vector(
  1, -1, 1,
  -1, 1, 0,
  1, -1, 0))

val nextConfig =
    MinMaxWithVectorConfig.bestNextConfigByMinMax(actConfig, WhitePlayer)

def main(args: Array[String]): Unit = {
  println(expected)
  println(nextConfig)
}

}
```

3.1.2 Funktionale Datenabstraktion: Typklasse

Typklassen sind das funktionale Äquivalent der Interfaces in der objektorientierten Programmierung. Das haben wir weiter oben schon ausführlich erläutert. Mit einer Typklasse statt einem Interface wird aus unser MinMax–Komponente:

```
import Trees.{TreeAna, TreeKata, TreeHylo}

trait MinMax {

  enum Player {
    case WhitePlayer
    case BlackPlayer
  }
  import Player._

  // Typklasse:
```

```
  // Ein Typ C gehört zur Klasse Config, wenn er folgende
  // Funktionen anbietet:
  trait Config[C] { // Trait als Typklasse
    extension (c: C) def score(): Int
    extension (c: C) def successors(player: Player): Seq[C]
    extension (c: C) def isFinal(): Boolean
  }

  private enum MinMaxNode[A](val config: A) {
    case MaxNode(override val config: A) extends MinMaxNode(config)
    case MinNode(override val config: A) extends MinMaxNode(config)
  }

  import MinMaxNode._

  def bestNextConfig[C: Config](config: C): C = {
    val ana: TreeAna[MinMaxNode[C], MinMaxNode[C]] =
      TreeAna(
        node => node.config.isFinal(),
        node => node,
        node => node match {
          case MaxNode(c) => c.successors(WhitePlayer).map(c => MinNode(c))
          case MinNode(c) => c.successors(BlackPlayer).map(c => MaxNode(c))
        }
      )
    val kata: TreeKata[MinMaxNode[C], Int] =
      TreeKata[MinMaxNode[C], Int](0, (node, ys) =>
        node match {
          case MaxNode(c) => ys.max
          case MinNode(c) => ys.min
        }
      )

    val treeHylo: TreeHylo[MinMaxNode[C], MinMaxNode[C], Int] =
        TreeHylo(kata, ana)

    config.successors(WhitePlayer).maxBy(c => treeHylo(MaxNode(c)))
  }

}
```

3.2 Algebraischer Entwurf

3.2.1 APIs: Mehr als eine Datenabstraktionen

Eine *API* ist ein *Application Programming Interface*, also die Schnittstelle einer Software-Komponente zu dem Code oder auch dem Codierer der sie benutzt. Eine API ist der Quellcode, "gegen" den eine Anwendung entwickelt und compiliert wird. Jede Implementierung der Komponente, die zu dieser API passt, kann dann als ausführbarer Code hinzugefügt werden, um so ein insgesamt lauffähiges System zu erzeugen.

Eine API ist meist mehr als nur eine Klasse oder ein einziges Interface. Wir gehen damit von der Datenabstraktion aus einen Schritt weiter zu Kollektionen von zueinander passenden Abstraktionen, die insgesamt für eine eine bestimmte (Problem- / Anwendungs-) *Domäne* einsetzbar

sind.

Betrachten wir unser Beispiel zur MinMax-Berechnung. Die funktionale Variante der MinMax-Komponente oben hat einen öffentlichen und einen privaten Teil. Der öffentliche macht die Schnittstelle aus:

```
trait MinMax {

  enum Player {
    case WhitePlayer
    case BlackPlayer
  }
  import Player._

  trait Config[C] {
    extension (c: C) def score(): Int
    extension (c: C) def successors(player: Player): Seq[C]
    extension (c: C) def isFinal(): Boolean
  }

  def bestNextConfig[C: Config](config: C): C // privates wird zur
                                              // abstrakten Methode
}
```

Software-Komponenten zur Bestimmung des optimalen nächsten Zugs in einem zwei-Personen Null-Summen Spiel sind unter algorithmischen Aspekten sicher recht recht interessant. Der Entwurf der API einer einer solchen Komponente ist ziemlich herausfordernd, wenn sie für beliebige Spiele geeignet sein soll.

Wir verlassen das Beispiel hier aber erst einmal und betrachten statt dessen die Frage, warum etwas Offensichtliches und unter Umständen Einfaches wie die Gestaltung der Schnittstelle (API) mit Funktionen und Typklassen einen derart hochtrabenden Namen hat: “Algebraischer Entwurf”.

3.2.2 Algebra und Algebraische Struktur

Ein *algebraischer Entwurf* beginnt mit dem Entwurf einer *Algebra*. Das gibt der Entwurfstechnik ihren Namen. Der Entwurf einer Algebra umfasst:

- Die Definition der für die Anwendungsdomäne relevanten Mengen von Werten als (Daten-) *Typen*.
- Die Identifikation der relevanten *Funktionen* auf den Typen mit der Definition ihrer Signatur.
- Die Definition der Regeln bzw. Gesetze die bei der Anwendung der Regeln gelten sollen.

Im MinMax-Beispiel gilt beispielsweise das “Gesetz”, dass bestNextConfig die direkte Nachfolge-Konfiguration mit der besten Bewertung liefern soll.

Eine Algebra $\mathcal{A}(A, F)$ besteht aus

- einer Menge A, die Trägermenge von $\mathcal{A}$ genannt wird, sowie
- Operationen F, die auf dieser Menge definiert sind und bei deren Anwendung

- bestimmte Gesetze G gelten.

Ein Beispiel für eine Algebra sind die natürliche Zahlen mit + und 0 und den üblichen Rechenregeln.

Eine *mehrsortige Algebra* umfasst mehrere Mengen und Funktionen zwischen diesen. In der Regel geht es beim algebraischen Entwurf um mehrere Mengen / Datentypen.

Eine *algebraische Struktur*[1] (auch *allgemeine Algebra*) ist eine Verallgemeinerung des Konzepts. Die Menge bleibt dabei unbestimmt. Eine algebraische Struktur besteht aus allen Algebren, mit irgendeiner Trägermenge A mit Operationen F für die die Gesetze G gelten. Eine algebraische Struktur wird darum

- mit einer Variablen für die Menge,
- den Signaturen der Operationen und den
- Gesetzen definiert.

Das Standardbeispiel ist das *Monoid* $\mathcal{M}(M, \{e, \circ\})$ mit einer unbestimmten Menge M und zwei Operationen: Das neutrale Element e als nullstellige und die Komposition $\circ$ als zweistellige Operation. Die Gesetze fordern die Assoziativität der Komposition sowie die Links- und Rechts-Neutralität von e.

Die Begiffsbildung ist in der Mathematik leider etwas inkonsistent[2]. Algebra und algebraische Struktur ist nicht immer klar unterschieden. Informatiker sind oft noch ungenauer. (Theoretische Informatiker natürlich nicht!) Eine Algebra ist nach Bedarf und Kontext das Abstrakte oder das Konkrete. Wie auch immer. Bleiben wir bei der Algebra als dem Konkreten und der algebraischen Struktur als dem Abstrakten, dann gelten folgenden groben(!) Äquivalenzen:

Mathematik	**objektorientiert**	**funktional**
Algebra	Klasse	(algebraischer Daten-) Typ
Algebraische Struktur	Interface	Typklasse

3.2.3 Softwareentwurf als Konzeption einer DSL

Eine *DSL (Domain Specific Language)* ist eine (Programmier-) Sprache, die für eine bestimmte Anwendungsdomäne konzipiert wurde. Man unterscheidet

- Externe und
- interne (eingebettete) DSLs.

Eine *externe DSL* ist eine eigenständige Sprache mit einem Interpreter oder eventuell sogar einem Compiler. Eine *interne DSL* ist dagegen in eine "Wirtssprache" eingebettet. Funktionen

[1] https://de.wikipedia.org/wiki/Algebraische_Struktur
[2] siehe etwa https://en.wikipedia.org/wiki/Universal_algebra

und Typen dieser Sprache erlauben es Programme der DSL auszuführen ohne eine komplette Sprachimplementierung mit Scanner, Parser, Typsystem, etc. liefern zu müssen.[3]

Wenn nun ein (funktionales) Programm, oder ein Modul[4] nichts anderes ist, als ein Ausdruck, dessen Wert in einem bestimmten Kontext zu bestimmen ist, dann ist ein Programm oder eine Programmkomponente einer bestimmten Anwendungsdomäne am besten in einer DSL dieser Domäne zu verfassen. Der erste Schritt dazu ist dann die Definition dieser DSL, als interne DSL natürlich.

Eine extrem vereinfachtes Beispiel wäre eine DSL, die aus einer einzigen Typklasse besteht. Nehmen wir die das unvermeidliche Monoid:

```
// Typklasse / algebraische Struktur: DSL
trait Monoid[M] {
  def combine(x: M, y: M): M
  def unit: M
}
object Monoid {
  def apply[M: Monoid] = summon[Monoid[M]]
}
```

Eine Komponente (ein Modul), verfasst in dieser DSL, wäre dann beispielsweise:

```
// Programmkomponente / Modul
def sumList[M: Monoid](lst: List[M]): M =
  lst.foldLeft(summon[Monoid[M]].unit)( (acc, a) =>
    summon[Monoid[M]].combine(acc, a))

def s[M: Monoid] =
  sumList(
    List[M](
      Monoid[M].unit,
      Monoid[M].unit))
```

Im Kontext einer bestimmten Algebra, also einer Instanz der Typklasse / algebraischen Struktur lässt sich dann die Komponente ausführen und ein Wert bestimmen:

```
// Kontext der Komponente / des Moduls
given Monoid[Int] with {
  def combine(x: Int, y: Int): Int = x + y
  def unit: Int = 0
}

//Bestimmung des Werts der Komponente in diesem Kontext
val v = s
```

Wir halten fest:

> (Funktionale) Softwarekomponenten bestehen aus Ausdrücken, die in einer DSL verfasst sind, deren Definition der Kern des Entwurfs der Komponente darstellen. Die Definition einer DSL ist dabei die Definition einer "Algebra": einer Kollektion von Typen, Typklassen, Werten und Funktionen.

Dieser *algebraische Entwurf*, oder besser *Algebra-getriebener Entwurfstechnik* ist ein stark spra-

[3] https://en.wikipedia.org/wiki/Domain-specific_language
[4] Ein Modul ist eine Softwarekomponente auf der Ebene des Quellcodes.

chorientierter Ansatz. Kein Wunder, dass es sich bei unserem nächsten Thema um Ausdrücke und ihre Interpreter dreht.

3.3 Syntax und Semantik – Ein initialer Ansatz

3.3.1 Das Interpreter-Muster – Objektorientiert und Funktional

Interpreter ist ein klassisches Muster der GOF, der sogenannten Viererbande *(Gang of Four)*, deren Buch *Design Patterns: Elements of Reusable Object-Oriented Software* den Begriff *Entwurfsmuster* bekannt gemacht und den objektorientierten Software-Entwurf maßgeblich geprägt hat.[5] Die Bestandteile des Interpreter-Musters sind:

- Die Syntax einer Ausdruckssprache, die als eine interne DSL verstanden werden kann.
- Ein Interpreter, der Ausdrücke dieser DSL auswerten kann.
- Ein optionaler Kontext in dem die Ausdrücke ausgewertet werden.

Syntax (ein ADT) und Interpreter werden dabei im klassischen OO-Stil modelliert: Als Interface mit abstrakter Methode (Interpreter) + konkrete Klassen die das Interface implementieren. (Siehe Abb. 3.1)

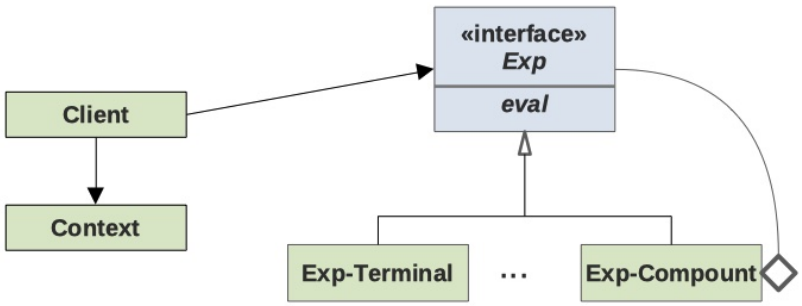

Abbildung 3.1: GOF: Interpreter-Muster.

Die Syntax ist ein ADT der in objektorientierter Art als abstrakte Klasse bzw. Interface mit abstrakter eval-Methode, sowie den das Interface implementierenden konkreten Klassen realisiert wird. Die Semantik, der Interpreter, ist die Implementierung der eval-Methode mit unterschiedlichen konkreten Klassen. In Enum-Notation im objektorientierten Stil:

```
// Expr
// - hat ein eval, und
// - gibt es in vier Varianten:
enum Expr(val eval: Map[String, Int] => Int) {
  case Const(number: Int)
    extends Expr(context => number)

  case Variable(name: String)
    extends Expr(context => context.getOrElse(name, 0))
```

[5] siehe https://en.wikipedia.org/wiki/Design_Patterns

```
  case Plus(left: Expr, right: Expr)
    extends Expr(context => left.eval(context) + right.eval(context))

  case Minus(left: Expr, right: Expr)
    extends Expr(context => left.eval(context) -right.eval(context))
}
```

Ein Anwendungsbeispiel ist:

```
import Expr._
val expr = Minus(Const(50), Plus(Variable("five"), Variable("three")))
val context = Map("three" -> 3, "five" -> 5)
val res = expr.eval(context) // 42
```

Betrachten wir das Ganze jetzt mit funktionalen Augen. In funktionalem Stil wird die eval-Methode zu einer Funktion:

```
enum Expr {
  case Const(number: Int)
  case Variable(name: String)
  case Plus(left: Expr, right: Expr)
  case Minus(left: Expr, right: Expr)
}
import Expr._

type Env = Map[String, Int]

def eval(expr: Expr): Env => Int = expr match {
  case Const(number) =>
    env => number
  case Variable(name: String) =>
    env => env(name)
  case Plus(left: Expr, right: Expr) =>
    env => eval(left)(env) + eval(right)(env)
  case Minus(left: Expr, right: Expr) =>
    env => eval(left)(env) -eval(right)(env)
}

val expr = Minus(Const(50), Plus(Variable("five"), Variable("three")))
val context = Map("three" -> 3, "five" -> 5)
val res = eval(expr)(context) // 42
```

3.3.2 Syntax und Semantik

Syntax

Das Interpreter-Muster kann als *initialer Ansatz* zur Beschreibung von Ausdrücken und ihrer Auswertung angesehen werden. Der Kerngedanke des initialen Ansatzes ist die Auffassung, dass syntaktische Strukturen *Datenstrukturen* als *Instanzen einer Typklasse* sind.

Beginnen wir damit Ausdrücke ganz allgemein als eine Typklasse zu definieren:

```
trait F_Exp[E] { // Typklasse, algebraische Struktur.
  def const(number: Int): E
```

```
  def variable(name: String): E
  def plus(left: E, right: E): E
  def minus(left: E, right: E): E
}
```

Das ist noch keine Syntax. Syntax das sind Datenstrukturen, abstrakte Syntaxbäume beispielsweise:

```
enum Expr { // ADT, Algebra
  case Const(number: Int)
  case Variable(name: String)
  case Plus(left: Expr, right: Expr)
  case Minus(left: Expr, right: Expr)
}
import Expr._
```

Die Syntax ist eine Instanz der Typklasse:

```
// ADT als Instanz von F_Exp / Algebra Expr ist eine F_Exp Algebra
given InitialAlgebra as F_Exp[Expr] with {
  def const(number: Int): Expr =
    Const(number)
  def variable(name: String): Expr =
    Variable(name)
  def plus(left: Expr, right: Expr): Expr =
    Plus(left, right)
  def minus(left: Expr, right: Expr): Expr =
    Minus(left, right)
}
```

Wir haben also zuerst von dem ADT Expr abstrahiert und sind zu undefinierten Konstruktions-Funktionen gekommen: Zu der Typklasse F_Exp. Dann haben wir wenig überraschend deklariert, dass Exp eine Instanz seiner Verallgemeinerung F_Exp ist.

Jetzt können "generische" Ausdrücke definiert werden. Das heißt Ausdrücke, die nur die von der Typklasse definierten Operationen nutzen und im Kontext unterschiedlicher Instanz unterschiedliche Werte liefern:

```
trait F_Exp[E] { // Typklasse
  def const(number: Int): E
  def variable(name: String): E
  def plus(left: E, right: E): E
  def minus(left: E, right: E): E
}
object F_Exp {
  def apply[E:F_Exp] = summon[F_Exp[E]]
}

// ein "generischer Ausdruck" entsprechend der Typklasse:
def expr[E: F_Exp] =
  F_Exp[E].minus(
    F_Exp[E].const(50),
    F_Exp[E].plus(
      F_Exp[E].variable("five"),
      F_Exp[E].variable("three")))
```

Oder wenn die die Ausdrücke etwas schöner aussehen sollen:

```
trait F_Exp[E] {
  def const(number: Int): E
  def variable(name: String): E
  def plus(left: E, right: E): E
  def minus(left: E, right: E): E
}
object F_Exp {
  def apply[E:F_Exp] = summon[F_Exp[E]]
  def C[E:F_Exp](number: Int) = F_Exp[E].const(number)
  def V[E:F_Exp](name: String) = F_Exp[E].variable(name)
  def P[E:F_Exp](left: E, right: E) = F_Exp[E].plus(left, right)
  def M[E:F_Exp](left: E, right: E) = F_Exp[E].minus(left, right)
}

// ein 'generischer Ausdruck' entsprechend F_Expr
import F_Exp._
def expr[E: F_Exp] = M(C(50), P(V("five"), V("three")))
```

expr ist jetzt in beiden Varianten kein (!) Ausdrucksbaum. Es ist eine generische Funktion, die bei einer gegebenen Instanz der Typklasse F_Exp mit irgendwelchen Funktionen const, variable, etc. einen Wert erzeugen kann.

Die zu benutzenden Funktionen const, variable, etc. werden als Instanz der Typklasse F_Exp implizit übergeben. Definieren wir den ADT Expr als Instanz, dann kann der generische Ausdruck konkrete Ausdrucksbäume erzeugen:

```
enum Expr { //ADT
  case Const(number: Int)
  case Variable(name: String)
  case Plus(left: Expr, right: Expr)
  case Minus(left: Expr, right: Expr)
}
import Expr._

// Gibt generischen Ausdrücken einen konkreten ADT-Wert als Syntax
given Syntax as F_Exp[Expr] with {
  def const(number: Int): Expr =
    Const(number)
  def variable(name: String): Expr =
    Variable(name)
  def plus(left: Expr, right: Expr): Expr =
    Plus(left, right)
  def minus(left: Expr, right: Expr): Expr =
    Minus(left, right)
}

// expr wird interpretiert als Syntax-Baum von Typ Expr
val syntaxTree = expr // Ausdrucksbaum:
  // Minus(Const(50),Plus(Variable(five),Variable(three)))
```

Generische Ausdrücke wie exp werden so als Syntaxbäume vom Typ Expr interpretiert:

exp (+ Syntax als implizites Argument) ⇒ Expr-Wert

Die F_Exp-Operationen const, variable, plus, minus werden als Baumerzeugungsoperationen interpretiert. Das ergibt die Syntax der Ausdrücke.

Semantik

Andere Interpretationen der F_Exp-Operationen sind natürlich auch möglich:

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

given Semntics as F_Exp[Env => Int] with {
  def const(number: Int): Env => Int =
    env => number
  def variable(name: String): Env => Int =
    env => env(name)
  def plus(left: Env => Int, right: Env => Int): Env => Int =
    env => left(env) + right(env)
  def minus(left: Env => Int, right: Env => Int): Env => Int =
    env => left(env) -right(env)
}

// expr wird interpretiert als Berechnung vom Typ Env => Int
val evaluationInEnv = expr

val env = Map("three" -> 3, "five" -> 5)
val result = evaluationInEnv(env) // 42
```

Syntax und Semantik sind jetzt Instanzen einer Typklasse. Statt einer eval-Funktion geben wir eine alternative Interpretation der Typklasse an. Das Ganze ist letztlich äquivalent zur Definition einer Auswertungsfunktion eval die rekursiv über die Struktur der Ausdrücke definiert ist. (Siehe Abb. 3.2)

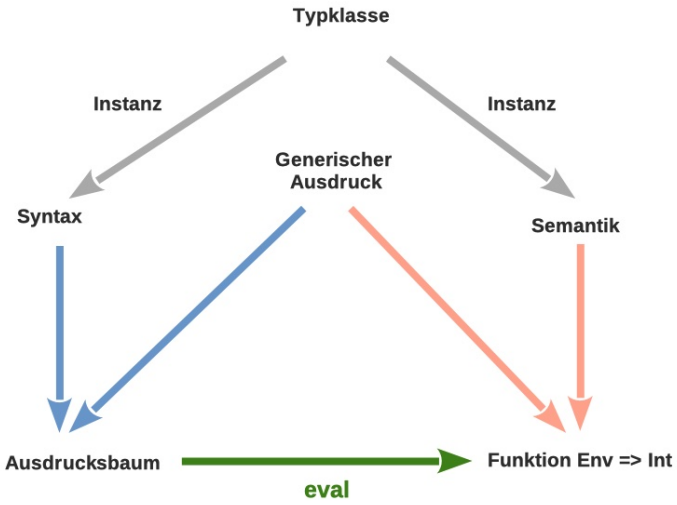

Abbildung 3.2: Syntax und Semantik als Instanzen der gleichen Typklasse.

Bei der Definition der Semantik unserer Ausdrücke haben wir viele Wahlmöglichkeiten. Die Typklasse gibt ja nicht vor, wie Operationen const, variable, etc. zu interpretieren sind. Die Berechnung eines Werts in einer bestimmten Umgebung mit der Belegung der Variablen mit Werten ist nur eine Möglichkeit. Die Ausdrücke könnten beispielsweise auch einfach für ihre textuelle Darstellung stehen:

```
given Semntics as F_Exp[String] with {
```

```
  def const(number: Int): String =
    number.toString
  def variable(name: String): String =
    name.toString
  def plus(left: String, right: String): String =
    s"($left + $right)"
  def minus(left: String, right: String): String =
    s"($left -$right)"
}

// generischer Wert interpretiert
// als Berechnung vom Typ Env => Int
val stringRep = expr // (50 -(five + three))
```

Die Syntax, also der ADT der Terme / Ausdrucksbäume, ist eine *ganz besondere Instanz* der Typklasse F_Exp: Sie ergibt sich quasi automatisch aus der Typklasse. Für eine bestimmte Semantik, also eine andere Instanz der Typklasse, kann eine eindeutig rekursiv über die Struktur der Ausdrücke definierte Funktion eval diesen ihre Bedeutung geben. (Siehe Abb. 3.3.)

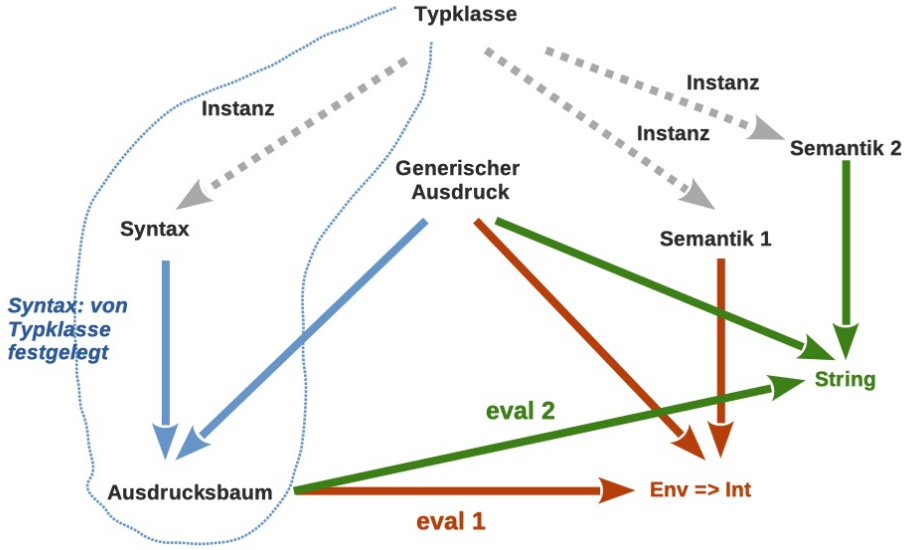

Abbildung 3.3: Triviale Syntax und und viele Semantiken.

Die Syntax ist ein initiales Objekt

Die Syntax spielt also in zweifacher Art eine besondere Rolle unter all den Instanzen der Typklasse:

- Sie ist schon komplett durch die Typklasse festgelegt, sie repräsentiert ja nichts anderes, als deren Ausdrucksmöglicheiten in Form einer Datenstruktur.
- Von der Syntax führt eine eindeutig bestimmte, strukturell rekursive Funktion zu jeder denkbaren Semantik der Ausdrücke.

In etwas mathematischer Ausdrucksweise sagt man, dass die Syntax *initiales Objekt* in der Kategorie der F-Algebren sind. Dabei ist Ist F der Funktor, der mit der Typklasse `F_Exp` indirekt, aber auch "automatisch" definiert wird.

Ein Objekt ist *initial* in einer Kategorie, wenn es einen Morphismus von diesem Objekt zu jedem anderen Objekt der Kategorie gibt. Die Kategorie besteht hier aus den Algebren in denen die in der Typklasse definierten Operationen existieren.

Der eindeutig bestimmte Morphismus ist die *eval*-Funktion: Sie führt von der Syntax, dem initialen Objekt zu jeder anderen Algebra mit den verlangten Operationen. (Siehe Abb. 3.4.)

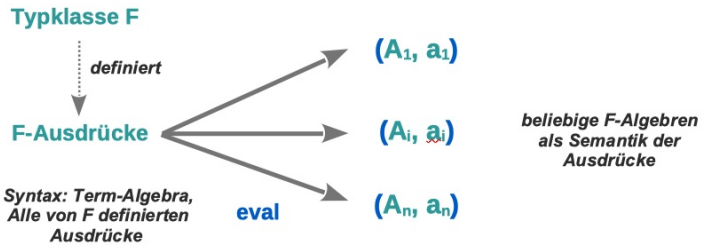

Abbildung 3.4: Syntax als initiales Objekt in der Kategorie der F-Algebren.

Das Interpreter-Muster haben wir damit etwas vage und informell aber ausreichend weit auseinander genommen und analysiert, um ansatzweise zu verstehen, warum es als *initialer* Ansatz bezeichnet wird. Die mathematischen Details können mit der einschlägigen Literatur vertieft werden. [6]

Wir halten einfach fest, dass die initiale Sicht auf Syntax und Semantik eine völlig natürliche und offensichtliche Angelegenheit ist: Ein ADT (Kollektion Baum-erzeugende Funktionen) als Syntax-Instanz der Typklasse und Homomorphismen (also strukturell rekursive Funktionen) die von ihr in passende Algebren als Semantik-Instanzen der Typklasse abbilden. Passend bedeutet nichts anderes, als dass es für die Konstrukte der Syntax Operationen in der Semantik-Algebra geben muss.

3.4 Syntax und Semantik – Ein finaler Ansatz

3.4.1 Ausdrücke als ihre Auswertungs-Funktionen

Dem initialen kann man den *finalen Ansatz* zum Verständnis von Syntax und Semantik gegenüber stellen. Die Frage, was an dieser Sicht "final" ist, stellen wir erst einmal zurück. Man ahnt sicher schon, dass es etwas mit Kategorientheorie zu tun hat.

Beginnen wir mit der Beobachtung, dass es nicht zwingend notwendig ist, einen Ausdruck als Datenstruktur zu verstehen. Nach allem, was man je über Syntax, abstrakte Syntax und Ausdrucksbäume gelernt hat, kostet es erst einmal etwas Überwindung zu akzeptieren, dass das alles unnötig sein soll. Aber so überraschend ist es dann doch wieder nicht, schon in *SIPC*[7] finden wir die Idee, dass es reicht, Ausdrücke durch die Funktionen zu repräsentieren, die sie auswerten.

[6] Kurz und bündig ist B. G. Pierces: *Basic category theory for computer scientists*, MIT-Press August 1991

[7] H. Abelson, G. Sussman, J. Sussman: *Structure and Interpretation of Computer Programs* MIT-Press 2^{te} Auflage, 1996. – Für funktionale Programmierer das was Euklids *Elemente* für Mathematiker ist.

> *For the evaluator, this means that the syntax of the language being evaluated is determined solely by the procedures that classify and extract pieces of expressions.*[8]

Auf unsere einfachen Ausdrücke und Scala übertragen sieht das folgendermaßen aus:

```
trait Expr { // ein Ausdruck 'ist' seine eval-Funktion
  def eval(context: Map[String, Int]): Int
}

object Expr {
  def const(number: Int): Expr =
    (context: Map[String, Int]) => number

  def plus(left: Expr, right: Expr): Expr =
    (context: Map[String, Int]) =>
     left.eval(context) + right.eval(context)

  def minus(left: Expr, right: Expr): Expr =
    (context: Map[String, Int]) =>
     left.eval(context) -right.eval(context)

  def variable(name: String): Expr =
    (context: Map[String, Int]) =>
     context.getOrElse(name, 0)
}
import Expr._

val expr = minus(const(50), plus(variable("five"), variable("three")))
val ctxt = Map("three" -> 3, "five" -> 5)
val res = expr.eval(ctxt) // 42
```

Die Interpretation der Ausdrücke ist jetzt fix. Sie werden als Funktionen vom Typ $Env \Rightarrow Int$ interpretiert. Wollten wir eine andere Repräsentation, etwa als Text, dann wird eine komplett neue Definition benötigt:

```
trait Expr {
  def eval(): String
}

object Expr {
  def const(number: Int): Expr =
    () => number.toString

  def plus(left: Expr, right: Expr): Expr =
    () => s"(${left.eval()} + ${right.eval()})"

  def minus(left: Expr, right: Expr): Expr =
    () => s"(${left.eval()} -${right.eval()})"

  def variable(name: String): Expr =
    () => name
}

import Expr._
```

[8] Abschnitt 4.1.2, Seite 501

```
val expr = minus(const(50), plus(variable("five"), variable("three")))
val res = expr.eval() // (50 -(five + three))
```

Es überrascht sicher wenig, dass die Sache mit einer Abstraktion in eine Typklasse generisch gemacht werden kann:

```
trait Expr[A] {
  def const(number: Int): A
  def variable(name: String): A
  def plus(left: A, right: A): A
  def minus(left: A, right: A): A
}

object Expr {
  def apply[A](using exprRep: Expr[A]) = exprRep
  // ~ def apply[A: Expr] = summon[Expr[A]]
}

def expr[A: Expr]: A =
  Expr[A].minus(
    Expr[A].const(50),
    Expr[A].plus(
      Expr[A].variable("five"),
      Expr[A].variable("three")))
```

Oder wenn etwas schönere Ausdrücke gewünscht sind:

```
trait Expr[A] {
  def const(number: Int): A
  def variable(name: String): A
  def plus(left: A, right: A): A
  def minus(left: A, right: A): A
}

object Expr {
  def apply[A: Expr] = summon[Expr[A]]
  def C[A: Expr](number: Int): A = Expr[A].const(number)
  def V[A: Expr](name: String): A = Expr[A].variable(name)
  def P[A: Expr](left: A, right: A): A = Expr[A].plus(left, right)
  def M[A: Expr](left: A, right: A): A = Expr[A].minus(left, right)
}

import Expr._

// generischer Ausdruck
def expr[A: Expr]: A = M( C(50), P( V("five"), V("three")))
```

Unterschiedliche Instanzen der Typklasse geben unterschiedliche Interpretationen des Ausdrucks. Soll es beispielsweise eine Auswertung in einer Umgebung sein, dann nehmen wir diese Instanz:

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

given Expr[Context => Int] with {
  def const(number: Int): Context => Int =
    context => number
  def variable(name: String): Context => Int =
```

```
    context => context.getOrElse(name, 0)
  def plus(left: Context => Int, right: Context => Int): Context => Int =
    context => left(context) + right(context)
  def minus(left: Context => Int, right: Context => Int): Context => Int =
    context => left(context) -right(context)
}

val context: Context = Map("three" -> 3, "five" -> 5)
val evalInContext = expr // implicitly pass typeclass instance
val res = evalInContext(context) // 42
```

Oder als textuelle Darstellung mit dieser Instanz:

```
given Expr[String] with {
  def const(number: Int): String =
    number.toString
  def variable(name: String): String =
    name
  def plus(left: String, right: String): String =
    s"($left + $right)"
  def minus(left: String, right: String): String =
    s"($left -$right)"
}

val evalToString = expr // implicitly pass typeclass instance
val res = evalToString // (50 -(five + three))
```

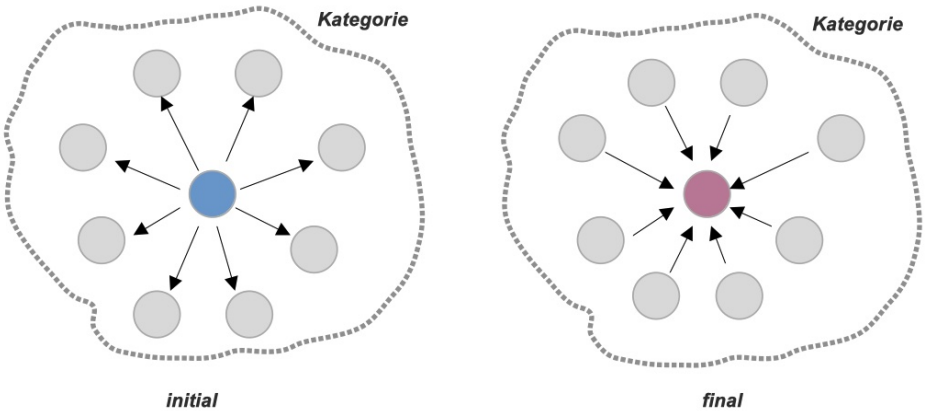

Abbildung 3.5: Initiale und finale Objekte.

Das ist alles im Prinzip das Gleiche wie weiter oben im "initialen Ansatz". Der einzige Unterschied ist, dass wir die Syntaxbäume komplett eliminiert haben. Damit ist die "initiale Algebra" eliminiert worden. An ihre Stelle ist Interpretation der Ausdrücke als Funktionen getreten. – Und genau das ist der *finale Ansatz* zur Darstellung von Syntax und Semantik.

Der Begriff *final*[9] kommt aus der Kategorientheorie und bezeichnet ein Objekt zu dem es einen Pfeil (Morphismus) von jedem anderen Objekt der gleichen Kategorie gibt. (Siehe Abb. 3.5.)

Die Signatur der Operationen, so wie sie in einer Typklasse F ausgedrückt wird, definiert indirekt

[9] Statt *final* wird auch der Begriff *terminal* verwendet.

den ADT der Terme, der initiales Objekt in der Kategorie der F-Algebren sind. F kann auch als implizite Definition eines finalen Objekts in einer anderen Kategorie gesehen werden, des finalen Objekts in der von F definierten Kategorie der *Coalgebren*. (Siehe Abb. 3.6.) Die mathematischen Details ersparen wir uns hier. [10]

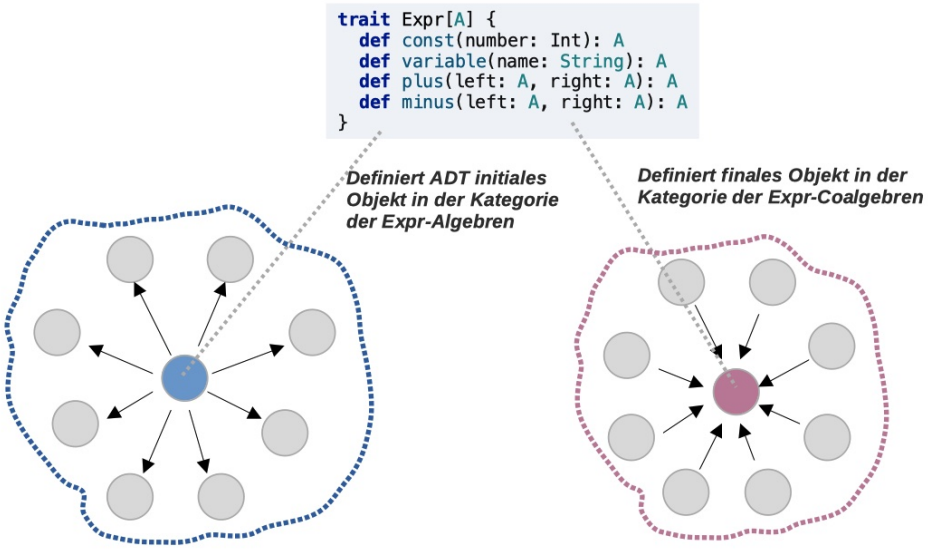

Abbildung 3.6: Kategorie der Algebren und Coalgebren mit initialem und finalem Objekt.

3.4.2 Das Ausdrucksproblem

Bei dem Ausdrucksproblem[11] (*Expression Problem*) handelt es sich um ein bekanntes Problem aus dem Grenzbereich der Softwaretechnik und Programmiersprachen.

> *Das Ausdrucksproblem: Finde eine Möglichkeit um sowohl die Struktur, als auch die Auswertung von Ausdrücken in modularer Art erweitern zu können.*

Das Problem ist keinesfalls ein esoterisches Thema der Programmiersprachen. Es ist ein zentrales Problem der Softwaretechnik, vor allem, wenn man Programmkomponenten in funktionaler Interpretation als Ausdrücke einer speziellen DSL versteht.

Bekanntlich können in einer objektorientierten Sprachen neue Strukturvarianten und in einer funktionalen Sprache neue Auswertungsvarianten in modularer Art, also ohne die die Modifikation existierenden Codes hinzugefügt werden.

Nehmen wir als Beispiel unsere Ausdrücke die

- in der Struktur um eine neue Operation, die Multiplikation, und
- in ihrer Auswertung um eine weitere Auswertungsfunktion

erweitert werden sollen.

[10] Das Thema ist aktuell und praktisch relevant. Eine gewisse Festigkeit in Kategorientheorie wird von der aktuellen Literatur jedoch vorausgesetzt. Zum Einstieg siehe etwa
https://www.mathematik.uni-marburg.de/~gumm/Papers/Cubo.pdf

[11] https://en.wikipedia.org/wiki/Expression_problem

Strukturerweiterung der OO-Variante

In der objektorientierten Variante ist die *Struktur*erweiterung modular möglich. Da in Scala 3 Enums nicht erweiterbar sind, nutzen wir die klassische OO-Codierung der Ausdrücke:

