Exercise sheet for lecture 07— Monads

In this exercise we deal with monads and their laws. You can find the signatures of the methods shown below, as well as predefined implementations, in the Git repository at https://gitlab2.informatik.uni-wuerzburg.de/intro-to-fp/tasksheets.

1 Sequence in Cats

Implement the function `sequence` for any monadic type. We already saw an implementation for `Option` and `Either`.

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

Use the `Monad` type class from Cats as well as `foldRight` and `map2` for your implementation.

2 Identity Monad

Sometimes it is helpful to use functions defined for monads on types, which are not "‘wrapped"’ in a monadic type. For this we create a pseudo type, resp. a type alias, which converts simple types into type constructors.

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

Attention, this type is not identical to the one from the lecture. There we used a case class, here a type alias.

Implement a Monad instance for `Id`. The function `tailRecM` is provided.

`tailRecM`

In Cats, a monad instance has to provide an implementation of the function `tailRecM` in addition to the functions known from the lecture. This function implements stack safe recursive calls to `flatMap`. This implementation detail allows to ensure for all monads, that functions based on recursive `flatMap` calls, which are pretty common in practice, don't cause `StackOverflowError`s when they are implemented using `tailRecM`, as long as `tailRecM` is tail recursive for the monad instance. You can find details in the docs

3 Monad Laws

a) Proof, that the following formulations of the associativity law for monads (based on `flatMap` on the one hand, based on `compose` on the other) are equivalent:

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

The idea is to reshape one of the formulations into the other. Remember how `compose` was implemented.

b) Proof that the formulations of the *identity laws* are equivalent. Use a similar appoach as in exercise **a)**.

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

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

c) Proof that the *identity laws* (in their `flatMap` formulation) hold for the Option monad.

4 Monad Combinators

In the lecture, monads were introduced with the "‘minimal set of monad combinators"’ `flatMap` and `pure`. We'll learn two more such sets, which are sufficient for the existence of a monad. Use the functions from Cats' `Monad` type class for `pure`, `map`, `flatten` and `flatMap`. An own implementation is provided for `compose`.

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

Implement the function `flatten` via `flatMap`. `flatten` removes one layer of nesting from a nested monadic type.

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

Now implement `flatMap` and `compose` via `flatten` and `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` and `compose`**

We've seen that we can define `compose` in terms of `flatMap` in the lecture. But what about the other way round?

Implement `flatMap`, `flatten` and `map` using `pure` and `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] = ???
```