Software-Komponenten

Th. Letschert

THM

University of Applied Sciences

Filterbare Funktoren und natürliche Transformationen

- Filtern / Filter-Gesetze
- Natürliche Transformationen

Filter: Verpacktes (eventuell) aus Verpackung entfernen

Funktor

Typkonstruktor mit map: Wende eine Funktion auf den Inhalt einer Packung an, erzeuge so eine gleichartige Packung mit transformiertem Inhalt.

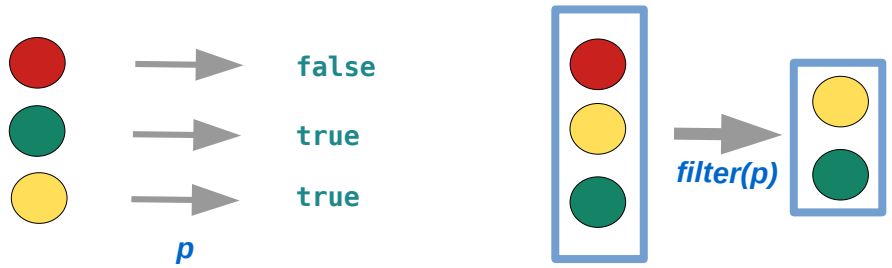
Filterbarer Funktor

Funktor mit filter: Wende eine Bewertungs-Funktion auf den Inhalt einer Packung an, erzeuge so gleichartige Packung mit eventuell weniger Inhalt.

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

Oder gleichwertig mit *for-Comprehension*:

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



Filter: Verpacktes eventuell aus der Verpackung entfernen

Filter und WithFilter in For-Ausdrücken / in Enum-ADT

Die Methode withFilter wird vom Compiler bei einer *For-Comprehension* erwartet.

Beispiel mit Enum als ADT-Implementierung:

```
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
```

Filter: Verpacktes eventuell aus der Verpackung entfernen

Was ist filterbar?

Filtern: etwas mit (eventuell) weniger Inhalt konstruieren

Filtern basiert auf der Intuition etwas auf Basis einer Bewertung zu verkleinern.

Manches kann definitiv nicht verkleinert werden.

Bei anderem kann man sich fragen, ob eine Verkleinerung möglich und das Ergebnis akzeptabel / gesetzeskonform ist.

Bei einigen Typkonstrukoren, wie List und Option, ist das offensichtlich so, bei anderen nicht.

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

*Dieses Tripel kann **nicht** gefiltert werden:*
*Wie sollte da „**weniger Inhalt**“ möglich sein?*

Filter: Verpacktes eventuell aus der Verpackung entfernen

Was ist filterbar?

Filtern: etwas mit weniger Inhalt konstruieren

Weniger, kann das vielleicht auch als „mehr None's“ interpretiert werden?

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

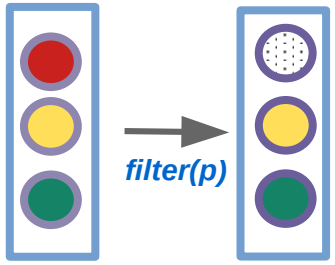
*Dieses Tripel kann **nicht** gefiltert werden!*

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

Wie sieht es mit diesem Tripel aus?

Als filterbarer Funktor

```
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))
  }
}
```



OK, da geht das Filtern natürlich

Filter: Verpacktes eventuell aus der Verpackung entfernen

Was ist filterbar?

Filterbar: Muss Summentyp sein

Filtern reduziert (eventuell) die Anzahl der Elemente, dabei wird ein Ergebnis vom gleichen Typ erzeugt wird.

Ein filterbarer Typ muss darum ein Typ mit Varianten (mit unterschiedlicher Zahl von Elementen) sein, also ein Summentyp.

Man beachte: List und Option sind Summentypen

- **Option[A] ~ Unit + A**
- **List[A] ~ Unit + (A, Unit) + (A, (A, Unit)) + ...**

Filter: Verpacktes eventuell aus der Verpackung entfernen

Was ist filterbar?

Filterbar: Filtern nur bei direkter Reduktion der Anzahl der Elemente einer Struktur?

Triple optionaler Werte haben „irgendwie“, aber nicht „direkt“ einen SummenTyp

TripleOpt[A] ~ ((A + Unit), (A + Unit), (A + Unit))

Ein Filtern bei dem (eventuell) A‘s durch None ersetzt werden, reduziert nicht die Zahl der Elemente in einer Struktur,

es reduziert lediglich die Zahl der definierten Elemente.

