Software-Komponenten

Th. Letschert

THM

University of Applied Sciences

Funktoren

- map: Verpacktes verarbeiten
- Funktor-Gesetze: vernünftiges map
- Funktor-Beispiele
- Contra-Funktoren

Map: Strukturinhalte transformieren

Verpacktes verarbeiten: map

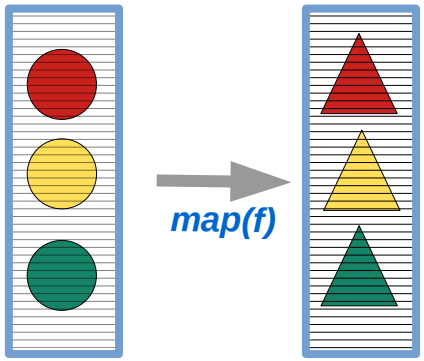
Idee von map: Wende eine Funktion auf den Inhalt eines Containers / einer Struktur an, erzeuge so gleichartigen Container mit transformiertem Inhalt.

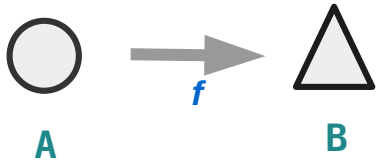
Map als Interface:

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

case 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)
```





WithMap: Etwas mit Inhalt vom Typ A das mit f: A => B in etwas gleichartiges mit Inhalt vom Typ B transformiert werden kann.

Map: Strukturinhalte transformieren

Verpacktes verarbeiten

map als OO-Abstraktion (generisch durch Subtyp-Polymorphismus):
Klassen mit einer map-Methode die kovariant im Ergebnistyp ist.

Muss in jeder Ableitung der selbe Typ sein!

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

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)
```

```
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))
}
```

***Notwendig: Kovariante Überschreibung** des Ergebnistyps* **WithMap** *!*
Das kann nur informal gefordert, und nicht vom Typsystem erzwungen werden.

Map: Strukturinhalte transformieren

Verpacktes verarbeiten

map als funktionale Abstraktion (Typklasse)

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

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

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))
  }
}
```

```scala
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
  }
}
```

Falsche Definitionen von map werden vom Compiler zurück gewiesen.

Map: Strukturinhalte transformieren

Verpacktes verarbeiten

map als funktionale Abstraktion (Typklasse)

oder so, wenn Infix- / Methoden-Syntax gewünscht ist:

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

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

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

val triple1 = Triple("Hallo", "Welt", "!")
val triple2 = triple1 map ( (x: String) => x.length )
```

Map gehört zu Typkonstruktoren

Vernünftiges map

Map-Intuition: Verpacktes mit Funktion in einer zu erwarteten Art verarbeiten

Wer f kennt, sollte eine zutreffende Vorstellung der Wirkung von map(f) haben:

- **SW wird sich ansonsten nicht erwartungsgemäß verhalten.**
- **Vor allem wichtig wenn es sich um Erwartungen von Anwendungs-Programmierern an Bibliothekscode handelt – den er meist nicht liest und wenn doch, dann eventuell nicht versteht!**

```
val a = List(x,y,z).map(f) // List(f(x), f(y), f(z))
val b = Try(x).map(f)      // Try(f(x))
val c = Future(x).map(f)   // Future(f(x))
```