```
// vorhandener Code
sealed abstract class Expr {
  val eval: Map[String, Int] => Int
}
final class Const(number: Int) extends Expr {
  override val eval = context => number
}
final class Variable(name: String) extends Expr {
  override val eval = context => context.getOrElse(name, 0)
}
final class Plus(left: Expr, right: Expr) extends Expr {
  override val eval = context => left.eval(context) + right.eval(context)
}
final class Minus(left: Expr, right: Expr) extends Expr {
  override val eval = context => left.eval(context) -right.eval(context)
}

// hinzugefügt ohne Eingriff in bestehnden Code
final class Mult(left: Expr, right: Expr) extends Expr {
  override val eval = context => left.eval(context) * right.eval(context)
}
```

Perfekt. Die Objektorientierung erlaubt die modulare Erweiterung der Struktur.

Erweiterung der Auswertung bei der OO-Variante

Das Hinzufügen einer neuen Auswertungsroutine ist dagegen nicht modular, also ohne Eingriff in bestehenden Code möglich:

```
sealed abstract class Expr {
  val eval: Map[String, Int] => Int
  // hinzugefügt:
  val evalStr: String
}
final class Const(number: Int) extends Expr {
  override val eval = context => number
  // hinzugefügt:
  override val evalStr: String = number.toString
}
final class Variable(name: String) extends Expr {
  override val eval = context => context.getOrElse(name, 0)
  // hinzugefügt:
  override val evalStr: String = name.toString
}
final class Plus(left: Expr, right: Expr) extends Expr {
  override val eval = context => left.eval(context) + right.eval(context)
  // hinzugefügt:
  override val evalStr: String = s"(${left.evalStr} + ${right.evalStr})"
}
final class Minus(left: Expr, right: Expr) extends Expr {
  override val eval = context => left.eval(context) -right.eval(context)
```

```
  // hinzugefügt:
  override val evalStr: String = s"(${left.evalStr} -${right.evalStr})"
}
```

Die Erweiterung erfordert heftige Eingriffe in den bestehenden Code. Das ist alles andere als modular.

Erweiterung der Auswertung bei der Funktionalen Variante

Bei einem funktionalen Programm sind die Verhältnisse genau umgekehrt. Neue Auswertungsfunktionen können problemlos hinzugefügt werden:

```
// vorhanden
enum Expr {
 case Const(number: Int)
 case Variable(name: String)
 case Plus(left: Expr, right: Expr)
 case Minus(left: Expr, right: Expr)
}
import Expr._

type Env = Map[String, Int]

def eval(expr: Expr): Env => Int = expr match { ... }

// hinzugefügt
def evalStr(expr: Expr): String = expr match { ... }
```

Erweiterung der Struktur bei der Funktionalen Variante

Eine Erweiterung der Struktur ist dagegen nicht modular möglich:

```
enum Expr {
  case Const(number: Int)
  case Variable(name: String)
  case Plus(left: Expr, right: Expr)
  case Minus(left: Expr, right: Expr)

  // hinzugefügt:
  case Mult(left: Expr, right: Expr)
}
import Expr._

def eval1(expr: Expr): A = expr match {
  ...
  // hinzugefügt:
  case Mult(left: Expr, right: Expr) => ...
}

def eval1(expr: Expr): A = expr match {
  ...
  // hinzugefügt:
  case Mult(left: Expr, right: Expr) => ...
}
```

Der Typ muss angepasst und alle Auswertungsfunktionen müssen erweitert werden.

Insgesamt sind beiden Varianten, funktional und objektorientiert, eine nicht völlig zufriedenstellend. Modulare Erweiterungen sind jeweils nur in einer Dimension möglich. (Siehe Abb. 3.7.)

Erweiterung	OO	λ
Struktur	🙂	☹
Auswertung	☹	🙂

Abbildung 3.7: Das Ausdrucksproblem: Klassische oo/funktionale Behandlung

Das Ausdrucksproblem und der finale Ansatz

Nehmen wir als Ausgangspunkt die Ausdrücke ohne Multiplikation mit nur einer Auswertungsfunktion:

```
trait Expr[A] {
  def const(number: Int): A
  def variable(name: String): A
  def plus(left: A, right: A): A
  def minus(left: A, right: A): A
}

given Expr[A] with {
   ...
}

// hinzugefügt:
given Expr[B] with {
   ...
}
```

Anscheinend gibt es da keinen Unterschied zu der funktionalen Variante oben. Eine weitere Auswertungsfunktion wird einfach als weitere Instanz der Typklasse hinzugefügt. Eine offensichtlich modulare Erweiterung. Für eine Strukturerweiterung muss in die Definition von Expr und all seiner Instanzen eingegriffen werden. Erst mal keine Verbesserung – aber:

Mit einem kleinen Trick lässt sich das aber vermeiden. Typklassen können erweitert werden! Das lässt sich für eine modulare Erweiterung der Struktur ausnutzen:

```
// vorhanden
trait Expr[A] {
  def const(number: Int): A
  def variable(name: String): A
  def plus(left: A, right: A): A
  def minus(left: A, right: A): A
}
import Expr._

given Expr[String] with {
  def const(number: Int): String = number.toString
  def variable(name: String): String = name
```

```
  def plus(left: String, right: String): String =
    s"($left + $right)"
  def minus(left: String, right: String): String =
    s"($left -$right)"
}

// Modular (!) hinzugefügt:
trait Multiplication[A] {
  def mult(left: A, right: A): A
}
import Multiplication._

given Multiplication[String] with {
  def mult(left: String, right: String): String =
    s"($left * $right)"
}
```

Perfekt! Der finale Ansatz erlaubt eine zufriedenstellende Lösung des Ausdrucksproblems. Struktur und Interpretation von Ausdrücken können modular erweitert werden.

3.4.3 Das Deserialisierungsproblem

Als *Serialisierung* bezeichnet man die Transformation einer beliebigen Struktur in eine äquivalente strikt lineare (serielle) Darstellung, beispielsweise als Text oder als Bytestrom. *Deserialisierung* ist der umgekehrte Prozess der Rekonstruktion der Struktur aus ihrer seriellen Darstellung. Serialisierung und Deserialisierung finden unter Akademikern wenige Interesse, es handelt sich aber um Thematiken von enormer praktischer Bedeutung. Hier betrachten wir vor allem die Deserialisierung, also die Rekonstruktion der Struktur eines Ausdrucks aus seiner linearen Darstellung.

Zur Vereinfachung nehmen wir eine von *JSON*[12] inspirierte Variante der Problemstellung. JSON beschreibt die serielle (Text-) Form beliebig strukturierter Daten. Dabei wird eine universelle Struktur als *"JSON-Wert"*, definiert. Diesen Werten entspricht dann eine exakt festgelegte Textcodierung.

Ein *JSON-Wert* ist

- ein Objekt bestehend aus Name–Wert–Paaren,
- ein Array, also eine Folge von Objekten,
- ein String,
- eine Zahl,
- ein boolescher Wert oder
- null.

Ein JSON-Wert ist also also eine universelle, generische Baumstruktur, die in Scala als Enum-codierter ADT dargestellt werden kann:

```
enum JValue {
  case JObject(prop: Map[String, JValue])
```

[12] siehe https://de.wikipedia.org/wiki/JavaScript_Object_Notation

```
  case JArray(elements: List[JValue])
  case JString(value: String)
  case JNumber(value: String)
  case JTrue
  case JFalse
  case JNull
}
```

RFC 8259[13] definiert dazu eine textuelle Darstellung die leicht zu erzeugen und zu erkennen (parsen) ist.

Bei einer *Serialisierung* in das JSON-(Text-)-Format erzeugt eine Anwendung typischerweise aus einer anwendungsorientierten Struktur eine JSON-Struktur, die dann von einer standardisierten JSON-Komponente in Textform entsprechend RFC-8259 gebracht wird.

Bei einer *Deserialisierung* aus dem JSON-(Text-)-Format parst eine standardisierte JSON-Komponente den strukturierten Text in einen JSON-Wert, der dann von der Anwendung in eine anwendungsorientierte Struktur umgewandelt wird. (Siehe Abb. 3.8.)

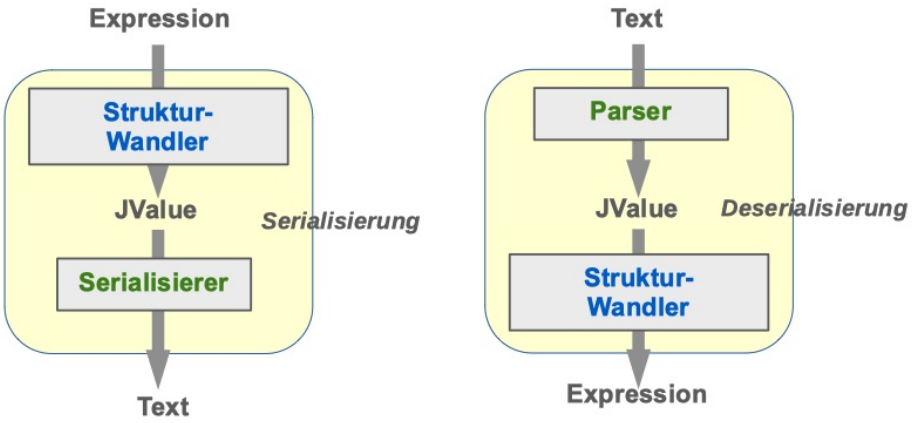

Abbildung 3.8: Serialisierung und Deserialisierung im JSON-Stil.

Viele gut verfügbare JSON-Bibliotheken stellen Parser zur Verfügung, mit denen JSON-kompatible Texte in die universelle Struktur eines JValue geparst werden können und die umgekehrt einen JValue in eine JSON-kompatible Textform bringen können. Das Problem, das eine Anwendung darum noch lösen muss, ist

- die Konvertierung eigener Strukturen in JSON-Strukturen ("Serialisierung")
- und umgekehrt die Konvertierung von JSON-Strukturen in eigene Strukturen ("Deserialisierung").

Wir schließen uns einer üblichen Praxis an, und bezeichnen diese *Struktur-Umwandlungen* als *Serislisierung* bzw. *Deserislisierung*.

Serialisierung

Statt der universellen JSON-Struktur (JValue) betrachten eine vereinfachte aber prinzipiell gleichwertige Version einer universellen Struktur:

[13] https://tools.ietf.org/html/rfc8259

```
enum Tree(override val toString: String) {
  case Leaf(value: String)
    extends Tree ( toString = value )
  case Node(value: String, elements: List[Tree])
    extends Tree ( toString = s"{$value: [${elements.mkString(", ")}]}" )
}
```

Die Transformation in Text übernimmt eine toString-Methode.

Als Beispiel für eine anwendungsorientierte Struktur dienen Ausdrücke in finaler Codierung:

```
trait Expr[A] {
  def const(number: Int): A
  def variable(name: String): A
  def plus(left: A, right: A): A
  def minus(left: A, right: A): A
}
```

Der syntaktischen Bequemlichkeit halber werden Ausdrücke mit einem Begleiterobjekt ausgestattet:

```
object Expr {

  def apply[A](using exprRep: Expr[A]) =
    exprRep

  def Const[A: Expr](number: Int) =
    Expr[A].const(number)

  def Variable[A:Expr](name: String): A =
    Expr[A].variable(name)

  def Plus[A: Expr](left: A, right: A): A =
    Expr[A].plus(left: A, right: A): A

  def Minus[A: Expr](left: A, right: A): A =
    Expr[A].minus(left: A, right: A): A
}
```

Ein final codierter (“generischer”) Ausdruck wie etwa

```
def expr[A: Expr]: A =
  Minus(
    Plus(
      Variable("five"),
      Const(45)),
    Plus(
      Variable("five"),
      Variable("three")))
```

kann einfach mit einer passenden Instanz der Typklasse Expr serialisiert werden:

```
given ToTree as Expr[Tree] with {

  def const(number: Int): Tree =
    Node("C", List[Tree](Leaf(number.toString)))
```

```
  def variable(name: String): Tree =
    Node("V", List[Tree](Leaf(name)))

  def plus(left: Tree, right: Tree): Tree =
    Node("P", List[Tree](left, right))

  def minus(left: Tree, right: Tree): Tree =
    Node("M", List[Tree](left, right))
}

val tree: Tree = expr
```

Deserialisierung – initial

Bei der Deserialisierung wird ein Text eine anwendungsorientierte Struktur konvertiert. Wie üblich gehen wir davon aus, dass ein Parser zur Verfügung steht, der Texte in Bäume vom universellen Strukturtyp Tree umwandelt. Es bleibt nur noch Trees in Expr umzuwandeln.

Dummerweise ist ein *final* codierter Ausdruck generisch. (Siehe Abb. 3.9.)

```
val tree: Tree =
  Node("minus",
    List(
      Node("plus",
      List(
        Node("var",
          List(Leaf("five"))),
        Node("const",
          List(Leaf("45"))))
      ),
      Node("plus",
        List(
          Node("var", List(Leaf("five"))),
          Node("var",List(Leaf("three")))))))
```

```
def expr[A: Expr]: A =
  Expr[A].minus(
    Expr[A].plus(
      Expr[A].variable("five"),
      Expr[A].const(45)),
    Expr[A].plus(
      Expr[A].variable("five"),
      Expr[A].variable("three")))
```

Abbildung 3.9: Deserialisierung: problematische generische Zielstruktur.

Die Deserialsierung in einen *initial* codierten Ausdruck ist dagegen völlig unproblematisch. Initial codierte Ausdrücke sind ja Ausdrücke in ihrer üblichen ADT-Darstellung (abstrakte Syntax):

```
// ADT der Ausdrücke, initiale Algebra in der Kategorie der Expr-Algebren
enum ExprADT {
  case Const(number: Int)
  case Variable(name: String)
  case Plus(left: ExprADT, right: ExprADT)
  case Minus(left: ExprADT, right: ExprADT)
}
```

Mit einer rekursiv über die Struktur von Trees können diese in ExprADT-Werte transformiert werden. (Siehe Abb. 3.10.)

```
def treeToADT(tree: Tree): ExprADT = tree match {
  case Node(tag, nodes) =>
    tag match {
      case "const" =>
        nodes match {
          case Leaf(str) :: Nil => Const(str.toInt)
```

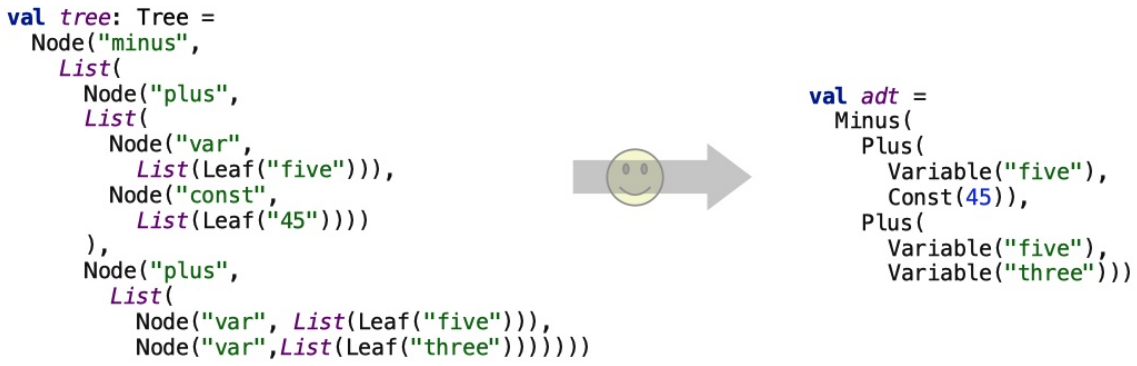

Abbildung 3.10: Deserialisierung: unprobkematische Zielstruktur ADT.

```
      }
    case "var" =>
      nodes match {
        case Leaf(str) :: Nil => Variable(str)
      }
    case "plus" =>
      nodes match {
        case node0 :: node1 :: Nil =>
          Plus(treeToADT(node0), treeToADT(node1))
      }
    case "minus" =>
      nodes match {
        case node0 :: node1 :: Nil =>
          Minus(treeToADT(node0), treeToADT(node1))
      }
  }
}
```

Der Übersicht halber haben wir auf jede Fehlerbehandlung verzichtet.

Die Interpretation generischen Ausdrucks mit ToADT führt zum gleichen Ergebnis wie die Deserialisierung der Serialisierung des generischen Ausdrucks. (Siehe Abb. 3.11.)

Deserialisierung – final

Initial ist also einfach. Soll die Serialisierung aber zu einem *final* codierten Ausdruck führen, dann muss die Deserialisierung einen *generischen Wert* erzeugen:

```
val tree: Tree = ...
val genExpr: [A: Expr] => A = ??? Deserialisierung von tree ???
```

d.h. einen Wert, der einen Typ-Parameter hat. Normalerweise unterliegen generische Dinge gewissen Beschränkungen. Klassen und Funktionen können in der Regel generisch definiert werden, aber es ist beispielsweise nicht möglich, etwas Generisches als Parameter zu übergeben, der Typ-Parameter wird bei der Übergabe gebunden und das Argument auf einen Typ instantiiert. Die Werte einer Funktion sind ebenfalls in der Regel nicht generisch. Kurzum, Generisches darf nicht überall erscheinen und ist so Bürger zweiter Klasse.

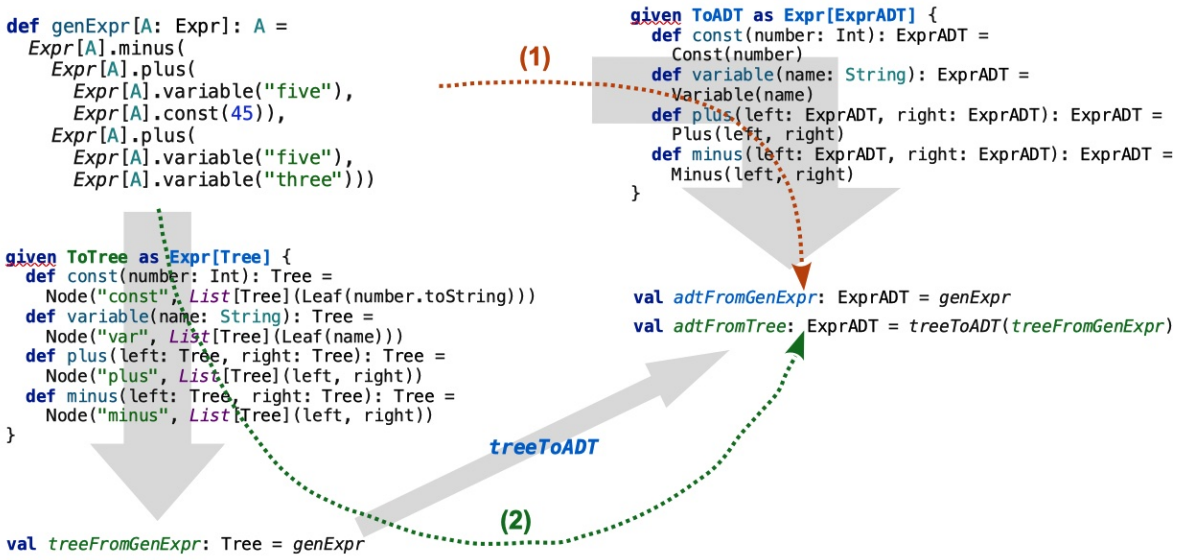

Abbildung 3.11: Interpretation als ExprADT = treeToADT(Interpretation als Tree)

Sind generische Werte vollwertige Mitglieder einer Sprache, dann spricht man von *Rang-2- Polymorphismus*. Scala 3 unterstützt diesen "höheren Polymorphismus" mit seinen polymorphen Funktionstypen (*Polymorphic Function Types*[14]) und wir können schreiben:

```
val genExpr: [A] => Expr[A] => A = fromTree(tree)
```

Syntaktischer Zucker wie [A: Expr] => A wird leider nicht unterstützt. Das lässt sich aber leicht verschmerzen. Fehlt nur noch die Definition von fromTree. Das ist aber nur noch eine einfache Rekursion über die Struktur der Trees, die wir ober schon öfter gesehen haben:

```
// rank-2: erzeugt einen generischen (!) Wert
def fromTree(tree: Tree): [A] => Expr[A] => A =
  [A] => (eRep: Expr[A]) => tree match {
    case Node(tag, nodes) =>
      tag match {
        case "const" =>
          nodes match {
            case Leaf(str) :: Nil => eRep.const(str.toInt)
          }
        case "var" =>
          nodes match {
            case Leaf(str) :: Nil => eRep.variable(str)
          }
        case "plus" =>
          nodes match {
            case node0 :: node1 :: Nil =>
              eRep.plus(fromTree(node0)(eRep), fromTree(node1)(eRep))
          }
        case "minus" =>
          nodes match {
            case node0 :: node1 :: Nil =>
```

[14] https://dotty.epfl.ch/docs/reference/new-types/polymorphic-function-types.html
Nicht zu verwechseln mit *higher kinded types*.

```
            eRep.minus(fromTree(node0)(eRep), fromTree(node1)(eRep))
        }
      }
  }

val tree: Tree = ...
val genExpr: [A] => Expr[A] => A = fromTree(tree)
```

Der erzeugte Ausdruck ist jetzt ein generischer Wert. Er kann jetzt mit einer Instanz der Typklasse Expr in einen konkreten, nicht-generischen "richtigen" Wert verwandelt werden:

```
def deserialize[A: Expr](tree: Tree) =
   fromTree(tree)(summon[Expr[A]])
```

Beispielsweise zu einem String:

```
given StrExpr as Expr[String] with {
  def const(number: Int): String = number.toString
  def variable(name: String): String = name
  def plus(left: String, right: String): String = s"($left + $right)"
  def minus(left: String, right: String): String = s"($left -$right)"
}

val res = deserialize(tree)
```

Mit Scala-3's Kontext-Funktionen *(Context Functions)*[15] können wir die lästige explizite Übergabe der Typklassen-Instanz los werden. Kontext-Funktionen sind Funktionen mit einem "Kontext-Parameter", also einem Parameter, der implizit aus dem Kontext übernommen wird. Damit wird unsere Deserialisierung zu:

```
def fromTree(tree: Tree): [A] => Expr[A] ?=> A =
  [A] => (using eRep: Expr[A]) => tree match {
  case Node(tag, nodes) =>
    tag match {
      case "const" =>
        nodes match {
          case Leaf(str) :: Nil => summon[Expr[A]].const(str.toInt)
        }
      case "var" =>
        nodes match {
          case Leaf(str) :: Nil => summon[Expr[A]].variable(str)
        }
      case "plus" =>
        nodes match {
          case node0 :: node1 :: Nil =>
            eRep.plus(fromTree(node0)(using summon[Expr[A]]),
                fromTree(node1)(using summon[Expr[A]]))
        }
      case "minus" =>
        nodes match {
          case node0 :: node1 :: Nil =>
            eRep.plus(fromTree(node0)(using summon[Expr[A]]),
                fromTree(node1)(using summon[Expr[A]]))
        }
    }
```

[15] https://dotty.epfl.ch/docs/reference/contextual/context-functions.html

```
}

val tree: Tree = ...
val genExpr: [A] => Expr[A] ?=> A = fromTree(tree)
```

Der (implizite) Kontext-Parameter wird im Typ recht anschaulich mit ?=> markiert. Damit können wir jetzt wie gewohnt die Typklassen-Instanz implizit übergeben:

```
given StrExpr as Expr[String] with {
  def const(number: Int): String = number.toString
  def variable(name: String): String = name
  def plus(left: String, right: String): String =
    s"($left + $right)"
  def minus(left: String, right: String): String =
    s"($left -$right)"
}

val res: String = genExpr[String] //((five + 45) -(five + three))
```

Texte können also insgesamt zu generischen (final codierten) Ausdruckswerten deserialisiert werden. Damit ist das Deserialisierungsproblem, also die Erzeugung eines final codierten Ausdruckswerts aus einer serialisierten Form gelöst: statisch unbekannte Daten, also Daten, die das Programm zur Laufzeit aus irgendwelchen Quellen liest oder empfängt können in die final codierte Form gebracht und damit in eine “algebraisch” strukturierte Anwendung integriert werden.

Teil II

Monadische Entwurfsmuster

Kapitel 4

Funktoren und Natürliche Transformationen

4.1 Funktoren

4.1.1 Daten in Strukturen transformieren

Mit *map* kann man die Inhalte einer Struktur (eines Containers) verändern, ohne dabei in die Struktur hinein sehen zu müssen. Oder, funktional, mit *map* lässt sich aus einer Struktur eine neue Struktur gleicher Art aber mit geändertem Inhalt erzeugen. (Siehe Abb. 4.1)

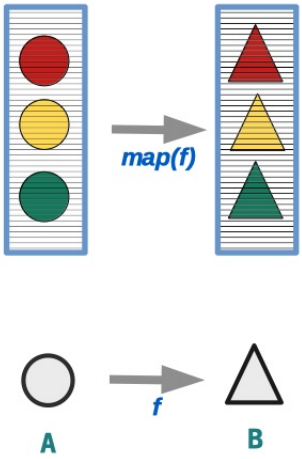

Abbildung 4.1: Map: Verpacktes manipulieren.

Umgekehrt kann man die Verfügbarkeit einer map-Funktion auch als die Essenz der Idee eines Containers oder einer Verpackung sehen. Die Daten sind verpackt, aber nicht "weg", sie können immer noch genauso verarbeitet werden, als fänden sich in Freiheit außerhalb des Containers. Das ist die Essenz einer Hülle, die ihren Inhalt unangetastet lässt. Etwas, das "ge-map-t" werden kann, ist darum das Ergebnis einer Informationsverarbeitung (Verpacken) bei der keine Information vernichtet oder hinzugefügt wird.

map-Funktionen oder -Methoden sind gehören zum Babybrei er funktionalen Programmierung und sind natürlich auch in Scala verfügbar. Alle Container-Typen, die das Trait Iterable imple-

mentieren haben ein map.

```
val lst_1: List[Int] = List(1,2,3)
val lst_2: List[String] = lst_1.map( _.toString)
```

OO-Sicht: Interface der Objekte, die eine map-Methode haben

In der Scala-API kann ein trait ein *Interface* im objektorientierten Sinn darstellen, also eine Funktionalität die *Objekte* anbieten. Das *Interface*:

```
// Interface
trait WithMap[A] { // "Etwas (ein Objekt) mit map" als Interface
  def map[B](f: A => B): WithMap[B]
}
```

ist die Basis aller Objekte, die eine map-Methode haben. Eine Klasse die (genauer: deren Objekte) dieses Interface erfüllt (erfüllen) ist beispielsweise

```
class Triple[A](a: A, b: A, c: A) extends WithMap[A] {
  override def map[B](f: A => B): Triple[B] =
    Triple(f(a), f(b), f(c))
}

val p1: Triple[String] = Triple("Hallo", "Welt", "!")
val p2: Triple[Int] = p1.map((x: String) => x.length)
```

Die an map übergebene Funktion f muss *kovariant* im Ergebnistyp WithMap[B] sein. Beispielsweise ist folgende Definition von Triple "falsch" bzw. "nicht erlaubt":

```
trait WithMap[A] {
  def map[B](f: A => B): WithMap[B]
}

 // OK: WithMap wird kovarianat verändert
class Pair[A](x: A, y: A) extends WithMap[A] {
  override def map[B](f: A => B): Pair[B] =
    Pair(f(x), f(y))
}

// OK --aber nicht wirklich, nur für den Compiler
class Triple[A](a: A, b: A, c: A) extends WithMap[A] {
  override def map[B](f: A => B): Pair[B] =
    Pair(f(a), f(b)) // Pfui: falsches aber vom Compiler akzeptiertes map
}
```

Die Typprüfung findet dieses "falsche" map nicht. Aber was ist überhaupt falsch daran, die geforderte Signatur wird eingehalten. Das Falsche ist der Verstoß gegen unsere *Intuition*, dass map die *gleiche* Struktur mit verändertem Inhalt liefern muss. Mit objektorientierten Mitteln kann diese intuitive Forderung nicht im Typsystem ausgedrückt und damit vom Compiler prüfbar gemacht werden.

Funktionale Sicht: Typen mit einer map-Funktion

Ersetzen wir die objektorientierte Interface-Sicht durch ihre funktionale Entsprechung, dann wird aus dem *Interface* WithMap, das Objekte beschreibt, die *Typklasse* WithMap, also die Klassen der Typen mit einer map-Funktion:

```
// Typklasse
trait WithMap[F[_]] { // "Etwas (ein Typ) mit map" als Typklasse
  def map[A, B](fa: F[A], f: A => B): F[B]
}

case class Pair[A](x: A, y: A)

case class Triple[A](x: A, y: A, z: A)

given WithMap[Pair] with {
  def map[A, B](fa: Pair[A], f: A => B): Pair[B] = fa match {
    case Pair(x, y) => Pair(f(x), f(y))
  }
}

given WithMap[Triple] with {
  def map[A, B](fa: Triple[A], f: A => B): Triple[B] = fa match {
    case Triple(x, y, z) => Triple(f(x), f(y), f(z))
  }
}
```

Hier, bei der Definition der *Typklasse*, ist genau wie weiter oben, bei der Definition des *Interface*, WithMap ein Trait. Die unterschiedlichen Bedeutungen, Typklasse oder Interface, erkannt man bei genauerem Hinsehen.

Will man jetzt "falsche" unintuitive map-Funktionen definieren, dann schreitet der Compiler ein:

```
given WithMap[Triple] with { // Typfehler!
  def map[A, B](fa: Triple[A], f: A => B): Pair[B] = fa match {
    case Triple(x, y, z) => Pair(f(x), f(y))
    // error overriding method map
    // Pair[B] has incompatible type
    }
}
```

4.1.2 Funktor-Gesetze

Map gehört zu einem Typkonstruktor

Die map-Funktionen hebt eine Funktion f von der Ebene der Typen auf die Ebene der generischen Typen. (Siehe Abb. 4.2) Sie ist darum eng mit dem Typkonstruktor F verbunden, in dessen Bereich sie eine Funktion hebt. F bestimmt wie *map* arbeitet. Um genau zu sein, müsste man stets map_F schreiben, statt einfach nur *map*, denn jedes F hat sein eigenes *map*.

F ist eine Abbildung auf der Ebene der Typen, ein Typkonstruktor. Es nimmt einen Typ A und macht daraus den Typ $F[A]$. *map* dagegen nimmt einen Wert vom Typ A => B und macht daraus einen Wert vom Typ $F[A]$ => $F[B]$. (Siehe Abb. 4.3)

Ist F in der Typklasse *Mapable*, dann gehört zu ihm eine Funktion map_F. Dieses map_F lässt

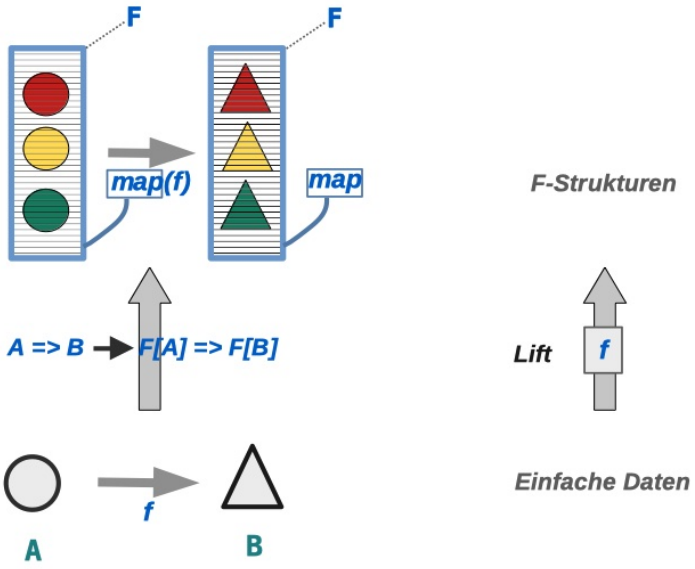

Abbildung 4.2: Map: Ein Aufzug nach F.

sich ziemlich trivial bestimmen (Siehe Abb. 4.4):

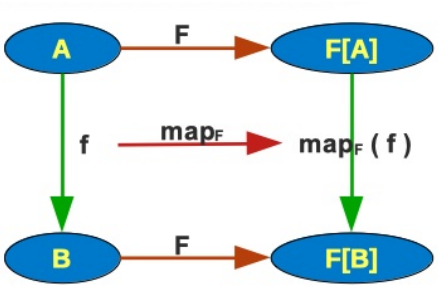

Abbildung 4.3: F und map_F sind Abbildungen auf unterschiedlichen Ebenen.

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

// map_F: map für ein F, das Mapable ist.
// ?=> : Verwendung einer Kontext-Funktion.
// Die im Kontext verfügbare Instanz von der Typklasse WithMap wird
    verwendet
def map_F[F[_]: WithMap]:
     WithMap[F] ?=> [A, B] => (A => B) => F[A] => F[B] =
  [A, B] => (f: A => B) => (fa : F[A]) => fa.map(f)
```

Listen können “gemapt” werden:

```
given WithMap[List] with {
  extension[A, B] (fa: List[A]) def map(f: A => B): List[B] = fa.map(f)
}
```

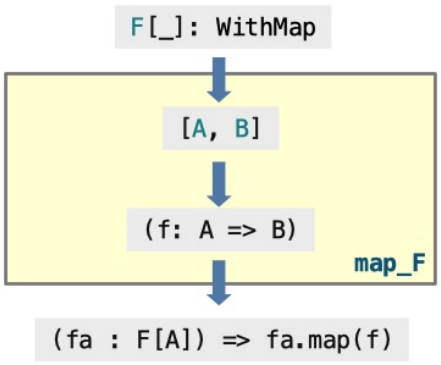

Abbildung 4.4: map_F aus einem F mit map konstruieren: Keine Kunst.

Also kann man ein passendes map_{List} konstruieren:

```
val mapList = map_F[List]
```

und mit ihm "mapen":

```
val lst_0 = List(1,2,3)
val lst_1 = mapList( (x:Int) => 2*x )(lst_0) // List(2, 4, 6)
```

Funktor: Typkonstruktor mit vernünftig definiertem Map

Wenn eine Funktion oder Methode den Namen "map" trägt, dann erwartet man ein bestimmtes Verhalten. Wer `f` kennt, hat eine Vorstellung von der Wirkung von `map(f)`. Dieser Vorstellung sollte die Implementierung `map(f)` dann auch erfüllen. Software sollte sich erwartungsgemäß verhalten. Vor allem dann, wenn es sich um Erwartungen von Anwendungs-Programmierern an Bibliothekscode handelt, den dieser meist nicht liest und wenn doch, dann eventuell nicht versteht.

> Einen Typkonstruktor F mit einem Typargument A, der für all seine Exemplare $F[A]$ ein vernünftig definiertes map bietet, nennt man *Funktor*.

Die Intuition sagt, dass ein "mapen" mit der identische Funktion keine Wirkung haben sollte:

$map(id_A) = id_{F[A]}$.

