Exercise sheet 07— Solutions

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)(_ :: _))
```

Here not much has changed from the lecture. The only difference is that the implementation is no longer on the Monad trait, but receives the Monad for `F` as parameter. Therefore the monad instance has to be named when calling pure.

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)
```

This encoding for the identity monad may be a bit confusing, but is pretty elegant. The type alias defines `Id[A]` to be the same as `A`. This way, methods like `pure` and `flatMap` can be implemented very simply

- `pure` lifts an `A` into an `F[A]`. But as `F[A] = A`, it can just return the input unchanged.
- `flatMap` usually has to unwrap the `A`. As the value packed in an Id is identical to the Id itself, the value can be passed to the function `f` directly.

3 Monad Laws

a)

We want to show that both formulations of the associativity law are equivalent:

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

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

As a reminder, this is how we defined `compose`:

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

We start from the `compose` formulation and work our way to the `flatMap` formulation. First we replace the outer `compose` calls in the `compose` formulation with `flatMap` calls, according to the above definition.

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

Then we do the same for the inner compose calls.

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

We simplify the left side. We can see, that the inner lambda, which takes a `b`, is called directly with `a`. So we replace `b` with `a` and eliminate the inner lambda that way.

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

Now we have lambdas of the form `a => ...` on both sides. We remove those. Then we replace `f(a)` on both sides with `x`. This is not a problem, as `f` was not restricted in any way, i.e. it can produce any arbitrary `x`.

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

We now have our original formulation, except for the names. So we replace: `g` $\mapsto$ `f`, `h` $\mapsto$ `g` and `b` $\mapsto$ `a` and get:

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

□

b)

We want to show that the formulations of the identity laws using `compose` and `flatMap` are equivalent:

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

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

We first look at the **left identity**:

We first add a variable for the passed in value. As functions are the same iff they return the same value for the same call, this is not a problem.

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

Now we replace `compose` with `flatMap`, just like in exercise a).

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

As before, we can replace the lambda's `a` by `y`, as the lambda is called directly with `y`.

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

Similar to the previous exercise, we substitute a function call with its result:

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

Now let's look at the **right identity**:

We again add a variable for the passed in value:

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

Then we replace `compose` by `flatMap`.

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

The passed `y` again replaces the anonymous `a`.

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

□

c)

We have to show that the following equations hold for the `Some` as well as the `None` part of the Option monad:

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

and

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

- Left Identity with `None`:

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

  Based on the implementation of `flatMap`, we know that calling it on `None` returns `None` again.

- Left Identity with `Some`:

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

Based on the implementation of `flatMap`, `y` is "unpacked". Then it is wrapped back into a `Some` using `pure`.

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

- For the right identity we don't need a case analysis, as the variable `y` is a non-monadic value that is lifted into the monad.

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

Similar to left identity with Some, a `flatMap` on a `Some` (which we get from pure) simply unwraps the contained value.

4 Monad Combinators

a)

There is not much to explain to the following solutions, they result from following the types.

- `flatten` via `flatMap`:

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

- `flatMap` via `flatten` and `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` and `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` and `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)
```