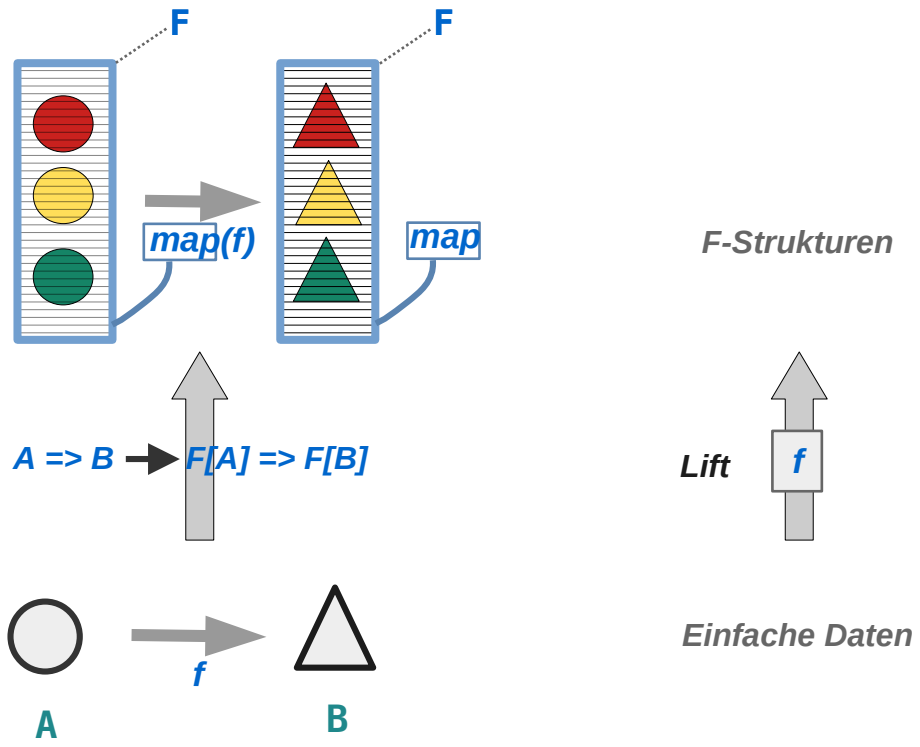
map(f) tut das was von ihm zu erwarten ist

Map gehört zu Typkonstruktoren

Map als „Liften“ einer Funktion

```
f : A => B =>> map(f): F[A] => F[B]
```

Diese Fähigkeit zum Liften hängt an F, einem generischen Typ F / Typkonstruktor F

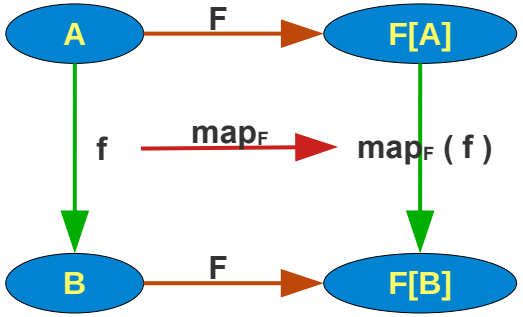


Map gehört zu Typkonstruktoren

F und map_F sind Abbildungen

```
F   : T       =>  F[T]
```
Abbildung auf der Ebne der Typen

```
mapF: A => B => F[A] => F[B]
```
Abbildung auf der Ebne der Werte

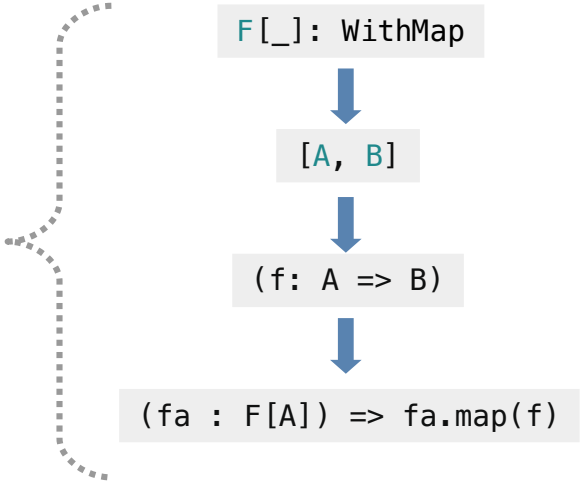



```scala
trait WithMap[F[_]] {
  extension[A, B] (fa: F[A]) def map(f: A => B): F[B]
}

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)

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

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



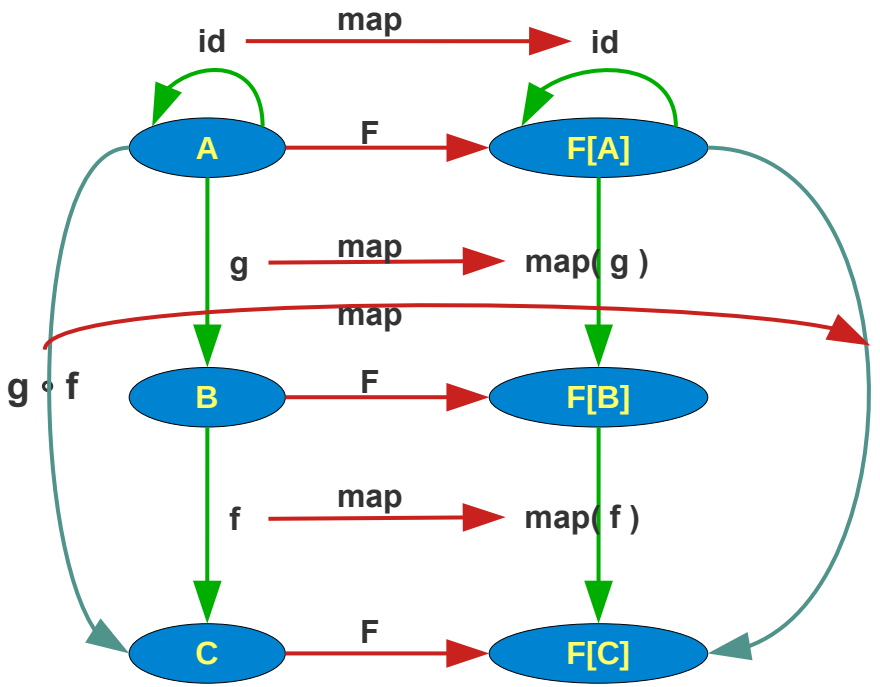
Map: Gesetzestreu und Vernünftig

Vernünftiges map = Gesetzestreues map

Map-Intuition als Regeln / Gesetze

– **Erhaltung der Identität:**
Die identische Abbildung A => A wird in die identische Abbildung F[A] => F[A] überführt
$map(id_A) = id_{F[A]}$

– **Distribution über die Funktionsverknüpfung**
map „distribuiert“ über die Funktionskomposition:
map(g ∘ f) = map(g) ∘ map(f)



Funktor: Konstruktion von Strukturen mit (vernünftigem) map

Triple-Funktor erfüllt die Funktor-Gesetze

1. Identität: map mit der Identität ändert nichts.

