Übungsblatt zu Vorlesung 07— Monads

In dieser Übung beschäftigen wir uns mit Monaden und ihren Gesetzen. 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 Sequence in Cats

Implementieren Sie die Funktion `Sequence`, die wir bereits für `Option` und `Either` kennen gelernt haben, für alle monadischen Typen.

```
def sequence[F[_], A](fas: List[F[A]])(using Monad[F]): F[List[A]] = ???
```

Benutzen Sie die `Monad` Type Class aus Cats sowie `foldRight` und `map2` für Ihre Implementierung.

2 Identity-Monad

Manchmal ist es hilfreich, Funktionen, die auf Monad definiert sind, auch für Typen zu verwenden, die nicht ein einen monadischen Typen „verpackt“ sind. Dazu erzeugen wir einen Pseudotyp, bzw. einen Typalias, der einfache Typen in unäre Typkonstruktoren verwandelt.

```
type Id[A] = A
```

Achtung, dieser Typ ist nicht identisch zu Id aus der Vorlesung. Dort wurde eine case class benutzt, hier ein Typalias.

Implementieren Sie eine Monad-Instanz für `Id`. Die Funktion `tailRecM` ist vorgegeben.

`tailRecM`

In Cats muss zusätzlich zu den aus der Vorlesung bekannten Funktionen die Funktion `tailRecM` implementiert werden, die wiederholte, rekursive Aufrufe von `flatMap` stack-safe implementiert. Dieses Implementierungsdetail ermöglicht es, für alle Monaden sicherzustellen, dass Funktionen, die auf rekursive Aufrufe von `flatMap` aufbauen, was in der Praxis relativ häufig ist, keinen `StackOverflowError` auslösen, wenn Sie stattdessen mit `tailRecM` implementiert wurden, sofern `tailRecM` für die jeweilige Monad-Instanz tail-rekursiv ist. Genaueres finden Sie wie immer in der Doku

3 Monad Laws

a) Zeigen Sie, dass die beiden folgenden Formulierungen des Assoziativgesetzes für Monaden (zum einen mit `flatMap`, zum anderen mit `compose`) äquivalent sind.

```
flatMap(flatMap(x)(f))(g) == flatMap(x)(a => flatMap(f(a))(g))
compose(compose(f)(g))(h) == compose(f)(compose(g)(h))
```

Die Idee ist hier, eine der beiden Formulierungen in die andere umzuformen. Erinnern Sie sich daran, wie `compose` implementiert wurde.

b) Zeigen Sie, dass die Formulierungen der *Identity Laws* äquivalent sind. Gehen Sie hier ähnlich vor wie in Aufgabe **a)**.

```
compose(f, pure) == f
compose(pure, f) == f

flatMap(x)(pure) == x
flatMap(pure(y))(f) == f(y)
```

c) Zeigen Sie für den Option-Monad, dass die *Identity Laws* (in ihrer `flatMap`-Formulierung) gelten.

4 Monad Combinators

In der Vorlesung wurden Monaden eingeführt mit der „minimalen Menge an Monad combinators“ `flatMap` und `pure`. Wir lernen hier noch zwei weitere Mengen kennen, deren Existenz ein Monad jeweils hinreichend bestimmen. Verwenden Sie für `pure`, `map`, `flatten` und `flatMap` jeweils die angegebenen Funktionen auf der Cats `Monad` Type Class. Für `compose` ist eine eigene Implementierung vorgegeben.

a) **`pure`, `map` und `flatten`**

Implementieren Sie die Funktion `flatten` via `flatMap`. `flatten` entfernt eine Schachtelungsebene von einem mehrfach verschachtelten monadischen Typ.

```
def flattenViaFlatMap[F[_], A](ffa: F[F[A]])(using Monad[F]): F[A] = ???
```

Implementieren Sie nun `flatMap` und `compose` via `flatten` und `map`.

```
def flatMapViaFlattenAndMap[F[_],A,B](fa: F[A])(f: A => F[B])(using Monad[F]): F[B]
↪  = ???
def composeViaFlattenAndMap[F[_],A,B,C](afb: A => F[B], bfc: B => F[C])(using
↪  Monad[F]): A => F[C] = ???
```

b) **`pure` und `compose`**

Dass wir `compose` via `flatMap` definieren können, haben wir in der Vorlesung gezeigt. Aber wie sieht es andersherum aus?

Implementieren Sie `flatMap`, `flatten` und `map` via `pure` und `compose`.

```
def flattenViaCompose[F[_],A](ffa: F[F[A]])(using Monad[F]): F[A] = ???
def mapViaCompose[F[_],A,B](fa: F[A])(f: A => B)(using Monad[F]): F[B] = ???
def flatMapViaCompose[F[_],A,B](fa: F[A])(f: A => F[B])(using Monad[F]): F[B] = ???
```