Außerdem sollte map über die Funktionsverkettung distribuieren, das heißt es sollte egal sein, ob zuerst mit f und dann mit g "gemapt" wird oder gleich mit $g \circ f$ (Siehe Abb. 4.5)

$map(g) \circ map(f) = map(g \circ f)$.

Mit einen kleinen Testfunktionen kann die Einhaltung der Gesetze geprüft werden:

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

def checkId[A, F[_]: Functor](fa: F[A]): Boolean = {
```

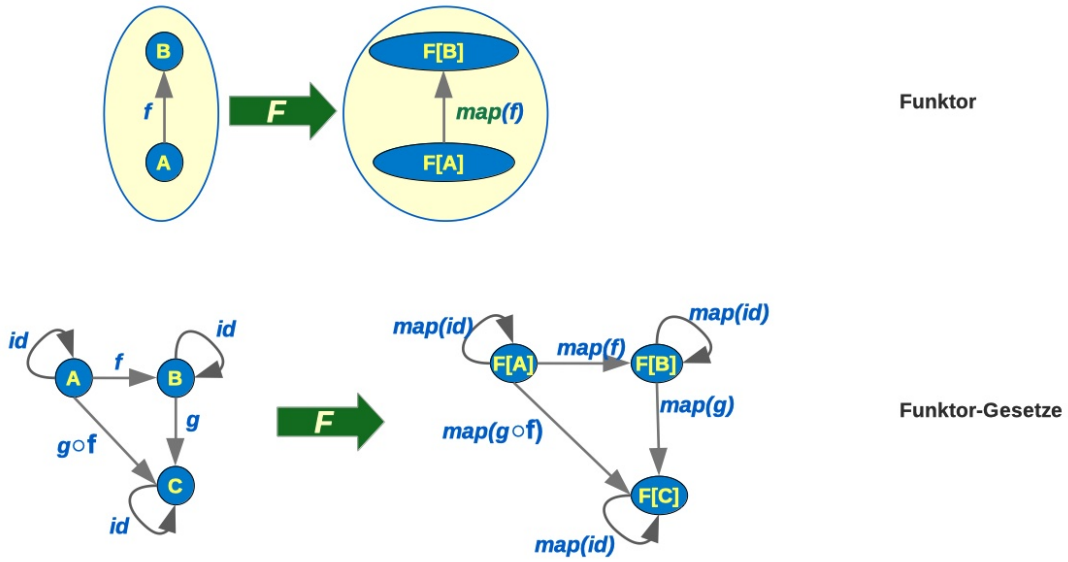

Abbildung 4.5: *map* und F in legaler Kooperation.

```
  def id: A => A = a => a

  fa.map(id) == fa
}

def checkDistr[A, B, C, F[_]: Functor](fa: F[A], f: A => B, g: B => C):
    Boolean =
  fa.map(f andThen g) ==
    fa.map(f).map(g) // =
    // ( {(fa:F[A]) => fa.map(f)} andThen {(fb:F[B]) => fb.map(g)} ) (fa)

case class Triple[A](a: A, b: A, c: A)

given TripleFunctor as Functor[Triple] with {
  extension[A, B] (fa: Triple[A]) def map(f: A => B): Triple[B] =
    Triple(f(fa.a), f(fa.b), f(fa.c))
}

val t: Triple[String] = Triple("Hallo", "Welt", "!")
val testId = checkId(t) // true

val f: String => Int = { str => str.length }
val g: Int => String = { i => s"[${i.toString}]" }

val testDistr = checkDistr(t, f, g) // true
```

Mit einem solchen Test lässt sich natürlich nichts Positives beweisen. Schlägt er aber fehl, dann ist klar, dass map "falsch" implementiert wurde. Ein Beispiel für einen offensichtlich "falsch" implementierten Funktor ist:

```
given Functor[Triple] with { // FALSCH
  extension[A, B] (fa: Triple[A]) def map(f: A => B): Triple[B] =
    Triple(f(fa.c), f(fa.b), f(fa.a))
```

```
}
```

Generische Containertypen sind von Natur aus Funktoren

Alle generischen Containertypen sind Funktoren. Beispielsweise List:

```
given Functor[List] with {
  extension[A, B] (fa: List[A]) def map(f: A => B): List[B] =
    fa.map(f)
}

val lstString: List[String] = List("ABC","die", "Katze", "lief", "im",
    "Schnee")
val lstInt: List[Int] = lstString.map( (s:String) => s.length) //
    List(3, 3, 5, 4, 2, 6)
```

Die Treue gegenüber den Funktor-Gesetzen wird als gegeben angenommen. Wir wären schon sehr überrascht, wenn nicht gar erbost, wenn eine Implementierung der Scala-API hier etwas anderes liefern würde. Mit Option verhält es sich natürlich genauso:

```
given Functor[Option] with {
  extension[A, B] (fa: Option[A]) def map(f: A => B): Option[B] =
    fa.map(f)
}

def length(os: Option[String]): Option[Int] =
  os map( (s:String) => s.length )

val optString: Option[String] = Some("ABC")
val optInt: Option[Int] = length(optString) // Sme(3)
```

Etwas weniger offensichtlich ist die Sache bei Either. Either hat zwei Typargumente, ein Funktor hat aber nur ein Typargument. In der Scala-API wirkt map nur auf die "rechte" Komponente. Damit ist Either dann ein Funktor in seinem rechten Typargument:

```
type Error = Throwable
type EitherThrowable[R] = Either[Error, R]

given Functor[EitherThrowable] with {
  extension[A, B] (fa: EitherThrowable[A]) def map(f: A => B):
      EitherThrowable[B] =
    fa.map(f)
}

val eitherString : EitherThrowable[String] = Right("12")
val eitherInt : EitherThrowable[Int] = eitherString map( (s:String) =>
   s.toInt )
```

Üblicherweise wird die linke Komponente von Either als Fehlerfall interpretiert. An diese Konvention hält sich auch Scala.

4.1.3 Was ist ein Funktor?

Nicht jeder Container-Typ kann ein Funktor sein

Ein Funktor muss zwingend generisch in genau einem Typargument sein. Das sieht man an der Definition:

```
trait Functor[F[_]] { ... }
```

F hat hier genau ein Typ-Argument. Generische Containertypen wie List oder Option sind nicht zufällig Funktoren. Da sie keinerlei Annahmen über ihren Inhalt machen, können mit den Inhalten auch nicht viel mehr machen, als sie unverändert abspeichern. Es darum leicht möglich eine gesetzestreue map-Funktion zu definieren. Natürlich kann man es noch falsch machen, wie bei unserem Triple-Beispiel:

```
given Functor[Triple] with { // FALSCH
  extension[A, B] (fa: Triple[A])
    def map(f: A => B): Triple[B] =
      Triple(f(fa.c), f(fa.b), f(fa.a))
}
```

Aber es ist *möglich* und *einfach* ein gesetzeskonformes *map* zu definieren.

Die Sache sieht jedoch ganz anders aus bei einem generischen Container wie etwa folgendem Typ der sortierten Listen:

```
class SortedList[A](val sortedAs: List[A]) {
  def toList: List[A] = sortedAs
}

object SortedList {
  def apply[A: Ordering](as: A*): SortedList[A] =
    new SortedList[A](as.toList.sorted) // die Elemente werden sortiert
        eingefügt
}
```

Dazu kann natürlich map-Funkion definiert werden. Beispielsweise als folgende Typklassen-Instanz:

```
  given Functor[SortedList] with { // FALSCH: distribuiert nicht
    extension[A, B] (fa: SortedList[A])
      def map(f: A => B): SortedList[B] =
        new SortedList( fa.sortedAs.map( (a: A) => f(a) ) )
  }
```

Dieses map ist sicher nicht Gesetzes-konform: Nur durch Zufall wird ein f distribuieren. Normalerweise wird es aber schon einen Unterschied machen, ob man zuerst f anwendet und dann die sortierte Liste bildet, oder eine sortierte Liste mit f transformiert:

```
def f(str: String): Int = str.toInt

val sList_1: SortedList[Int] = SortedList(f("200"),f("10"),f("7"))
  // ~> List(7, 10, 200)
val sList_2: SortedList[Int] = SortedList("200", "10", "7" ) map(f)
  // ~> List(10, 200, 7)
```

Future: Ein Funktor der kein Container-Typ ist

Nicht jeder Container-Typ ist ein Funktor. Um ein Funktor zu sein, muss man ein generischer Typ mit einem Parameter-Typ sein und man sollte möglichst nichts über den Parameter-Typ wissen. Wenn ich nicht weiß, dass er vergleichbar ist, dann kann ich den Inhalt nicht auf der Basis von Vergleichen organisieren – und diese Organisation dann durch ein map(f) verlieren.

Ein sehr wichtiger Funktor ist der generische Typ Future. Er hat einen Typ-Parameter, an den keinerlei Anforderungen gestellt werden, aber er ist definitiv kein Container-Typ.

```
import scala.concurrent.{Future, Await}
import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.duration._
import scala.util.{Success, Failure}

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

def factors(n: Long): List[Long] =
  if (n < 2) List() else
    Range.Long(2L, n/2+1, 1)
      .filter( (i: Long) => n%i == 0 && isPrime(i) ).toList

given Functor[Future] with {
  extension[A, B] (fa: Future[A]) def map(f: A => B): Future[B] =
    fa.map(f) // Future hat eine map-Methode
}

val futureFactors = Future {factors(125000001L)}

val futureResult: Future[String] =
  futureFactors map((l: List[Long]) => l.toString())

futureResult.onComplete {
  case Success(result) => println(result)
  case Failure(failure) => println("Failed because of " + failure)
}
```

Future ist mit einer map-Methode ausgestattet. Das macht es einfach Future zu einer Instanz der Typklasse Funktor zu deklarieren. Wie sieht es mit der Einhaltung der Funktor-Gesetze aus. Sie gelten:

- Die identische Funktion ändert nichts, auch dann nicht, wenn sie asynchron ausgeführt wird, und
- ob f und g jeweils hintereinander asynchron ausgeführt werden, oder $g \circ f$ insgesamt asynchron ausgeführt wird, das macht auch keinen Unterschied.

Der Typ Future ist kein Container-Typ, er speichert nichts, außer einem einmal berechneten Ergebnis, ist aber ein Funktor. Oder besser: kann als Funktor verstanden werden.

Funktion: Ein Funktor der kein Container-Typ ist

In einem Future wird ein einmal berechneter Wert für beliebig häufige Leseoperationen gespeichert. Es ist damit zumindest mit einem Container-Typ vage verwandt. Man kann aber auch

Typ-Konstruktoren als Funktoren auffassen, die ganz und gar nichts mit abgelegten Daten zu tun haben.

Der Typ $A \Rightarrow B$ mit fixiertem A ist ein Funktor der ganz und gar nichts mit einem Container zu tun hat. Ein Funktor hat *einen* Typparameter. $A \Rightarrow$ hat zwei, den Definitionsbereich A und den Wertebereich B.

Der Definitionsbereich der Funktionen wird darum fixiert, beispielsweise auf Int, und der Wertebereich wird zum Parameter. Das gibt dann den generischen Typ der Funktionen von Int nach irgendeinem anderen Typ:

```
type IntTo[B] = Int => B
```

IntTo ist ein Funktor mit der Funktionsverknüpfung als map (Siehe Abb. 4.6):

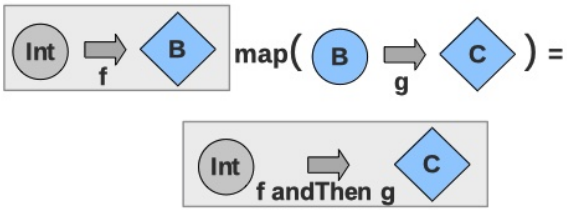

Abbildung 4.6: Funktionen sind ein Funktor in ihrem Ergebnistyp.

```
given Functor[IntTo] with {
  extension[B, C] (f: IntTo[B]) def map(g: B => C): IntTo[C] =
    f andThen g
}

val f: IntTo[Int] = i => i+1
val g: Int => String = i => s"<$i>"

val IncThenToString: IntTo[String] = f.map(g)
val v: String = IncThenToString(12) // <13>
```

Nicht jeder generische Typ ist ein Funktor

Funktionstypen, die generisch in ihrem Wertebereich sind, sind also Funktoren. Wie sieht es aus mit Funktionstypen, die generisch in ihrem Argumenttyp sind? Sind das auch Funktoren? (Siehe Abb. 4.7):

```
type ToIntFrom[A] = A => Int

given Functor[ToIntFrom] with {
  extension[B, A] (g: ToIntFrom[B]) def map(f: A => B): ToIntFrom[A] = ???
}
```

Diese Konstruktion ist unmöglich. Aus

- f, einer Funktion, die einen B-Wert konsumiert und daraus einen Int-Wert produziert und
- g, einer Funktion, die einen A-Wert konsumiert und daraus einen B-Wert produziert,

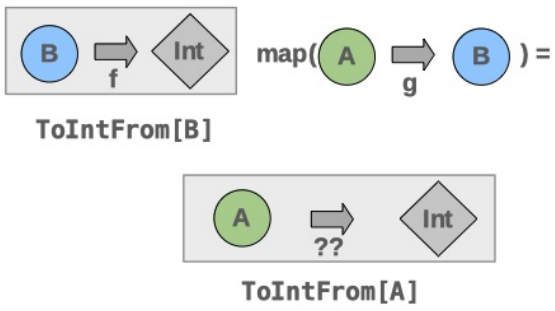

Abbildung 4.7: Sind Funktionen Funktoren in ihrem Argumenttyp?

kann man mit keiner Funktion, deren Signatur zu map passt, eine Funktion konstruieren, die aus einem A-Wert einen Int-Wert konstruiert.

Natürlich kann aus den beiden, f und g, eine Funktion $A \Rightarrow Int$ konstruieren werden. Aber nur, indem g *vor* f gesetzt wird. Die Signatur von map erlaubt aber kein "Davor-Setzen". Bei einem map-konformen "Dahinter-Setzen" muss ein a eines völlig unbekannten Typs A ge-/er-funden werden, auf das g angewendet wird. Das geht nicht, da wir dazu definitiv zu wenig über A wissen. (Siehe Abb. 4.8):

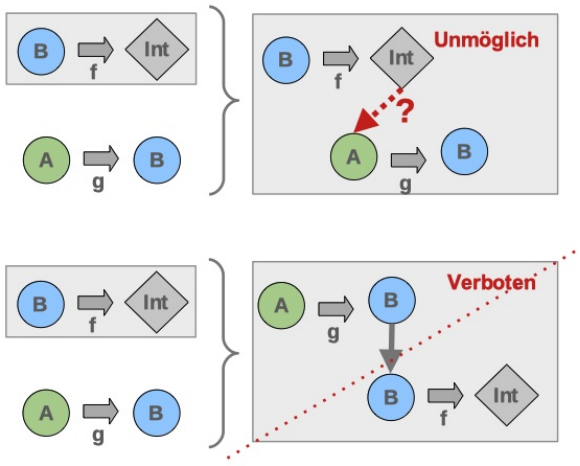

Abbildung 4.8: Die Unmöglichkeit ein Funktor zu sein.

Eine Funktion deren Argumente nur Typen haben, die mit Typ-Parametern beschrieben sind, wird *vollständig parametrisch* genannt. Eine vollständig parametrische Funktion kann nicht von bestimmten Werten abhängig sein. Die map-Funktion ist für jeden Funktor vollständig parametrisch, und damit unabhängig von irgendwelchen konkreten Werten.

Insgesamt halten wir für Funktoren fest:

Ein Typkonstruktor mit

- genau *einem, völlig unbeschränkten* Typ-Parameter,
- für den eine map-Funktion
  - *passender Signatur*, die

– die *Gesetze* beachtet,

definiert werden kann,

ist mit diesem map ein *Funktor*.

Es ist nicht immer klar, ob für einen gegebenen generischen Typ eine korrekte map-Funktion definiert werden kann und wenn ja, wie diese aussehen soll. Manchmal hat man keine Möglichkeit und gelegentlich auch mal mehr als eine, ein map zu definieren.

Funktoren in Einsatz

Funktoren sind Kontexte / Hüllen von Daten, die es erlauben verkettbare Operationen auf dem Inhalt auszuführen. Am deutlichsten sieht man das an den Container-Typen mit map. Beispielsweise:

```
val v =
  Some(scala.io.StdIn.readLine()) // Verarbeitungskette
    .map( str => toInt(str) )
    .map( i => even(i) )
```

In fast allen ernsthaften modernen Programmiersprachen gibt es in Form von *Funktor-Blöcken* (auch *For-Comprehension* oder *For-Ausdrücke*) angenehmen "syntaktischen Zucker" mit dem die Verarbeitungsketten intuitiv und elegant ausgedrückt werden können.

```
val list = List("one", "two", "three", "four", "five")

val result_1: List[Boolean] = // Verkettung mit map
  list
    .map( str => str.length)
    .map( i => i % 2)
    .map( j => j == 0)

val result_2: List[Boolean] = // entsprechender Funktor-Block
  for (str <- list;
       i = str.length;
       j = i % 2)
    yield j == 0
```

Man beachte die Benennung der Zwischenergebnisse in den beiden Varianten und die Verwendung von Pfeilen und Gleichheitszeichen in der For-Variante.

Funktorblöcke sind vor allem dann eine Erleichterung, wenn Zwischenergebnisse nicht nur in der nächsten, sondern auch in weiteren Verarbeitungsstufen verwendet werden sollen. Bei der "rohen" Anwendung von map müssen alle Zwischenergebnisse mühsam mitgeschleppt werden:

```
val result =
  list
    .map( str => (str, str.length))
    .map { case (str, i) => (str, i) }
    .map { case (str, i) => (str, i, i%2) }
    .map { case (str, i, j) => (
       str,
       i,
       if (j == 0) "even length" else "odd length" )}
    .map { case (str, i, e) => (s"'$str' has $e ($i)") }
```

Bei einem Funktorblock übernimmt der Compiler diese lästige Sache:

```
val result =
  for (str <- list;
       i = str.length;
       j = i % 2;
       e = if (j == 0) "even" else "odd"
    ) yield (s"'$str' has $e length ($i)")
```

Der Begriff "Inhalt" kann auf beliebige *Kontexte* von Daten verallgemeinert werden. Damit kommen wir zur praktischen Bedeutung von Funktoren:

> Der Begriff *Funktor* bringt
>
> - das Konzept eines Daten-*Kontextes* zum Ausdruck, der
> - die *Verkettung*
> - *beliebiger* Operationen
>
> auf seinem Inhalt zulässt – weil er praktisch nichts über das Wesen seines Inhalts weiß.

Nichts weniger aber vor allem auch auch nicht mehr. Es ist sinnlos nach irgendwelchen weiteren Eigenschaften oder tieferen Bedeutungen eines Funktors zu suchen.

4.1.4 Contra-Funktoren

Funktoren sind *covariant*, die Pfeilrichtung überträgt sich: Wenn f eine Funktion von A nach B ist und F ein Funktor, dann ist map(f) eine Funktion von F[A] nach F[B|.

$f : A \to B \Rightarrow map(f) : F[A] \to F[B]$

Bei einem *contra-varianten Funktor*, oder kurz *Contra-Funktor* kehrt sich die Pfeilrichtung um. Wenn f eine Funktion von A nach B ist und F ein Contra-Funktor, dann ist contraMap(f) eine Funktion von F[B] nach F[A|. (Siehe Abb. 4.9.):

$f : A \to B \Rightarrow contraMap(f) : F[A] \leftarrow F[B]$

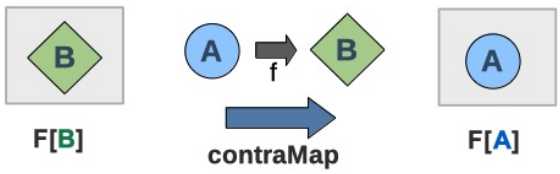

Abbildung 4.9: Contra-Funktor.

Die Typklasse der Contra-Funktoren bringt das zum Ausdruck:

```
trait ContraFunctor[F[_]] {
  extension[A, B] (fb: F[B])
    def contraMap(f: A => B): F[B] => F[A]
}
```

Oben haben wir gesehen, dass man Funktionen durch eine nachgeschaltete weitere Funktion "nach hinten verlängern" kann. Sie sind damit Funktoren mit dem "nach hinten verlängern" als map. Man kann aber auch eine Funktion durch Vorschalten einer anderen Funktion "nach vorn erweitern". Das ist keine Funktor-Eigenschaft: eine entsprechendes map kann nicht definiert werden. Es ist eine Contra-Funktor-Eigenschaft, die mit einem contraMap ausgedrückt werden kann. (Siehe Abb. 4.10.)

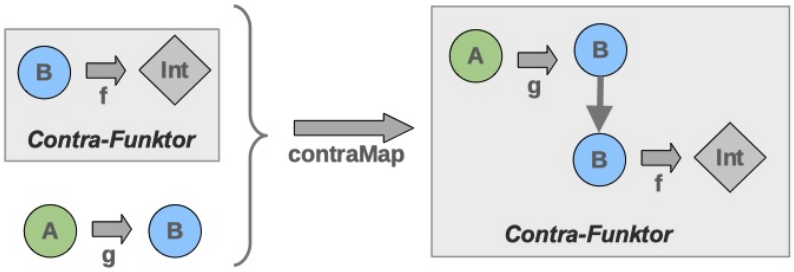

Abbildung 4.10: Funktionen als Contra-Funktoren.

Vergleichbarkeit: ein Contra-Funktor

Als Beispiel für einen Contra-Funktor betrachten wir die Vergleichbarkeit Comparable. Wenn ein Typ vergleichbar ist, dann kann man auch Werte von Typen vergleichen, die durch irgendeine Funktion auf den vergleichbaren Typ abgebildet werden können. Eine solche Funktion, die auf einen vergleichbaren Typ abbildet, nennt man meist "messen", oder auch "wiegen", sie führt von einem Typ, dessen Werte nicht *per se* vergleichbar sind, zu einem vergleichbaren Typ. In der Regel sind das numerische Werte. Die Vergleichbarkeit wird damit "nach vorn erweitert". (Siehe Abb. 4.11.)

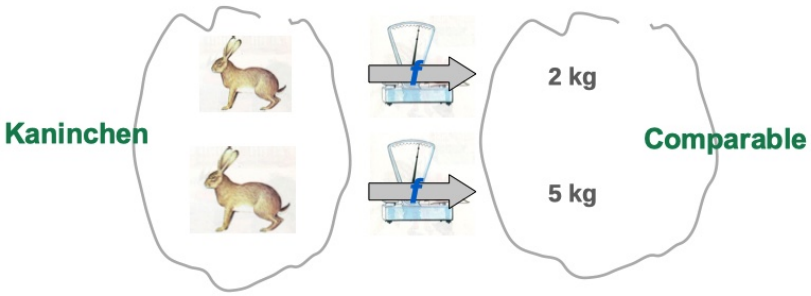

Abbildung 4.11: Wiegen: Vergleichbarkeit kontravariant erweitern.

Das Überraschende an der Umsetzung in Programmcode ist bestenfalls der Begriff "Contra-Funktor":

```
trait ContraFunctor[F[_]] {
  def contraMap[A, B](fb: F[B], f: A => B): F[A]
}

trait Comparable[B] {
  def compare(b1: B, b2: B): Int
```

```
}

// Comparable als ContraFunctor
// mache mit f: A => B aus einem Comparable[B] ein Comparable[A]
given ContraFunctor[Comparable] with {
  def contraMap[A, B](fb: Comparable[B], f: A => B) : Comparable[A] =
    new Comparable[A] {
      def compare(a1: A, a2: A): Int = fb.compare(f(a1), f(a2))
    }
}
```

Damit kann man beispielsweise einen Sortieralgorithmus so erweitern, dass er auch Dinge sortieren kann, die in etwas Vergleichbares konvertiert werden können:

```
def insertionSort[B: Comparable, A](lst: List[A])(using f: A => B):
    List[A] = {

  def insert(a: List[A], v: A): List[A] = a match {
    case Nil =>
      v :: Nil
    case x :: rest =>
      if ( summon[ContraFunctor[Comparable]]
          .contraMap(
            summon[Comparable[B]],
            f)
          .compare(v, x) < 0 ) {
        v :: a
      } else {
        x :: insert(rest, v)
      }
  }

  lst match {
    case Nil => Nil
    case head :: tail =>
      insert(insertionSort(tail), head)
  }
}
```

Damit können dann Kaninchen nach Gewicht sortiert werden:

```
case class Rabbit(name: String, weight: Int)

// Int ist in der Typklasse Comparable
given Comparable[Int] with {
  def compare(x: Int, y: Int) = x -y
}

// Konversion als implizite Funktion
given Conversion[Rabbit, Int] with {
  def apply(rabbit: Rabbit): Int = rabbit.weight
}

val alex = Rabbit("Alex", 77)
val bert = Rabbit("Bert", 72)
val claus = Rabbit("Claus", 96)
val dominik = Rabbit("Dominik", 111)
```

```
val emil = Rabbit("Emil", 96)
val flo =  Rabbit("Flo", 122)
val gerd = Rabbit("Gerd", 75)

val rabbits = List(alex, bert, claus, dominik, emil, flo, gerd)
val rabbitsSorted = insertionSort(rabbits)
```

"Funktor-artige" generische *Klassen* wie etwas die Klasse List sind mit einer map-Methode ausgestattet. Die generische *Klasse* Comparable ist "Contra-Funktor-artig", wir können sie nach dem Vorbild der "Funktor-artigen" gleich mit einer Methode contraMap-Methode ausstatten:

```
trait ContraFunctor[F[_]] {
  extension[A, B] (fb: F[B])
    def contraMap(f: A => B): F[A]
}

trait Comparable[B] { self =>
  def compare(b1: B, b2: B): Int
  def contraMap[A](f: A => B): Comparable[A] =
    new Comparable[A] {
      def compare(a1: A, a2: A): Int = self.compare(f(a1), f(a2))
    }
}

given ContraFunctor[Comparable] with {
  extension[A, B] (fb: Comparable[B])
    def contraMap(f: A => B) : Comparable[A] = fb.contraMap(f)
}
```

Damit lässt sich dann wieder InsertionSort definieren:

```
def insertionSort[B : Comparable, A](lst: List[A])(using f: A => B):
    List[A] = {

  def insert(a: List[A], v: A): List[A] = a match {
    case Nil =>
      v :: Nil
    case x :: rest =>
      if ( ( summon[Comparable[B]].contraMap(f).compare(v, x) < 0) ) {
        v :: a
      } else {
        x :: insert(rest, v)
      }
  }

  lst match {
    case Nil => Nil
    case head :: tail =>
      insert(insertionSort(tail), head)
  }
}
```

4.2 Filterbare Funktoren und Natürliche Transformation

4.2.1 Filterbare Funktoren

Filter sind eng mit Funktoren verwandt. Funktoren sind generische Typen mit einer assoziierten map-Funktion. *Filterbare Funktoren* haben zusätzlich noch eine filter-Funktion. Beide gehen über den Inhalt ohne Rücksicht auf den Kontext der Verpackung.

- map: Wende eine Funktion auf den Inhalt eines Kontexts an, erzeuge so einen gleichartigen Kontext mit *transformiertem* Inhalt.
- filter: Wende eine Bewertungs-Funktion auf den Inhalt einer Kontextes an, erzeuge so einen gleichartigen Kontext mit eventuell *weniger* Inhalt. (Siehe Abb. 4.12)

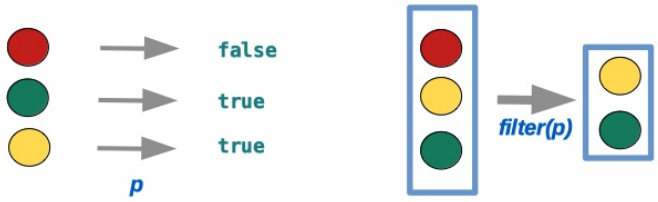

Abbildung 4.12: Filter: Verpacktes eventuell aussortieren.

Filterbare Funktoren sind ein Standardmittel der funktionalen Programmierung und in vielen Sprachen bekannt und üblicherweise in die Syntax von Funktor-Blöcken *(For-Comprehension)* integriert:

```
val lst_1 = List(0, 1, 2).filter( i => i % 2 == 0)

val lst_2 = // äquivalent
  for (
    i <- List(0, 1, 2);
    if i % 2 == 0
  ) yield i
```

In Scala erwartet der Compiler eine withFilter-Methode, wenn er einen Funktor-Block mit if übersetzt:

```
// Eine Box als filterbarer Funktor
enum MyBox[+A] {
  case EmptyBox      extends MyBox[Nothing]
  case FilledBox[A](a: A) extends MyBox[A]

  def withFilter(p: A => Boolean): MyBox[A] = this match {
    case EmptyBox => EmptyBox
    case FilledBox(a) =>
      if (p(a)) FilledBox(a) else EmptyBox
  }

  def map[B](f: A=>B): MyBox[B] = this match {
    case EmptyBox => EmptyBox
    case FilledBox(a) => FilledBox(f(a))
```

```
  }
}
import MyBox._

val filteredBox = // = FilledBox(8)
  for ( x <- FilledBox(4);
        if x % 2 == 0)
    yield 2*x
```

Was ist Filterbar

Filterbare Funktoren sind zunächst einmal Funktoren, also generische Typen mit einem einzigen unbeschränkten Typ-Parameter. Filtern ist wie "Mappen" eine Manipulation von Verpacktem, ohne "die Verpackung öffnen zu müssen". Man könnte damit den Eindruck bekommen, dass jeder Funktor filterbar ist.

Dem ist aber nicht so. Filtern kann die Größe einer Struktur verändern. Strukturen mit fester Größe können darum nicht filterbar sein. Ein Tripel

```
case class Triple[A](x: A, y: A, z: A)
```

ist ein Funktor: eine gesetzeskonforme map-Funktion auf Tripeln kann problemlos definiert werden:

```
given Functor[Triple] with {
  extension[A, B] (fa: Triple[A]) {
    def map(f: A => B): Triple[B] = Triple(f(fa.x), f(fa.y), f(fa.z))
  }
}
```

Es kann aber offensichtlich nicht gefiltert werden. Es gibt drei Elemente, davon kann ich nicht einfach mal eines heraus filtern. Filtern basiert auf der Intuition, etwas auf Basis einer Bewertung zu *verkleinern*. Manches kann, anderes kann definitiv nicht verkleinert werden.

Bei anderem kann man sich fragen, was genau eine Verkleinerung ist. Ob eine Verkleinerung akzeptabel, also konform zu den Gesetzen der Vernunft ist und den üblichen Erwartungen entspricht, die an Aktionen mit diesen Namen gestellt werden.

Bei einigen Typkonstrukoren, wie List und Option, oder MyBox oben, ist eine Verkleinerung möglich, bei anderen wie dem Tripel dagegen nicht.

Verpackt man die Tripel-Komponenten in Option:

```
case class TripleOpt[A](x: Option[A], y: Option[A], z: Option[A])
```

dann sieht die Sache schon besser aus. Ein Some(a), mit einem nicht passenden a, kann ja einfach durch None ersetzt werden:

```
given FilterableFunctor[Triple] with {
  extension[A, B] (fa: Triple[A]) {
    def map(f: A => B): Triple[B] =
      Triple(fa.x.map(f), fa.y.map(f), fa.z.map(f))
  }
  extension[A, B] (fa: Triple[A]) {
    def filter(f: A => Boolean): Triple[A] =
      Triple(fa.x.filter(f), fa.y.filter(f), fa.z.filter(f))
```

```
    }
  }
```

Wenn Filtern bedeutet, etwas von gleicher Struktur, aber mit eventuell weniger Inhalt zu produzieren, dann muss der Typ der Struktur Exemplare unterschiedlicher Größe haben. Für einen solchen Typ gibt es im Prinzip nur eine Möglichkeit: Es muss ein *Summentyp* sein. List und Option sind Summentypen:

- Option[A] $\sim Unit + A$
- List[A] $\sim Unit + (A \times Unit) + (A \times (A \times Unit)) + \cdots$

Bei TripleOpt wird durch Filtern nicht die Zahl der Elemente verringert, die Zahl *der definierten* Elemente wird verringert.

- TripleOpt[A] $\sim (A + Unit) \times (A + Unit) \times (A + Unit)$

Darf man so etwas auch "Filtern" nennen? Klar! Die Intuition sagt ja. Betrachten wir das mal etwas genauer.

Filtergesetze

Natürlich ist TripleOpt filterbar. Die definierte filter-Funktion erfüllt ganz sicher unsere intuitiven Erwartungen an etwas, das sich "filtern" nennt. Können diese "intuitiven Erwartungen" aber auch etwas präziser ausgedrückt werden? Was genau erwarten wir also intuitiv?

Klar ist schon mal:

> *Gesetz der Identität:*
> Filtern mit einem Prädikat, das stets true liefert, sollte ohne Wirkung sein:
>
> $$fa\ filter\ (x \to x) \equiv fa$$
>
> *Gesetz der Komposition:*
> Filtern mit p und dann mit q, sollte das das Gleiche sein, wie Filtern mit $p \wedge q$:
>
> $$fa\ filter\ (p)\ filter(q) \equiv fa\ filter(p \wedge q)$$

Etwas, das was gefiltert werden kann, kann stets auch "gemapt" werden. Sollten die beiden, filter und map, darum in irgendeiner Form harmonieren? Kann man also beim Entwurf eines filterbaren Funktors gegen intuitive Erwartungen des Nutzers verstoßen, weil filter und map nicht erwartungsgemäß harmonieren? Klar! Schauen wir das mal genauer an:

Hinter einem Filter kann mit einer *partiellen* Funktion gemapt werden, die auf herausgefilterten Elementen nicht definiert ist (Siehe Abb. 4.13):

> *Gesetz der partiellen Funktionen:*
>
> $$filter(p); map(f) \equiv filter(p); map(f^P)$$

Im Beispiel:

```
def checkPartialLaw[
  F[_]: FilterableFunctor,
  A,
```

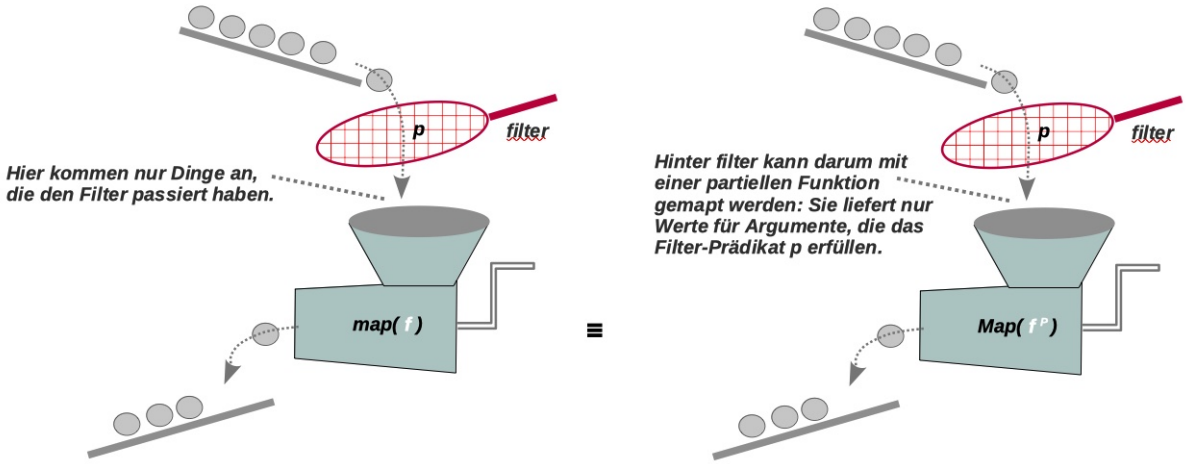

Abbildung 4.13: Gesetz der partiellen Funktionen.