```
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 = {
  def id: A => A = a => a
  fa.map(id) == fa
}

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

given TripleFunctor : 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
```

map mit der Identität ändert nichts

Funktor: Konstruktion von Strukturen mit (vernünftigem) map

Triple-Funktor erfüllt die Funktor-Gesetze

2. Komposition: map distribuiert über die Funktionsverknüpfung

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

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

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

map distribuiert über die Funktionsanwendung

Ein erfolgreicher Test ist natürlich kein Beweis, aber map ist sicher vernünftig definiert.

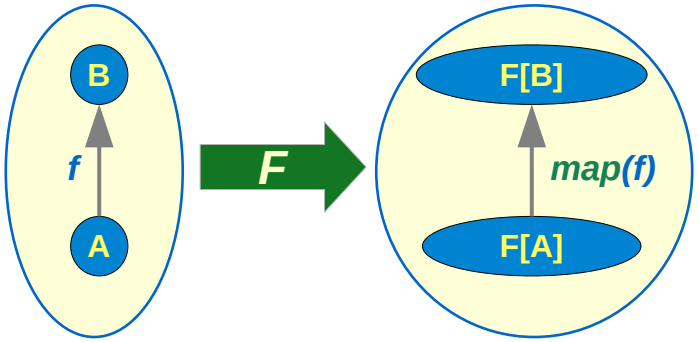
Funktor: Konstruktion von Strukturen mit (vernünftigem) map

Ein „falscher“ Triple-Funktor – ein vermeintlicher Funktor, der keiner ist

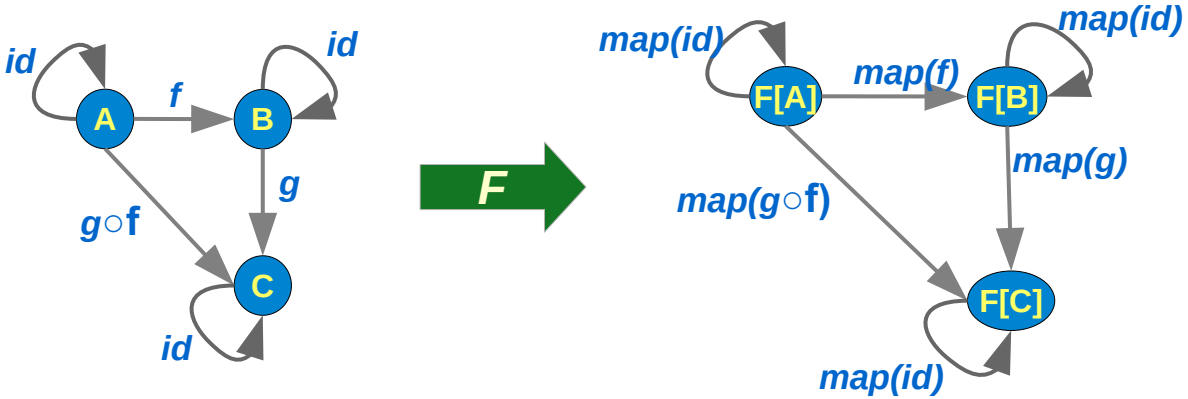
```
given Functor[Triple] with {
  extension[A, B] (fa: Triple[A]) def map(f: A => B): Triple[B] =
    Triple(f(fa.c), f(fa.b), f(fa.a))
}
```

Funktor: Konstruktion von Typen mit (vernünftigem) map

Funktor-Gesetze in der Übersicht



Funktor



Funktor-Gesetze

Funktor – Beispiele offensichtlicher Funktoren

List

List ist (schon von Natur aus) ein Funktor