Darf man es trotzdem „Filtern“ nennen?

Filter-Gesetze: Natürliche Erwartungen an Filter-Prozesse

Filterbar: Filtern nur bei direkter Reduktion der Anzahl der Elemente einer Struktur?

Triple optionaler Werte haben „irgendwie“, aber nicht „direkt“ einen SummenTyp

TripleOpt[A] ~ ((A + Unit), (A + Unit), (A + Unit))

Ein Filtern bei dem (eventuell) A‘s durch None ersetzt werden, reduziert nicht die Zahl der Elemente in einer Struktur,

es reduziert lediglich die Zahl der definierten Elemente.

Darf man es trotzdem „Filtern“ nennen?

Natürlich darf man !

Man sollte

- **es aber nur dann „Filtern“ nennen,**
- **wenn es die natürlichen Erwartungen an Filter erfüllt.**

Erwartungen, die als „Filter-Gesetze“ formalisiert werden können.

Filter-Gesetze

Identität

filter(x => true) ≡ id

Filtern mit einem Prädikat, das stets true liefert, sollte der identischen Abbildung entsprechen.

Beispiel: Für MyBox gilt dies offensichtlich.

```
val v1a = FilledBox(4).filter( _ => true) // = FilledBox(4)
val v2a = EmptyBox.filter( _ => true)     // = EmptyBox
```

als *For-Comprehension*:

```
val v1b =
  for (
    x <- EmptyBox;
    if true
  ) yield x

val v2b =
  for (
    x <- FilledBox(4);
    if true
  ) yield x
```

```
enum MyBox[+A] {
  case EmptyBox           extends MyBox[Nothing]
  case FilledBox[A](a: A) extends MyBox[A]

  def filter(p: A => Boolean): MyBox[A] = withFilter(p)
  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))
    }
}
```

Filter-Gesetze

Komposition

filter(p) ; filter(q) ≡ filter(p ∧ q)

Filtern mit p und dann filtern mit q ist Gleiche wie filtern mit p ∧ q.

Für TripleOpt gilt dies. Beispiel:

```
val ta1 = TripleOpt(Some("Hallo"), Some("Welt"), Some("Ha"))
  .filter(_.length() > 2)
  .filter(_.charAt(0) == 'H')

val ta2 = TripleOpt(Some("Hallo"), Some("Welt"), Some("Ha"))
  .filter( s => s.length() > 2 && s.charAt(0) == 'H')
```

Alle Varianten liefern:

TripleOpt(Some(Hallo),None,None)

als *For-Comprehension*:

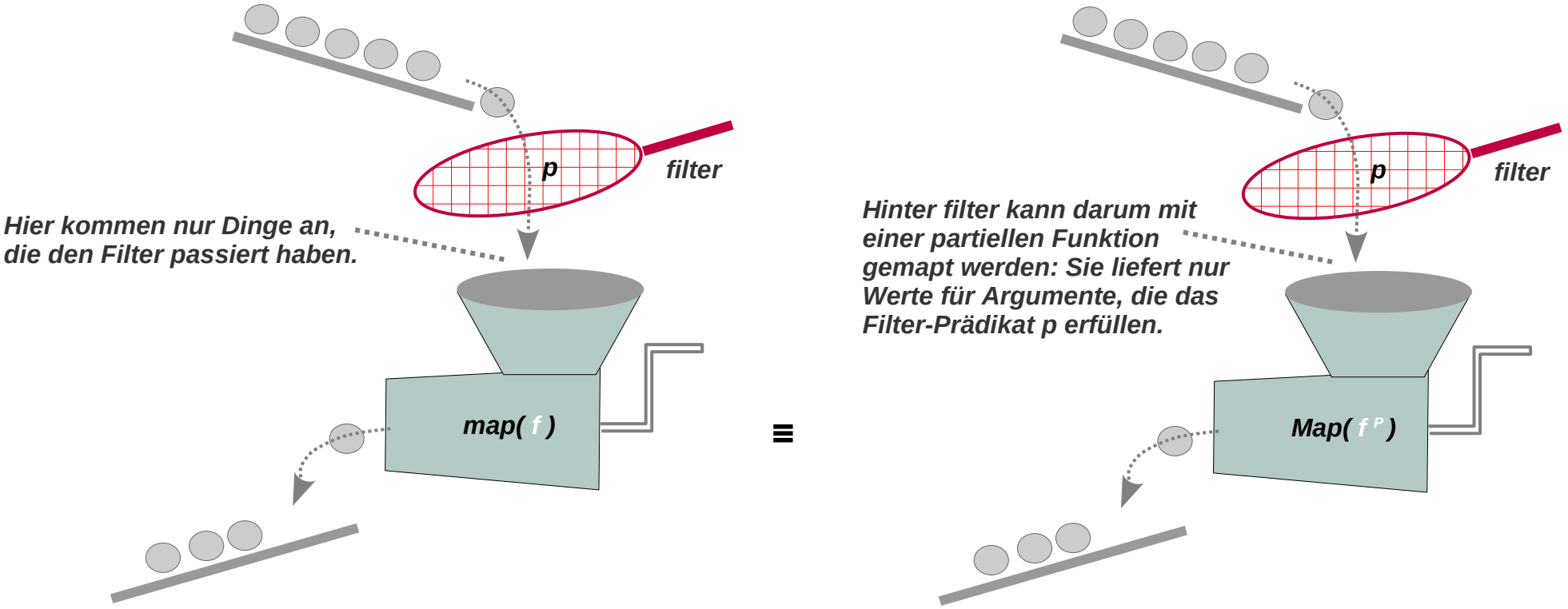
```
val tb1 =
  for ( s <- TripleOpt(Some("Hallo"), Some("Welt"), None);
        if s.length() > 2;
        if s.charAt(0) == 'H'
  ) yield s

val tb2 =
  for ( s <- TripleOpt(Some("Hallo"), Some("Welt"), None);
        if s.length() > 2 && s.charAt(0) == 'H'
  ) yield s
```

filter und map / Partielle Funktionen

Kooperation von filter und map (1)

Partielle Funktionen: Hinter *filter* darf mit einer partiellen Funktion *ge-map-t* werden

filter(p) ; map(f) ≡ filter(p) ; map(f^p)



Filter-Gesetze

filter und map / Partielle Funktionen

filter(p) ; map(f) ≡ filter(p) ; map(f^p)

Eine Testfunktion in Scala:

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

def checkPartialLaw[
  F[_]: FilterableFunctor,
  A,
  B](fa: F[A],
     p: A => Boolean,
     f: A =>B): Boolean = {

  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))
}
```

f^p : f nur dort definiert, wo p zutrifft

Filter-Gesetze

filter und map / Partielle Funktionen

filter(p) ; map(f) ≡ filter(p) ; map(f^p)

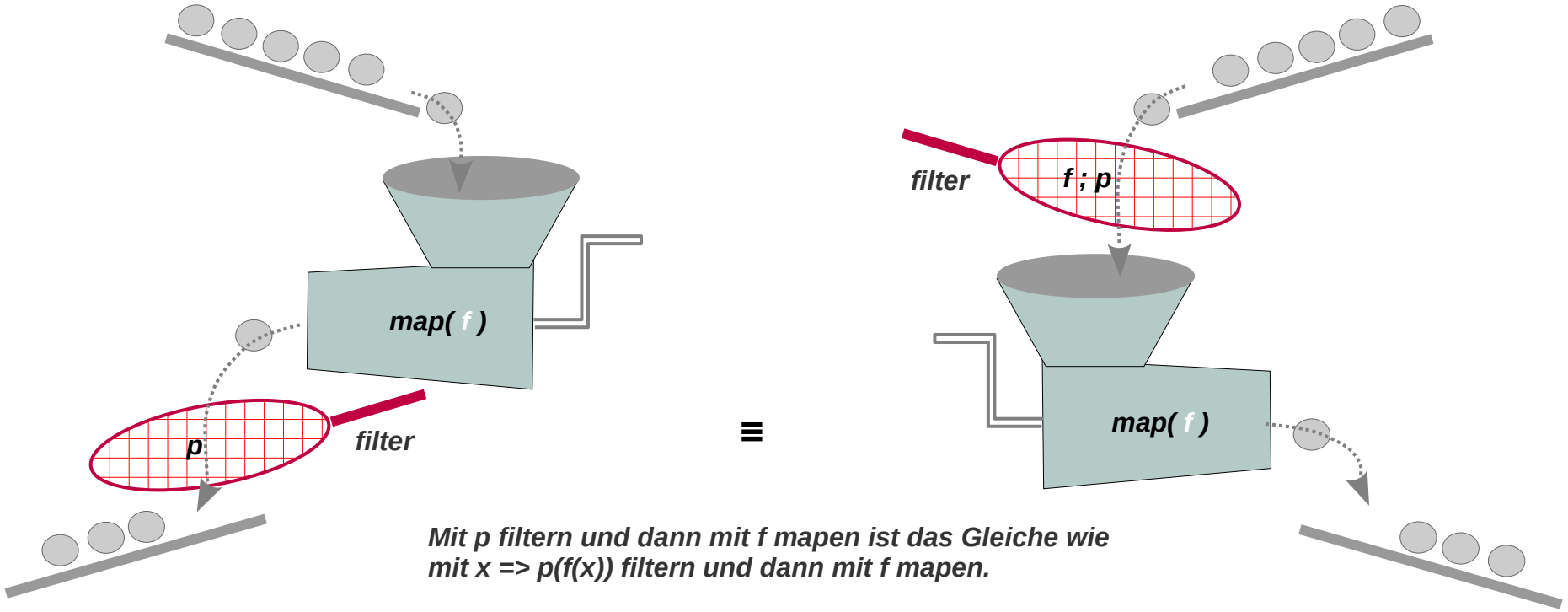
Ein Test:

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

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

filter und map / Gesetz der Natürlichkeit

Kooperation von filter und map (2)

Gesetz der Natürlichkeit: map(f) ; filter(p) ≡ filter(f ; p) ; map(f)



Mit p filtern und dann mit f mapen ist das Gleiche wie mit x => p(f(x)) filtern und dann mit f mapen.

Filter-Gesetze

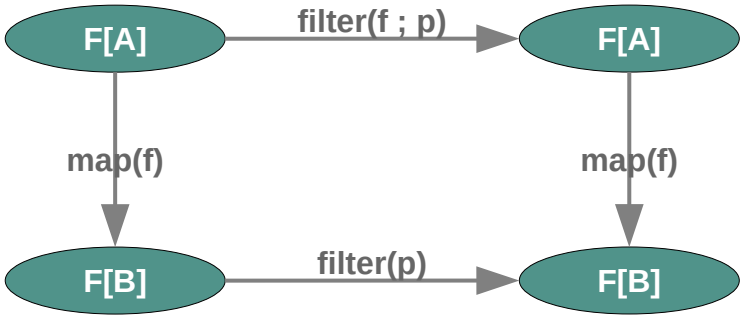
Natürlichkeits-Regel

map(f) ; filter(p) ≡ filter(f ; p) ; map(f)

map und dann filter ist das Gleiche wie mit p ○ f filtern und dann map.

In Scala:

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

def checkNturality[
  F[_]: FilterableFunctor,
  A,
  B](fa: F[A],
     p: B => Boolean,
     f: A =>B): Boolean = {
  fa.map(f).filter(p) == fa.filter(f andThen p).map(f)
}
```



Die „Natürlichkeit“ der Natürlichkeitsregel

Filter-Gesetze

Natürlichkeits-Regel

map(f) ; filter(p) ≡ filter(f ; p) ; map(f)

Beispiel TripleOpt:

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

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

Filter-Gesetze

Die Filter-Gesetze

- **Identität** filter(x => true) ≡ id
- **Komposition** filter(p) ; filter(q) ≡ filter(p ∧ q)
- **Partielle Funktion** filter(p) ; map(f) ≡ filter(p) ; map(f^p)
- **Natürlichkeit** map(f) ; filter(p) ≡ filter(f ; p) ; map(f)

werden von einem filterbarer Funktor erfüllt / müssen erfüllt werden.

Alle Containertypen wie List oder Option sind „auf natürliche Art“ filterbar.

Filterbar sind Funktoren

Ein Funktoren ist filterbar, wenn er Exemplare mit „unterschiedlich vielen Elemente“ zulässt. Dazu muss er ein Summentyp sein.

Produkttypen sind nicht filterbar. z.B.: Tupel, Tripel, etc.

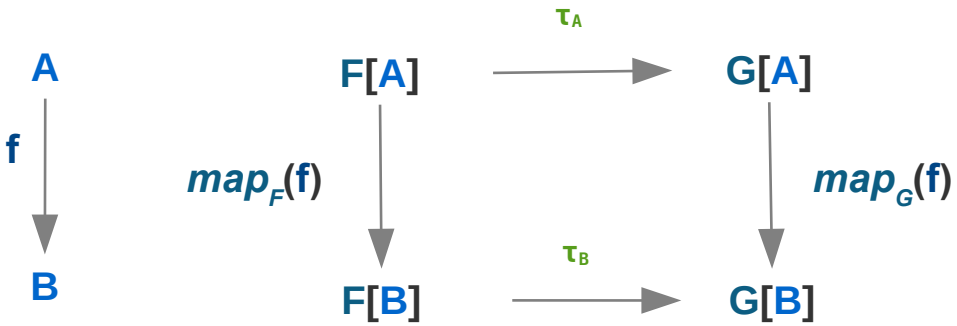
Natürliche Transformation

Generische Transformation einer Struktur (Funktor)

- in eine andere Struktur (Funktor),
- bei der alle Daten der Struktur erhalten bleiben

Die Transformation darf also nicht von den Daten in der Struktur abhängig sein.
Die Unabhängigkeit von den Daten ist die „Natürlichkeit“ der Transformation.