```
  B](fa: F[A],
     p: A => Boolean,
     f: A =>B): Boolean = {

   // f mit von p beschränktem Definitionsbereich
   def fp(f: A =>B ) = (a: A) => p(a) match {
    case true => f(a)
   }

   fa.filter(p).map(f) == fa.filter(p).map(fp(f))
 }

 val triple: TripleOpt[String] = TripleOpt(Some("1"), None, Some("three"))

 def f(s: String): Int = s.toInt

 def p(s: String): Boolean =
   s.toIntOption match {
     case None => false
     case _ => true
   }

 val checkTriple = checkPartialLaw(triple, p, f)
```

Das zweite Kooperationsgesetz (Siehe Abb. 4.14) für map und filter ist:

Gesetz der Natürlichkeit:

$$map(f); filter(p) \equiv filter(f; p); map(f)$$

Es sagt, dass es egal ist, ob man zuerst mit p filtert und dann mit f mapt, oder mit $p \circ f$ filtert und dann mit f mapt. Im Beispiel:

```
 def checkNturality[
   F[_]: FilterableFunctor,
   A,
   B](fa: F[A],
```

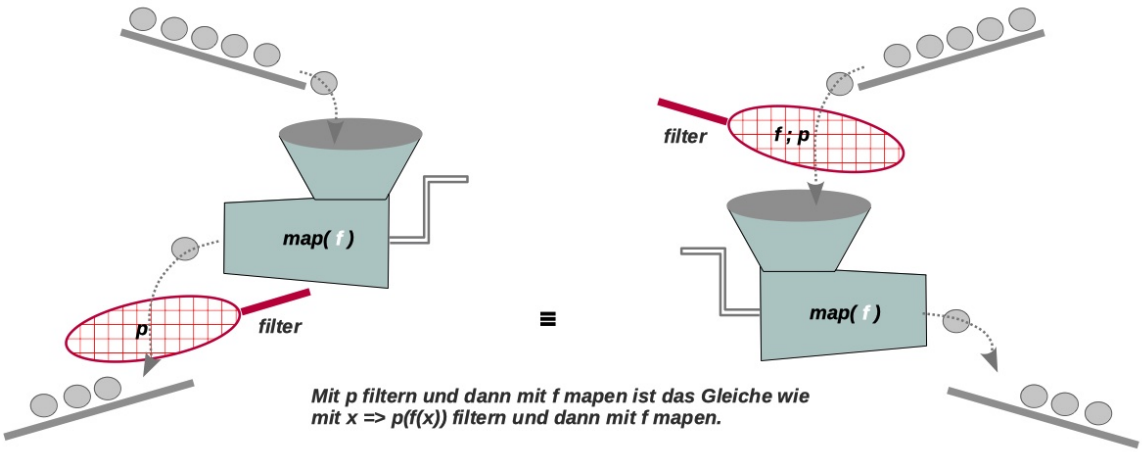

Abbildung 4.14: Gesetz der Natürlichkeit.

```
    p: B => Boolean,
    f: A =>B): Boolean = {
  fa.map(f).filter(p) == fa.filter( f andThen p).map(f)
}

val triple: TripleOpt[String] = TripleOpt(Some("abc"), None,
    Some("Katze"))

val f: String => Int = _.length

val p: Int => Boolean = _ > 4

val checkTriple = checkNturality(triple, p, f)
```

Der etwas seltsame Name "Gesetz der Natürlichkeit" kommt von der Struktur der Beziehungen, der beteiligten Typen und Funktionen. (Siehe Abb. 4.15.)

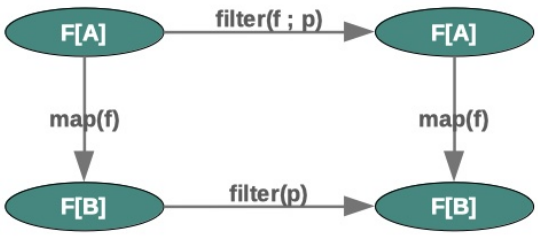

Abbildung 4.15: Die Natürlichkeit des Gesetzes der Natürlichkeit.

Die *Filter-Gesetze* noch einmal in der Übersicht:

$$
\begin{aligned}
&fa\ filter\ (x \to x) && \equiv fa \\
&fa\ filter\ (p)\ filter(q) && \equiv fa\ filter(p \wedge q) \\
&filter(p); map(f) && \equiv filter(p); map(f^P)
\end{aligned}
$$

$$map(f); filter(p) \equiv filter(f; p); map(f)$$

Die *Filtergesetze* beschreiben die Erwartungen, die ein Nutzer intuitiv an einen filterbaren Typ hat und die von einer entsprechenden Bibliothekskomponente *unbedingt beachtet* werden müssen.

Containertypen wie List oder Option sind "auf natürliche Art" filterbar. Ist etwas filterbar, dann ist es auch ein Funktor. Umgekehrt ist ein Funktor ist filterbar, wenn er Exemplare mit unterschiedlich vielen (definierten) Elementen zulässt. Dazu muss er ein *Summentyp* sein. Reine Produkttypen, wir Tupel, Tripel, etc., sind nicht filterbar.

4.2.2 Natürliche Transformationen

Daten kann man umorganisieren. Beispielsweise kann man alles was in einem Baum gespeichert ist, auch in eine Liste packen. Die Struktur der Daten wird dabei transformiert, ohne dass Daten verloren gehen oder neue hinzu kommen. Eine sortierte Liste kann man unter Erhaltung der Sortierung in einem sortierten Baum speichern. Das ist ein anderes Beispiel einer Struktur-Transformation: eine bei der der Inhalt eine Rolle spielt

Wenn die Umorganisation völlig unabhängig von den Daten ist, dann spricht man von einer *natürlichen Transformation*. Die erste Transformation – Baum in Liste – ist "natürlich", die zweite – sortierter Baum in sortierte Liste – ist "nicht natürlich". Der Begriff der "Natürlichkeit" bezieht sich, darauf, dass die Transformation auf naheliegende, natürliche Art, ohne Bezug auf irgendwelche besonderen Kriterien erfolgt. [1]

Strukturen sind Funktoren und eine *natürliche Transformation* transformiert einen Funktor F[A] in einen anderen Funktor G[A]. Das Typ-Argument A ist dabei ein völlig unbeschränkter Parameter, der bei der Transformation keine Rolle spielt. Die völlige Unabhängigkeit von den Daten, die "Natürlichkeit" der Transformation, kann über map definiert werden: Jeder Weg im Diagramm 4.16 ist äquivalent:

$$\tau_A(fa).map(f) = \tau_B(fa.map(f))$$

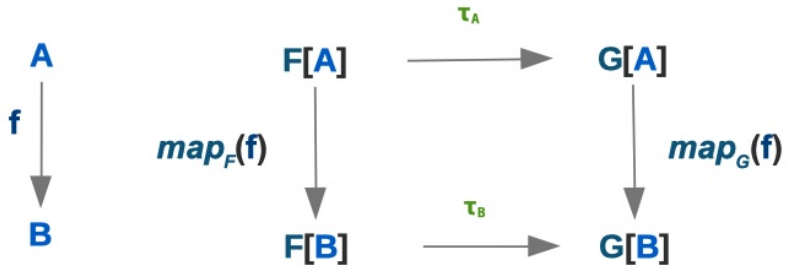

Abbildung 4.16: Narürliche Transformation τ.

Eine natürliche Transformation ist eine Typrelation, also eine Typklasse mit mehr als einer Typvariablen:

```
trait Functor[F[_]] {
```

[1] An dem Begriff der "Natürlichkeit" tiefer Interessierte mögen mathematische Quellen konsultieren, z.B. https://mathoverflow.net/questions/56938/what-does-the-adjective-natural-actually-mean/56956

```
  extension[A, B] (fa: F[A]) {
    def map(f: A => B): F[B]
  }
}

// natürliche Transformation als Typklasse
trait NT[F[_]: Functor, G[_]: Functor] {
  def tau[A](fa: F[A]): G[A]
}
```

Eine Transformation von Option nach List, bei der ein None zur leeren Liste und ein Some(a) zu List(a) wird, kann man mit Fug und Recht als "natürlich" bezeichnen. Man hat auch keine andere Wahl: Wird None zu einem List(a), dann muss zu einem komplett unbekannten Typ A irgendein a: A ge-/erfunden werden. Das ist unmöglich. Die Transformation von Option nach List als Instanz der Typrelation ist darum eindeutig bestimmt:

```
given NT[Option, List] with {
  def tau[A](o: Option[A]): List[A] = o match {
    case None => List()
    case Some(x) => List(x)
  }
}
```

Umgekehrt gibt schon mehrere Möglichkeiten eine Liste auf "natürliche" Art in ein Option zu transformieren. Auch wenn eine *natürliche* Transformation kein Wissen über den Typ der Komponenten verwenden darf. Eine leere Liste kann nur auf ein None abgebildet werden. Bei nichtleeren Listen können wir irgendein Element auswählen: Das erste:

```
given NT[List, Option] with {
  def tau[A](l: List[A]): Option[A] = l match {
    case Nil => None
    case x :: _ => Some(x)
  }
}
```

Das letzte:

```
given NT[List, Option] with {
  def tau[A](l: List[A]): Option[A] = l.lastOption
}
```

Oder irgendein anderes:

```
given NT[List, Option] with {
  def tau[A](l: List[A]): Option[A] = l match {
    case Nil => None
    case x :: Nil => Some(x)
    case x :: y :: tail => Some(y)
  }
}
```

Alles OK und (mehr oder weniger) natürlich. Es gibt also nicht unbedingt immer nur eine einzige Transformation zwischen zwei Funktoren, die natürlich ist.

Ein weiteres Beispiel für eine natürliche Transformation ist die Umwandlung einer beliebigen Struktur in einen Text. Die Struktur muss dazu zuerst in eine lineare Form, beispielsweise in

eine Listenform gebracht werden. Dann kann man die Listenelemente in Strings umwandeln und verketten. Die Struktur muss dazu auf natürliche Art in eine Liste transformiertbar sein. Diese Transformierbarkeit können wir einem Typ ausdrücken:

```
type NaturallyTransformableToList[F[_]] = NT[F,List]
```

Damit lässt sich eine Funktion definieren, die Strukturen in Text transformiert:

```
def format[
   T,
   F[_]:NaturallyTransformableToList
 ](
   structure: F[T]
  ) : String =
 s"[${summon[NaturallyTransformableToList[F]].tau(structure).mkString(",")}]"
```

Oder mit der Übergabe der Transformation mit einem impliziten Argument statt ihrer "Beschwörung" mit summon:

```
def format[
   T,
   F[_]](structure: F[T])
       (using nt : NT[F,List]) : String =
 s"[${nt.tau(structure).mkString(",")}]"
```

Damit sind dann Formatierungen wie folgende möglich:

```
val str_1: String = format(List("A", "B", "C")) // [A,B,C]
val str_2: String = format(Option("A")) // [A]
val str_3: String = format(List(format(Option("A")), format(None))) //
    [[A],[]]
```

Ein weiteres Beispiel für eine natürliche Transformation ist flatten, das Flachklopfen einer Liste:

```
type ListList[T] = List[List[T]]

given Functor[ListList] with {
  extension[A, B] (fa: ListList[A]) def map(f: A => B): ListList[B] =
    fa.map(_.map(f))
}

given NT[ListList, List] with {
  def tau[T](lstLst: ListList[T]): List[T] = lstLst.flatten
}

val Lst: ListList[Int] = List(List(42), List(43, 44), List(43, 44, 45))
val str = format(Lst) // [42,43,44,43,44,45]
```

Das Umkehren einer Liste ist eine natürliche Transformation von List nach List:

```
given NaturallyTransformableToList[List] with {
  def tau[T](lst: List[T]): List[T] = lst match {
    case Nil => Nil
    case h :: t => tau(t) ::: (h :: Nil)
  }
}
```

```
val lst: List[Int] = List(1,2,3,4,5)
val str = format(lst) // [5,4,3,2,1]
```

Man kann sich in all den Fällen davon überzeugen, das die Transformationen tatsächlich die formalen Anforderungen an die Natürlichkeit erfüllen und mit map in der gewünschten Form kooperieren. Fassen wir zum Schluss noch einmal das Wesentliche einer natürlichen Transformation zusammen:

> Eine *natürliche Transformation* τ ist eine rein generische Transformation einer rein generischen Struktur (Funktor) F in eine rein generischen Struktur (Funktor) G.
>
> So wie die "reine Generizität" eines Funktors durch die Funktor-Gesetze konkretisiert wird, so wird die "reine Generizität" der natürlichen Transformation durch das Kriterium der Natürlichkeit ausgedrückt:
>
> $\tau_A(fa).map(f) = \tau_B(fa.map(f))$

Kapitel 5

Monaden

Monaden sind ein weiteres Muster zur Organisation von Verarbeitungsschritten.

Ein *Funktor* bringt die Idee von verkettbaren transparenten Berechnungen in einem Kontext zum Ausdruck. Die zentrale Operation ist die map-Funktion, die in transparenter Art verkettbare Aktionen auf dem Inhalt anwendet.

Eine *Semi-Monade*[1] ist eine weiterentwickelter Funktor. Mit ihm werden verkettbare transparente Berechnungen in einem Kontext auf eine andere, erweiterte Art unterstützt: *Geschachtelte Iterationen* in einem Kontext können einfach und übersichtlich ausgedrückt werden. Die weitere zentrale Operation ist *flatMap*.

Eine *Monade* ist dann eine weiterentwickelte Semi-Monade. Monaden sind Semi-Monaden mit einer ("Konstruktor-") Operation pure.

Damit ist das Thema der Monaden aber nicht am Ende. Im Gegenteil, es beginnt erst. *Spezialisiere Monaden* sind Monaden mit zusätzlichen Fähigkeiten, beispielsweise zur Fehlerbehandlung, zur Verwaltung eines Zustands und so weiter. Diese speziellen Monaden werden von Bibliotheken wie etwa *Cats*[2] zur Verfügung gestellt und sind eine handhabbare Basis für die Anwendungsentwicklung.

5.1 Monaden

5.1.1 Listenartige Monaden: geschachtelte Iterationen

Mit map werden Funktionen auf einem Inhalts-Typ in den Kontext einer Struktur gehoben. Dagegen wird mit flatMap eine *Struktur-erzeugende* Funktion in den Kontext einer Struktur gehoben. (Siele Abb. 5.1.)

Damit lässt sich viel mehr machen, als es zunächst den Anschein hat. Die erste Anwendung von flatMap sind *geschachtelte Iterationen*. Es geht auch ohne flatMap, aber mit geht es einfacher und übersichtlicher. Betrachten wir dazu ein Beispiel: Die Erzeugung aller Kombinationen der Elemente von zwei Listen:

$$([a, b], [1, 2]) \;\Rightarrow\; [(a, 1), (a, 2), (b, 1), (b, 2)]$$

Ohne flatMap:

[1] Semi-Monaden werden oft auch "FlatMap" genannt.
[2] https://typelevel.org/cats/

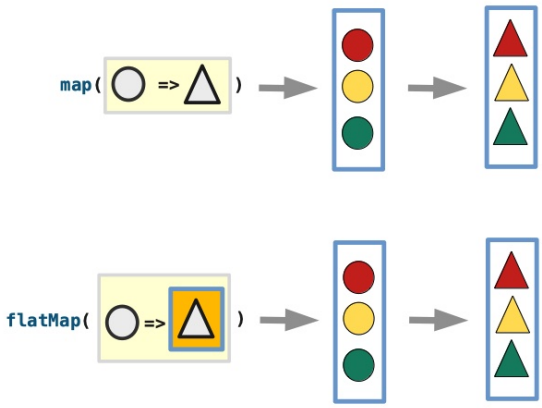

Abbildung 5.1: map und flatMap.

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

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

Und mit flatMap spart man sich das flatten:

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

Nicht verwunderlich, flatMap ist ja eine Kombination aus map und flatten. Die "inneren Iterationen" erzeugen die zusätzlichen Strukturen, die dann "flach geklopft" werden müssen. Je verschachtelter die Iterationen sind, um so mehr wird man flatMap schätzen. Mit *For-Comprehension* wird es noch übersichtlicher:

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

Ein weiteres Beispiel für flatMap in Aktion ist die Erzeugung aller 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)
   .map(_.mkString(""))
   .mkString(", ")
// -> "abc, bac, bca, acb, cab, cba"
```

Monade: Semi-Monade mit pure

Wie bereits erwähnt sind *Monaden Semi-Monaden* mit pure, also *Funktoren* mit einer flatMap- und einer pure-Funktion. (Siehe Abb. 5.2.)

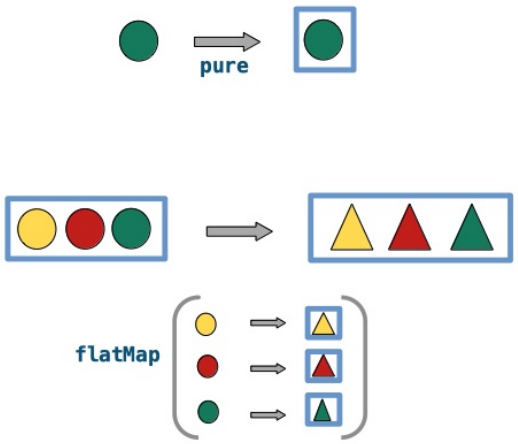

Abbildung 5.2: Monade: Funktor mit pure und flatMap.

Die Typklasse Monad repräsentiert die ‘monadischen Typen”:

```
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]]
}
```

Hier ist map über flatMap und pure definiert. (Siehe Abb. 5.3.)

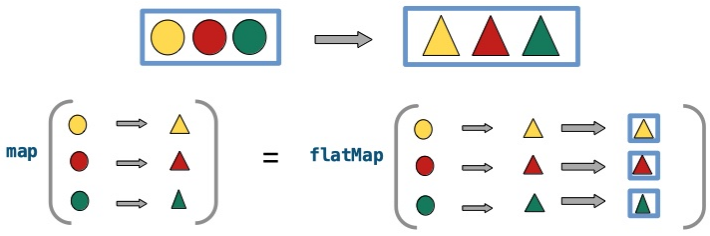

Abbildung 5.3: pure: pure plus flatMap.

In Scala – und nicht nur dort – sind alle sequenziellen Strukturen wie Seq, List, LazyList, Array, Vector, filterbare Monaden mit einem Konstruktor als pure-Funktion.

Geschachtelte Iterationen können generisch definiert werden:

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

Oder schöner:

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

Und jeder generische sequenzielle Typ kann dann die Rolle der Monade F übernehmen. Beispielsweise List:

```
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 von List, wir ignorieren das vordefinierte map
  }
}

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

Geschachtelte Iterationen auf listenartigen Strukturen sind aber nur eine der vielen Anwendungsmöglichkeiten des Entwurfsmusters “Monade”.

5.1.2 Fehlermanagement-Monade: Iterationen mit Fehlerbehandlung

Wenn es *unterschiedliche Arten der Verpackung* gibt, dann eröffnet flatMap eine weitere Dimension der Ausdrucksmöglichkeiten. Abhängig vom Argument kann es mal ein auf die eine Art verpacktes Ergebnis produzieren und ein anderes Mal eins, das auf die andere Art verpackt wurde.

Die eine naheliegende Anwendung dieser Idee ist die Fehlerbehandlung. Der monadische Typ muss dazu zumindest zwei Varianten haben. Eine ist die "gute", die einen Wert repräsentiert, die andere Variante, die "böse", repräsentiert einen Fehler. Wenn flatMap die "böse" Variante als "leer" betrachtet, weil sie tatsächlich leer ist und es darum nichts zu (flat-) mapen gibt, oder flatMap den vorhanden Wert ignoriert, dann können Fehler nach ihrem Auftreten einfach durchgereicht werden, ohne dass dazu der Code mit Prüfungen voll gepflastert werden muss. (Siehe Abb. 5.4.)

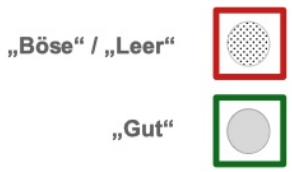

Abbildung 5.4: Monadischer Typ mit guter und böser Variante.

Viele Typen haben zwei Varianten, die man als "böse" bzw. "leer" oder "gut" bzw. "gefüllt" klassifizieren kann:

- List: Nil / ::
- Option: None / Some
- Either: Left / Right
- Try: Failure / Success
- ...

Das sind *Fehler(-Management)-Monaden* oder auch *Pass-/Failure*-Monaden. Die beiden Varianten des Kontexts geben flatMap (also jeder Iteration) die Möglichkeit entweder

- ein neues Element in einem "guten‘" Kontext zu erzeugen, oder
- einen "böse" / "leeren" Kontext zu erzeugen, der im weiteren ignoriert wird – flatMap und map haben ja nichts mehr zu mapen – ist ja leer (oder "leer")!

Das gibt dem Konzept "verkettbare Berechnung in einem Kontext" erweiterte Ausdrucksmöglichkeiten.

Ein Funktor kann als Fehlermangement-Monade agieren, wenn er zwei Varianten hat und mit einem flatMap ausgestattet werden kann, das eine der beiden Varianten als "leer" behandelt. *List* kann problemlos diese Rolle erfüllen. "Leer" ist dabei dann tatsächlich leer. (Siehe Abb. 5.5.)

Die idealtypische Besetzung der Rolle *Fehlermanagement-Monade* ist aber nicht List sondern Option. Eine Verarbeitungskette, bei der es auf jeder Stufe zu einem Fehler kommen kann, lässt sich einfach und elegant mit mit Option formulieren. Beispiel:

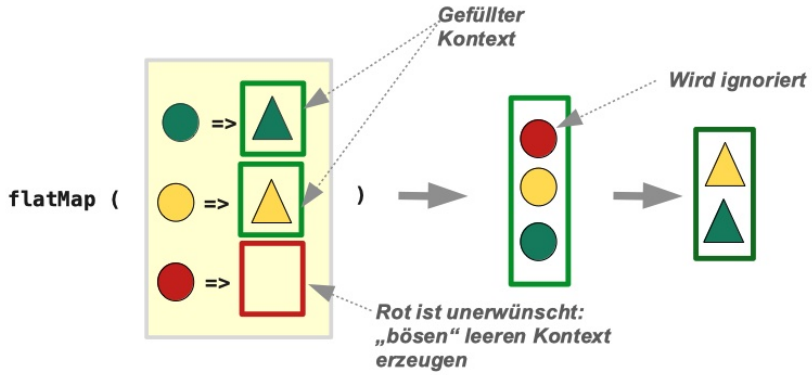

Abbildung 5.5: List als Fehlermangement-Monade.

```
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
    )
  }
}

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

Jede Stufe einer Iteration liefert ein optionales Ergebnis und alle Stufen können mit flatMap kombiniert werden, ohne dabei mühsam eine Fehlerbehandlung einstreuen zu müssen.

Ein weiteres Beispiel einer Fehlermonade ist Either. Die "leere" Variante ist hier im Gegensatz zu Option und List nicht tatsächlich leer. Sie enthält einen Fehlerwert. (Siehe Abb. 5.6.)

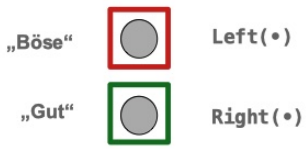

Abbildung 5.6: Either als Fehlermangement-Monade.

Als kleines Beispiel eine Ausdrucksauwertung, bei der eine Division durch Null abgefangen wird:

```
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)
}
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) => // etwas umständlich da Either nicht filterbar ist
    eval(l).flatMap((v1: Int) =>
      eval(r).flatMap((v2: Int) =>
        if (v2 != 0) Right(v1 / v2) else Left("Divide by zero")
      )
    )
}
```

Weitere bekannte Fehlermanagement-Monaden ist Try und Future, die beide ein Throwable als

Abbildung 5.7: Try und Future als Fehlermangement-Monaden.

Fehlerwert haben. (Siehe Abb. 5.7.) Auf entsprechende Beispiele verzichten wir hier.

Fehlermanagement-Monade: Verallgemeinerung

Eine Fehlermangement-Monade dient der Verkettung von Funktionen, die fehlschlagen können. Funktionen, die fehlschlagen können, sind partielle Funktionen. Eine *partielle Funktion*

$$f_p : A \Rightarrow B$$

ist eine Funktion, die auf einigen Elementen von A undefiniert ist. Eine solche partielle Funktion kann mit einer *totalen*, also überall definierten Funktion modelliert werden. (Siehe Abb. 5.8.) Aus einer partiellen Funktion

$$f_p : A \Rightarrow B$$

wird dazu eine totale Funktion:

$$f : A \Rightarrow F[B]$$

mit F als einer Monade mit zwei Varianten

$$F[\cdot] = F_{good}[\cdot] + F_{bad}[\cdot]$$

und einer totalen Funktion f mit

$$f(x) = \begin{cases} F_{good}(f(x)) & f \text{ auf x definiert} \\ F_{bad}(\cdot) & f \text{ auf x nicht definiert} \end{cases}$$

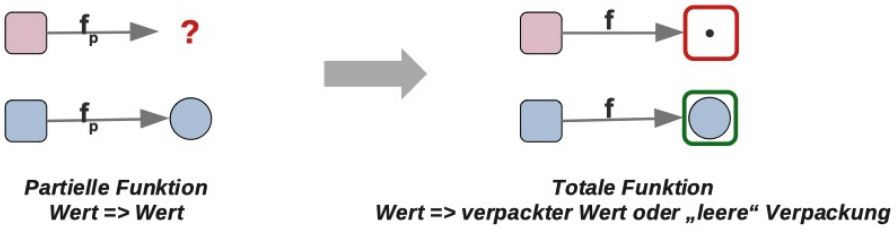

Abbildung 5.8: Fehlermangement-Monaden: Partielle zu totaler Funktion.

Eine Fehlermanagement-Monde hat mindestens folgende Besonderheiten:

- map und flatMap sind ohne Wirkung bei der “bösen” Variante.
- Als Komplement zu pure gibt es auch eine Möglichkeit die “böse” Variante zu erzeugen.

Folgende Definition bringt diese Minimalanforderungen zum Ausdruck:

```
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]]
}

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

Damit haben wir ein *generisches Fehlermanagement*, das es uns erlaubt Anwendungen mit einer "offenen", noch nicht spezifizierten Strategie der Fehlerbehandlung zu definieren. Beispielsweise:

```
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
    )
  }
}
```

Das konkrete Fehlermanagement kann dann nach Bedarf gewählt werden. Es ergibt sich aus verwendetet Instanz der Typklasse ErrorMonad. Wir könnten beispielsweise List als Instanz von ErrorMonad einsetzen:

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

Natürlich wäre Option oder Try näherliegender als List.

Cats bietet eine wesentlich ausgefeiltere Variante der Fehlermangement-Monade als Typklasse. Sie hat wie unere simple Variante ebenfalls den Namen MonadError.[3]

5.1.3 Monaden-Definitionen

Monaden sind ein Kernkonzept, das für unterschiedliche Anwendungen auf unterschiedliche Arten erweitert werden kann. Listenartige Monaden beschränken sich auf das Kernkonzept. Fehler(-Management)-Monaden sind ein Beispiel für Monaden, die darüber hinaus weitere Möglichkeiten bieten.

Es gibt aber nicht nur bei den Erweiterungen ein Vielzahl von Möglichkeiten. Das Kernkonzept kann schon auf unterschiedliche Arten definiert werden. Es zeigt sich aber, dass diese Varianten äquivalent sind. Die Unterschiede bestehen in der Grundausstattung an Funktionen, die unterschiedliche Art zusammengestellt werden können. Die *"monadischen Grundfunktionen"* sind:

- pure — auch: unit, return, lift, point, η (eta)
- flatMap — auch: bind, »=
- flatten — auch: multiplication, join, μ (mü)
- map — map kann mit pure und flatMap definiert werden

Diese Funktionen sind nicht gleichermaßen essentiell, da einige mit Hilfe anderer definiert werden können. So kann beispielsweise flatten mit flatMap und pure definiert werden. (Siehe Abb. 5.9.):

fa.flatMap(f) = fa.map(f).flatten.

Die Definition von map mit Hilfe von flatMap und pure haben wir schon weiter oben gesehen:

fa.map(f) = fa.flatMap(f.andThen(pure))

[3] Siehe https://typelevel.org/cats/api/cats/MonadError.html.

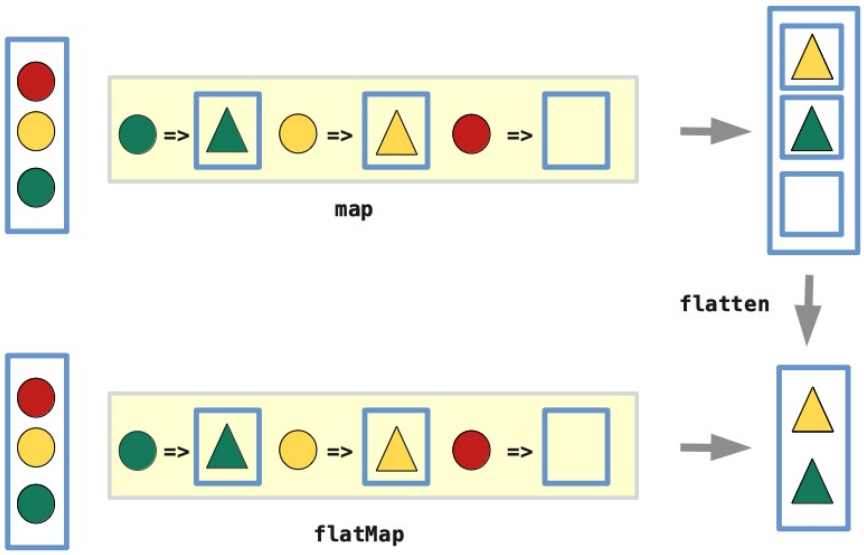

Abbildung 5.9: map, flatMap, und flatten.

Damit haben wir eine etwas erweiterte Definition der Typklasse Monad:

```
trait Monad1[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)
  }
}
```

Monaden können auch völlig anders definiert werden: Nämlich als Kombination eines Funktors mit zwei *natürlichen Transformationen*. In dieser Definitionsvariante besteht eine *Monade* aus

- einem Funktor F und
- zwei natürlichen Transformationen
  - $\eta : T \Rightarrow M[T]$ und
  - $\mu : M[M[\cdot]] \Rightarrow M[\cdot]$

(η entspricht pure und μ entspricht flatten). Die beiden natürlichen Transformationen sollen sich dabei in Kombination mit der map-Funktion des Funktors in vernünftiger und erwartbare Art vertragen. (Siehe Abb. 5.10.)

Als alternative Typklasse Monad haben wir dann:

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

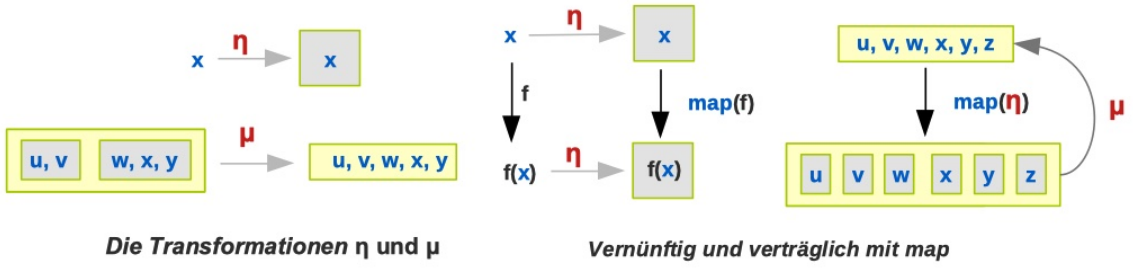

Abbildung 5.10: Monde als Funktor plus zwei natürliche Transformationen.

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

// Monade wird mit natürlichen Transformationen definiert
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]
}
```

Option als Monade ist damit:

```
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))
    }
}
```

Eine kleine Test-Anwendung wäre:

```
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")
```

5.1.4 Monaden-Gesetze

Die *Gesetze* garantieren, dass die Funktionen in erwartbarer Art zusammen arbeiten und eine Monade sich damit insgesamt in erwartbarer Art verhält. Das erste Gesetz das der *Links-Natürlichkeit* von flatMap. (Siehe Abb. 5.11.)

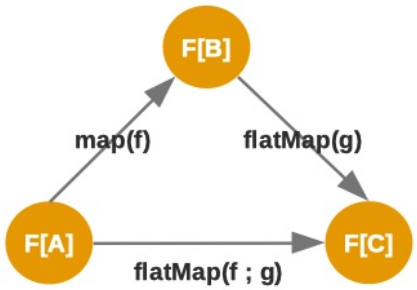

Abbildung 5.11: Die Links-Natürlichkeit von flatMap.

Es sagt dass

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

und

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

völlig äquivalent sind. – Sicherlich eine berechtigte Forderung.

Die *Rechts-Natürlichkeit* von flatMap (siehe Abb. 5.12) entspricht ebenfalls der intuitiven Erwartung an eine vernünftig definierte Monade. Es sagt dass

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

und

```
for (
     x <- m1;
```

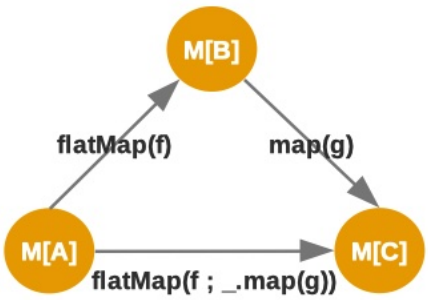

Abbildung 5.12: Die Rechts-Natürlichkeit von flatMap.

```
      y <- f(x).map(g) )
  yield y
// = m.flatMap(f andThen (m => m.map(g)) )
```

völlig äquivalent sind. – Oder es sein sollten.
Das dritte Gesetz verlangt die *Assoziativität von* flatMap (siehe Abb. 5.13)

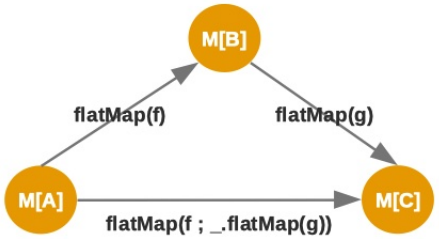

Abbildung 5.13: Assoziativität von flatMap.

Das heißt die Äquivalenz von

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

und

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

Natürlich gilt auch (siehe Abb. 5.14):

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

pure sollte *Rechts- und Links-Neutral* sein:

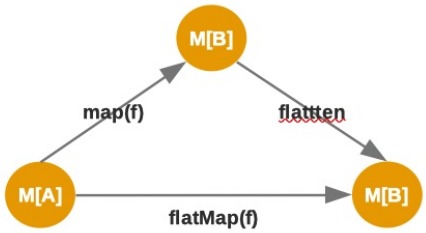

Abbildung 5.14: Kooperation von flatMap, map und flatten.

pure(a).flatMap(f) = f(a)	Links-Neutral
m.flatMap(pure) = m	Rechts-Neural

Auch dies wird man vernünftigerweise von der Interaktion von pure und flatMap erwarten.

5.2 Leser und Schreiber

Metaphern sind ein beliebtes Mittel zur bildhaften Darstellung abstrakter Sachverhalte. Im Falle der Monaden erfreuen sich zwei Metaphern besonderer Beliebtheit:

- die *Container-Methapher*: Monaden sind "Burritos",
- die *Berechnungs-Metapher*: "Berechnungen in einem Kontext" *"computational effects"*.

Diese beiden Interpretationen des Konzepts *Monade* sind oft hilfreich, aber nicht immer streng zu trennen. Sie beschreiben unterschiedliche Szenarios der Verwendung des Monaden-Begriffs, die in der Regel inhaltlich wenig miteinander zu tun haben. Diese Metaphern sollten aber nicht zu ernst genommen werden. Letztlich ist "Monade" nicht mehr als ein Konzept zur Strukturierung von Code nach einem einheitlichen Prinzip, mit dem Generizität und Wiederverwendbakeit erreicht werden sollen. Inhaltliche Bezügen können, sie müssen aber nicht bestehen. [4]

Die Container-Methapher haben wir im ersten Abschnitt ausführlich behandelt. Im Kern geht es bei ihr um flatMap als Mittel zur transparenten geschachtelten Iteration.

Bei der zweiten Monaden-Metapher, "Monaden sind Berechnungen in einem Kontext", geht es um die Verallgemeinerung von *Container* zu einem *Kontext*, der im Falle der Leser- und der Schreiber-Monade recht wörtlich zu nehmen ist.

- *Reader-Monade*: Der Kontext "liefert Werte", man kann ihn *lesen*.
- *Writer-Monade*: Der Kontext "konsumiert Werte", man kann ihn be-*schreiben*.

5.2.1 Reader-Monade

Bei einer Reader-Monade stellt der Kontext einige für die Berechnung benötigte Werte zur Verfügung. Mit einer solchen Monade strukturiert man darum Berechnungen, die von (unveränderlichen) Kontext-Werten abhängen. Sie sind damit ein Idiom, oder Muster für eine funktionale *Dependency Injection*.

Beispiel 1: Login mit DB-Abhängigkeit

Ein übliches Beispiel für den Einsatz einer Reader-Monade ist der Login-Prozess, der von Daten abhängt, die beispielsweise in einer Datenbank abgelegt sind. Die Datenbank ist der Kontext. Wir nehmen an, er hat folgende Schnittstelle:

```
trait DB {
  def getId(name: String): Option[Int]
  def getPassword(id: Int): Option[String]
  def getAccessRight(id: Int): Option[AccessRight]
}

enum AccessRight {
  case NoAccess
  case ReadAccess
  case WriteAccess
  case ReadWriteAccess
}
```

[4] Siehe dazu etwa http://tomasp.net/academic/papers/monads/

Das klassische Verfahren zum Umgang einer solchen Kontext-Abhängigkeit ist die Übergabe als Parameter. Beispielsweise beim mehrstufigen Feststellen der Zugriffsrechte:

```
def findId(db: DB, name: String): Int = db.getId(name).get

def checkPassword(db: DB, id: Int, password: String): Boolean =
  db.getPassword(id) match {
    case Some(pw) if (pw == password) => true
    case _ => false
  }

def getAccessRight(db: DB, id: Int): AccessRight =
  db.getAccessRight(id).get

def login(db: DB, name: String, password: String): AccessRight = {
  val id = findId(db, name)
  val pwOk = checkPassword(db, id, password)
  if (pwOk) getAccessRight(db, id) else AccessRight.NoAccess
}

val someDB: DB = ...

val acessRight = login(someDB, "Hugo", "56")
```

Die Funktionen arbeiten im Kontext einer Datenbank. (Siehe Abb. 5.15.) Der Kontext wird als Parameter übergeben und muss durch alle Aufrufe geschleift werden.

