Übungsblatt 07— Lösungen

1 Sequence in Cats

```
def sequence[F[_], A](fas: List[F[A]])(using mf: Monad[F]): F[List[A]] =
  fas.foldRight[F[List[A]]](mf.pure(Nil))((a, b) => a.map2(b)(_ :: _))
```

Hier hat sich zur Vorlesung eigentlich nichts geändert. Der einzige Unterschied ist, dass die Implementation jetzt nicht auf dem Monad Trait ist, sondern einen Monad für `F` bekommt. Deswegen muss die Monad-Instanz beim Aufruf von pure explizit benannt werden.

2 Identity Monad

```
given Monad[Id] with
  def pure[A](x: A): Id[A] = x
  def flatMap[A, B](fa: Id[A])(f: A => Id[B]): Id[B] = f(fa)
```

Dieses Encoding für den Identity-Monad ist vielleicht etwas verwirrend, aber sehr elegant. Der Typalias gibt an, dass `Id[A]` das gleiche ist wie `A`. Hierdurch lassen sich die Methoden `pure` und `flatMap` sehr einfach implementieren.

- `pure` hebt ein `A` in ein `F[A]`. Nachdem aber `F[A] = A` gilt, kann der übergebene Wert einfach zurück gegeben werden.
- `flatMap` muss normalerweise das `A` auspacken. Nachdem in diesem Id-Monad das eingepackte und ausgepackte A identisch sind, kann der Wert einfach direkt an die Funktion `f` übergeben werden.

3 Monad Laws

a)

Zu zeigen ist, dass die beiden Formulierungen des Assoziativgesetzes äquivalent sind:

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

```
compose(compose(f)(g))(h) == compose(f)(compose(g)(h))
```

Zur Erinnerung ist hier noch einmal die Definition von `compose`

```
def compose[A, B, C](f: A => F[B])(g: B => F[C]): A => F[C] =
  a => flatMap(f(a))(g)
```

Wir arbeiten uns hier von der `compose`-Formulierung zur `flatMap`-Formulierung vor und ersetzen dabei in der `compose`-Formulierung zunächst die äußeren `compose`-Aufrufe durch `flatMap`s entsprechend der obigen Definition.

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

Dann werden die inneren Aufrufe auf die gleiche Art ersetzt.

```
a => flatMap((b => flatMap(f(b))(g))(a))(h) == a => flatMap(f(a))(b => flatMap(g(b))(h))
```

Wir vereinfachen die linke Seite. Wir sehen, dass das innere Lambda, welches ein `b` nimmt, direkt mit `a` aufgerufen wird. Wir ersetzen also das `b` durch das `a` und eliminieren dadurch das innere Lambda.

```
a => flatMap(flatMap(f(a))(g))(h) == a => flatMap(f(a))(b => flatMap(g(b))(h))
```

Wir haben nun auf beiden Seiten Lambdas der Form `a => ...` stehen. Diese entfernen wir. Dann ersetzen wir das `f(a)` auf beiden Seiten durch `x`. Das ist kein Problem, weil `f` eine uneingeschränkte Funktion war. Das heißt, sie kann jedes beliebige `x` erzeugen.

```
flatMap(flatMap(x)(g))(h) == flatMap(x)(b => flatMap(g(b))(h))
```

Abgesehen von den Namen steht unsere ursprüngliche Formulierung bereits da. Wir ersetzen also einfach: `g` $\mapsto$ `f`, `h` $\mapsto$ `g` und `b` $\mapsto$ `a` und landen bei:

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

□

b)

Zu zeigen: Die Formulierungen der identity laws mit `compose` und `flatMap` sind je äquivalent

```
//left identity
compose(f)(pure) == f
flatMap(x)(pure) == x

//right identity:
compose(pure)(f) == f
flatMap(pure(y))(f) == f(y)
```

Widmen wir uns zuerst der **Links-Identität**:

Zunächst fügen wir eine übergebene Variable hinzu. Da Funktionen genau dann gleich sind, wenn sie bei Aufruf das gleiche Ergebnis liefern, ist das kein Problem.

```
compose(f)(pure)(y) == f(y)
```

Nun ersetzen wir `compose` durch `flatMap`, genau wie bei Aufgabe a).

```
(a => flatMap(f(a))(pure))(y) == f(y)
```

Wie zuvor können wir das `a` des Lambdas, nachdem das Lambda mit `y` aufgerufen wird, direkt durch `y` ersetzen.

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

Ähnlich wie bei der vorherigen Aufgabe substituieren wir einen Funktionsaufruf wieder durch dessen Ergebnis:

```
flatMap(x)(pure) == x
```

Widmen wir uns nun der **Rechts-Identität**:

Wir fügen wieder eine übergebene Variable hinzu.

```
compose(pure)(f)(y) == f(y)
```

Dann ersetzen wir `compose` wieder durch `flatMap`.

```
(a => flatMap(pure(a))(f))(y) == f(y)
```

Das übergebene `y` ersetzt wieder das anonyme `a`.

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

□

c)

Zu zeigen ist, dass die folgenden Gleichungen sowohl für den `Some` als auch den `None` Teil des Option-Monads erfüllt sind.

```
flatMap(x)(pure) == x
```

und

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

- Left Identity mit `None`:

  ```
  flatMap(None)(Some(_)) == None
  ```

  Wir wissen anhand der Implementation von `flatMap`, dass ein `None` direkt wieder ein `None` zurück gibt.

- Left Identity mit `Some`:

```
flatMap(Some(y))(Some(_)) == Some(y)
```

Laut der Definition von `flatMap`, wird das `y` einfach ëntpackt". Dann wird es mit `pure` wieder in ein `Some` gewrapped.

```
Some(y) == Some(y)
```

- Bei der Right Identity brauchen wir keine Fallunterscheidung, da die Variable `y` ein nicht monadischer Wert ist, der in den Monad verpackt wird.

```
flatMap(Some(y))(f) == f(y)
f(y) == f(y)
```

Wie bei der Left Identity für Some packt ein `flatMap` auf einem `Some` (das wir von Pure bekommen) den enthaltenen Wert einfach aus.

4 Monad Combinators

a)

Zu den folgenden Aufgaben gibt es wenig zu erklären. Man folgt den Typen.

- `flatten` via `flatMap`:

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

- `flatMap` via `flatten` und `map`:

```
def flatMapViaFlattenAndMap[F[_],A,B](fa: F[A])(f: A => F[B])(using Monad[F]): F[B]
↪  =
  fa.map(f).flatten
```

- `compose` via `flatten` und `map`:

```
def composeViaFlattenAndMap[F[_],A,B,C](afb: A => F[B])(bfc: B => F[C])(using
↪  Monad[F]): A => F[C] =
  a => afb(a).map(bfc).flatten
```

b)

- `flatten` via `compose`:

```
def flattenViaCompose[F[_],A](ffa: F[F[A]])(using Monad[F]): F[A] =
  compose(identity[F[F[A]]])(identity[F[A]]).apply(ffa)
```

- `map` via `compose` und `pure`:

```
def mapViaCompose[F[_],A,B](fa: F[A])(f: A => B)(using mf: Monad[F]): F[B] =
  compose(identity[F[A]])(a => mf.pure(f(a))).apply(fa)
```

- `flatMap` via `compose`:

```
def flatMapViaCompose[F[_],A,B](fa: F[A])(f: A => F[B])(using Monad[F]): F[B] =
  compose(identity[F[A]])(f).apply(fa)
```