# Übungsblatt 10— Lösungen

## 1 `map` via `traverse`

```
def mapViaTraverse[F[_],A,B](fa: F[A])(f: A => B)(using Traverse[F]): F[B] = ???
```

Wie in der Vorlesung bereits erwähnt, entspricht `traverse` einem `map` gefolgt von einem `sequence`. Dabei erwartet `traverse` eine Funktion, die einen Applicative zurückgibt, wir müssen unsere an `map` übergebene Funktion also entsprechend modifizieren.

Da die Signatur von `map` keinen Applicative vorgibt, können wir einen beliebigen verwenden, solange wir ihn danach wieder auspacken können (`traverse` gibt uns das gewünschte Ergebnis ja in den Applicative verpackt zurück).

Für diesen Zweck ist `Id` der naheliegendste Typ:

```
fa.traverse(f(_).pure[Id])
```

(Die Schreibweise `x.pure[F]` ist in `cats` die Abkürzung für `Applicative[F].pure(x)`)

So gibt `traverse` uns ein `Id[F[B]]` zurück, was identisch zu `F[B]` ist.

Jeder andere in der Vorlesung behandelte `Applicative` ist allerdings genau so möglich, zum Beispiel `List`:

```
fa.traverse(f(_).pure[List]).head
```

## 2 Traverse-Instanz für Bäume

### 2.1 Implementation

```
given Traverse[Tree] with
  import fp06.given
  import fp06.Tree.*

  def traverse[G[_],A,B](fa: Tree[A])(f: A => G[B])(using Applicative[G]): G[Tree[B]] =
    fa match
      case Leaf(a)             => f(a).map(Leaf(_))
      case Branch(left, right) => left.traverse(f).map2(right.traverse(f))(Branch(_, _))

  def foldLeft[A, B](fa: Tree[A], b: B)(f: (B, A) => B): B =
    summon[Foldable[Tree]].foldLeft(fa, b)(f)

  def foldRight[A, B](fa: Tree[A], lb: Eval[B])(f: (A, Eval[B]) => Eval[B]): Eval[B] =
    summon[Foldable[Tree]].foldRight(fa, lb)(f)
```

In der Implementation von `traverse` schauen wir uns zunächst den übergebenen Tree an. Ist er ein `Leaf`, haben wir einen Wert, auf dem wir `f` aufrufen können. Wir bekommen ein `G[B]`

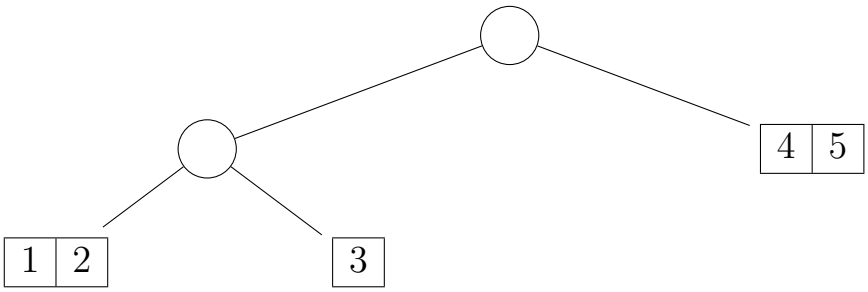
zurück, benötigen aber ein `G[Tree[B]]`, also verpacken wir mittels `map` den Wert *im Applicative* in ein `Leaf`.

Haben wir stattdessen einen `Branch`, rufen wir `traverse` rekursiv für beide Teilbäume auf, was uns zwei `G[Tree[B]]` liefert. Diese kombinieren wir mittels `map2` wieder in einen `Branch` innerhalb von `G`.

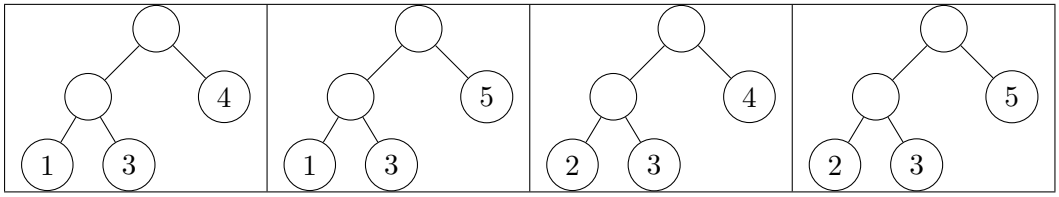
Für die beiden Folds kann wie gesagt die Foldable-Instanz aus dem sechsten Übungsblatt genutzt werden. Wir importieren die givens aus `fp06` und können dann mit `summon[Foldable[Tree]]` die Instanz erhalten, auf der wir dann `foldLeft` bzw. `foldRight` aufrufen können. Alternativ kann natürlich auch der Code der Methoden kopiert werden.