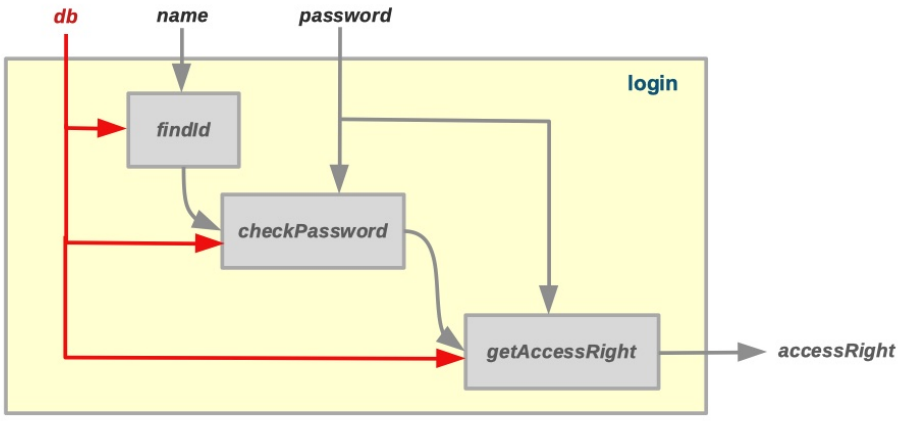

Abbildung 5.15: Berechnungen in einem Kontext als Kombination berechneter Werte.

Die Funktion *login* hat eine Struktur, bei der *berechnete Werte kombiniert werden.*

Um das explizite Herumreichen von db zu vermeiden, können wir login so umgestalten, dass es mit seinen Helfer-Funktionen findId, checkPassword und getAccessRight, sowie einem Passwort und einem Namen eine Funktion vom Typ

$$\mathsf{DB} \Rightarrow \mathsf{AccessRight}$$

generiert. Der gewünschter Effekt ist dabei, dass *Funktionen kombiniert* werden und das *Argument* dieser Funktionen db unerwähnt bleiben kann. Es muss dann nicht mehr explizit herum geschleppt werden. (Siehe Abb. 5.16.)

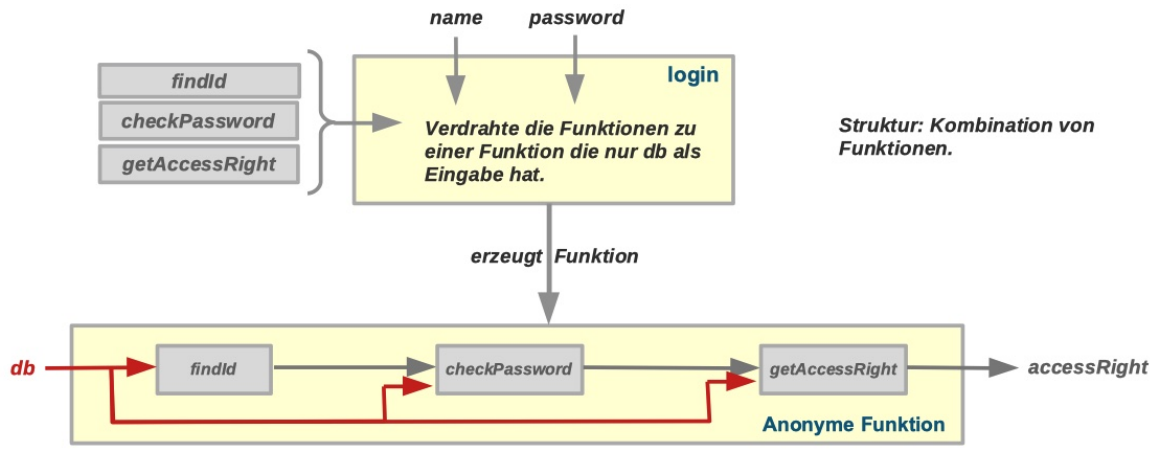

Abbildung 5.16: Berechnungen als Kombination kontextabhängiger Funktionen.

In einem ersten Schritt dahin kann die Datenbank aus der Parameterliste heraus "gecurryt" werden (Siehe Abb. 5.17.):

```
def findId(name: String): DB => Int =
  db => db.getId(name).get

def checkPassword(id: Int, password: String): DB => Boolean =
  db => db.getPassword(id) match {
    case Some(pw) if (pw == password) => true
    case _ => false
  }

def getAccessRight(id: Int): DB => AccessRight =
  db => db.getAccessRight(id).get

def login(name: String, password: String): DB => AccessRight =
  db => {
    val id = findId(name)(db)
    val pwOk = checkPassword(id, password)(db)
    if (pwOk) getAccessRight(id)(db)
    else AccessRight.NoAccess
  }
```

Das Currying bringt leider noch nicht viel. Hier wird immer noch mit db herum hantiert. Von einer "Funktionsverdrahtung" ist erst mal noch keine Spur zu sehen.

Wir können aber die val-Definition in login zurück in Parameterübergaben transformieren. Das macht die Funktion erst einmal etwas unübersichtlicher:

```
// val-Definitionen zu Parameterübergaben aufgelöst
def login(name: String, password: String): DB => AccessRight = db =>
  ((id: Int) =>
    ((pwOk: Boolean) =>
      if (pwOk) getAccessRight(id)(db) else AccessRight.NoAccess
    )(checkPassword(id, password)(db))
```

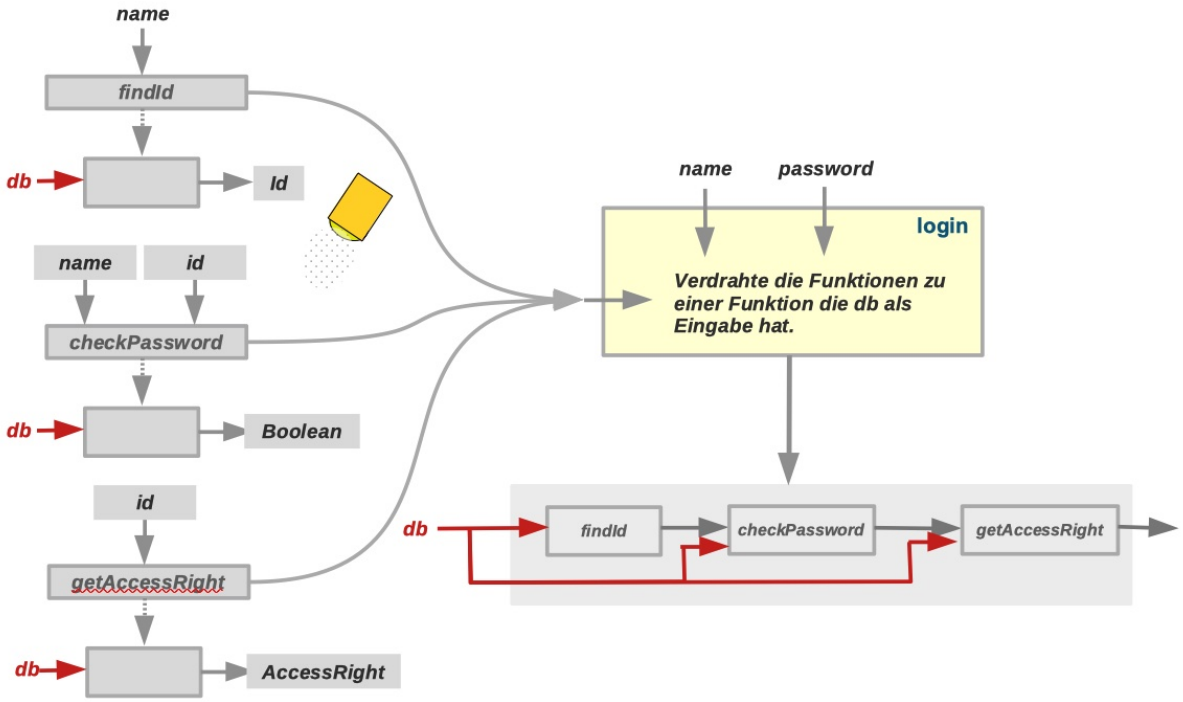

Abbildung 5.17: Kontextabhängige Funktionen mit etwas Currypulver.

```
)(findId(name)(db))
```

Wir erinnern uns aber, dass Funktionen mit der Hintereinanderausführung als *map*-Funktion, mit fixiertem Definitionsbereich, hier Z genannt, *Funktoren* sind (Siehe Abb. 5.18.):

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

type ZTo[T] = Z => T

given Functor[ZTo] with {
  extension[A, B] (f: ZTo[A]) def map(g: A => B): ZTo[B] = f andThen g
}
```

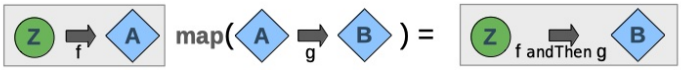

Abbildung 5.18: Funktionen als Funktoren.

Eine passende flatMap-Funktion macht sie sogar zu *Semi-Monaden* (Monaden ohne pure). (Siehe Abb. 5.19):

```
trait SemiMonad[F[_]] extends Functor[F] {
  extension[A, B] (x: F[A]) {
```

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

type ZTo[T] = Z => T

given SemiMonad[ZTo] with {
  extension[A, B] (f: ZTo[A]){
    def map(g: A => B): ZTo[B] = f andThen g
    def flatMap(g: A => ZTo[B]): ZTo[B] = z => { // == z => g(f(z))(z)
      val v1 = f(z)
      val v2 = g(v1)
      v2(z)
    }
  }
}
```

Abbildung 5.19: Funktionen als Semi-Monaden.

Damit haben wir aber schon den gesuchten Typ. ZTo reicht in seiner flatMap-Funktion ein Z weiter. Wenn wir ein z: Z als Kontext ansehen, dann haben wir mit ZTo[T] den gesuchten Typ der verkettbaren und damit iterierbaren, kontextabhängigen Aktionen.

Benennen wir ZTo[T] um in Reader[Z,T], dann haben wir mit einem fixierten Z schon die gesuchte Reader-Monade: Kontextabhängige transparent verkettbare Operationen:

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

Reader sind also im Kern *Funktionen* mit map und flatMap. Zum besseren Handling hier in einer Umschlag-Klasse Reader gekapselt. Fixiert man den ersten Typ-Parameter, dann hat man einen generischen Typ mit einem Parameter, der mit den entsprechenden Funktionen ein Funktor und eine Monade ist:

```
trait DB { ... }

type DBReader[T] = Reader[DB, T]

given Monad[DBReader] with {

  def pure[A](a: A): DBReader[A] = Reader(db => a)

  extension[A, B] (dbReader: DBReader[A]) {

    def flatMap(g: A => DBReader[B]): DBReader[B] =
      Reader(z => {
        val v1 = dbReader.f(z)
        val v2 = g(v1)
        v2.f(z)
```

```
      } )

    override def map(g: A => B) = Reader( dbReader.f andThen g )
  }

}
```

Die Hilfsfunktionen sind vom Typ A ⇒ DBReader (siehe Abb. 5.20)

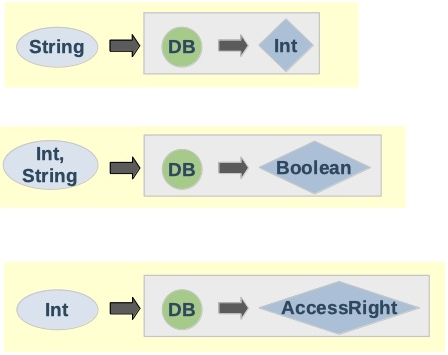

Abbildung 5.20: Struktur der Hilfsroutinen.

```
def findId(name: String): DBReader[Int] =
   Reader( (db: DB) => db.getId(name).get )

def checkPassword[R[_]: Monad](id: Int, password: String):
    DBReader[Boolean] =
  Reader( (db:DB) => db.getPassword(id) match {
     case Some(pw) if (pw == password) => true
     case _ => false
    }
  )

def getAccessRight[R[_]: Monad](id: Int): DBReader[AccessRight] =
  Reader( (db:DB) => db.getAccessRight(id).get)
```

und damit kompatibel mit flatMap (siehe Abb. 5.21)

Abbildung 5.21: flatmap und Reader.

Die login-Funktion kann mit flatMap:

```
def login(name: String, password: String): DBReader[AccessRight] =
  findId(name) flatMap(
    (id:Int) => checkPassword(id, password) flatMap(
      (pwOk: Boolean) =>
       if (pwOk) getAccessRight(id)
       else Monad[DBReader].pure(AccessRight.NoAccess)
```

```
    )
  )
```

oder schöner als Funktor-Block (*for-Comprehension*) formuliert werden:

```
def login(name: String, password: String): DBReader[AccessRight] =
  for ( id <- findId(name);
        pwOk <- checkPassword(id, password);
        res <- (if (pwOk) getAccessRight(id)
                else Monad[DBReader].pure(AccessRight.NoAccess))
    ) yield res
```

Es spricht auch nichts dagegen, die Klasse Reader gleich mit den "monadischen Methoden" auszustatten. Die Definition einer Monaden-Instanz ist dann nicht (zwingend) notwendig:

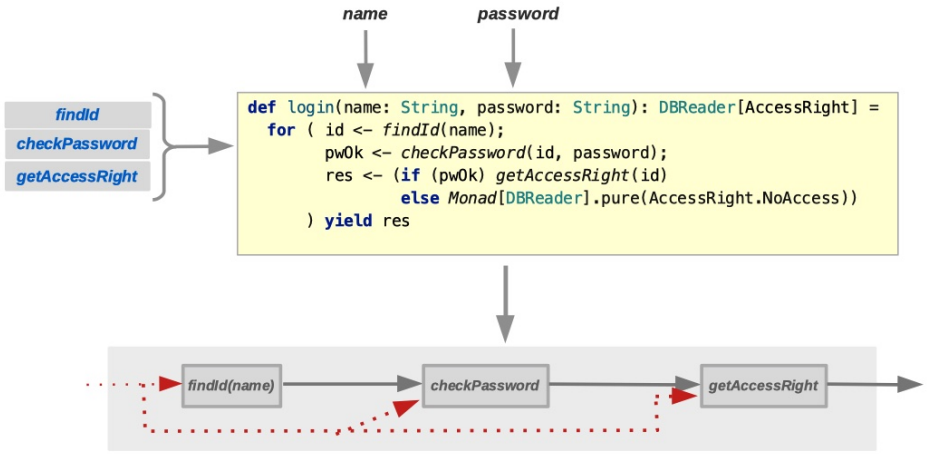

Abbildung 5.22: Die kombinatorische Form die Reader ermöglichen.

```
class Reader[Z, A](f: Z => A) {
  def this(a: A) = this((f:Z) => a)
  def apply(db: Z): A = f(db)
  def map[B](g: A => B) = Reader(f andThen g)
  def flatMap[B](g: A => Reader[Z, B]): Reader[Z, B] = Reader(z =>
      g(f(z)) (z) )
}

def findId(name: String): Reader[DB, Int] =
  Reader( (db: DB) => db.getId(name).get )

def checkPassword(id: Int, password: String): Reader[DB, Boolean] =
  Reader( (db:DB) => db.getPassword(id) match {
    case Some(pw) if (pw == password) => true
    case _ => false
  }
  )

def getAccessRight(id: Int): Reader[DB, AccessRight] =
  Reader( (db:DB) => db.getAccessRight(id).get)

def login(name: String, password: String): Reader[DB, AccessRight] =
```

```
  for ( id <- findId(name);
        pwOk <- checkPassword(id, password);
        res <- (if (pwOk) getAccessRight(id)
                else Reader(AccessRight.NoAccess)))
   yield res
```

Die Reader-Monade liefert uns in beiden Varianten die gewünschte "kombinatorische Form" der login-Funktion: Eine Funktion, in der andere Funktionen zu einer neuen Funktion kombiniert werden ohne, dass dabei explizit auf die einen db-Wert eingegangen werden muss: In login taucht db nicht ein einziges Mal auf! (Siehe Abb. 5.22)

Beispiel 2: Ausdrücke mit definierten Konstanten

Terme (Ausdrücke) mit Konstanten können einen Kontext ausgewertet werden, der den Konstanten einen Wert zuweist. Der Kontext wird oft auch *Umgebung* (engl. *Environment*) genannt und ist im einfachsten Fall eine Abbildung *Name* → *Wert*. Eine Auswertungsroutine hängt von diesen Kontext ab. Wir formulierten sie schon mal gleich "gecurryt":

```
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._

type Env = Map[String, Int] // der Kontext

def eval(term: Term): Env => Int =
  env => term match {
    case Literal(v) => v
    case Const(n) => env(n)
    case Add(t1, t2) => eval(t1)(env) + eval(t2)(env)
    case Sub(t1, t2) => eval(t1)(env) -eval(t2)(env)
    case Mult(t1, t2) => eval(t1)(env) * eval(t2)(env)
    case Div(t1, t2) => eval(t1)(env) / eval(t2)(env)
  }

val term: Term =
   Add(Literal(18),
     Div(Mult(Literal(12),
       Literal(4)), Const("two")))
val termValue = eval(term)(Map("two" -> 2)) // 42
```

Man sieht wieder, wie das env-Argument mühsam und wenig elegant herum gereicht werden muss.

Mit einer Reader-Klasse wird daraus:

```
class Reader[Z, A](f: Z => A) {
  def this(a: A) = this((f:Z) => a)
  def apply(db: Z): A = f(db)
  def map[B](g: A => B) = Reader(f andThen g)
```

```
  def flatMap[B](g: A => Reader[Z, B]): Reader[Z, B] = Reader(z =>
      g(f(z)) (z) )
}

type EnvReader[T] = Reader[Env, T]

def eval(term: Term): EnvReader[Int] = term match {
  case Literal(v) => Reader(v)
  case Const(n) => Reader(env => env(n))
  case Add(t1, t2) =>
    for(l <- eval(t1);
        r <- eval(t2))
      yield l+r
  case Add(t1, t2) =>
    for(l <- eval(t1);
        r <- eval(t2))
      yield l+r
  case Sub(t1, t2) =>
    for(l <- eval(t1);
        r <- eval(t2))
      yield l-r
  case Mult(t1, t2) =>
    for(l <- eval(t1);
        r <- eval(t2))
      yield l*r
  case Div(t1, t2) =>
    for(l <- eval(t1);
        r <- eval(t2))
      yield l/r
}

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

val termValue = eval(term)(Map("two" -> 2)) // 42
```

Beispiel 2: Programme mit definierten Konstanten

Das Beispiel der Termauswertung kann auf "Programme" mit Konstanten-Definitionen erweitert werden. Sehr einfache Programme wie etwa

```
x = 2
y = x*x
y = y + y
r = x+y*10/2
```

bestehen aus einer Folge von Konstanten-Definitionen, die jeweils mit einem Ausdruckswert belegt werden können. In den Ausdrucken können jeweils vorher definierte Konstanten verwendet werden. Die (abstrakte) Syntax dieser Programme ist:

```
enum Term {
  ...
  case Variable(name: String) // Ansonsten unverändert
```

```
  ...
}
import Term._

case class Assign(varName: String, value: Term)
type Prog = List[Assign]
```

und die Auswertung mit einem Reader:

```
def eval(term: Term): EnvReader[Int] = term match {
  case Literal(v) => Reader(v)
  case Variable(n) => Reader(env => env(n))
  case Add(t1, t2) =>
    for(l <- eval(t1);
        r <- eval(t2))
      yield l+r
  ... etc. ...
}

def executeAssgin(assign: Assign): Env => Env =
  assign match {
    case Assign(varName, exp) =>
      env => env + (varName -> eval(exp)(env))
  }

def execute(prog: Prog): Env => Env =
  prog.foldLeft((env: Env) => env)( (acc, assign) =>
    acc andThen executeAssgin(assign)
  )
```

5.2.2 Writer-Monade

Die Reader-Monade repräsentiert Berechnungen in einem Kontext, der als *Datenquelle* agiert. Die Writer-Monade repräsentiert das komplementäre Konzept von Berechnungen in einem Kontext, der als *Datensenke* fungiert. Mit einer Writer-Monade lassen darum Berechnungen strukturieren, die von einem Daten-konsumierenden Kontext abhängen. Damit sind sie ein weiteres Idiom / Muster für eine (funktionale) *Dependency Injection*.

Das Standard-Beispiel für Writer-Monaden sind (funktionale) *Logger*.

Beginnen wir mit einem einfachen *imperativen* (!) Logger:

```
trait Logger {
  def log(msg: String): Unit // mit Seiteneffekt !!
  def getLogs(): String
}

object logger extends Logger {
  private var logBuffer: StringBuffer = new StringBuffer
  def log(msg: String): Unit = logBuffer.append(msg+"\n")
  def getLogs(): String = logBuffer.toString
}


// Logs verändern den Logger:
```

```
def f(x:Int): String = {
  logger.log(s"calling f($x)")
  (x+2).toString
}

def g(s:String): Int = {
  logger.log(s"calling g($s)")
  2 * s.length
}

def h(i: Int): Boolean = {
  logger.log(s"calling h($i)")
  i % 2 == 0
}

def loggedComputation(x: Int): Boolean = {
  val y = f(x)
  val z = g(y)
  h(z)
}

val result = loggedComputation(2)
val logs = logger.getLogs()
```

Das *Logging* ist hier ein Seiteneffekt der ausgeführten Funktionen. – Die übliche imperative Vorgehensweise, bei der *logs* in einem *Log*-Verzeichnis gesammelt werden, das sich dabei verändert.

In einer einer funktionalen Version gibt es nichts, das sich "im Hintergrund" ändert. Alle Änderungen müssen als neue Versionen des sich ändernden Objekts explizit gemacht werden. In unserem Fall haben wird dann:

```
case class Log(msg: String) {
  def conc(other: Log): Log = Log(s"$msg\n${other.msg}")
}


def f(x:Int): (String, Log) =
  ((x+2).toString, Log(s"calling f($x)"))


def g(s:String): (Int, Log) =
  (2 * s.length, Log(s"calling g($s)"))


def h(i: Int): (Boolean, Log) =
  (i % 2 == 0, Log(s"calling h($i)"))


def loggedComputation(x: Int): (Boolean, Log) = {
  val (y, log1) = f(x)
  val (z, log2) = g(y)
  val (u, log3) = h(z)
  (u, log1.conc(log2).conc(log3))
}

val log = loggedComputation(2)._2
```

Es ist dieses explizite Herumreichen der Versionen, mit denen die Veränderungen funktional modelliert werden müssen, das die funktionalen Programme so umständlich macht, wenn sie sich in die reale Welt der veränderlichen Dinge bewegen.

Hier können Monaden helfen, in diesem Fall die *Writer*-Monade. Der *Logger* wird beschrieben und soll sich dabei im Hintergrund halten. Dazu müssen wir wieder die Verarbeitung der *Logs* in das flatMap eines geeigneten Typs schieben. Genau wie oben das Hantieren mit db durch die Kombination (mit Hilfe von flatMap) von Funktionen mit db-Argument ersetzt wurde.

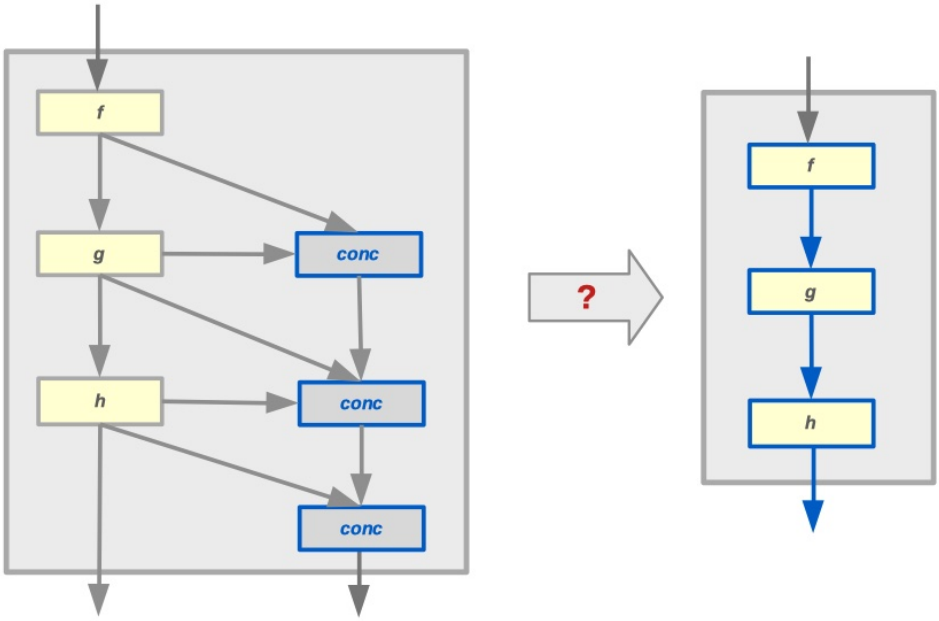

Abbildung 5.23: Von der Wert-Manipulation (mit conc) zur Kombination von Funktionen.

Die implizite Verarbeitung der Ergebnisse von f, g, h durch die Kombination der Funktionen f, g und h soll die explizite Verarbeitung ihrer Ergebnisse ersetzen. (Siehe Abb. 5.23.)

Als erstes vereinheitlichen wir den Umgang mit *Logs*. In

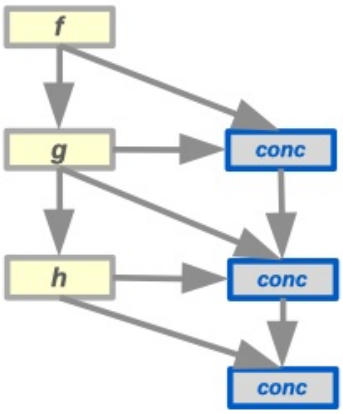

Abbildung 5.24: Schrittweise Verarbeitung von Logs.

```
def loggedComputation(x: Int): (Boolean, Log) = {
  val (y, log1) = f(x)
  val (z, log2) = g(y)
  val (u, log3) = h(z)
  (u, log1.conc(log2).conc(log3))
}
```

werden *Logs* in einem Schritt am Schluss der Funktionen verarbeitet. Das ersetzen wir durch eine schrittweise Verarbeitung (siehe Abb. 5.24):

```
def loggedComputation(x: Int): (Boolean, Log) = {
  val (y, log1) = f(x)
  val (z, log2) = g(y).match {case (v, l) => (v, log1.conc(l))}
  h(z).match {case (v, l) => (v, log2.conc(l))}
}
```

Die Hilfsfunktionen sind vom Typ $\mathsf{A} \Rightarrow (\mathsf{B}, \mathsf{Log})$. Mit einem Funktor $\mathsf{F}[\cdot] = (\cdot, \mathsf{Log})$ haben sie ein Typ der Form $\mathsf{A} \Rightarrow \mathsf{F}[\mathsf{B}]$ und sind damit "flatMap-kompatibel". Nennen wir den Typ $\mathsf{F}[\cdot]$ $\mathsf{LogWriter}[\cdot]$ dann erhalten wir (siehe Abb. 5.25):

```
case class Log(msg: String) {
  def conc(other: Log): Log = Log(s"$msg\n${other.msg}")
}

type LogWriter[A] = (A, Log)

def f: Int => LogWriter[String] = x =>
  ((x+2).toString, Log(s"calling f($x)"))

def g: String => LogWriter[Int] = s =>
  (2 * s.length, Log(s"calling g($s)"))

def h: Int => LogWriter[Boolean] = i =>
  (i % 2 == 0, Log(s"calling h($i)"))
```

Hüllen wir den Typ

$\mathsf{type\ LogWriter[A]} = (\mathsf{A}, \mathsf{Log})$

in eine Klasse, dann ergibt sich:

```
case class Log(msg: String) {
  def conc(other: Log): Log = Log(s"$msg\n${other.msg}")
}

case class LogWriter[A](a: A, log: Log) {
  def flatMap[B](f: A => LogWriter[B]): LogWriter[B] = {
    f(a) match {
      case LogWriter(v, l) =>
        LogWriter[B](v, log.conc(l))
    }
  }
}

def f: Int => LogWriter[String] = x =>
  LogWriter((x+2).toString, Log(s"calling f($x)"))
```

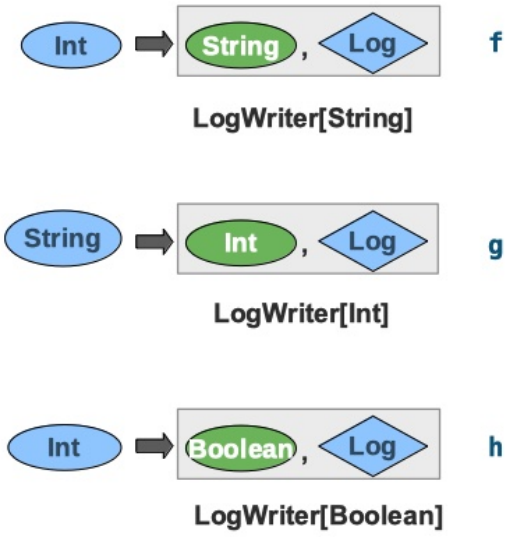

Abbildung 5.25: Der monadische Charakter der loggenden Funktionen.

```
def g: String => LogWriter[Int] = s =>
  LogWriter(2 * s.length, Log(s"calling g($s)"))

def h: Int => LogWriter[Boolean] = i =>
  LogWriter(i % 2 == 0, Log(s"calling h($i)"))
```

Da die *Logs* nur konkateniert werden, kann String durch eine beliebiges Monoid ersetzt werden:

```
trait Monoid[T] {
  def unit: T
  def combine(x: T, y: T): T
}

object Monoid {
  def apply[T: Monoid]: Monoid[T] = summon[Monoid[T]]
}

case class Writer[A, W:Monoid](a: A, w: W) {
  def flatMap[B](f: A => Writer[B, W]): Writer[B, W] = {
    val Writer(v, w1) = f(a)
    Writer[B, W](v, Monoid[W].combine(w, w1))
  }
}

case class Log(msg: String) {
  def conc(other: Log): Log = Log(s"$msg\n${other.msg}")
}

type LogWriter[A] = Writer[A, Log]

def f: Int => LogWriter[String] = x =>
  Writer((x+2).toString, Log(s"calling f($x)"))
```

```
def g: String => LogWriter[Int] = s =>
  Writer(2 * s.length, Log(s"calling g($s)"))

def h: Int => LogWriter[Boolean] = i =>
  Writer(i % 2 == 0, Log(s"calling h($i)"))

given Monoid[Log] with {
  def unit = Log("")
  def combine(l1:Log, l2:Log): Log = l1.conc(l2)
}

def loggedComputation: Int => Writer[Boolean, Log] = x =>
  f(x).flatMap(g).flatMap(h)

val resultAndlog = loggedComputation(2)
val log = resultAndlog.w
```

Der LogWriter kann wir noch zu einer Monade aufgeplustert werden:

```
trait Monoid[T] {
  def unit: T
  def combine(x: T, y: T): T
}

object Monoid {
  def apply[T: Monoid]: Monoid[T] = summon[Monoid[T]]
}

trait Functor[F[_]] {
  extension[A, B] (fa: 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]]
}

case class Writer[A, W:Monoid](value: A, w: W) {
  def flatMap[B](f: A => Writer[B, W]): Writer[B, W] = {
    val Writer(v, w1) = f(value)
    Writer[B, W](v, Monoid[W].combine(w, w1))
  }
}

case class Log(msg: String) {
  def conc(other: Log): Log = Log(s"$msg\n${other.msg}")
}
```

```
type LogWriter[A] = Writer[A, Log]

given Monoid[Log] with {
  def unit = Log("")
  def combine(l1:Log, l2:Log): Log = l1.conc(l2)
}

given Monad[LogWriter] with {
  def pure[A](a: A): LogWriter[A] = Writer(a, Monoid[Log].unit)

  extension[A, B] (logWriter: LogWriter[A]) {
    def flatMap(f: A => LogWriter[B]): LogWriter[B] = {
      val lw = f(logWriter.value)
      Writer(lw.value, Monoid[Log].combine(logWriter.w, lw.w))
    }
  }
}

def f: Int => LogWriter[String] = x =>
  Writer((x+2).toString, Log(s"calling f($x)"))

def g: String => LogWriter[Int] = s =>
  Writer(2 * s.length, Log(s"calling g($s)"))

def h: Int => LogWriter[Boolean] = i =>
  Writer(i % 2 == 0, Log(s"calling h($i)"))

// monadisch -funktional
def loggedComputation(x: Int): LogWriter[Boolean] = {
  for (y <- f(x);
       z <- g(y);
       u <- h(z))
    yield u
}

val resultAndlog = loggedComputation(2)
val log = resultAndlog.w
```

Man vergleiche loggedComputation hier mit der imperativen Version:

```
// imperativ
def loggedComputation(x: Int): Boolean = {
  val y = f(x)
  val z = g(y)
  val u = h(z)
  return u
}
```

Man sieht: mit etwas monadischer Würze wird auch die gesunde streng vegane funktionale Magerkost schmackhaft und genießbar.

5.3 Zustand

Ein *Reader* repräsentiert eine Berechnung, die von einem Werte-*liefernden* Kontext abhängt. Ein *Writer* repräsentiert eine Berechnung, die von einem Werte-*konsumierenden* Kontext abhängt.

Die *Zustands-Monade State* kombiniert beides. Sie repräsentiert eine Berechnung in einem Kontext der Werte liefern und konsumieren kann. Die Werte, die der Zustand liefert, können dabei auch vorher von ihm konsumiert worden sein. (Siehe Abb. 5.26.) Betrachten wir darum die Leser und Schreiber noch mal etwas genauer.

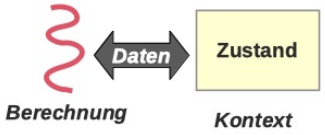

Abbildung 5.26: Berechnungen in einem Kontext der Werte liefert und konsumiert.

Ein Reader ist die Berechnung eines Wertes $a \in A$ abhängig von / im Kontext einer zu "lesenden" Datenquelle vom Typ Z:

$$Reader[Z, A] \simeq Z => A$$

Als Funktor mit fixiertem Z:

$$ZReader[A] \equiv Reader[Z, A] \simeq Z => A$$

Ein Writer ist die Berechnung eines Wertes $a \in A$ im Kontext einer "zu beschreibenden" Datensenke vom Typ W. Das Beschreiben wird dabei funktional als neue Version der Datensenke modelliert.

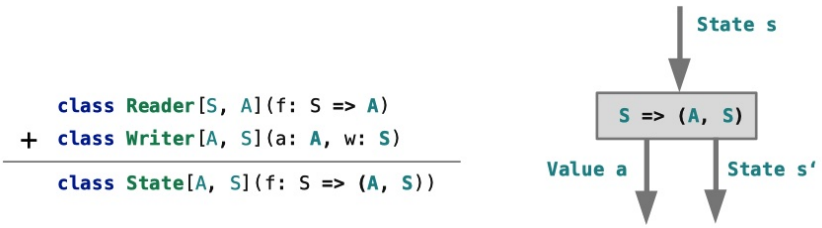

Abbildung 5.27: State = Reader + Writer.

Die Berechnung liefert darum ein $a \in A$ und ein $w \in W$.

$$Writer[A, W] \simeq (A, W)$$

Als Funktor mit fixiertem W:

$$WWriter[A] \equiv Wrtiter[A, W] \simeq (A, W)$$

Bei zustandsabhängigen Berechnungen ist der Kontext sowohl Datenquelle, als auch Datensenke. Z und W sind also der gleiche Typ S:

$$\begin{aligned} Reader[S, A] &\simeq \quad S => A \\ Writer[A, S] &\simeq \quad (A, S) \end{aligned}$$

Vereinheitlicht man beides dann ergibt sich die *Essenz der Zustandsmonade*:

$$State[S, A] \simeq S => (A, S)$$

Berechnungen die Werte aus dem aktuellen Zustand $s \in S$ konsumieren und neben einem Ergebnis einen neuen Zustand liefern können. (Siehe Abb. 5.27.)

