Übungsblatt 08— Lösungen

1 Tupel-Komposition für Applicative

Da die beiden Applicatives in einem Tupel stehen und nicht verschachtelt sind, können wir sie vollständig unabhängig voneinander behandeln.

In `pure` verpacken wir das übergebene A einfach jeweils einmal in F und G, und geben die Ergebnisse dann in einem Tupel zurück.

```
def pure[A](x: A): (F[A], G[A]) =
  (Applicative[F].pure(x), Applicative[G].pure(x))
```

In `ap` bekommen wir bereits zwei Tupel, wir wenden `F.ap` auf die jeweils ersten Tupelelemente an, `G.ap` dann auf die jeweils zweiten.

```
def ap[A, B](ffgg: (F[A => B], G[A => B]))(faga: (F[A], G[A])): (F[B], G[B]) =
    (ffgg._1 <*> faga._1, ffgg._2 <*> faga._2)
```

Wir benutzen hier die von Cats zur Verfügung gestellte Operatorsyntax:
`ffgg._1 <*> faga._1` ist identisch zu `Applicative[F].ap(ffgg._1, faga._1)`.

Falls wir für `ap` unsere Implementation aus der Vorlesung via `map2` verwenden, kann auch `map2` implementiert werden, auch hier verwenden wir wieder jeweils die Implementationen der beiden kombinierten Applicatives:

```
override def map2[A,B,C](faga: (F[A], G[A]), fbgb: (F[B], G[B]))(f: (A, B) => C) =
  (Applicative[F].map2(faga._1, fbgb._1)(f),
   Applicative[G].map2(faga._2, fbgb._2)(f))
```

oder mit zusätzlicher Cats-Syntax:

```
override def map2[A,B,C](faga: (F[A], G[A]), fbgb: (F[B], G[B]))(f: (A, B) => C) =
  ((faga._1, fbgb._1).mapN(f),
   (faga._2, fbgb._2).mapN(f))
```

2 Applicative Combinators

Zur Erinnerung, hier die Signatur von product:

```
def product[F[_],A,B](fa: F[A], fb: F[B])(using Applicative[F]): F[(A,B)]
```

2.1 ap via product und map

```
def apViaProductAndMap[F[_],A,B](ff: F[A => B])(fa: F[A])(using Applicative[F]): F[B] =
  Applicative[F].product(ff, fa).map((fab, a) => fab(a))
```

Wir kombinieren zuerst die beiden Fs mit `product` und erhalten ein `F[(A => B, A)]`. Auf diesem einzelnen `F` können wir dann `map` benutzen, um die Funktion auf das A anzuwenden.

2.2 product via ap

Zuerst definieren wir `ap` via `product` und `map`:

```
def productViaApAndMap[F[_],A,B](fa: F[A], fb: F[B])(using Applicative[F]): F[(A,B)] =
  fa.map((a: A) => (b: B) => (a, b)) <*> fb
```

Mittels `map` wandeln wir das `A` in eine Funktion um, die ein `B` erwartet, und es mit dem A in ein Tupel verpackt. Diese Funktion können wir nun mit `<*>`, also `ap`, auf das `F[B]` anwenden.

Damit das ganze nur `ap` und nicht `map` benutzt, ersetzen wir den Aufruf noch durch die `map`-Definition via `ap` aus der Vorlesung:

```
def productViaApAndPure[F[_],A,B](fa: F[A], fb: F[B])(using Applicative[F]): F[(A,B)] =
  Applicative[F].pure((a: A) => (b: B) => (a, b)) <*> fa <*> fb
  //                  F[A => B => (A, B)]            F[A]   F[B]   -> F[(A, B)]
```

3 Applicative-Instanz für Binärbäume

3.1 Applicative für Binärbäume

Zuerst einmal die Methode `map` aus dem Functor-Übungsblatt:

```
override def map[A, B](fa: Tree[A])(f: A => B): Tree[B] =
  fa match
    case Branch(left, right) => Branch(left map f, right map f)
    case Leaf(value) => Leaf(f(value))
```

Wir wollen nun `pure` und `ap` implementieren. In `pure` verpacken wir den übergebenen Wert in ein `Leaf`:

```
override def pure[A](x: A): Tree[A] = Leaf(x)
```

In `ap` haben wir nun zwei Bäume, die wir durchlaufen müssen. Da wir bereits `map` haben, welches eine Funktion `A => B` erwartet, bietet es sich an, zuerst den Baum mit solchen Funktionen zu durchlaufen und diese dann per `map` auf den anderen anzuwenden:

```
override def ap[A, B](ff: Tree[A => B])(fa: Tree[A]): Tree[B] =
  ff match
    case Branch(left, right) => Branch(left <*> fa, right <*> fa)
    case Leaf(f) => fa.map(f)
```

Wir matchen also auf `ff` und rufen im Fall eines Branches `ap` rekursiv mit dem Baum `fa` auf. Wenn wir ein Leaf erreichen, nehmen wir die Funktion daraus und mappen diese über unseren `Tree[A]`.