### 2.2 Verhalten von sequence

Überlegen wir uns zuerst einmal, wie sich `sequence` für einen `Tree[List[Int]]` verhalten würde, also mit der `sequence`-Methode der List-Traverse-Instanz. Wir haben einen Baum gegeben, in dessen Blättern sich Listen befinden:

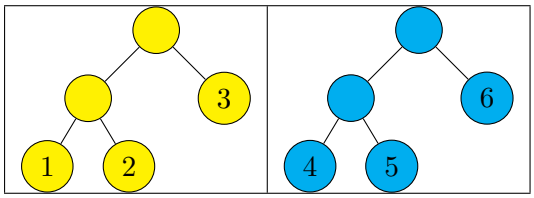



Mittels `sequence` drehen wir das um: `(x: Tree[List[Int]]).sequence` → `List[Tree[Int]]`

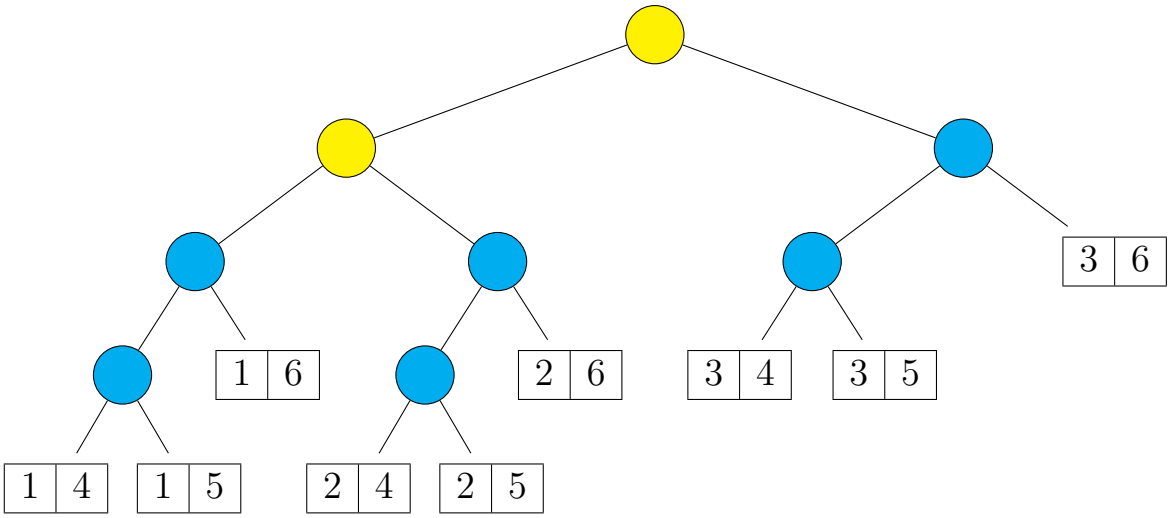



Wir erhalten also eine Liste mit Bäumen, die die gleiche Struktur haben wie der Ausgangsbaum. Für alle Blätter mit mehreren Elementen erhalten wir je einen Baum mit jeder möglichen Kombination mit den anderen Blättern (also insgesamt so viele Bäume wie das Produkt aller Listenlängen, hier $2 \cdot 1 \cdot 2 = 4$).

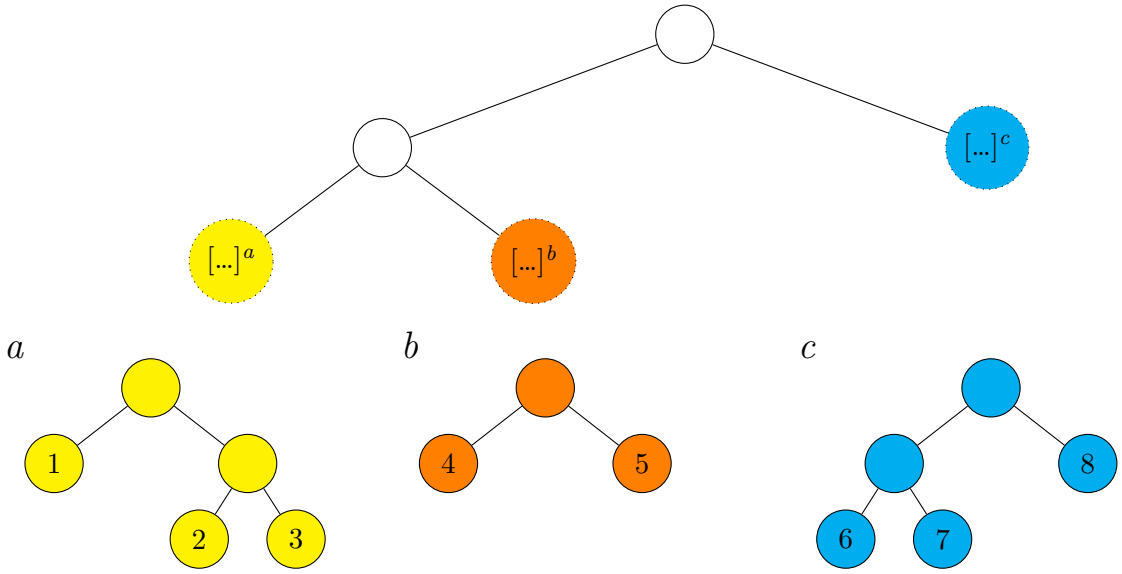
Wie schaut es nun in der anderen Richtung aus, also mit `(x: List[Tree[Int]]).sequence` → `Tree[List[Int]]`? Hier wieder ein Beispiel, zwei Bäume in einer Liste:




Da wir am Ende einen einzelnen Baum herausbekommen wollen, muss `sequence` diese kombinieren. Dies passiert dadurch, das jeweils die Blätter des ersten Baums durch die Struktur des zweiten Baums ersetzt werden. In den Blättern steht dann jeweils eine Liste mit dem Wert, der im ersten Baum im ersetzten Blatt enthalten war, und dem Wert, der im zweiten Baum an der jeweiligen Stelle stand, quasi der Pfad im Baum. So ergibt sich folgender größerer Baum:

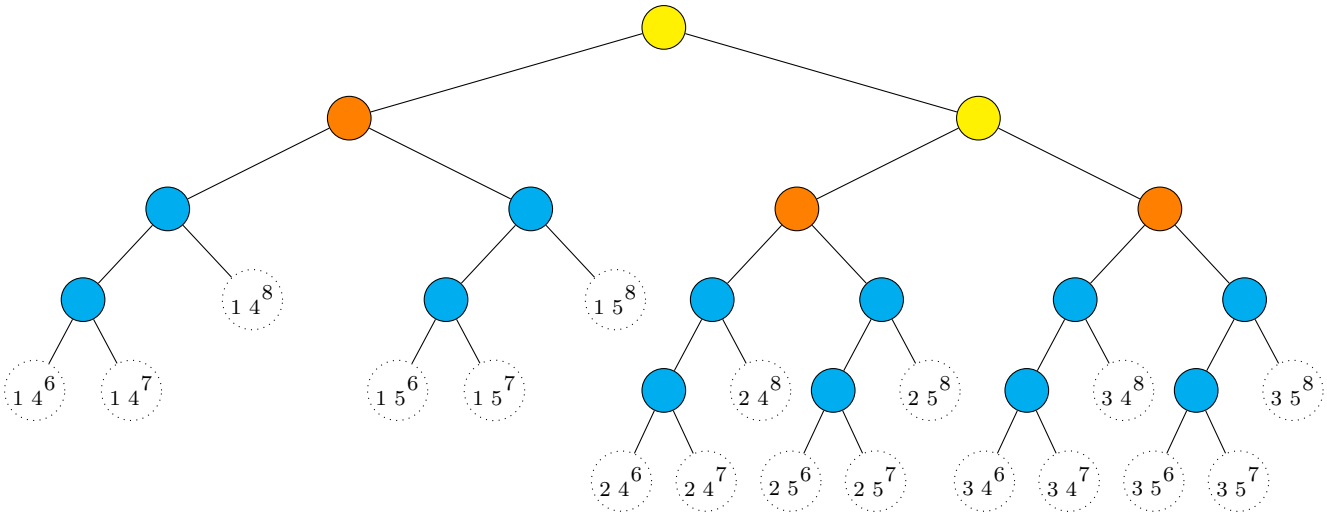



Etwas ähnliches passiert, wenn wir einen verschachtelten Tree mit `sequence` umkehren. In der folgenden Abbildung sehen wir einen Baum (weiße Knoten), welcher in jedem Blatt wiederum einen Baum mit Integern (farbige Knoten) speichert, der Typ ist also `Tree[Tree[Int]]`:




Ähnlich wie bei `sequence` auf `List[Tree[Int]]` ist die Struktur des ersten (also hier am weitesten links liegenden) Baumes an der Wurzel zu finden, die des zweiten dann an Stelle der Blätter des ersten, die des dritten an Stelle der Blätter des zweiten (und so weiter).

In den Blättern des letzten Baums ist dann jeweils *als Element* ein Baum mit der Struktur des vorher äußeren Baums gespeichert. Dieser enthält als Elemente wiederum die Zahlen, die als Blätter an den Stellen der einzelnen farbigen Bäume auf dem Pfad zu diesem Blatt standen:




# 3 Akkumulieren mit State

## 3.1 reverse

Zur Erinnerung hier die Implementation von `mapAccum`:

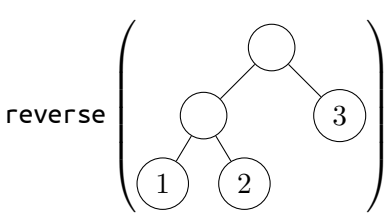
```
def mapAccum[F[_]:Traverse,S,A,B](fa: F[A], s: S)(f: (S,A) => (S,B)): (S,F[B]) =
  fa.traverse(a => State(s => f(s, a))).run(s).value
```

Da Listen bereits `reverse` implementieren und wir jedes `Traverse` in eine Liste umwandeln können, nutzen wir dies aus, um unseren Startwert für die Akkumulation zu setzen: die Liste der Elemente in umgekehrter Reihenfolge.

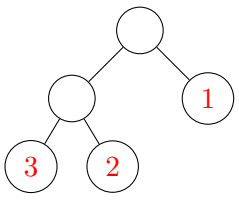
In der Funktion, die wir an `mapAccum` übergeben, ignorieren wir das aktuelle Element unseres Traversable Functors, wir benötigen diesen nur, um seine Struktur zu erhalten. Die Werte erhalten wir aus der umgekehrten Liste. Von dieser geben wir den Head zurück, und geben den Tail als neuen Zustand zurück. Wir nutzen also die umgekehrte Liste als Stack, von der wir nach und nach Elemente entnehmen und an die Positionen setzen, die uns unser `fa` vorgibt:

```
def reverse[F[_],A](fa: F[A])(using Traverse[F]): F[A] =
  mapAccum(fa, fa.toList.reverse)((l, _) => (l.tail, l.head))._2
```

Hier ein Schritt-für-Schritt-Beispiel mit einem Baum. Links der Aufruf, in der Tabelle die einzelnen Schritte, rechts das Ergebnis:




| A | S | B |
|---|---|---|
| – | 3 \| 2 \| 1 | – |
| ① | 2 \| 1 | 3 |
| ② | 1 | 2 |
| ③ | ∅ | 1 |




Zu Beginn des Aufrufs ist unser Zustand die Liste aller Blattwerte in umgekehrter Reihenfolge. Im ersten Schritt ist das an die Funktion übergebene `A` der Wert aus dem Knoten ganz links. Der Wert wird ignoriert, aber an diese Stelle der erste Wert aus unserem Stack (S) gesetzt. Dies führen wir fort, bis wir alle Blätter durchlaufen haben.

### 3.2 foldLeft via mapAccum

```
def foldLeftViaMapAccum[F[_]:Traverse,A,B](fa: F[A], z: B)(f: (B, A) => B): B =
  mapAccum(fa, z)((s, a) => (f(s, a), ())).1
```

Ähnlich wie bei der `toList`-Implementation in der Vorlesung, benötigen wir hier am Ende von `mapAccum` nicht den produzierten Wert in `F`, sondern unseren State `S`. Dementsprechend geben wir wie auch in `toList` immer unit als Wert zurück.

Den Startwert für unseren State setzen wir auf den `foldLeft` übergebenen Wert `z`. In der anonymen Funktion wird dieser dann jeweils auf das Ergebnis des an `foldLeft` übergebenen `f` gesetzt.