State-Monaden konsumieren (lesen) S-Werte und produzieren (schreiben) S-Werte Durch die monadische Herangehensweise soll es möglich sein, mit den S-Werten einfach, elegant und möglichst im Hintergrund umzugehen.

In der Regel sind die S-Werte "funktionale Repräsentanten" eines veränderlichen Zustands in dem die Berechnung stattfindet. Zustands-Monaden ermöglichen darum das, was bei imperativen Programmen mit Seiteneffekten erreicht wird: Einfach im Hintergrund etwas ändern, ohne ständig über diesen Hintergrund reden zu müssen.

5.3.1 Beispiel: Ausdrücke mit Seiteneffekten auswerten

Ein Ausdruck mit einem Seiteneffekt liefert einen Wert und beeinflusst darüber hinaus die folgende Auswertung anderer Ausdrücke. Ein sehr einfacher Ausdruck mit einem Seiteneffekt ist der aus C bekannte *Inkrement-Operator* ++. i++ liefert den aktuellen Wert der Variablen i und erhöht ihn gleichzeitig. Das ist natürlich ein softwaretechnischer Irrweg, das sei aber einmal dahin gestellt. Wir wollen Ausdrücke mit einem solchen Inkrement-Operator auswerten. Erst einmal *imperativ* mit einem globalen Zustand, der im Hintergrund weiter gereicht wird. (Siehe Abb. 5.28.)

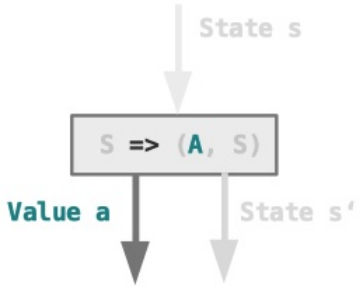

Abbildung 5.28: Imperativer Umgang mit einem globalen Zustand.

```
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]
var globalEnv: Env = Map() // globaler Zustand
```

```
def eval(exp: Term): Int = exp match {
  case Literal(v) => v
  case Variable(n) => globalEnv(n)
  case Inc(name) => // initial 0, sonst Post-Inkrement
    val v: Int = globalEnv.getOrElse(name, 0)
    globalEnv = globalEnv + (name -> (v+1))
    v
  case Add(t1, t2) => eval(t1) + eval(t2)
  case Sub(t1, t2) => eval(t1) -eval(t2)
  case Mult(t1, t2) => eval(t1) * eval(t2)
  case Div(t1, t2) => eval(t1) / eval(t2)
}

val term =
  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")))

val result = eval(term) // 42
```

Für alle Variablen wird ein initialer Wert von Null angenommen. Bei jedem Zugriff mit inc wird der aktuelle Wert erhöht. Die Variablenwerte sind in einem globalen Zustand vom Typ

$$\mathsf{Env} = \mathsf{Map[String, Int]}$$

gespeichert.

In einer funktionalen Version gibt es keinen globalen Zustand mit den aktuellen Variablenwerten. Der implizite "Hintergrund" der Variablenbelegungen muss explizit gemacht werden. – Mit dem damit verbundenen Umstand des Herumschleppens eines Env-Wertes:

```
def eval(exp: Term): Env => (Int, Env) = exp match {
  case Literal(v) => env => (v, env)
  case Variable(n) => env => (env(n), env)
  case Inc(name) => env => {
    val v: Int = env.getOrElse(name, 0)
    (v, env + (name -> (v+1)))
  }
  case Add(t1, t2) => env =>
    val (v1, env1) = eval(t1)(env)
    val (v2, env2) = eval(t2)(env1)
    (v1 + v2, env2)
  case Sub(t1, t2) => env =>
    val (v1, env1) = eval(t1)(env)
    val (v2, env2) = eval(t2)(env1)
    (v1 -v2, env2)
  case Mult(t1, t2) => env =>
    val (v1, env1) = eval(t1)(env)
    val (v2, env2) = eval(t2)(env1)
```

```
    (v1 * v2, env2)
  case Div(t1, t2) => env =>
    val (v1, env1) = eval(t1)(env)
    val (v2, env2) = eval(t2)(env1)
    (v1 / v2, env2)
}

val resultAndState = eval(term)(Map()) // (42,Map(x -> 6, y -> 2))
```

Damit sind wir funktional: Der Zustand, hier ein Wert vom Typ Env, ist ein *expliziter Wert* statt einer impliziten Variablen. (Siehe Abb. 5.29.)

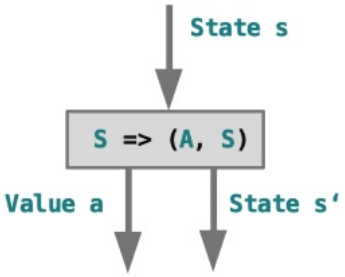

Abbildung 5.29: Funktionaler Umgang mit einem globalen Zustand.

Die Auswertungsfunktion für solche Ausdrücke ist

ein Reader[Env, Int]: sie beachtet die aktuellen Werte der Variablen, und

ein Writer[Int, Env]: die aktuellen Werte der Variablen werden verändert.

und damit insgesamt ein State[Env, Int].

Es fehlt noch die Definition von State[S, A]. Es ist klar, dass sich State als Kombination von Reader[Z, A] und Writer[A, W] ergeben muss:

```
class Reader[Z, A](f: Z => A) {
  def this(a: A) = this((f:Z) => a)
  def apply(db: Z): A = f(db)
  def map[B](g: A => B) = Reader(f andThen g)
  def flatMap[B](g: A => Reader[Z, B]): Reader[Z, B] = Reader(z =>
      g(f(z)) (z) )
}
```

plus

```
class Writer[A, W:Monoid](value: A, w: W) {
  def flatMap[B](f: A => Writer[B, W]): Writer[B, W] = {
    val Writer(v, w1) = f(value)
    Writer[B, W](v, Monoid[W].combine(w, w1))
  }
}
```

ergibt

```
class State[A, S](ma: S => (A, S)) { ??? }
```

Der Inhalt ist klar, aber die Methoden?

Betrachten wir flatMap. Es nimmt ein x: State[A, S] und eine Funktion f: A ⇒ State[B, S] und soll daraus ein State[B, S] machen. (Siehe Abb. 5.30.) *x* dabei nichts weiter als ein verpacktes *ma: S => (A, S)* und *f* erzeugt aus einem *a: A* ein verpacktes *mb: S => (B, S)*. Damit ist klar, wie die beiden kombiniert werden müssen (Siehe Abb. 5.30.):

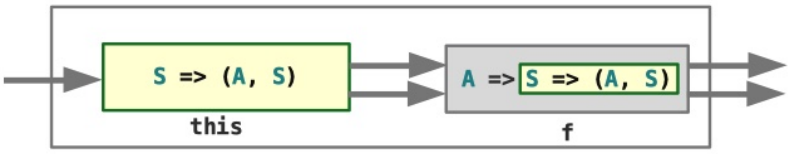

Abbildung 5.30: this.flatMap(f).

```
class State[A, S](ma: S => (A, S)) {
  ...
  def flatMap[B](f: A => State[B, S]): State[B, S] =
    State(s => {
      val (a, s1) = ma(s) // erst this.ma
      f(a)(s1)            // dann f
    })
    ...
}
```

flatMap verknüpft this.ma mit f. Dazu wird erst this.ma angewendet und dann f. Es wird also ein (verpacktes)

S ⇒ (A, S)

mit einem

A ⇒ (S ⇒ (B ⇒ S))

zu einem (verpackten)

S ⇒ (B, S)

kombiniert. Die Methode map ist offensichtlich: sie modifiziert den berechneten Wert mit einer Funktion. Insgesamt haben wir dann:

```
class State[A, S](ma: S => (A, S)) {

  // einfache Verwendung als Funktion.
  def apply(s: S) = ma(s)

  // map: wende this.ma an und transformiere den erzeugten Wert mit f.
  def map[B](f: A => B): State[B, S] =
    State(s => {
      val (a, newState) = ma(s)
      (f(a), newState)
    })

  // verknüpfe this.ma mit f: wende this.ma an und dann f.
  def flatMap[B](f: A => State[B, S]): State[B, S] =
    State(s => {
      val (a, s1) = ma(s)
      f(a)(s1)
```

```
    })
}

object State {
  // liefere einen Wert a unabhängig vom Zustand.
  // Verändere den Zustand nicht.
  def pure[A, S](a: A): State[A, S] = State(s => (a, s))
}
```

Unsere Ausdrucksauswertung wird damit zu:

```
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) =>
    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))
      yield v1 / v2
}

val term = ... wie oben ....

val resultAndState = eval(term)(Map()) // (42, Map(x -> 6, y -> 2))
```

Schön! So schön wie die imperative Variante und dabei auch noch völlig ungiftig und frei von Nebenwirkungen und Seiteneffekten.

5.3.2 Beispiel: Postfix-Ausdrücke auswerten

Betrachten wir noch ein Beispiel, die Auswertung von *Postfix-Ausdrücken*. Postfix-Ausdrücke sind Ausdrücke bei denen der Operator den Operanden folgt. Also

$12 + 3 *$ statt $(1 + 2) * 3$.

Postfix-Ausdrücke sind sehr einfach zu parsen:

```
enum Operator(op: (Int, Int) => Int) {
  case Add extends Operator( (x, y) => x+y )
  case Sub extends Operator( (x, y) => x-y )
  case Mul extends Operator( (x, y) => x*y )
  case Div extends Operator( (x, y) => x/y )
```

```
  def apply(x: Int, y: Int): Int = op(x,y)
}

type OpOrInt = Operator | Int
type PostfixExp = List[OpOrInt]

val numPat = """(0|(?:[1-9][0-9]*))""".r
val opPat = """(\+|\-|\*|/)""".r

def parse(str: String): PostfixExp =
  str.split(" ").toList.map {
    case numPat(n) =>
      n.toInt
    case opPat(op) =>
      op match {
        case "+" => Operator.Add
        case "-" => Operator.Sub
        case "*" => Operator.Mul
        case "/" => Operator.Div
      }
  }

val exp = "1 2 + 3 *" // (1 + 2) * 3

val postfixExp = parse(exp) // List(1, 2, Add, 3, Mul)
```

Mit einem Stack können sie auch sehr einfach ausgewertet werden:

```
class Stack(private var rep: List[Int] = List()) {
  def push(x: Int): Unit =
    rep = x :: rep
  def pop(): Int = rep match {
    case top :: rest =>
      rep = rest
      top
    case _ =>
      throw IllegalStateException("stack is empty")
  }
}

val stack: Stack = new Stack

def eval(postFixExp: PostfixExp): Int = postFixExp match {

  case Nil => stack.pop()

  case first :: rest => first match {
    case x: Int =>
      stack.push(x)
      eval(rest)

    case op: Operator =>
      val v1 = stack.pop()
      val v2 = stack.pop()
      stack.push(op(v2,v1))
      eval(rest)
  }
```

```
}

val expVal = eval(postfixExp) // 9
```

Das ist *imperativer* Code! Die Variable stack wird während der Auswertung im Hintergrund verändert: Seiteneffekte! Igitt! Also schreiben wir es mal in eine funktionale Version um:

```
type Stack = List[Int]

def eval(postFixExp: PostfixExp): Stack => Int = postFixExp match {

  case Nil => stack => stack.head

  case first :: rest => first match {
    case x: Int =>
      stack => eval(rest)(x::stack)

    case op: Operator => stack => {
      val v1 = stack.head
      val v2 = stack.tail.head
      eval(rest)(op(v2,v1) :: stack.tail.tail)
    }
  }
}

val exp = "4 1 -3 *" // (4 -1) * 3
val postfixExp = parse(exp) // List(4, 1, Sub, 3, Mul)
val expVal = eval(postfixExp)(Nil) // 9
```

Der veränderliche Stack wird hier durch einen zusätzlichen Parameter modelliert. Aber auch wenn der Stack "sich ändert" (tatsächlich werden stets neue Versionen erzeugt), sieht doch sehr nach Reader aus. Mit

$$Stack \Rightarrow Int$$

hat der Wert von eval einen "Reader-Typ": Wertberechnung abhängig von einem Kontext Z. Der Kontext Z ist hier der Stack. Weiter oben hatten wir ja schon einen Reader zur Auswertung von Ausdrücken mit Konstanten eingesetzt. Also nichts Neues hier. Formulieren wir eval mit einem Reader:

```
class Reader[Z, A](f: Z => A) {
  def this(a: A) = this((f:Z) => a)
  def apply(db: Z): A = f(db)
  def map[B](g: A => B) = Reader(f andThen g)
  def flatMap[B](g: A => Reader[Z, B]): Reader[Z, B] = Reader(z =>
      g(f(z)) (z) )
}

type Stack = List[Int]
type StackReader[T] = Reader[Stack, T]

def eval(postFixExp: PostfixExp): StackReader[Int] = postFixExp match {

  case Nil => Reader(stack => stack.head)

  case first :: rest => first match {
```

```
      case x: Int =>
        Reader(stack => eval(rest)(x::stack))

      case op: Operator => Reader(stack => {
        val v1 = stack.head
        val v2 = stack.tail.head
        eval(rest)(op(v2,v1) :: stack.tail.tail)
      })
    }
  }
```

Diese Auswertungsfunktion hat zwei Auffälligkeiten im Vergleich zur Funktion zur Auswertung "normaler" Infix-Ausdrücke. flatMap kommt nicht zum Einsatz. Das hat seinen Grund darin, dass Zwischenergebnisse im Stack gespeichert werden und darum nicht an den nächsten Schritt weiter gegeben werden müssen. flatMap hat die Signatur:

$$A \Rightarrow S \Rightarrow A$$

beziehungsweise

$$Int \Rightarrow Stack \Rightarrow Int$$

Der Wert kommt aber in den Stack und muss darum nicht explizit weiter gereicht werden. (Siehe Abb. 5.30.):

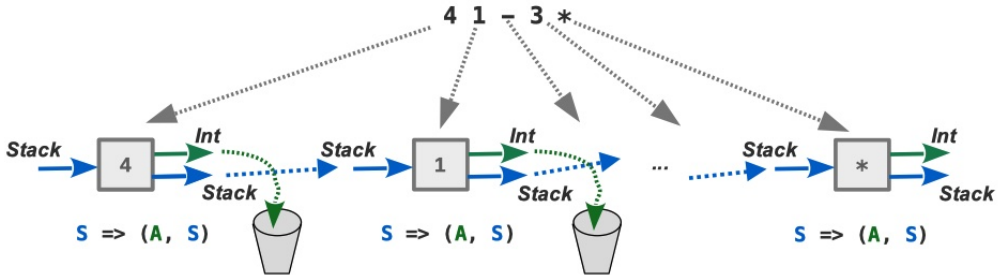

Abbildung 5.31: Zwischenergebnisse werden über den Stack weiter gereicht

flatMap verlangt aber ein "Zwischenergebnis". (Siehe Abb. 5.32.):

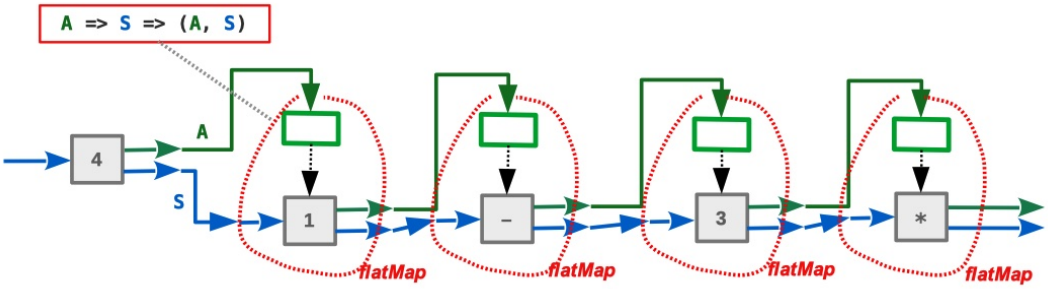

Abbildung 5.32: flatMap verlangt Zwischenergebnisse.

Nun, was nicht passt, wird passend gemacht. Wenn wir die Zustands-Monade nutzen wollen, dann müssen die mit flatMap verknüpften Funktionen ein Argument annehmen, auch wenn sie

es nur nehmen und dann wegwerfen (Siehe Abb. 5.33.):

```
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))
}

type Stack = List[Int]
type StackState[T] = State[T, Stack]

def processSymbol(symbol: OpOrInt): Int => StackState[Int] =
  symbol match {
    case x: Int =>
      _ => State(stack => (x, x :: stack))
    case op: Operator =>
      _ => State(
        stack => {
          val v1 = stack.head
          val v2 = stack.tail.head
          (op(v2, v1), op(v2, v1) :: stack.tail.tail)
        })
   }
```

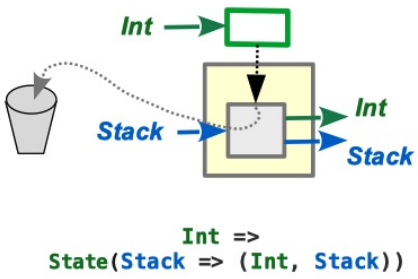

Abbildung 5.33: State-flatMap-kompatible Verarbeitung der Symbole.

Jetzt können wir auf hübsche Art[5] Postfix-Ausdrücke auswerten:

```
val expValInState = for {
   x <- processSymbol(4)(0);
   y <- processSymbol(1)(x);
```

[5] Monaden sind Softwaretechnik, es weniger darum etwas zu tun, als es auf hübsche Art zu tun.

```
    z <- processSymbol(Operator.Sub)(y);
    u <- processSymbol(3)(z);
    v <- processSymbol(Operator.Mul)(u)
) yield v

val expVal = expValInState(Nil) // (9,List(9))
```

Für eine Auswertungsfunktion können wir die Tatsache nutzen, dass Postfix-Ausdrücke in einer Schleife ausgewertet werden können. Natürlich nehmen wir statt einer imperativen Schleife ein foldLeft:

```
def eval(postFixExp: PostfixExp): StackState[Int] = {

  def processSymbol(symbol: OpOrInt): Int => StackState[Int] =
    symbol match {
      case x: Int =>
        _ => State(stack => (x, x :: stack))
      case op: Operator =>
        _ => State(
          stack => {
            val v1 = stack.head
            val v2 = stack.tail.head
            (op(v2, v1), op(v2, v1) :: stack.tail.tail)
          })
    }

  postfixExp.foldLeft( State.pure(0): StackState[Int] )(
    (stackState, symbol) =>
      stackState.flatMap(processSymbol(symbol) )
  )
}

val exp = "4 1 -3 *" // (4 -1) * 3
val postfixExp = parse(exp) // List(4, 1, Sub, 3, Mul)
val expVal = eval(postfixExp)(Nil) // (9,List(9))
```

Mit diesem Beispiel sollte gezeigt werden, dass bei einer monadischen Herangehensweise keine Naturgesetze oder mathematischen Wahrheiten umgesetzt werden sollen, sondern dass es darum geht, den Code in passender schöner Art zu strukturieren. Was passend oder schön ist, das liegt bekanntlich im Auge des Betrachters und kosmetische Mittel können unbeschränkt verwendet werden. – Im perfekten Make-up der einen sehen andere nur schmierige Schminke.

5.4 Fortsetzungsfunktionen

In diesem Abschnitt geht es um die "Mutter aller Monaden", die *Continuation-Monade*. Dazu vertiefen wir uns erst einmal in die Thematik der Fortsetzungsfunktionen (engl. *Continuations*) und dem *ContinuationPassing-Style*.

5.4.1 CPS – Continuation Passing Style

Beim *ContinuationPassing-Style (CPS)* (auch *Call-Back-Style*) der Programmierung hat eine Funktion den Verwender ihres Ergebnisses als zusätzlichen Parameter. Statt

```
def fac(x: Int) : Int =
  if (x == 0) 1
  else fac(x-1)*x

val res_direct = fac(10)
// res_direct = 3628800
```

schreibt man im *ContinuationPassing-Style*:

```
def facC(x: Int, cont: Int => Unit) : Unit =
  if (x == 0) cont(1)
  else facC(x-1,
    res => cont(res*x))

var res_cont = 0

facC(10, res => {res_cont = res} )
// res_cont = 3628800
```

Ein anderes Beispiel ist:

```
def mult2(x: Int): Int = x*2
def add1(x: Int): Int = x+1

val output_direct =
  println(
    mult2(
      add1(
        20)))
```

Hier liefern "normale Funktionen" Ergebnisse. Sie werden durch Parameterübergaben verkettet. Dabei werden Zwischenergebnisse weitergegeben. In der CPS-Variante wird daraus:

```
def pure(x: Int, k: Int => Unit): Unit = k(x)
def mult2C(x: Int, k: Int => Unit): Unit = k(x*2)
def add1C(x: Int, k: Int => Unit): Unit = k(x+1)

val output_cont =
  pure(20, x =>
    add1C(x, y =>
      mult2C(y, z =>
        println(z))))
```

Hier haben die Funktionen einen zusätzlichen Funktionsparameter (*Continuation*, oder *Call-*

back genannt), der das Ergebnis annimmt. Die *Callbacks* werden durch Parameterübergaben verkettet.

CPS und Asynchronität

Die *Struktur* des Codes der beiden Varianten unterscheidet sich. In aller Regel unterscheidet sich die *Ausführung* des Codes aber nicht wesentlich. In manchen Systemen gibt es allerdings drastische Unterschiede in der Ausführung. Dort werden gewisse (I/O-) Systemroutinen *asynchron* ausgeführt. Will man andere, weitere Aktionen asynchron ausführen, dann kann man sie dort per *Callback* an die asynchrone Systemroutine anhängen. Dort hat man beispielsweise mit:

```
fs.readFile(
  ...,
  function (err, data) {
    if (err) throw err;
     ....
});
```

die Möglichkeit folgendes zum Ausdruck zu bringen:

```
f = Future{ fs.readFile( ... ) }

f onComplete {
  case Success(...) => ...
  case Failure(err) => throw err
}
```

Da Future ein monadischer Typ ist, sieht eine mit Future organisierte Berechnung deutlich eleganter aus, als eine die den "rohen" Callback-Stil verwendet. *De facto* gibt es keinen Unterschied außer dem des besser organisierten Codes. *Asynchronität* wird von Future geliefert und der Callback-Stil kann problemlos damit kombiniert werden:

```
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global

def pure(x: Int, k: Int => Unit): Unit = Future { k(x) }
def mult2C(x: Int, k: Int => Unit): Unit = Future { k(x*2) }
def add1C(x: Int, k: Int => Unit): Unit = Future { k(x+1) }
def printC(x: Int): Unit = Future{ println(x) }


def run(): Unit =
 pure(20, x =>
   add1C(x, y =>
     mult2C(y, z =>
       printC(z)
     )
   )
 )
```

Eleganter ist aber

```
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global
```

```
def pure(x: Int): Future[Int] = Future { x }
def mult2C(x: Int): Future[Int] = Future { x*2 }
def add1C(x: Int): Future[Int] = Future { x+1 }
def printC(x: Int): Future[Unit] = Future{ println(x) }

def run(): Unit =
  for (x <- pure(20);
       y <- add1C(x);
       z <- mult2C(y) )
    yield printC(z)
```

Future ist ein monadischer Typ, der mit flatMap wesentlich elegantere Möglichkeiten bietet Verkettungen asynchroner Berechnungen auszudrücken.

CPS und Endrekursion

Rekursive Funktionen, bei denen der rekursive Aufruf die letzte Aktion im rekursiven Zweig ist, nennt man *end-rekursiv* (*tail recursive*). Die GGT-Berechnung mit folgendem Algorithmus ist end-rekursiv:

```
@tailrec
def gcd(x: Int, y: Int): Int =
  if (x == y) x
  else gcd(Math.max(x,y) -Math.min(x,y), Math.min(x,y))
```

Endrekursive Funktionen können in Schleifen umgewandelt werden. In anständigen funktionalen Sprachen erkennt der Compiler End-Rekursionen und wandelt sie selbständig in Schleifen um. gcd also in etwa zu:

```
def gcd(x: Int, y: Int): Int = {
  var (x_, y_) = (x, y)
  while (x_ != y_) {
    val (a, b) = (Math.max(x_, y_) -Math.min(x_, y_), Math.min(x_, y_))
    x_ = a
    y_ = b
  }
  x_
}
```

Die Annotation tailrec ist dazu nicht notwendig. Der Scala-Compiler wandelt alle end-rekursiven Funktionen in Schleifen um, mit oder ohne tailrec-Annotation. Da end-rekursive Funktionen die Wahrscheinlichkeit eines *Stack-Overflow*-Fehlers drastisch reduzieren, ist es wichtig, die richtige Vorstellung davon zu haben, was end-rekursiv ist, und was nicht. Mit der Annotation können Programmierer ihre Meinung zu diesem Thema zum Ausdruck bringen. Der Compiler gibt dann eine Fehlermeldung aus, wenn er diese Meinung nicht teilt.

Leider sind Funktionen, die "von Natur aus" end-rekursiv sind eher selten. Sehr viele Funktionen sind *linear rekursiv*. Bei einer linear-rekursiven Funktion enthält der rekursive Zweig auch nur einen rekursiven Aufruf, aber dieser Aufruf ist nicht zwingend die letzte Aktion. Ein Beispiel ist die Fakultätsfunktion:

```
def fac(x: Int) : Int =
  if (x == 0) 1
  else fac(x-1)*x
```

oder auch typische Funktionen auf Listen:

```
def append[A, B >: A](lst1: List[A], lst2: List[B]): List[B] =
  st1 match {
    case Nil => lst2
    case head :: tail => head :: append(tail, lst2)
  }

def reverse[A](lst: List[A]): List[A] = lst match {
  case Nil => Nil
  case head :: tail => append(reverse(tail), List(head))
}
```

Interessanterweise werden linear-rekursive Funktionen nach einer Transformation in den *CPS* end-rekursiv:

```
  @tailrec
  def facC(x: Int, k: Int => Unit) : Unit =
    if (x == 0) k(1)
    else facC(x-1, i => k(i*x))

  @tailrec
  def appendC[A, B >: A](lst1: List[A], lst2: List[B],
                         k: List[B] => Unit): Unit = lst1 match {
    case Nil => k(lst2)
    case head :: tail => appendC(tail, lst2, l => k(head::l))
  }

  @tailrec
  def reverseC[A](lst: List[A],
                  k: List[A] => Unit): Unit = lst match {
    case Nil => k(Nil)
    case head :: tail =>
      reverseC(tail, l => appendC(l, List(head), k))
  }

  def pure[A](a: A, k: A => Unit): Unit = k(a)
```

Ein Anwendungsbeispiel für die Listenfunktionen ist:

```
  val lst_1 = List(1, 2, 3)
  val lst_2 = List(4, 5, 6)

  def print42: Unit =
    pure(lst_1, x =>
      appendC(x, lst_2, y =>
        reverseC(y, z =>
          println(z) // List(6, 5, 4, 3, 2, 1)
        )
      )
    )
```

CPS und Backtracking

Unter *Backtracking* versteht man eine optimierte Version der algorithmischen Strategie "rohe Kraft / erschöpfende Suche". Wenn der Suchraum einer Probleminstanz aus den Blättern eines Baums besteht und man bereits bei inneren Knoten des Baums erkennen kann, dass der Unterbaum, der diesen Knoten als Wurzel hat, keine Lösung enthält, dann kann der Unterbaum bei der Lösungssuche komplett ausgelassen werden.

Das Standardbeispiel ist das Problem der n-Damen.[6] Der Suchraum besteht aus allen Schachbrettern der Größe $n \times n$ die jeweils mit n Damen besetzt sind. Der Suchraum kann als Menge der Blätter eines Baums verstanden werden, deren Knoten aus teilweise besetzten Brettern besteht und die alle Bretter als Nachfolger haben, bei denen jeweils eine Dame irgendwo in der nächsten freien Zeile gesetzt ist. (Siehe Abb. 5.34.)

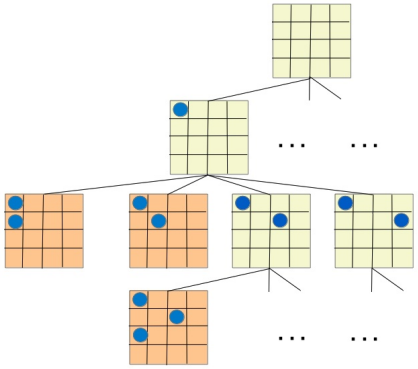

Abbildung 5.34: Backtracking: optimierte erschöpfende Suche in einem baumförmig organisierten Suchraum.

Beim Backtracking wird also ein Baum traversiert. Dabei könne zwei der üblichen Varianten verwendet werden

- preorder Tiefensuche, oder
- preorder Breitensuche.

Jeder Knoten wird dabei betrachtet, bevor seine Kinder betrachtet werden (vielversprechend, oder nicht). Darum wird immer preorder traversiert.

Die einfachste Variante einer *imperativen* Implementierung ist eine preorder Tiefensuche mit einem rekursiven Algorithmus, bei der der Baum "im Fluge" erzeugt und begutachtet wird. Der aktuelle Knoten wird oft als "Zustand" bezeichnet. Aus ihm können sich "Folgezustände" (Nachfolgerknoten) ergeben. Ein "Zustand" (Knoten) ohne Nachfolger ist dann entweder

- eine "Sackgasse", oder
- eine Lösung.

In einer "Sackgasse" geht man zurück zum Vorgänger und untersucht dessen nächsten Nachfolger. Gibt es keine weiteren Nachfolger mehr, dann ist auch dieser Vorgänger eine "Sackgasse" und es geht wieder zurück.

[6] https://de.wikipedia.org/wiki/Damenproblem.

Dieses Verfahren kann **imperativ** mit Hilfe von *Exceptions* realisiert werden. Vor jeder Wahl des nächsten "Zustands" wird ein Auffangpunkt gesetzt und in einer Sackgasse wird eine Exception geworfen. Ein Algorithmus zur Lösung des Problems der n Damen ist mit diesem Verfahren:

```
// Gibt es einen Konflikt der Damen bei dieser Konstellation?
def Ok(board: List[Int]): Boolean =
  (for (i <- 0 until board.length;
        j <- i + 1 until board.length
        ) yield {
    val (x, y) = (board(i), board(j))
    val d = j -i
    !(x == y || y == x -d || y == x + d)
  }).find(_ == false)
    .getOrElse(true)

def NQueens(n: Int): List[Int] = {

  class BTException extends Throwable

  def solve(t: List[Int]): List[Int] = {
    if (t.length == n)
      return t
    else {
      for (x <- 0 until n) {
        val t_extended = t ++ List(x)
        if (Ok(t_extended))
          try {
            return solve(t_extended)
          } catch { // gesicherten Stand festhalten
            case _: BTException =>
          }
      }
      throw new BTException // Sprung (Backtrack) zum letzten gesicherten
          Stand
    }
  }

  try {
    solve(Nil)
  } catch {
    case e:BTException => Nil
  }
}
```

Mit den *Exceptions* wird hier eine spezielle *Kontrollstruktur* realisiert, die es erlaubt in der Ausführung wieder an einen bereits passierten Punkt zurück zu kehren und von dort aus einen alternativen Weg zu gehen.

In der funktionalen Programmierung gibt es natürlich keine *Exceptions*. Trotzdem kann der imperative Algorithmus oben auch rein funktional implementiert werden. Auffangstellen von *Exceptions* entsprechen alternativen *Fortsetzungs-Funktionen* (*Continuations*, *Callbacks*). Sie werden dem nächsten Versuch als Parameter mitgegeben und können aktiviert werden, wenn die Suche in einer Sackgasse endet.

Der Algorithmus von oben kann mit dieser Idee in eine rein funktionale Form gebracht werden:

```
def NQueens(n: Int): Unit = {
```

```
  def solve(
    t: List[Int],
    ksucc: List[Int] => Unit,
    kfail: => Unit): Unit = {

   // die imperative Schleife von oben hier als Rekursion
   def loop(x: Int): Unit = {
    if (x == n)
      kfail // Backtrack
    else {
      val t_extended = t ++ List(x)
      if (Ok(t_extended)) {
        // Versuche es mit t ++ [x],
        // wenn das schief geht, dann mache weiter
        // mit der nächsten Runde (x+1) in der Schleife.
        solve(t_extended, ksucc, loop(x + 1))
      } else {
        loop(x + 1)
      }
    }
   }

   if (t.length == n)
     ksucc(t)
   else
     loop(0) // ~for (x <- 0 until n)
  }

  solve(Nil, lst => println(lst), println("Failed"))
}

NQueens(4) // List(1, 3, 0, 2)
```

Der Kontrollfluss in NQueens wird durch zwei *Continuations*, ksucc und kfail, modelliert.

- ksucc ist die Fortsetzungs-Funkion für den Erfolgsfall. Sie wird weiter gegeben bis irgendwann eine Lösung gefunden wird, die an sie übergeben wird.
- kfail ist die Fortsetzungs-Funkion für den Fall, dass ein Backtracking notwendig ist. Sie wird ausführt, wenn der Algorithmus in eine Sackgasse gerät.

In der imperativen Version geht eine Schleife über alle möglichen Erweiterungen (0 until n) der der aktuellen Teillösung t:

```
def solve(t: List[Int]): List[Int] = {
   ...
   for (x <- 0 until n) {
     ...
   }
}
```

In der funktionalen Variante wird diese Schleife durch eine rekursive Funktion loop ersetzt.

```
def solve(
    t: List[Int],
```

```
    ksucc: List[Int] => Unit, kfail: => Unit): Unit = {

  def loop(x: Int): Unit = {
    ...
  }

  if (t.length == n)
    ksucc(t)
  else
    loop(0) // ~for (x <- 0 until n)
  }

  solve(Nil, lst => println(lst), println("Failed"))
}
```

Wie in der imperativen Version in der Schleife, wird in loop eine mögliche Erweiterung nach der anderen ausprobiert, bis schließlich eine zu Erfolg geführt hat und ksucc aktiviert wird, oder alle Alternativen erschöpft sind und kfail aktiviert wird. (Siehe Abb. 5.35.)

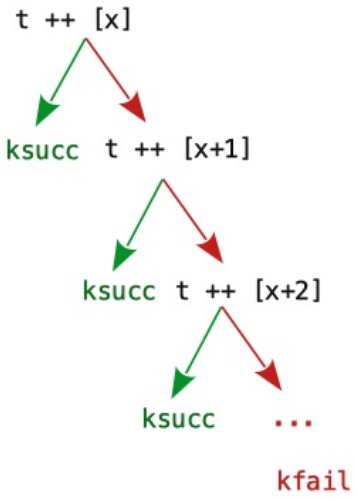

Abbildung 5.35: Backtracking mit *Success-* und *Failure-Continuations*.

So, nach der Betrachtung möglicher Anwendungen, wenden wir uns jetzt der Continuation-Monade zu, die es erlauben sollte, diese Anwendungen eleganter und übersichtlicher zu gestalten als das mit "rohen" Fortsetzungsfunktionen möglich ist.

5.4.2 Die Continuation-Monade

Der *Continuation Passing Style, CPS* ist eine vielseitig verwendbare Technik zur Lösung algorithmischer Probleme. Leider ist er etwas unhandlich und führt oft zu komplexem und schwer verständlichen Code. Hier kommen jetzt die Monaden ins Spiel. Mit einem monadischen Stil kann Übersicht und Ordnung in die Sache gebracht werden.

Im *CPS* müssen eigentlich einfache *Verknüpfungen* von Funktionen als kompliziere *Verschachtelungen* dargestellt werden. Kommen wir noch einmal zurück zu unserem ersten einfachen Beispiel von oben:

```
def pure(x: Int, k: Int => Unit): Unit = k(x)
def mult2(x: Int, k: Int => Unit): Unit = k(x*2)
def add1(x: Int, k: Int => Unit): Unit = k(x+1)
def out(x: Int): Unit = println(x)

def print42 : Unit =
  pure(20, x =>
    add1(x, y =>
      mult2(y, out)
    )
  )
```

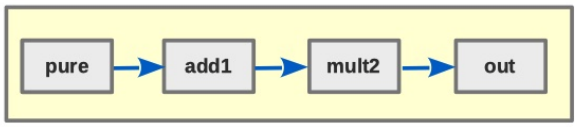

Abbildung 5.36: Funktionsverkettung: Das ist das, was wir wollen.

Eigentlich eine ganz einfache und übersichtliche Sache: Eine Funktion soll nach der anderen ausgeführt werden. (Siehe Abb. 5.36.) Das was wir tatsächlich hinschreiben müssen, ist aber eine unübersichtliche Verschachtelung: Soll eine Folgeoperation *angehängt* werden, dann muss sie *im Inneren* einer existierenden Struktur *eingebaut* werden. (Siehe Abb. 5.37.)

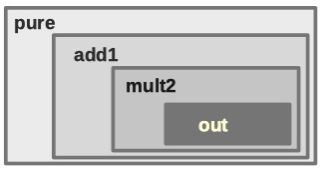

Abbildung 5.37: Funktionsverkettung: Das ist das, was wir konstruieren müssen.

Unschön, aber es gibt eine Kur. Wir müssen die Sache monadisch machen.

Etwas Curry

Beginnen wir mit dem üblichen Mittel und streuen ein wenig Curry auf die Funktionen. Das transformiert sie zu:

```
def pure(x: Int): (Int => Unit) => Unit =
  kInt => kInt(x)

def mult2(x: Int): (Int => Unit) => Unit =
  kInt => kInt(x*2)

def add1(x: Int): (Int => Unit) => Unit =
  kInt => kInt(x+1)

def out: Int => Unit = x => println(x)

def print42 : Unit =
```

```
pure(20)( x =>
  add1(x)(y =>
    mult2(y)(out)
  )
)
```

Man erkennt sofort, dass pure, mult2 und add1 eine "flatMap-Signatur" haben. (Siehe Abb. 5.38.) Der Funktor F ist dabei so definiert, dass

$\mathsf{F}[\mathsf{Int}] \simeq (Int \Rightarrow Unit) \Rightarrow Unit$

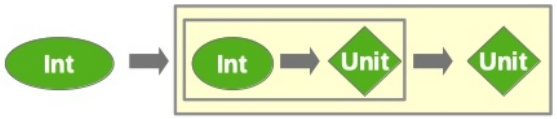

Abbildung 5.38: Die "flatMap-Signatur" der Funktionen.

Der Typ $(Int \Rightarrow Unit) \Rightarrow Unit$ kann verallgemeinert werden: $Unit$ verallgemeinern wir zu einem beliebigen *Return*-Typ R und Int zu einem beliebigen Argument-Typ A. Das Ganze ist dann unsere *Continuation-Monade* $Cont$:

$Cont[R, A] = (Int \Rightarrow R) \Rightarrow R$

Mit einem auf $Unit$ fixierten R:

$ContToUnit[A] = Cont[Unit, A]$

Mit den passenden Funktionen ist $ContToUnit$ eine Monade.

Eine Umschlag-Klasse

Definieren wir also eine Umschlagklasse für $(Int \Rightarrow R) \Rightarrow R$ und sehen, ob wir sie "monadisch" machen können:

```
case class Cont[R, A](kToR: (A => R) => R){

  // zum bequemen Auspacken der Funktion
  def apply(ka: A => R): R = kToR(ka)

  def map[B](f: A => B): Cont[R, B] = ???

  def flatMap[B](f: A => Cont[R, B]) = ???
}
```

Die Verwendung sieht schon mal ganz vernünftig aus:

```
def pure[A,R]: A => Cont[R, A] =
  a => Cont(kA => kA(a))

type ContToUnit[A] = Cont[Unit, A]

def mult2(x: Int): ContToUnit[Int] =
  Cont(kInt => kInt(x*2))

def add1(x: Int): ContToUnit[Int] =
```

```
  Cont(kInt => kInt(x+1))

def compute42 : ContToUnit[Int] =
  for (
      x <- pure(20);
      y <- add1(x);
      z <- mult2(y))
   yield z

def print42 = compute42(x => println(x))
```

Alle Typen passen. Jetzt fehlen nur noch die Definition der Methoden map und flatMap.

Eine map-Implementierung für die Umschlag-Klasse

map nimmt eine Funktion

f: A => B

und macht damit aus einem Cont[R, A] ein Cont[R, B]. Diese Transformation erzeugt mit dem in this gekapselten Wert

kToR: (A ⇒ R) ⇒ R)

ein neues und wieder einzukapselndes

kToR: (B ⇒ R) ⇒ R)

Mit map soll also etwas konstruiert werden, das mit einer Fortsetzungsfunktion

kB: B ⇒ R

zu einem Ende mit R führt. Dazu kann es ein f und ein *kToR* nutzen:

f: A => B

kToR: (A ⇒ R) ⇒ R.

Das *kToR* ist das "eigene", die eingepackte Funktion. Dazu wird

this.kToR

mit

a => kb(f(a))

gefüttert. kb wird dabei in die verschachtelte Struktur hineingeschoben. Das ist die umständliche Implementierung der Verkettung, die wir automatisieren wollten. (Siehe Abb. 5.39.) Insgesamt haben wir folgende Implementierung von *map*:

```
def map[B](f: A => B): Cont[R, B] =
   Cont( (kb: B => R) => kToR((a:A) => kb( f(a)) ))}
```

Eine flatMap-Implementierung für die Umschlag-Klasse

flatMap geht in ähnlicher Weise vor. Das Ziel ist wieder das Einfädeln einer Stufe einer Verarbeitungskette in eine verschachtelte Struktur. flatMap nimmt eine Funktion

f: A => Cont[R, B]

und erzeugt damit und mit this.kToR

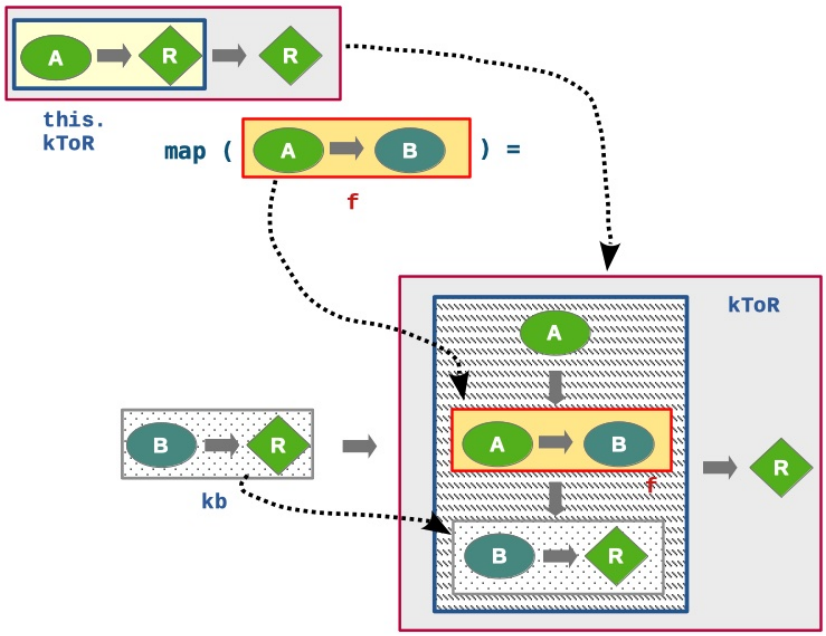

Abbildung 5.39: map in Aktion.

ein neues kToR: (B => R) => R, das in Cont verpackt wird. (Siehe Abb. 5.40.)

```
def flatMap[B](f: A => Cont[R, B]): Cont[R, B] =
  Cont( (kb: B => R) => kToR(a => f(a)(kb)) )
```

Damit haben wir eine vollständige Definition unserer Continuation-Monade:

```
case class Cont[R, A](kToR: (A => R) => R){

  def apply(ka: A => R): R = kToR(ka)

  def map[B](f: A => B): Cont[R, B] = //flatMap(a => pure(f(a)))
    Cont( (kb: B => R) => kToR((a:A) => kb( f(a)) ))

  def flatMap[B](f: A => Cont[R, B]): Cont[R, B] =
    Cont( (kb: B => R) => kToR(a => f(a)(kb)) )

}

def pure[A,R]: A => Cont[R, A] =
  a => Cont(kA => kA(a))
```

Die Monaden-Gesetzen verlangen, dass

```
def map[B](f: A => B): Cont[R, B] =
  flatMap(a => pure(f(a)))
```

Wir haben map aber schon mit inhaltlichen Überlegungen definiert als:

```
def map[B](f: A => B): Cont[R, B] =
 Cont( (kb: B => R) => kToR((a:A) => kb( f(a)) ))
```

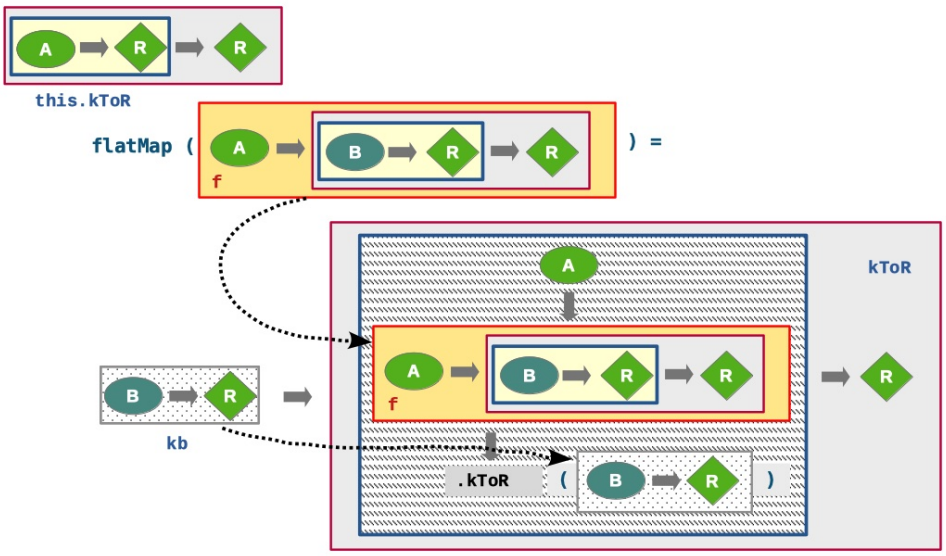

Abbildung 5.40: flatMap in Aktion.

Wenn Cont tatsächlich monadisch sein soll, dann müssen beide Definitionen äquivalent sein. In der Tat sind sie es:

$$
\begin{aligned}
& flatMap(a => pure(f(a))) \\
& = flatMap(a => Cont(kA => kA(f(a)))) \\
& = Cont(kb => kToR(a => (a => Cont(kA => kA(f(a))))(a))(kb)) \\
& = Cont(kb => kToR(a => (a => kb(f(a)))(a))) \\
& = Cont(kb => kToR(a => kb(f(a)))
\end{aligned}
$$

Unser kleiner Test offenbart wie gewünscht die Antwort auf alle essentiellen Fragen: 42. (Siehe Abb. 5.40.)

```
def pure[A,R]: A => Cont[R, A] =
  a => Cont(kA => kA(a))

type ContToUnit[A] = Cont[Unit, A]

def mult2(x: Int): ContToUnit[Int] =
  Cont(kInt => kInt(x*2))

def add1(x: Int): ContToUnit[Int] =
  Cont(kInt => kInt(x+1))

def compute42 : ContToUnit[Int] =
  for {
      x <- pure(20);
      y <- add1(x);
      z <- mult2(y)}
    yield z
```

```
def print42 = compute42(x => println(x))

print42 // -> 42
```

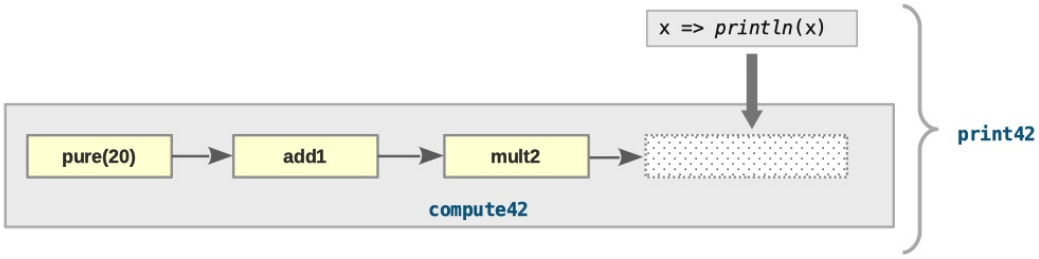

Abbildung 5.41: Lösung des Rätsels der Welt mit einer Continuation-Monade.

Jetzt noch ein weiteres Beispiel für eine Berechnung von universeller Bedeutung:

```
def computeFac: Int => Cont[Unit, Int] = x =>
  if (x == 0)
    pure(1)
  else
    for( y <- computeFac(x-1) )
      yield x * y

def printfac(n: Int) = computeFac(n)(res => println(res))

printfac(10) // -> 3628800
```

Continuation-Monade in Future transformieren

Cont[R, A] repräsentiert / kapselt eine Funktion kToR: (A ⇒ R) ⇒ R. Das Argument von kToR ist der Verarbeiter des von Cont[R, A] produzierten Ergebnisses. Der Verarbeiter kann vieles mit dem Ergebnis tun, beispielsweise kann mit ihm ein Versprechen (Promise) erfüllt werden. (Siehe Abb. 5.42.)

```
import scala.concurrent.{Future, Promise, ExecutionContext}
import ExecutionContext.Implicits.global

// Cont => Future
def futureRes[A](c: Cont[Unit, A]): Future[A] = {
  val promise = Promise[A]
  c(a => promise.success(a))
  promise.future
}

for (
  x <- futureRes(fac(10))
) yield println(x)

 // warte auf das Ende der asynchronen Aktion
Thread.sleep(1000)
```

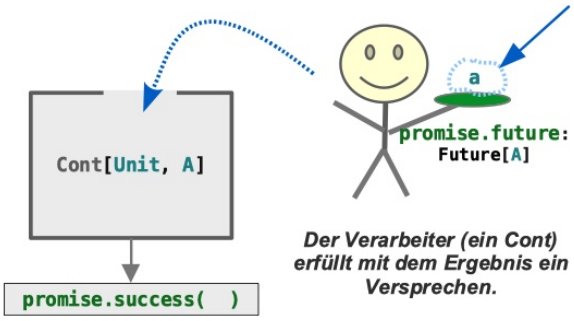

Abbildung 5.42: Continuations und die Erfüllung eines Versprechens.

5.5 Monadisches Backtracking

Im letzten Kapitel haben wir schon dargelegt, dass der *Continuation Passing Style (CPS)* genutzt werden kann, um eine funktionale Variante des imperativen *Backtrackings* mit Exceptions zu realisieren. Hier in diesem Kapitel wird der Gedanke weiter ausgearbeitet und der CPS mit einer *Continuation*-Monade realisiert.

5.5.1 Backtracking und Nichtdeterministische Programme

Nichtdeterministische Algorithmen sind Algorithmen, bei denen nicht in jeder Situation die Wahl der als nächstes auszuführenden Aktion vollständig durch den Algorithmus festgelegt ist, sondern von irgendwelche äußeren Einflüssen (mit) bestimmt wird. Streng genommen, d.h. im Sinne der Algorithmentheorie, sind nichtdeterministische Algorithmen gar keine Algorithmen. Es ist aber trotzdem gelegentlich sinnvoll mit ihnen zu arbeiten, beispielsweise als Mittel zur vereinfachten Definition "richtiger", also deterministischer Algorithmen.

Bei der Spezifikation von richtigen, deterministischen Algorithmen wird Nichtdeterminismus auf zwei Arten eingesetzt:

- *Don't care*-Nichtdeterminismus : Es ist für den Ablauf des algorithmischen Verfahrens egal welche Entscheidung getroffen wird. Die Auswahl kann darum als Implementierungsdetail angesehen und bei der Definition des Verfahrens vernachlässigt werden.
- *Don't know*-Nichtdeterminismus: Es ist nicht bekannt, was die richtige Wahl ist, es muss danach gesucht werden. Die Definition eines Suchverfahrens ist aber ein algorithmisches Detail, das man einer Implementierung überlassen. Bei der Definition des Verfahrens wird es der Einfachheit und Übersichtlichkeit halber vernachlässigt.

Bein *Don't care*-Nichtdeterminismus ist die fehlende "Determiniertheit" irrelevant und wird dem Implementierer überlassen. Beim *Don't know*-Nichtdeterminismus wird sie durch einen nicht genau spezifizierten Suchprozess geliefert.

Ein bekanntes Beispiel *Don't know*-Nichtdeterminismus sind *nicht-deterministische endliche Automaten* mit denen sich auf eine vereinfachte Art Automaten definieren lassen, die in (deterministische!) Algorithmen zur Erkennung regulärer Sprachen umgeformt werden können. (Siehe Abb. 5.43.)

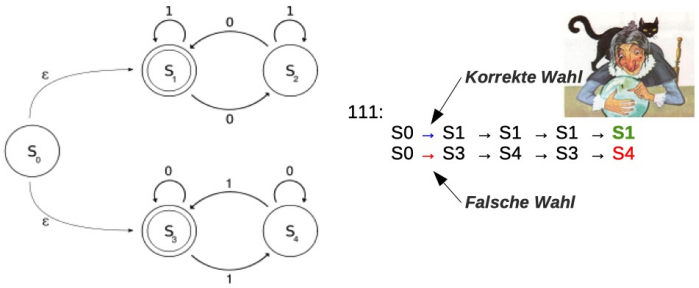

Abbildung 5.43: Dont't Know Nichtderteminismus bei einem NDEA.

Die Grundidee bei dieser Art von *Nichtdeterminismus* ist folgende: Der Algorithmus enthält Entscheidungspunkte, an denen der richtige unter mehreren möglichen Werten gewählt wird

und man tut so, als sei es möglich die richtige Wahl zu treffen und mit dieser den Algorithmus fortführen zu können. Tatsächlich akzeptiert man aber, das nicht möglich ist und die richtige Wahl mit irgendeinem Verfahren gesucht werden muss.

Ein solcher *Don't know* nichtdeterministischer Algorithmus spezifiziert eine erschöpfende Suche in folgender Form:

```
teilLösung <~ leereLösung
while ! istFertigeLösung (teilLösung) {
     s <~ Orakel ( {s | s ist möglicher nächster Schritt} )
     teilLösung = teilLösung + s
}
```

Das Orakel wählt "mit magischen Mitteln" einen jeweils richtigen nächsten Schritt derart, dass das Ziel einer vollständigen Lösung erreicht wird. Das ist, wie bereits gesagt, erst mal kein Algorithmus! Bei einem Algorithmus muss der nächste Schritt stets determiniert sein. Es handelt sich um die abstrakte Spezifikation eines Algorithmus, bei dem nach der / einer richtigen Folge von Auswahlschritten gesucht wird. Die Organisation der Suche bleibt als "Implementierungsdetail" unerwähnt.

Die Organisation der Suche kann, wie bei der Implementierung regulärer Ausdrücke, einem speziellen Verfahren überlassen werden. Sie kann auch als Sprachfeature zur Verfügung stehen (wie in Prolog), oder es muss bei der Implementierung des Algorithmus "händisch" realisiert werden.

Als Sprachfeature nimmt die Organisation einer als Nichtdeterminismus formulierten Suche in der Regel folgende linguistische Gestalt an:

```
teilLösung <- leereLösung
while ! istFertigeLösung (teilLösung) {
     s <- choose ({s | s ist möglicher nächster Schritt})
     teilLösung = teilLösung + s
     if (nichtOK(teilLösung))
       fail
}
```

Die choose / fail-Konstrukte können auf unterschiedliche Art implementiert werden. Eine naheliegende ist *Backtracking*. Eine andere ist die systematische Suche nach allen Lösungen.

Statt choose / fail als Sprachfeature oder als Spezifikation eines Algorithmus zu sehen, die entweder bei der Implementierung einer Programmiersprache, oder bei der Implementierung eines Algorithmus' in reale Konstrukte umgesetzt werden, kann man auch einen Mittelweg wählen, und choose / fail als Bestandteil einer *generischen (Bibliotheks-) Komponente* realisieren. Eine solche Komponente kann dann auf unterschiedliche Arten implementiert und zu unterschiedlichen Zwecken verwendet werden.

Bevor wir uns aber an die Abstraktion und Vereinheitlichung geben, betrachten wir noch einmal kurz die beiden konkreten Varianten zur Implementierung des Nachtdeterminismus: die funktionale Suche nach allen Lösungen und die imperative Suche nach einer Lösung, der mit Fortsetzungsfunktionen ihr imperativer Charakter ausgetrieben werden kann.

Pythagoreische Tripel: funktionale Suche nach allen Lösungen

Als einfaches Beispiel nehmen wir die Bestimmung aller oder eines *pythagoreischen Tripels*. Ein Tripel von natürlichen Zahlen (x, y, z) wird "pythagoreisch" genannt, wenn $x^2 + y^2 = z^2$. Das

Tripel $(3, 4, 5)$ ist pythagoreisch. Mit einem choose / fail-Mechanismus kann die Suche nah einem solchen Tripel übersichtlich beschrieben werden:

```
def triple() = {
  val i = choose(2, 3, 4, 5)
  val j = choose(2, 3, 4, 5)
  val k = choose(2, 3, 4, 5)
  if (i*i + j*j != k*k) fail()
  succeed(i, j, k)
}
```

Man kann diesen Code als Spezifikation eines Suchalgorithmus' betrachten. Eine mögliche *funktionale Implementierung* wäre dann:

```
type Triple = (Int, Int, Int)

// Alle Lösungen -funktional
def allPTriples: List[Triple] =
  for (i <- List(1, 2, 3, 4, 5);
       j <- List(1, 2, 3, 4, 5);
       k <- List(1, 2, 3, 4, 5);
       if i*i + j*j == k*k)
    yield (i, j, k)

val pTripleList = allPTriples //List((3,4,5), (4,3,5))
```

Diese funktionale Implementierung bestimmt *alle* Lösungen. Der Suchraum wird komplett traversiert und die Menge aller Lösungen in einer Liste gesammelt. Die Implementierung von fail ist der Verzicht darauf, ein Tripel zur Lösungsmenge hinzuzufügen. Die Implementierung von success ist yield.

imperative Suche nach einer Lösung

Ein andere, diesmal *imperative Implementierung* ist:

```
// Eine Lösung -imperativ
def aPtriple: Triple = {
  for (i <- List(1, 2, 3, 4, 5))
    for (j <- List(1, 2, 3, 4, 5))
      for (k <- List(1, 2, 3, 4, 5)) {
        if (i*i + j*j == k*k)
          return (i, j, k) // return !!
      }
  throw new Exception("no triple found")
}

val pTriple = aPtriple //(3,4,5)
```

Die Implementierung von fail ist: Weiter suchen, bzw. ein throw. Die Implementierung von success ist return.

Backtracking mit Continuations: Funktionale Suche nach einer Lösung

Die Suche nach allen Lösungen lässt sich mit einer bequem mit einer *For-Comprehension* formulieren. So etwas hätten wir auch gerne für die Suche nach *einer* Lösung: einen schönen Algorithmus, der bequem und übersichtlich mit einer *For-Comprehension* ausgedrückt werden kann.

Im letzten Kapitel haben wir gesehen, wie der imperative Geist verscheucht und die Suche nach einer Lösung in eine funktionale Form gebracht werden kann. Wir brauchen dazu einen starken Exorzismus, den *Continuation Passing Style*. Im letzten Abschnitt haben wir uns bereits mit dem CPS und seiner Beziehung zur Backtracking beschäftigt. Das Beispiel dort waren die n-Damen. Wir übernehmen die Überlegungen und wenden sie auf das einfachere Problem hier an. Eine funktionale Variante der (Backtracking-) Suche nach einer Lösung ist:

```
type SuccessCont = Triple => Unit
type FailureCont = () => Unit

def OK(chosen: List[Int]): Boolean = chosen match {
  case i :: j :: k :: _ => i * i + j * j == k * k
}

extension (lst: List[Int]) {
  def toTriple: Triple = lst match {
    case i :: j :: k :: _ => (i, j, k)
  }
}

def triple_CPS(ksucc: SuccessCont, kfail: FailureCont): Unit = {

  def solve(chosen: List[Int],
     ksucc: SuccessCont,
     kfail: FailureCont): Unit =

    if (chosen.length == 3) {
      if (OK(chosen)) {
        ksucc(chosen.toTriple)
      } else {
        kfail()
      }
    } else {
      solve(
        chosen.appended(1),
        ksucc,
        () => solve(
          chosen.appended(2),
          ksucc,
          () => solve(
            chosen.appended(3),
            ksucc,
            () => solve(
              chosen.appended(4),
              ksucc,
              () => solve(
                chosen.appended(5),
                ksucc,
                kfail
              )
```

```
          )
        )
      )
    )
  }

  solve(Nil, ksucc, kfail)
}

triple_CPS(triple => println(triple), () => println("Failure"))
// Ausgabe (3,4,5)
```

Die Schleife der imperativen Variante muss zu einer Verschachtlung der Aufrufe werden, da der Aufruf der Fortsetzungsfunktion die letzte Aktion sein muss: Wir haben in der funktionalen Version ja kein return, mit dem aus einer Aufrufhierarchie heraus gesprungen werden könnte.

Das ganz kann natürlich mit einer Rekursion zur Steuerung der Verschachtlung noch etwas geschönt werden:

```
def triple_CPS(ksucc: SuccessCont, kfail: FailureCont): Unit = {

  def solve(chosen: List[Int], ksucc: SuccessCont, kfail: FailureCont):
      Unit = {
    def loop(x: Int) : Unit =
      if (x > 5)
        kfail()
      else {
        val chosen_extended = chosen.appended(x)
        solve(chosen_extended, ksucc, () => loop(x+1) )
      }

    if (chosen.length == 3) {
      if (OK(chosen)) ksucc(chosen.toTriple)
      else kfail()
    } else {
      loop(1)
    }
  }

  solve(Nil, ksucc, kfail)
}
```

5.5.2 Plus-Monade: Nichtdeterminismus als generische Komponente

Die Mechanismen des Nichtdeterminismus:

Auswahl: choose

Erfolg: succeed

Fehlschlag: fail

können in einer Typklasse gekapselt werden, die wir *Plus-Monade*, MonadPlus, nennen: Eine Monde die um eine plus-Operation erweitert wird.

Sowohl die (funktionale) Suche nach allen Lösungen, als auch die Suche nach einer Lösung mit funktionalem *Backtracking* sollten dann als Instanzen von MonadPlus definierbar sein.

Tun wir also einfach mal so, als gäbe es die gewünschte Typklasse. Der "nicht-derministische" Algorithmus für unsere Tripel wäre dann:

```
type Triple = (Int, Int, Int)

def triple[M[_]: MonadPlus](): M[Triple] =
  for ( i <- MonadPlus[M].choose(2, 3, 4, 5);
        j <- MonadPlus[M].choose(2, 3, 4, 5);
        k <- MonadPlus[M].choose(2, 3, 4, 5);
        r <- (if (i*i + j*j == k*k) MonadPlus[M].succeed(Tuple3(i, j, k))
           else MonadPlus[M].fail))
    yield r
```

MonadPlus muss eine Monade sein und zudem noch fail / succeed und choose definieren:

```
trait Functor[F[_]] {
  extension[A, B] (fa: 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]]
}

trait MonadPlus[M[_]] extends Monad[M] {

  def fail[A]: M[A]

  // succeed ist pure
  def succeed[A](a:A) = pure(a)

  // choose wird auf plus reduziert
  def choose[A](alternatives: List[A]): M[A] =
    alternatives.foldLeft(fail[A])(
      (acc, i) => acc plus pure(i)
    )

  // der Bequenlichkeit halber
  def choose[A](alternatives: A*): M[A] =
    choose(alternatives.toList)

  extension[A, B] (x: M[A]) {
    def plus(y: M[A]): M[A]
  }
}
object MonadPlus {
```

```
  def apply[M[_]: MonadPlus] = summon[MonadPlus[M]]
}
```

Die intuitive Definition der Funktionen ist:

- succeed nimmt einen Wert und macht ihn zu einer (eventuell nur vorerst) erfolgreichen Wahl. Der Wert wird in die Plus-Monade gepackt, dazu haben wir aber schon das pure der Monade.
- choose wählt einen aus vielen Werten. Wir reduzieren es auf eine Basis-Operation plus die zwischen *zwei* Werten wählt.

  Die zweite Variante von choose dient der Bequemlichkeit beim Aufruf.
- plus wählt zwischen zwei Werten. Die letzte Wahl, wenn es sonst nichts mehr zu wählen gibt, ist natürlich fail.

5.5.3 List als Instanz der Plus-Monade

Die Menge aller Lösungen kann als Liste erzeugt werden. Dazu muss List als Instanz von MonadPlus definiert werden. Das ist erstaunlich einfach und offensichtlich:

```
given MonadPlus[List] with {

// Die Lösungsmenge enthält a
  def pure[A](x: A): List[A] = List(x)

  // Die Lösungsmenge ist leer
  def fail[A]: List[A] = Nil

  extension [A, B](xs: List[A]) {
    // List enthält schon die passenden map und flatMap
    def flatMap(f: A => List[B]): List[B] = xs.flatMap(f)
    override def map(f: A => B) = xs.map(f)

  // Die Lösungsmengen werden vereinigt
    def plus(y: List[A]): List[A] = xs ::: y
  }

}
```

Der generischen Algorithmus Backtracking-Algorithmus ist dann:

```
def triple_A[M[_]: MonadPlus](): M[Triple] =
  for (
        i <- MonadPlus[M].choose(2, 3, 4, 5);
        j <- MonadPlus[M].choose(2, 3, 4, 5);
        k <- MonadPlus[M].choose(2, 3, 4, 5);
        r <- if (i*i + j*j == k*k)
               MonadPlus[M].succeed((i, j, k))
             else MonadPlus[M].fail)
    yield r
```

oder auch in einer rekursiven Variante:

```
def triple_B[M[_]: MonadPlus](): M[Triple] = {
```

```
  def OK(chosen: List[Int]): Boolean = chosen match {
   case i :: j :: k :: _ => i * i + j * j == k * k
  }

  extension (lst: List[Int]) {
   def toTriple: Triple = lst match {
    case i :: j :: k :: _ => (i, j, k)
   }
  }

  def solve(ijk: List[Int]): M[Triple] =
   if (ijk.length < 3) {
    for (x <- MonadPlus[M].choose(2,3,4,5);
        s <- solve(ijk ++ List(x)))
     yield s
   } else
    if (OK(ijk))
     MonadPlus[M].succeed(ijk.toTriple)
    else MonadPlus[M].fail

  solve(Nil)
 }
```

Und auch die Positionierung der feindlichen Damen als generisches Backtracking:

```
 def queens[M[_]:MonadPlus](n: Int): M[List[Int]] = {

  def Ok(board: List[Int]): Boolean =
   (for (i <- 0 until board.length;
         j <- i + 1 until board.length
       ) yield {
    val (x, y) = (board(i), board(j))
    val d = j -i
    !(x == y || y == x -d || y == x + d)
   }).find(_ == false)
    .getOrElse(true)

  val alternatives = (0 until n).map(List(_)).toList

  def solve(chosen: List[Int]): M[List[Int]] =
   if (Ok(chosen)) {
    if (chosen.length == n) {
     MonadPlus[M].pure(chosen)
    } else {
     for (i: List[Int] <- MonadPlus[M].choose(alternatives);
          s: List[Int] <- solve(chosen ::: i))
      yield s
    }
   } else MonadPlus[M].fail

  solve(Nil)
 }
```

Die generischen Algorithmen liefern in Gegenwart der Liste als Instanz der Typklasse MonadPlus die erwarteten Lösungen:

```
val allTriples_A = triple_A() // List((3,4,5), (4,3,5))
val allTriples_B = triple_B() // List((3,4,5), (4,3,5))
val fourQueens = queens(4) // List(List(1, 3, 0, 2), List(2, 0, 3, 1))
```

5.5.4 Funktionales Backtracking als Plus-Monade

Nicht ganz so offensichtlich, wie der Typ List, ist die Definition des funktionalen Backtrackings *mit einer Lösung*, die Continuation basierte Suche, als Instanz von MonadPlus. Werfen wir dazu noch einmal einen Blick auf die Backtrack-Lösung der n-Damen von weiter oben:

```
def NQueens(n: Int): Unit = {

  def solve(t: List[Int],
            ksucc: List[Int] => Unit,
            kfail: => Unit): Unit = {

    def loop(x: Int): Unit = {
      if (x == n)
        kfail // Backtrack
      else {
        val t_extended = t ++ List(x)
        if (Ok(t_extended)) {
          solve(t_extended, ksucc, loop(x + 1))
        } else {
          loop(x + 1)
        }
      }
    }

    if (t.length == n)
      ksucc(t)
    else
      loop(0) // ~for (x <- 0 until n)
  }

  solve(Nil, lst => println(lst), println("Failed"))
}
```

Der Algorithmus arbeitet mit zwei Continuations: einer, ksucc, für den Erfolgsfall und einer, kfail, für den Misserfolgs- / Sackgassen-Fall. Bei jeder Entscheidungsmöglichkeit wird wird die die erste Wahl getroffen und mit den restlichen die kfail erweitert.

Die etwas übersichtlichere Suche nach einen pythagoreischen Tripel in dieser Form ist:

```
type Triple = (Int, Int, Int)

def OK(chosen: List[Int]): Boolean = chosen match {
  case i :: j :: k :: _ => i * i + j * j == k * k
}

extension (lst: List[Int]) {
  def toTriple: Triple = lst match {
    case i :: j :: k :: _ => (i, j, k)
  }
}
```

```
type SuccessCont = Triple => Unit
type FailureCont = () => Unit

def triple(ksucc: SuccessCont, kfail: FailureCont): Unit = {

  def solve(ijk: List[Int], ksucc: SuccessCont, kfail: FailureCont): Unit
     = {
    def loop(x: Int) : Unit =
      if (x > 5)
        kfail()
      else {
        val chosen_extended = ijk.appended(x)
        solve(chosen_extended, ksucc, () => loop(x+1) )
      }

    if (ijk.length == 3) {
      if (OK(ijk)) ksucc(ijk.toTriple)
      else kfail()
    } else {
      loop(1)
    }
  }
  solve(Nil, ksucc, kfail)
}
```

Mit entfalteter Rekursion ist das:

```
def triple(ksucc: SuccessCont, kfail: FailureCont): Unit = {

  def solve(ijk: List[Int], ksucc: SuccessCont, kfail: FailureCont): Unit
     =
    if (ijk.length == 3) {
      if (OK(ijk)) ksucc(ijk.toTriple)
      else kfail()
    } else {
      solve(
        ijk.appended(1),
        ksucc, () => solve(
          ijk.appended(2),
          ksucc, () => solve(
            ijk.appended(3),
            ksucc, () => solve(
              ijk.appended(4),
              ksucc, () => solve(
                ijk.appended(5),
                ksucc, kfail )))))
    }
    solve(Nil, ksucc, kfail)
  }
```

Hier hört man geradezu wie der Code nach einer monadischen Form mit flatMap schreit. Wenden wir also unsere üblichen Schritte an, um triple zu “monadisieren”.

Die Continuations in einer Umschlag-Klasse

Die beiden Continuations können in eine Klasse ContDuo gepackt werden:

```
type SuccessCont = Triple => Unit
type FailureCont = () => Unit

case class ContDuo (ksucc: SuccessCont, kfail: FailureCont)

def triple(contduo: ContDuo): Unit = {

  def solve(ijk: List[Int], kduo: ContDuo): Unit = ...

  solve(Nil, contduo)
}

triple(ContDuo(triple => println(triple), () => println("Failure")))
```

Etwas Curry hilft auch immer:

```
type SuccessCont = Triple => Unit
type FailureCont = () => Unit
case class ContDuo (ksucc: SuccessCont, kfail: FailureCont)


def triple: ContDuo => Unit = {

  def solve(ijk: List[Int]) : ContDuo => Unit = {
    def loop(x: Int): ContDuo => Unit =
      if (x > 5)
        kduo => kduo.kfail()
      else
        kduo => solve(ijk.appended(x))(ContDuo(kduo.ksucc, () =>
            loop(x+1)(kduo)) )

    if (ijk.length == 3) {
      if (OK(ijk)) kduo => kduo.ksucc(ijk.toTriple)
      else kduo => kduo.kfail()
    } else {
      kduo => loop(1)(kduo)
    }
  }

  contDuo => solve(Nil)(contDuo)
}
```

Noch eine Typdefinition ändert nicht viel:

```
type SuccessCont = Triple => Unit
type FailureCont = () => Unit
case class ContDuo (ksucc: SuccessCont, kfail: FailureCont)

type ContDuoToUnit = ContDuo => Unit

def triple_CPS: ContDuoToUnit = {

  def solve(ijk: List[Int]) : ContDuoToUnit = ...
```

```
    contDuo => solve(Nil)(contDuo)
  }
```

Packen wir jetzt die Werte des Typs ContDuoToUnit in eine Umschlagklasse, mit dem Namen BT (wie *BackTracking*):

```
type SuccessCont[T] = T => Unit
type FailureCont = () => Unit

case class BT[A](
     kduo: (ksucc: SuccessCont[A], kfail: FailureCont) => Unit)

type Triple = (Int, Int, Int)

def triple(): BT[Triple] = ???
```

Damit ist noch nicht viel gewonnen. Wenn allerdings BT die Funktionalität einer *Plus-Monade* hätte, z.B.:

```
case class BT[A](
  kduo: (ksucc: SuccessCont[A], kfail: FailureCont) => Unit
)

extension[A, B] (x: BT[A]) {
  def flatMap(f: A => BT[B]): BT[B] = ???
  def map(f: A => B): BT[B] = ???
  def plus(y: BT[A]): BT[A] = ???
}

def chooseBT[A](alternatives: List[A]): BT[A] = ???
def chooseBT[A](alternatives: A*): BT[A] = chooseBT(alternatives.toList)
def pureBT[A](a: A): BT[A] = ???
def succeedBT[A](a: A): BT[A] = pureBT(a)
def failBT[A]: BT[A] = ???
```

dann könnten wir schöneren Code schreiben:

```
def triple(): BT[Triple] =
  for ( i <- chooseBT(2, 3, 4, 5);
        j <- chooseBT(2, 3, 4, 5);
        k <- chooseBT(2, 3, 4, 5);
        r <- if (i*i + j*j == k*k) succeedBT((i, j, k))
             else failBT[Triple] )
    yield r
```

oder auch

```
def triple(): BT[Triple] = {

  def OK(chosen: List[Int]): Boolean = chosen match {
    case i :: j :: k :: _ => i * i + j * j == k * k
  }
  extension (lst: List[Int]) {
    def toTriple: Triple = lst match {
      case i :: j :: k :: _ => (i, j, k)
    }
```

```
  }

  def solve(ijk: List[Int]): BT[Triple] =
    if (ijk.length < 3) {
      for (x <- chooseBT(2,3,4,5);
           s <- solve(ijk ++ List(x)))
        yield s
    } else {
      if (OK(ijk)) succeedBT(ijk.toTriple)
      else failBT[Triple]
    }

  solve(Nil)
}
```

Dann also frisch ans Werk. Beginnen wir mit den einfachen Dingen.

pure und fail

pure repräsentiert einen Wert, der der Success-Continuation übergeben werden kann. fail aktiviert die Failure-Continuation:

```
def pureBT[A](a: A): BT[A] =
  BT[A]( (ksucc, kfail) => ksucc(a) )

def failBT[A]: BT[A] =
  BT[A]( (ksucc, kfail) => kfail() )
```

map und flatMap

map delegieren wir der Einfachheit halber in der üblichen Art an flatMap. flatMap verkettet Berechnungen und setzt dabei eine erfolgreiche Berechnung fort:

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

  // f wird auf a angewendet und dann
  // geht es weiter mit ks und kf
  def flatMap(f: A => BT[B]): BT[B] =
    BT(
      (ks: SuccessCont[B], kf:FailureCont) =>
       x.kduo(
         (a:A) => f(a).kduo(ks, kf),
         kf)
    )

  def map(f: A => B): BT[B] =
    flatMap((a: A) => pureBT(f(a)))

  def plus(y: BT[A]): BT[A] = ???
}
```

choose und plus

choose wird wieder auf plus reduziert:

```
def chooseBT[A](alternatives: List[A]): BT[A] =
  alternatives.foldLeft(failBT[A])(
    (acc, i) => acc.plus(pureBT(i))
  )

def chooseBT[A](alternatives: A*): BT[A] =
  chooseBT(alternatives.toList)
```

plus repräsentiert alternative Ausführungen, wenn die erste fehlschlägt, nimm die zweite:

```
extension[A, B] (x: BT[A]) {
  ...
  // Probiere es mit der einen Berechnung, x.
  // Wenn das schief geht, dann probiere es mit der anderen, y.
  def plus(y: BT[A]): BT[A] =
    BT[A](
      (ks: SuccessCont[A], kf:FailureCont) =>
        x.kduo(
          a => ks(a),
          () => y.kduo(ks, kf)
        )
    )
}
```

Die Typen passen, das ist meist schon fast eine Garantie, dass der Algorithmus korrekt ist.

In unserem Fall leider nicht. Ein einfacher Test zeigt schon die Fehlfunktion:

```
tripleA().kduo( (res: Triple) => println(res), () => println("failure"))
    // failure
```

flatMap und plus harmonieren nicht

Ein wenig Nachdenken über die Regeln, die bei der Anwendung der Funktionen gelten müssen, zeigt den Fehler. flatMap muss "von links über plus distribuieren". Es muss also gelten:

(x plus y) flatMap f = (x flatMap f) plus (y flatMap f)

Es muss also egal sein ob ich eine Wahl zwischen x und y treffe und dann mit f weiter mache, oder ob ich eine Wahl treffe zwischen "x und dann f" oder "y und dann f". Dieses "von links distribuieren" ist bei den Definitionen so wie sie sind nicht gewährleistet:

```
def flatMap(f: A => BT[B]): BT[B] =
  BT(
    // f ist ohne Einfluss auf kf, es wird in Failure-Fall ignorieert
    (ks: SuccessCont[B], kf:FailureCont) =>
      x.kduo(
        (a:A) => f(a).kduo(ks, kf),
        kf)
  )
```

Um dem gerecht zu werden, muss BT umdefiniert werden.

Redefinition von BT

Damit flatMap von links über plus distribuieren kann, werden bei einem plus die beiden Verzweigungen aufgehoben. Damit hat dann flatMap die Möglichkeit sein f in beiden Varianten einzubringen.

```
enum BT[A] {

  // das Ergebnis einer plus Operation
  case Plus(x: BT[A], y: BT[A])

  // die bisherige Form
  case Cont(k2U: (SuccessCont[A], FailureCont) => Unit )

  def apply(ks: SuccessCont[A], kf: FailureCont): Unit = this match {

    case Cont(k2U) => k2U(ks, kf)

    case Plus(x, y) =>
      ( (ks: SuccessCont[A], kf: FailureCont) =>
         x.apply( a => ks(a), () => y(ks, kf) )
      ).apply(ks, kf)
  }

}
```

Bei einer Anwendung müssen die beiden Varianten unterschieden werden. Cont verhält sich wie BT in seiner bisherigen Form. Bei der Plus-Variante kann auf die Alternativen x und y zugegriffen werden. y wird dann zur Failure-Continuation von x.

Damit können jetzt plus und flatMap mit einem korrektem Zusammenspiel definiert werden:

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

  def flatMapBT(f: A => BT[B]): BT[B] = x match {

    case Cont(k2U) =>
      Cont(
        (ks, kf) =>
          k2U((a:A) =>
            f(a) match {

              case Cont(k2U) => k2U(ks, kf)

              case Plus(x, y) =>
                x(
                  a => ks(a),
                  () => y(ks, kf) // y geht in die Failure-Continuation ein
                )
            }, kf)
      )

    case Plus(x, y) =>
      Plus[B](x.flatMap(f), y.flatMapBT(f))
  }

  def mapBT(f: A => B): BT[B] = flatMapBT((a: A) => pureBT(f(a)))
```

```
  // plus speichert die Alternativen in einem Plus
  def plusBT(y: BT[A]): BT[A] = Plus(x, y)

}
```

Bei den anderen Funktionen ändert sich nichts:

```
def chooseBT[A](alternatives: List[A]): BT[A] =
  alternatives.foldLeft(failBT[A])(
    (acc, i) => acc.plusBT(pureBT(i))
  )

def chooseBT[A](alternatives: A*): BT[A] = chooseBT(alternatives.toList)

def pureBT[A](a: A): BT[A] =
  Cont[A]( (ksucc, kfail) => ksucc(a) )

def failBT[A]: BT[A] =
  Cont[A]( (ksucc, kfail) => kfail() )

def succeedBT[A](a: A): BT[A] = pureBT(a)
```

Um Konflikte mit den Namen der generischen Funktionen der Plus-Monade zu vermeiden, wurden alle Funktionen mit einem "BT" markiert. Damit ist es möglich BT als Instanz der Typklasse zu erklären, ohne dass die generische und die konkrete BT-Variante von map oder flatMap verwechselt werden.

Die Definition von BT als Instanz der Typklasse MonadPlus ist jetzt eine Kleinigkeit:

```
given MonadPlus[BT] with {

  def pure[A](x: A): BT[A] = pureBT(x)

  def fail[A]: BT[A] = failBT[A]

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

    def flatMap(f: A => BT[B]): BT[B] = x.flatMapBT(f)

    override def map(f: A => B) = x.mapBT(f)

    def plus(y: BT[A]): BT[A] = x plusBT (y)
  }
}
```

Damit sind die generischen Backtrack-Algorithmen mit einem Continuation-basiertem funktionalen Backtracking ausführbar. Beispielsweise die Suche nach n verträglichen Damen:

```
def queensGen[M[_]:MonadPlus](n: Int): M[List[Int]] = {

  def Ok(board: List[Int]): Boolean =
    (for (i <- 0 until board.length;
          j <- i + 1 until board.length
          ) yield {
      val (x, y) = (board(i), board(j))
      val d = j -i
```

```
      !(x == y || y == x -d || y == x + d)
    }).find(_ == false)
      .getOrElse(true)

  val alternatives = (0 until n).map(List(_)).toList

  def solve(chosen: List[Int]): M[List[Int]] =
    if (Ok(chosen)) {
      if (chosen.length == n) {
        MonadPlus[M].pure(chosen)
      } else {
        for (
            i: List[Int] <- MonadPlus[M].choose(alternatives);
            s: List[Int] <- solve(chosen ::: i))
          yield s
      }
    } else MonadPlus[M].fail

  solve(Nil)
}

def q[M[_]: MonadPlus]: M[List[Int]] = queensGen(4)

q.apply( (res: List[Int]) => println(res), () => println("failure"))
// ~> List(1, 3, 0, 2)
```

Das generische Backtracking kann also auch mit einem funktionalen Äquivalent der imperativen Exception-basierten Lösung instantiiert werden.

Im Sinne des algebraischen Entwurfs haben wir eine DSL für Backtracking-Probleme entworfen. Die Ausdrucksmittel erlauben es Lösungen generisch als Ausdrücke dieser DSL zu formulieren, die mit unterschiedlichen Instanzen der Typklassen – hier der Backtrakingmonde – als Implementierungen ausgeführt werden können. Das ist das, was wir weiter oben *algebraischen Entwurf* genannt haben.

5.6 Monadentransformer

5.6.1 Monaden kombinieren, nicht mischen

Gemischte Monaden

In einer *For-Comprehension* können unterschiedlichen Monaden nicht gemischt werden. Man bleibt besser "sortenrein". Das kommt beim Einstieg in die funktionale Programmierung für viele oft oft überraschend, denn schnell ist so etwas hingeschrieben wie:

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

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

def stringToFactorsString(str: String): List[String] =
  for (
      x <- StringToLong(str);
      f <- factors(x)) //Found: List[String], Required: Option[Any]
    yield f.toString
```

Aber man lernt dann schnell, dass List und Option nicht einfach in "funktionalen For-Schleifen" gemischt werden können. Man muss sich entscheiden, entweder List oder Option zu verwenden, aber nicht beides:

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

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)) // OK !
    yield f.toString
```

Der Grund ist klar. Eine *For-Comprehension* (ein Funktorblock) wird auf map und flatMap abgebildet und die beiden bewegen sich im Kontext eines (!) Funktors.

Das ist in vielen Fällen problemlos oder zumindest akzeptabel. Gelegentlich will oder muss man aber das Ergebnis einer Funktion wie stringToLong von Option in List ändern. So müssen eventuell Option und List bei vorgegebenen Funktionen wie etwa

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

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

akzeptiert werden und

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

hat sich gefälligst daran anzupassen.

Nun, man kann Funktionen mit unterschiedlichen Funktoren als Ergebnistyp schon in einem

Funktorblock mischen. Es wird nur etwas anstrengend. Der eine muss dann an den anderen angepasst werden:

```
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 { // List[Int] ~> Option[List[Int]]
        case Nil => None;
        case l => Some(l.map( _.toString))
      })
    yield f
```

Nennen wir das also besser *"Kombinieren von Monaden"* und nicht "mischen".

Monaden kombinieren: Beispiel Future und Option

Unterschiedlichen Monaden kann man also nicht einfach so mischen. Mit etwas Aufwand kann man sie aber *kombinieren*. Als Beispiel betrachten wir die Kombination von Future und Option. Am Anfang steht dabei die Erkenntnis, dass es ein Problem gibt:

```
import scala.concurrent.Future
import concurrent.ExecutionContext.Implicits.global

// 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)) // Future[String], Required: Option[Any]
    yield s"$l is ${if(!b) "not" else ""} prime"
```

Um das Problem zu lösen, wird in einem zweiten Schritt ein Typ definiert, der beide Monaden, hier Option und Future, kombiniert. Dabei gibt es zwei Möglichkeiten entweder:

- Future[Option[·]] oder
- Option[Future[·]]

Bei der Kombination wird ein monadischer Hülltyp definiert. Also entweder

- FutureOption[·] oder
- OptionFuture[·]

Dieser muss dann noch mit den angepassten map- und flatMap-Methoden ausgestattet werden.

Bevorzugen wir aus irgendeinem Grund FutureOption und passen isPrime und toLong an diesen Typ an, dann haben wir erst einmal:

```
type FutureOption[A] = Future[Option[A]]

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

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

def f(str: String): FutureOption[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")
    }
}
```

Ohne eine Anpassung der Basisoperationen des Funktor-Blocks wäre das:

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

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

def f(str: String): FutureOption[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")
    }
```

```
}
```

Etwas schöner wird die Sache, wenn wir FutureOption mit den Funktor-Methoden ausstatten. Beispielsweise durch die Definition einer Umschlagklasse mit den passenden Methoden. Aus

```
type FutureOption[A] = Future[Option[A]]
```

wird dann:

```
case class FutureOption[A](fo: Future[Option[A]]) {

  def map[B](f: A => B): FutureOption[B] =
    FutureOption(fo.map(_.map(f)))

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

Die Verlagerung der Anpassungsoperationen in den Typ macht die Nutzung übersichtlicher und schöner:

```
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 einer Helferfunktion lift kann auf eine Anpassung der Funktionen isPrime und toLong verzichtet werden, ohne dass dabei Bequemlichkeit verloren geht:

```
def isPrime(n: Long): Future[Boolean] = ...

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

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

// lift: Future ~> FutureOption
extension[A] (fut_a: Future[A]) {
```

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

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"
}
```

Monaden kombinieren: Future und List

Als zweites Beispiel für eine Kombination von Monaden betrachten wir List und Future. Auch diese beiden benötigt man öfter mal in einer *for-Comprehension*. Ein einfaches Beispiel ist:

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

Definieren wir eine Klasse in der List und Future kombiniert werden, dann stimmen zumindest schon mal die Typen:

```
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] = ???
}
```

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

Jetzt fehlt nur noch eine passende Definition der Methoden. Die lift-Funktionen 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 liftL: FutureList[A] = FutureList(Future.successful(lst_a))
}

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

Die Definition von flatMap ist nicht ganz so offensichtlich. Die entscheidenden Aktionen finden auf den verpackten Werten statt. Verschaffen wir uns also erst mal einen Überblick über das, was die Signaturen fordern. (Siehe Abb. 5.44.)

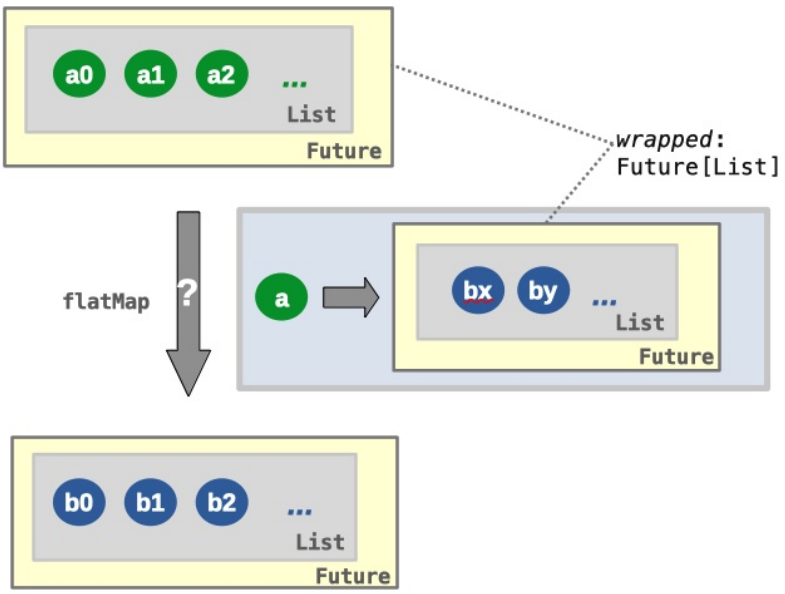

Abbildung 5.44: faltMap in Aktion.

Die b0, b1, etc. in Abbildung 5.44 sind eine Zusammenfassung der Ergebnisse der Anwendun-

gen von f auf die a*i* der Ausgangsliste. Es ist naheliegend die Ergebnisse von f(a*i*) einfach zu verketteten. (Siehe Abb. 5.45.)

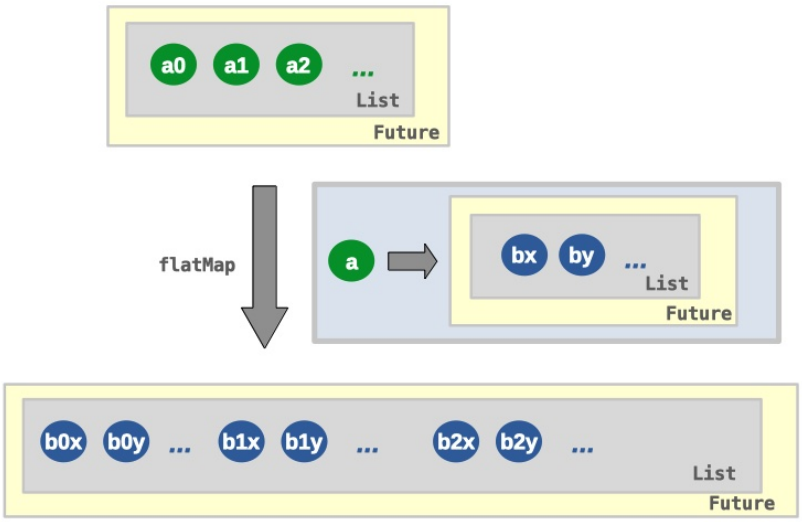

Abbildung 5.45: flatMap: Verkette die Ergebnisse in einer Liste.

Das ist dann recht schnell in Code umgesetzt, etwas umständlich, aber für einen in Continuation-Gymnastik geschulten Geist keine Herausforderung, die nicht zu meistern ist:

```
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)
      )
    )
  )
}
```

Insgesamt und mit den lift-Operationen haben wir:

```
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 liftL: FutureList[A] = FutureList(Future.successful(lst_a))
}

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

Jetzt können wir uns die Primfaktoren bequem asynchron berechnen lassen:

```
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)
```

Monaden kombinieren: Reader und Option

Wie wir weiter oben gesehen haben, ist die Reader-Monade bestens geeignet für die Auswertung von Ausdrücken mit definierten Konstanten. Für die Fehlerbehandlung eignet sich dagegen bestens der Typ Option, eine andere Monade. Für Ausdrücke mit definierten Konstanten und eventuellen Fehlern wäre eine Kombination von Reader und Option darum eine angemessene und naheliegende Monade.

Beginnen wir mit der Fehlerbehandlung, lassen dazu erst mal die Konstanten weg und kommen so vorerst ohne Reader aus. Die Terme sind:

```
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._
```

Eine generische Auswertungsfunktion für solche Terme ist :

```
def eval[M[_]: MonadWithFilter](term: Term): M[Int] =
  term match {
    case Literal(v) =>
      MonadWithFilter[M].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
  }
```

M ist dabei eine filterbare Monade. Wir müssen ja eine Division durch 0 ausschließen.

```
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]]
}

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

Selbstverständlich ist Option eine filterbare Monade:

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

given MonadWithFilter[Option] with {
  def pure[A](x: A): Option[A] =
    Some(x)
  extension [A, B](x: Option[A]) {
    def flatMap(f: A => Option[B]): Option[B] = x.flatMap(f)
    override def map(f: A => B) = x.map(f)
    def withFilter(f: A => Boolean): Option[A] = x.filter(f)
  }
}
```

Das harmoniert alles bestens und eval liefert die erwarteten Ergebnisse:

```
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
```

Lassen wir die Division weg und vermeiden damit fehlerhafte Terme und nehmen stattdessen Konstanten hinzu, dann ist ein Reader die geeignete Monade. Die Terme mit Konstanten und ohne Division sind:

```
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._
```

Ihre Auswertung mit einem Reader ist unproblematisch. Definieren wir zunächst einen passenden Reader:[7]

```
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]
```

[7] Reader wurden weiter oben ausführlich behandelt. Wir übernehmen die Funktionen einfach von dort.

```
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)
  }
}
```

Die Auswertungsfunktion ist dann:

```
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
}
```

Ein Anwendungsbeispiel ist:

```
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
```

Kombination: Reader + Option

Bei der Kombination der beiden Monaden entscheiden wir uns für Option in Reader, nicht Reader in Option! – Die Verschachtlung Reader[Option]] sieht es irgendwie "natürlicher" aus, als umgekehrt Reader[Option]].

```
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]]

given MonadWithFilter[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) )
  }
}
```

In den Methoden map und faltMap muss None abgefangen werden. In flatMap brauchen wir dazu eine explizite Fallunterscheidung. In map macht reicht dazu die Delegation an das map von Option.

Die Auswertungsfunktion für Terme mit Konstanten und eventuell auch Division durch Null:

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

ist damit:

```
def eval(term: Term): EnvReaderOpt[Int] =
  term match {
    case Literal(v) =>
      MonadWithFilter[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
  }
```

Ein Anwendungsbeispiel ist:

```
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
```

5.6.2 Monadenkombination: Möglichkeiten und Grenzen

Man sieht, dass mit etwas Bastelarbeit vieles möglich ist. Bevor wir uns mit der Frage beschäftigen, ob und wie die Basteleien reduziert werden können, werfen wir einen kurzen Blick auf die Frage, welche Monaden grundsätzlich wie miteinander kombiniert werden können. Wenn also M_1 und M_2 Monaden sind, können sie dann immer miteinander kombiniert werden? Wir hatten die Beispiele

$$\mathsf{Future} + \mathsf{Option} \Rightarrow \mathsf{Future}[\mathsf{Option}[\,\cdot\,]]$$

$$\mathsf{Future} + \mathsf{List} \Rightarrow \mathsf{Future}[\mathsf{List}[\,\cdot\,]]$$

$$\mathsf{Reader} + \mathsf{Option} \Rightarrow \mathsf{Reader}[\mathsf{Option}[\,\cdot\,]]$$

Wie wäre es mit einer umgekehrten Verschachtlung, also beispielsweise

$$\mathsf{Future} + \mathsf{List} \Rightarrow \mathsf{List}[\mathsf{Future}[\,\circ\,]]$$

oder

$$\mathsf{Reader} + \mathsf{Option} \Rightarrow \mathsf{Option}[\mathsf{Reader}[\,\circ\,]]$$

Hmm, klingt irgendwie falsch in dieser Kombination, das kann aber auch täuschen.

Allgemein: Ist

$$M_1[M_2[\,\cdot\,]]$$

eine Monade wenn M_1 und M_2 Monaden sind, und sind

$$M_1[M_2[\,\cdot\,]] \text{ und } M_2[M_1[\,\cdot\,]]$$

äquivalent? Testen wir einfach mal mit einem weiteren Beispiel die beiden Alternativen der Kombination zweier Monaden M_1 und M_2 zu $M_1[M_2[\,\cdot\,]]$ bzw. $M_2[M_1[\,\cdot\,]]$.

ListTry vs. TryList

List und Try können zu ListTry:

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

und zu TryList:

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

kombiniert werden. Gelingt es uns beide zu Instanzen der Typklasse Monade zu machen und wenn ja: zeigen sie dann ein äquivalentes Verhalten?

Fangen wir mal mit ListTry an:

```
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))
}
```

Die Definitionen von map und flatMap sind recht offensichtlich. Die übergebene Funktion f wird an die Listeninhalte weiter gereicht. Bei flatMap muss, wie weiter oben bei Option, wieder explizit auf Failure geprüft werden.

Der Bequemlichkeit halber haben wir wieder lift-Funktionen definiert.

Ein kurzer Test zeigt, dass sich ListTry wie erwartet verhält:

```
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

println(res.wrapped.mkString("\n"))
/* liefert das Erwartete:
   Success(1)
```

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

Jetzt zur umgekehrten Verschachtlung TryList :

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

Die lift-Operationen, pure und map sind recht offensichtlich:

```
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) ) )
  }

}
```

Die Definition von flatMap ist nicht ganz so klar.

```
given Monad[TryList] with {
 ...
  extension[A, B](x: TryList[A]) {

    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))
      )
    }

    ...
  }
}
```

x.wrapped hat den Typ Try[List[A]]. Wir haben ein

f: A ⇒ TryList[B]

und können

x.wrapped mit einer Funktion vom Typ List[A] ⇒ Try[List[B]] zu einem Try[List[B]] flatMapen. Diese Funktion sollte natürlich f nutzen. (Siehe Abb. 5.46.)

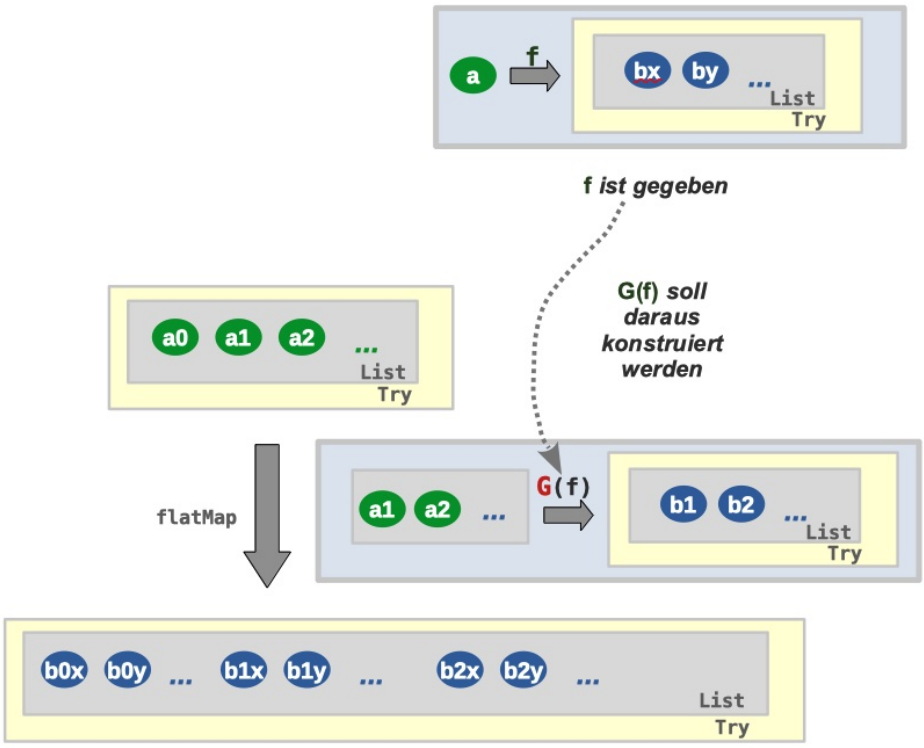

Abbildung 5.46: faltMap auf Try[List[A]].

Wir brauchen also ein

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

Eine naheliegende Implementierung von G besteht darin, die die verschachtelten Werte von f zu verketten:

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

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

}
```

Ein kurzer Test zeigt allerdings, dass mit TryList ein anderer Effekt erreicht wird, als mit ListTry:

```
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

val resStr = // java.lang.NumberFormatException: For input string: "two"
  res.wrapped match {
    case Success(lst) => lst.mkString("\n")
    case Failure(e) => e
  }
```

Wir sehen also, dass $M_1[M_2[\circ]]$ und $M_2[M_1[\circ]]$ *nicht* unbedingt immer *äquivalent* sind und unter Umständen nur eine der beiden Varianten zu einem gewünschten Verhalten führt. Welche Variante eher den Erwartungen entspricht, kann unter Umständen diskutiert werden. Die erste Variante unseres Beispiels verhält sich aber sicher weniger überraschend.

Try ist wie Option und Either eine *Pass / Fail* – Monade: Die "Iteration" ist bei ihnen eine Verkettung von Operationen, die mit mit dem ersten Fehlschlag beendet wird. Die Konsequenz ist, dass Try, Option, Either nur als "innere Monaden" sinnvoll sind, es sei denn man möchte nicht sofortigen Abbruch der "laufenden Aktion", sondern einen kompletten Fehlschlag aller Aktionen.

Die Kombination von Monaden ist mühsam und nicht immer möglich oder in sinnvoller Art möglich. Mit etwas Bastelarbeit können aber etliche, wenn auch nicht alle Monaden in der einen, der anderen, oder in beiden Verschachtlungen kombiniert werden.

5.6.3 Monadentransformer

Mit dem Konzept der *Monadentransformer* soll die Bastelarbeit bei der Kombination vereinfacht werden.

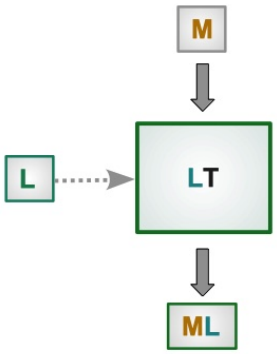

Abbildung 5.47: Das Prinzip der Monadentransformer.

Die Idee der Monadentransformer ist folgende: Angenommen, wir haben zwei Monaden L und M, die kombiniert werden sollen. Eine der beiden – L – sei die Basis. Zur Basismonade L wird

ein *Transformer* $LT[\,\cdot\,]$ definiert. Um eine Monadenkombination ML zu erzeugen wird der Transformer LT auf M angewandt und man erhält die Kombination LM. (Siehe Abb. 5.46.)

$L \rightsquigarrow LT[\,\cdot\,]$
$LT[M] \Rightarrow ML$

Ist beispielsweise $L =$ Try und $M =$ List dann liefert der Transformer TryT angewendet auf List die Kombination ListTry.

Mit einem Monadentransformer kann man sich viele mühsame Einzelfall-Kombinationen ersparen. Man füttert wechselnde Ms in eine fixen Transformer LT und kann so auf einfache Art ein LM erzeugen.

Transformer TryT

Betrachten wir als Beispiel den Transformer TryT. Er kann als Verallgemeinerung der Kombination ListTry gesehen werden, wobei List durch eine beliebige Monade M ersetzt wird.

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

  def flatMap[B](f: A => TryT[M, B]): TryT[M, B] =
    TryT(
      wrapped.flatMap { // Bearbeite erfolgreiche Listenelemente mit f
        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)))
}
```

Die passende pure-Methode ist

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

und die Lift-Funktionen können ebenfalls durch Ersetzen von List durch eine Monade M aus ihren konkreten Vorbildern abgeleitet werden:

```
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))
}
```

Ein kleiner Test zeigt, dass die Verallgemeinerung sich 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
  }
}

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) */
```

Transformer OptionT

Durch Verallgemeinerung von beispielsweise FutureOption (siehe oben) können wir eine Definition des Transformer OptionT entwickeln:

```
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(_)))
}
```

Eine Testanwendung ist:

```
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"
}

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)
```

Transformer ReaderT

Einen Transformer ReaderT erhalten wir beispielsweise durch eine Verallgemeinerung von OptionReader (siehe weiter oben) in Option. Als Beispiel nehmen wir wieder die Auswertung von Termen mit Division und Konstanten. Die Funktionalität des Readers wird für die Fehlerbehandlung gebraucht. Es gibt zwei Arten möglicher Fehler:

- Division durch Null.
- Undefinierte Konstante.

Die Behandlung der Division durch Null wird erleichtert, wenn die Monade M, mit der der Reader kombiniert wird, filterbar ist. Für die Behandlung der undefinierten Konstanten ist es nützlich wenn wir direkt auf eine "leere" Monade (einen "Fehler") zugreifen können.

```
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) )
}
```

In flatMap ist

```
val ma: M[A] = run(z)
ma.flatMap(a => f(a).run(z))
```

eine Verallgemeinerung der entsprechenden Aktion in flatmap der OptionReader-Kombination

```
run(z) match {
     case Some(a) =>
       f(a).wrapped(z)
     case None =>
       None
    }
```

Die generische Auswertungsfunktion der 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._
```

ist damit:

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

def eval[M[_]: MWithFilterZero](term: Term): ReaderT[M, Env, Int] = term
    match {

  case Literal(v) => ReaderT(env =>
    MWithFilterZero[M].pure(v))

  case Const(n) => ReaderT(
```

```
    (env: Env) =>
     if (env.isDefinedAt(n) ) MWithFilterZero[M].pure(env(n))
     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
}
```

Mit Option als Instanz der Typklasse MWithFilterZero:

```
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)
  }
}
```

können Terme mit der generischen Funktion ausgewertet werden:

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

val termValue1 = eval(term).run(Map("two" -> 2)) // Some(42)

// Division durch 0:
val termValue2 = eval(term).run(Map("two" -> 0)) // None
```

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

Transformer StateT

Die *State*-Monade stellt Berechnungen in einem veränderlichen Kontext dar und kann als eine Kombination von *Reader* und *Writer* gesehen werden:

$$Reader[S, A] \approx S \Rightarrow A \qquad Writer[S, A] \approx (A, S)$$
$$State[S, A] \approx S \Rightarrow (A, S)$$

Die *State*-Monade wird typischerweise mit einer anderen Monade kombiniert, wenn bei der Berechnung etwas schief gehen kann. *Try* oder *Option* kämen dann in Frage. Bei einer Kombination mit *Option* hätten wir dann

$$StateOption[S, A] \approx S => Option[(A, S)]$$

Eine Auswertung von Termen mit Seiteneffekten hat den Typ

```
def eval(exp: Term): State[Int, Env] = ...
```

Kann bei der Auswertung etwas schiefgehen, dann kann das zu

```
def eval(exp: Term): StateOption[Int, Env]] = ...
```

geändert werden, wobei StateOption einem Wert vom Typ S ⇒ Option[(A, S)] kapselt. Verallgemeinert man Option zu einer beliebigen Monade, dann kommt man zum Monaden-Transformer *StateT*:

```
case class StateT[M[_]: Monad, S, A](run: S => M[(A, S)]) { ... }
```

Für die Ausdrucksauswertung, die schief gehen kann, ist es sinnvoll eine Filterfunktion und eine "leeren Wert" zur Verfügung zu haben. Weiter oben hatten wir Ausdrücke mit einem Seiteffekt-behafteten inc-Operator:

```
  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._
```

Deren generische Auswertung mit einer beliebigen Fehlermonade mit Filter und 'leeren Wert" 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]]
}
```

ist dann:

```
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) ) {
         // Zugriff auf den Wert einer definierten Variablen
         MWithFilterZero[M].pure(env(name), env)
       } else {
         // Zugriff auf eine undefinierte Variable: Fehler
         MWithFilterZero[M].zero
       }
   )

  case Inc(name) => StateT(
     (env: Env) =>
       if (env.isDefinedAt(name) ) {
         val v = env(name)
         // definierte Variable: Zugriff und Erhöhung
         MWithFilterZero[M].pure(env(name), env + (name -> (v+1)))
       } else {
         // undefinierte Variable
         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 ) // Division durch 0 vermeiden
      yield v1 / v2
```

```
}
```

Fehlt nur noch die Definition von StateT. Dazu kombiniert man am besten State und Option und verallgemeinert dann Option. Das Ergebnis kann dann folgendermaßen aussehen:

```
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) ) )
}
```

Mit Option als einer Instanz von MWithFilterZero:

```
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)
  }
}
```

können Terme ausgewertet werden:

```
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 resultInState = eval(term)

val res_1 = resultInState.run( Map("x" -> 1, "y" -> 0, "z" -> 1) )
// Some((70,Map(x -> 7, y -> 2, z -> 1)))

val res_2 = val res_2 = resultInState.run(Map(Map("x" -> 1, "y" -> 0,
    "z" -> 0)))
// None (Division durch 0)

val res_3 = resultInState.run(Map("zett" -> 1))
// None (undefinierte Variable)
```

Monadentransformer werden gebraucht, wenn mehrere Monaden zusammenkommen und man sich die mühsame Arbeit der Kombination der beiden ersparen will. Selbstverständlich wird man im Entwickleralltag Monadentransformer nicht selbst mühsam und fehlerträchtig herleiten wollen. Hier ist wieder die Nutzung einer etablierten Bibliothek zu empfehlen. Deren Lösung wird korrekt, ausgefeilt und vielseitig sein. – Und damit natürlich auch wesentlich komplexer als die hier demonstrierten Prinzipien mit denen wir ja nicht mehr wollten als den Zugang zu einer solchen Bibliothek zu erleichtern.

Literaturverzeichnis

[1] Paul Chiusano and Rnar Bjarnason. *Functional Programming in Scala.* Manning Publications Co., 2014.

[2] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists.* MIT Press, 1991.

[3] Noel Welsh and Dave Grunell. *Scala With Cats.* Underscore Consulting LLP, Brighton, UK, 2017.

Index