3.2 map2 auf Binärbäumen

Um zu verstehen, was `map2` auf zwei Binärbäumen tut, schauen wir uns erst einmal `map2` auf einer einfacheren Datenstruktur, nämlich List, genauer an.

Vergleichen wir für Listen die Methoden `product` (= `map2` mit festem `f`) und `zip`:

```
def product[A,B](fa: List[A], fb: List[B]): List[(A,B)]
def zip[A,B](fa: List[A], fb: List[B]): List[(A,B)]
```

Die Signaturen sind identisch, jedoch haben die beiden Methoden unterschiedliche Ergebnisse. `zip` kombiniert jeweils ein Element der einen Liste mit einem der anderen Liste:

```
val fa = List(1, 2, 3)
val fb = List('a', 'b', 'c', 'd', 'e')
fa.zip(fb) == List((1, 'a'), (2, 'b'), (3, 'c'))
```

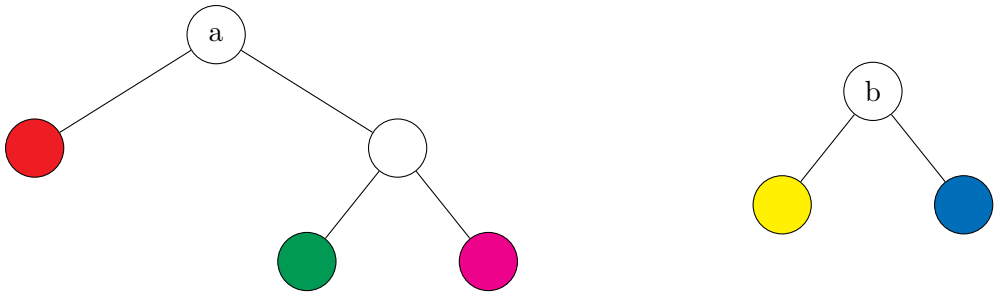
`product` hingegen kombiniert *jedes* Elemenent der einen mit *jedem* Element der anderen Liste. Wir erhalten quasi das „Kreuzprodukt“ der Listen:

```
Applicative[List].product(fa,fb) == List(
    (1, 'a'), (1, 'b'), (1, 'c'), (1, 'd'), (1, 'e'),
    (2, 'a'), (2, 'b'), (2, 'c'), (2, 'd'), (2, 'e'),
    (3, 'a'), (3, 'b'), (3, 'c'), (3, 'd'), (3, 'e')
  )
```

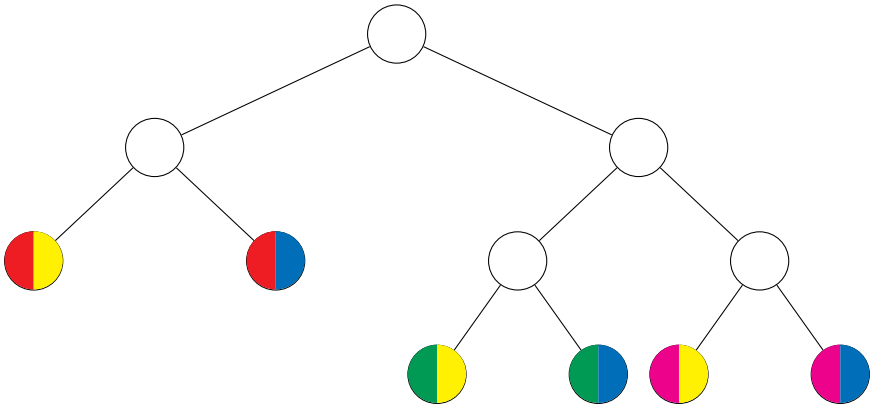
Ähnlich verhält es sich auch bei unseren Bäumen. Kombinieren wir diese mit `map2`, erhält man die Struktur die sich ergibt, indem man in der Struktur des ersten Baums jedes Blatt durch eine vollständige Struktur des zweiten Baums ersetzt.

(Siehe nächste Seite für bebildertes Beispiel)

Hier das ganze anhand eines Beispiels mit zwei Bäumen a und b:



Rufen wir `a.map2(b)(_ |+| _)` auf (wobei `|+|` eine beliebige Operation zum Kombinieren der beiden Werte ist, hier dargestellt durch die Kombination der Farben in den Knoten), erhalten wir einen Baum, der an der Wurzel erst mal die gleiche Struktur hat wie a. An jeder Stelle, die bei a ein Blatt ist, ist nun ein Zweig mit zwei Blättern: die Struktur von b:



Umgekehrt erhalten wir einen anderen Baum, wenn wir `b.map2(a)(_ |+| _)` aufrufen. Unser `map2` ist also nicht kommutativ: