Übungsblatt zu Vorlesung 06— Foldable and Functor

In dieser Übung beschäftigen wir uns mit Typklassen, Givens und Katzen[1]. Die Signaturen der in den Aufgaben geforderten Methoden sowie die vorgegebenen Implementierungen finden Sie online unter `https://gitlab2.informatik.uni-wuerzburg.de/intro-to-fp/tasksheets` als Git-Repository.

1 Typeclass Instanzen für Binärbäume

In der Datei `Tree.scala` finden Sie eine Implementierung eines Binärbaums (eine vereinfachte Variante der aus der *Huffman*-Aufgabe bekannten Bäume). Im Folgenden werden wir für `Tree` Instanzen der jeweiligen Cats-Pendants der aus der Vorlesung bekannten Typklassen implementieren. Cats ist eine der verbreitetsten Scala-Bibliotheken für funktionale Programmierung. Beachten Sie auch die API-Doku unter .

Erst einmal die Definition unserer Binärbäume:

```
enum Tree[+A]:
  case Branch[A](left: Tree[A], right: Tree[A])
  case Leaf[A](value: A)
```

Ist Cats in ein Projekt eingebunden, finden Sie im Paket `cats` die durch Cats bereitgestellten Typklassen, insbesondere diejenigen, die aus der Vorlesung bekannt sind. Im Paket `cats.syntax` finden sich zusätzlich Objekte, die es ermöglichen die Methoden aus den Typklassen in der uns vertrauten Syntax (also z.B. `l.map(_ * 2)` anstatt `Functor[List].map(l)(_ * 2)`) zu verwenden[2]. Im Git-Repository sind die entsprechenden Imports bereits vorgenommen.

IntelliJ zeigt sich leider hin und wieder überfordert mit den Definitionen aus `cats.syntax` und markiert formal korrekten Code als fehlerhaft. Hier hilft leider nur, die von IntelliJ gemeldeten Fehler mit dem Scala Compiler zu verifizieren (z.B. via `sbt compile`) und im Zweifel zu ignorieren, oder einen Editor zu benutzen, welcher den Metals Language Server unterstützt.

a) In der Vorlesung haben wir Monoide kennen gelernt und die Typklasse `MonoidK` für Monoide über Typkonstruktoren. Überlegen Sie sich, warum unsere Binärbäume keine Monoide sind!

Eine allgemeinere Variante von Monoiden sind Halbgruppen, die sehr ähnlich funktionieren aber auf die Definition eines Nullelements verzichten, also nur aus einer Menge und einer assoziativen Operation auf dieser bestehen. Die zugehörige Typklasse in Cats heißt `Semigroup`. Auch hier existiert mit `SemigroupK` wieder eine Variante für Typen höherer Ordnung.

Implementieren Sie eine Instanz von `SemigroupK` für `Tree`. Beachten Sie, dass `combineK` assoziativ sein muss!

b) Mit `Functor` hat Cats eine Typklasse für kovariante Funktoren, wie wir sie aus der Vorlesung kennen. Sie funktioniert wie die aus der Vorlesung bekannte, nur dass `map` keine Extension Method ist, sondern die Datenstruktur als ersten Parameter bekommt. Dank

[1] `cats` ist eine Library für funktionale Programmierung

[2] Da cats noch kompatibel mit Scala 2 ist, sind diese leider nicht direkt als Extensions definiert, daher ist der zusätzliche import nötig

des `cats.syntax` packages kann die Methode trotzdem genutzt werden, als wäre sie eine Extension.

Implementieren Sie eine Instanz von `Functor` für `Tree`.

c) Auch für die `Foldable`-Typklasse hat Cats ein gleichnamiges Äquivalent, das sehr ähnlich arbeitet.

Cats verwendet in `Foldable` bei `foldRight` die `Eval`-Klasse für Stack-sichere nicht strikte Evaluation. `Eval` ist nicht Teil des Vorlesungsstoffs und für die Bearbeitung der Aufgabe weitgehend irrelevant. Wer sich trotzdem dafür interessiert sei auf die Dokumentation verwiesen. Für Ihre Implementierung können Sie `Eval` ignorieren. Sie müssen an keiner Stelle `Eval`-Instanzen erzeugen und können `Eval[B]` genau wie `B` behandeln.

Implementieren Sie eine Instanz von `Foldable` für `Tree`. *Hinweis:* in diesem Fall muss `foldLeft` nicht tailrekursiv sein.

2 Eine halbautmatische Monoid-Fabrik

Hinweis: Die folgende Definition nutzt die mathematische Schreibweise für Monoide: Ein Tupel aus der Wertemenge (entspricht bei uns dem Typ), der Operation (combine) und dem Nullelement.

Ein Monoid-Isomorphismus ist eine bijektive Abbildung $f : A \to B$ zwischen zwei Monoiden $(A, \oplus, z_A)$ und $(B, \odot, z_B)$, für die gilt:

- $\forall x, y \in A : f(x \oplus y) = f(x) \odot f(y)$,
- $f(z_A) = z_B$.

Wir können uns diese Eigenschaften zunutze machen, um mit Hilfe von gegebenen Monoid-Isomorphismen neue Monoid-Instanzen zu erzeugen.

Implementieren Sie die Funktion `imap`, die aus einer Funktion `f: A => B` und ihrer Inversen `g` für ein Monoid `A` eine Monoid-Instanz für `B` erzeugt. Beachten Sie, dass in Cats das Nullelement `empty` genannt wird.

```scala
def imap[A, B](f: A => B, g: B => A)(using Monoid[A]): Monoid[B] = ???
```

Um Ihre Implementierung zu testen ist die Klasse `Box[A]` vorgegeben, die beliebige Werte vom Typ `A` „verpackt“:

```scala
case class Box[+A](value: A)
```

Implementieren Sie mit Hilfe von `imap` eine Monoid-Instanz für `Box[A]`, wobei `A` ein Monoid ist.

```scala
given boxMonoid[A](using Monoid[A]): Monoid[Box[A]] = ???
```

Hinweis: Die Angaben nebst einer `main`-Methode finden Sie in der Datei `InvariantMonoid.scala` im Git.