```
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 Methode List.map hat die gewünschte Funktionalität.
List „ist“ schon „irgendwie“ ein Funktor

Funktor – Beispiele offensichtlicher Funktoren

Option

Option ist (schon von Natur aus) ein Funktor

```
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) // Some(3)
```

Die Klasse Option „ist“ schon“ (irgendwie) ein Funktor, denn sie hat eine passende Map-Methode.

Funktor – Beispiele offensichtlicher Funktoren

Either

Either als Functor

Problem: Either hat zwei Typargumente.

Lösung: Das erste Typargument wird fixiert

```
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, und auch so in Scala, wird die linke Komponente von Either als Fehlerfall interpretiert.

Funktoren: Wer kann dazugehören

Was kann ein Funktor sein

- Map muss korrekt definiert sein
- Kann map überhaupt immer korrekt definiert werden?
  - Ja bei einem Container-Typ mit einem Typ-Argument
  - Der seinen Inhalt nur speichert und ansonsten nicht beachtet

Gegenbeispiel

```
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)
}
```

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

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

Eine sortierte Liste kann kein Funktor sein!

```
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)
```

Funktor – weniger offensichtlich

Future

Future: ein (wichtiger) Funktor aber kein Container

```
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)
}
```

Die Funktor-Gesetze gelten:

- ***die identische Funktion ändert nichts, auch dann nicht, wenn sie asynchron ausgeführt wird.***
- ***Ob f und g hintereinander, oder g • f einmal asynchron ausgeführt werden, macht keinen Unterschied.***

Funktor – weniger offensichtlich

Funktionen als Funktoren

Ein Funktor ist oft, muss aber kein Containertyp (mit map) sein

Funktionen (d.h. der Typkonstruktor =>) können als Funktor interpretiert werden

Funktoren sind generisch in einem Typparameter:

In A => B muss A oder B fixiert werden.

Fixiert man A z.B. auf Int dann erhält man den Typ IntTo[B] = Int => B

IntTo[B] ist ein Funktor:

```
IntTo[B].map(f: B => C) : IntTo[C]
map: (Int => B) => (B => C) => Int => C
```

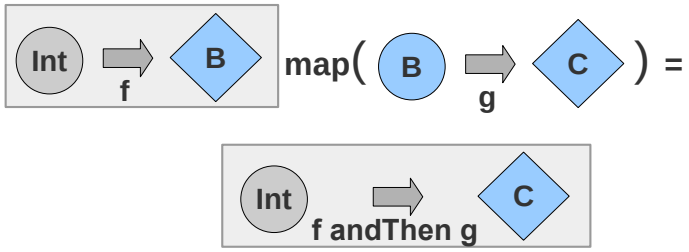
```
type IntTo[B] = Int => B