Formal kann diese Natürlichkeit über map definiert werden:



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

Natürliche Transformation

Natürliche Transformation als Typ-Relation

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

trait NT[F[_]: Functor, G[_]: Functor] {
  def tau[A](fa: F[A]): G[A]
}
```

Natürliche Transformationen:
Eine Typrelation, also eine Typklasse
mit mehr als einer Typvariablen

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

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

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

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

Option ~> List

Zwei natürliche Transformation.

List ~> Option

Natürliche Transformation

Beispiel: natürliche Transformation von *Option* in *List*

Natürliche Transformation: Option ~> List

Eine *natürliche Transformation* NTOptList: *Option* ~> *List* ist beispielsweise genau das, was man als „natürliche Transformation“ von Option nach List erwartet:

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

In der umgekehrten Richtung ist folgende Transformation völlig natürlich:

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

Die „Natürlichkeit“ verlangt die völlige Generizität, also die Unabhängigkeit von A.
Es ist nicht notwendig, dass der erste Wert gewählt wird.
Folgende Funktion ist also ebenfalls eine natürliche Transformation:

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

Natürliche Transformation

Beispiel: natürliche Transformation

Struktur ~NT~> List ~> String

Anwendung NT: eine Funktion die Objekte in Strings umwandelt indem sie sie zuerst mit einer (beliebigen) natürlichen Transformation in Listen transformiert:

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

Wandle eine beliebige Struktur mit einer passenden natürlichen Transformation in eine Liste und diese dann in einen String.

Natürliche Transformation

Beispiel: natürliche Transformation

Struktur ~NT~> List ~> String (2)

NT: Option ~> List

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

NT: List ~> List

```
given NT[List, List] with {
  def tau[A](lst: List[A]): List[A] = lst
}

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

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],[]]
```

Beispiel: natürliche Transformation

Struktur ~NT~> List ~> String (3)

Alternative Variante von *format*: *summon* satt Übergabe eines Zeugen

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

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

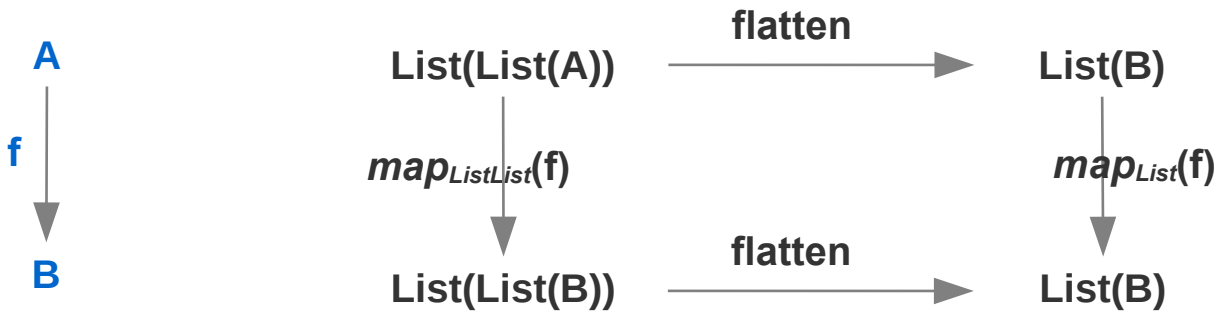
Natürliche Transformation

Beispiel: Flatten

flatten, die Umwandlung

- von Listen von Listen
- in Listen

ist eine natürliche Transformation `List[List[_]] ~> List[_]`



Natürliche Transformation

Beispiel: Flatten

***flatten*, die Umwandlung von Listen von Listen in Listen, ist eine natürliche Transformation**

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

Flatten als natürliche Transformation

Die Funktion format ist unverändert

Natürliche Transformation

Beispiel: Reverse

***reverse*, die Umkehrung von Listen, ist eine natürliche Transformation**

```
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]
```