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>
```



Funktor – Beispiele

Funktion als Funktor

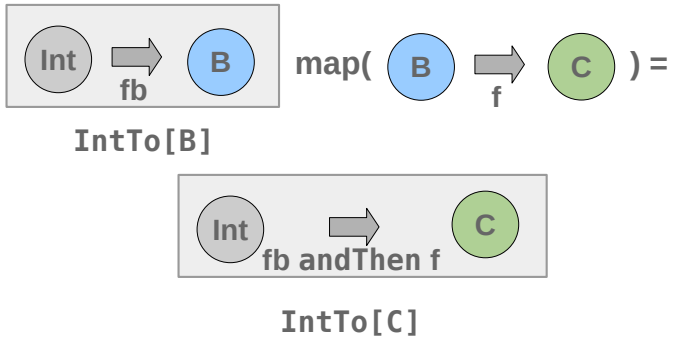
Funktionen

mit fixiertem Definitionsbereich (z,B: Int)

und map als „Fortsetzung“ sind Funktoren

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

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



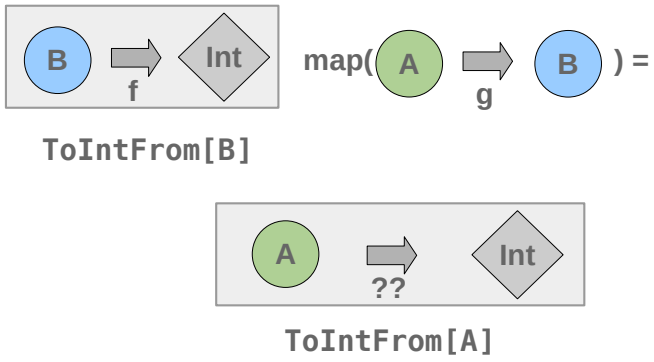
Was ist mit Funktionen

mit fixiertem Wertebereich (z.B: Int),

sind das auch Funktoren für eine geeignete Definition von map?

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

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



Funktor – Beispiele

Beispiel Non-Funktor

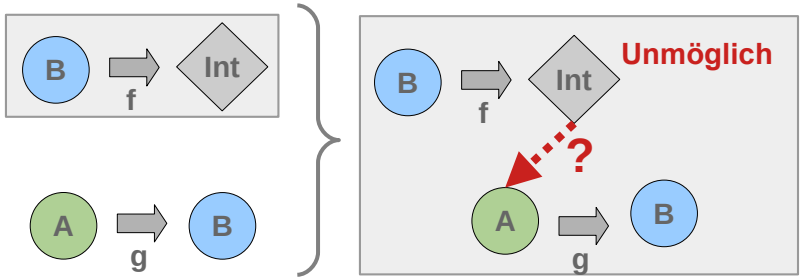
Funktionen

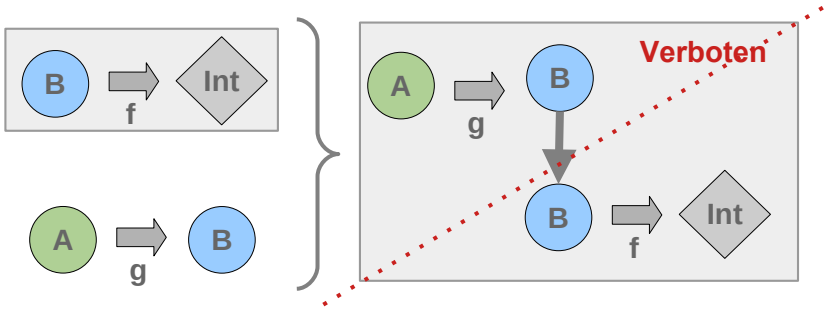
mit fixiertem Wertebereich (z,B: Int)

sind keine Funktoren:

Ein Typ-korrektes map kann (!) nicht definiert werden:

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





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,

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

Gegeben

- ***ein f, mit dem aus einem B-Wert ein Int-Wert konstruiert werden kann, und***
- ***ein g das einen A-Wert in einen B-Wert transformiert.***

Aus den beiden kann eine Funktion A => 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:A eines unbekannten Typs A ge-/er-funden werden, das geht nicht. – Wir wissen ja nichts über A.

Funktor – Beispiele

Map: Vollständig parametrisch

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.

Jede map-Funktion ist vollständig parametrisch

und damit unabhängig von irgendwelchen konkreten Werten. – Sie kann selbst nichts mit den Werten tun, ausser sie an andere weiter geben.

Funktor – das Wesentliche: Der praktische Einsatz

Funktor ~ Kontext der verkettbare Operationen auf dem Inhalt zulassen

Verarbeitungsketten

Funktoren:

- **Anwendung einer Funktion in einem Kontext**
- **Verkettung von Verarbeitungsschritten in einem Kontext**

Verallgemeinerung:

- **Es kann sich um beliebige Verarbeitungsschritte handeln,**
- **Der Kontext kann, ist oft, muss aber nicht zwingend eine Werte-Container sein**

map: Operationen auf dem Inhalt, ohne Rücksicht auf die Verpackung / den Kontext die verkettet werden können.

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

def even(x: Int): Boolean = (x%2 == 0)

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


Verarbeitungskette:
Transformiert den Inhalt

Funktor – das Wesentliche (das Praktische)

Funktor-Block-Ausdrücke

Zur Formulierung von Verarbeitungsketten gibt es spezielle Ausdrucksmittel

z.B. For-yield-Ausdrücke (*for-Comprehension*) in Scala „verkleiden“ map

Code wird leichter lesbar

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

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

val result_2: List[Boolean] = // List(false, false, false, true, true)

for (str <- list;
     i = str.length;
     j = i % 2)
  yield j == 0
```

Verarbeitungskette mit map

Äquivalenter Funktor-Block

Funktor-Block-Ausdrücke

Zur Formulierung von Verarbeitungsketten gibt es spezielle Ausdrucksmittel

map vs Funktor-Block

```
list                          // für alle Elemente von list:
  .map( str => str.length)    // nenne das Element str und berechne str.length, das Ergebnis ist namenlos
  .map( i   => i % 2)         // nenne das Ergebnis aus dem letzen Schritt i und berechne i % 2
  .map( j   => j == 0)        // nenne das Ergebnis aus dem letzen Schritt j und berechne j == 0
                              // füge das Ergebnis aus dem letzen Schritt zum Gesamtergebnis hinzu
```

```
for (str <- list;     // für alle Elemente str von list:
     i = str.length;  // berechne str.length, nenne das Ergebnis i
     j = i % 2        // berechne i % 2, nenne das Ergebnis j
) yield j == 0        // berechne j == 0 und füge das Ergebnis zum Gesamtergebnis hinzu
```

Funktor – das Wesentliche (das Praktische)

Funktor-Block-Ausdrücke

Zur Formulierung von Verarbeitungsketten gibt es spezielle Ausdrucksmittel

map vs Funktor-Block

Namen können flexibel verwendet werden

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

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

In einem Funktor-Block kann jeder Name nach seiner Definition an beliebigen Stellen genutzt werden.

```
List('one' has odd length (3), 'two' has odd length (3), 'three' has odd length (5), 'four' has even length (4), 'five' has even length (4))
```

```
val result_map : List[String] =
  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)") }
```

Äquivalente Verarbeitungskette mit map.

Namen müssen explizit mitgeschleppt werden. (Diese Arbeit übernimmt bei Scala der Compiler.)

Contravariante Funktoren

Co- und Contra-Varianz von Funktoren

Varianz: wie überträgt sich die Richtung der Pfeile

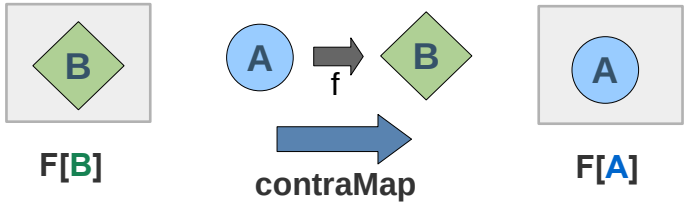
Funktoren sind covariant:

f: A => B ~> map(f): F[A] => F[B]

Bei einem contravarianten Funktoren kehrt sich die Pfeilrichtung um:

- **f: A => B ~> contraMap(f): F[A] <= F[B]**

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



Contravariante Funktoren

Beispiel

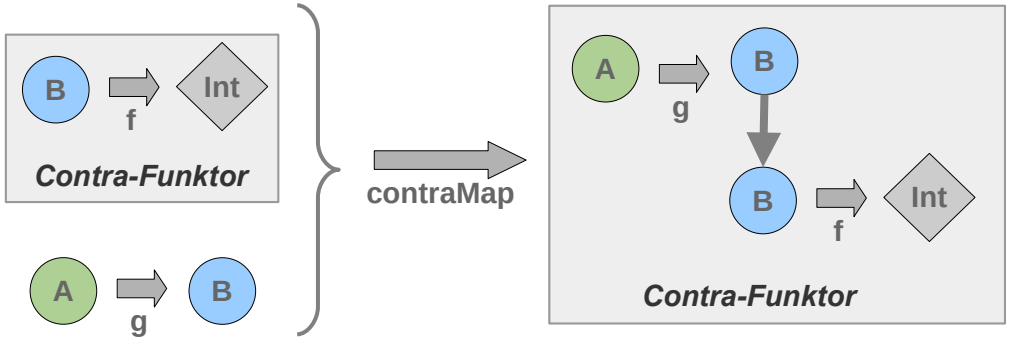
Bei einem contravarianten Funktoren kehrt sich die Pfeilrichtung um:

f: A => B ~> contraMap(f): F[B] => F[A]

Mit map wird eine Operation „angehängt“
Mit contraMap wird eine Operation „vorangestellt“

Anwendungsbeispiel: Aktion nach „vorne erweitern“:

- **F[B] kann verwendet werden**
- **Mit einer Funktion f: A => B wird auch F[A] verwendbar**



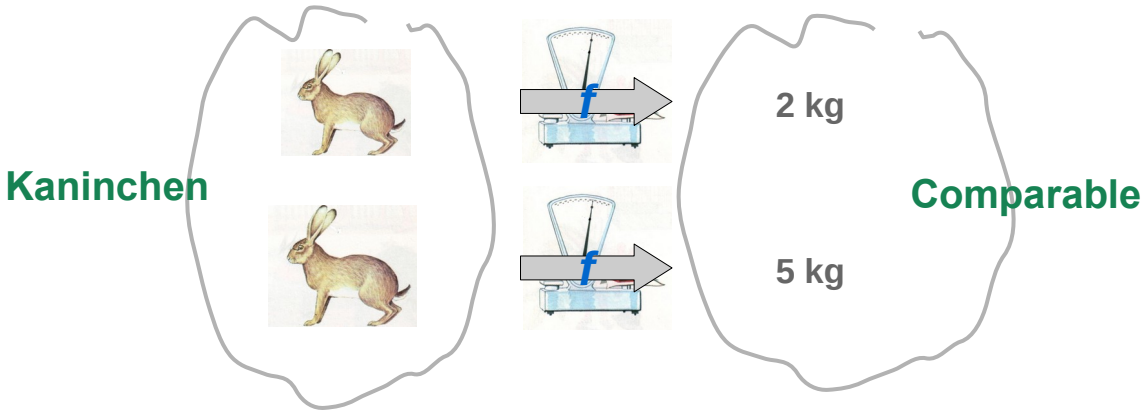
Beispiel Contravarianter Funktor

Contrafunktor, Aktion nach „vorne erweitern“:

- **F[B] kann verwendet werden**
- **Mit einer Funktion f: A => B wird auch F[A] verwendbar**

Anwendung: Vergleichbarkeit Comparable erweitern:

- **Gegeben sei Comparable[B]**
  mit f: A => B
- **Damit kann ein Comparable[A] erreicht werden:**
  vor dem tatsächlichen Vergleich muss nur
  mit f jedes a:A in ein (vergleichbares) b:B transformiert werden.



Kg sind vergleichbar. Diese Vergleichbarkeit kann mit der Funktion „wiegen“ kontravariant auf Kaninchen erweitert werden.

Beispiel Contravarianter Funktor

Comparable als contravarianter Funktor

```scala
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))
    }
}
```

Beispiel Contravarianter Funktor

Sortieren mit Hilfe eines contravarianten Funktors

```
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)
  }

}
```

Eine Liste von A-s kann mit InsertionSort sortiert werden, wenn es ein Konversion A => B gibt und B comparable ist.

Beispiel Contravarianter Funktor

Kaninchen nach Gewicht sortieren mit contravariantem Funktor

```scala
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
}

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)
```

Contravarianter Funktor

Beispiel

Comparable mit contraMap

So wie „Funktor-artige Klassen“ eine map-Methode haben, so kann man Comparable eine contraMap-Methode geben:

```
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)
}
```

Contravarianter Funktor

Beispiel

Comparable mit contraMap

… und sortieren

```
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)
  }

}
```