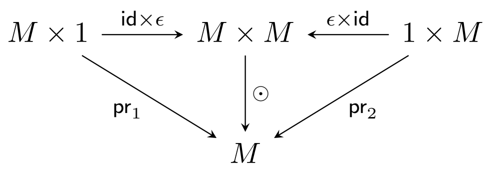
Monad-Based Programming SS 2019

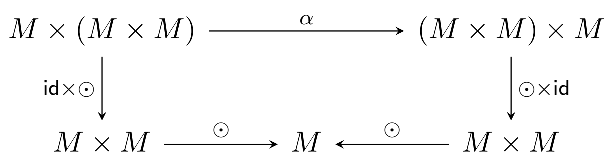
Assignment 3

Deadline for solutions: 13.06.2019, **12.15 a.m.**

Exercise 1 From Monoids to Monads (10 Points)

A *monoid* in a Cartesian category is a triple $(M, \epsilon, \odot)$ where M is an object; $\odot$ (*multiplication*) is a morphism $M \times M \to M$ and ϵ (*unit*) is a morphism $1 \to M$ such that the following diagrams commute:



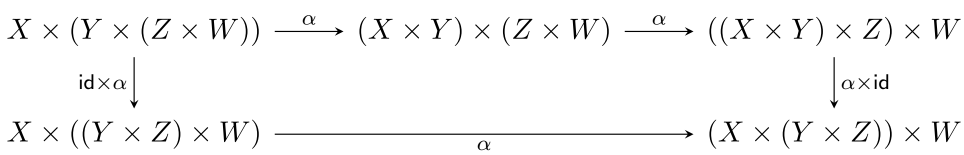


where $\alpha = \langle \mathsf{id} \times \mathsf{pr}_1, \mathsf{pr}_2\mathsf{pr}_2 \rangle : X \times (Y \times Z) \to (X \times Y) \times Z$ is the associativity morphism*.

The left diagram consists of two triangles expressing two unit laws: $x \odot \epsilon = x$ and $\epsilon \odot x = x$; the right diagram expresses associativity law: $x \odot (y \odot z) = (x \odot y) \odot z$.

1. A monoid M gives rise to a monad T_M, which we call an *M-module monad*, for which $T_M X = M \times X$; $\eta_X = \langle \epsilon \circ !, \mathsf{id} \rangle : X \to M \times X$; $f^* = (\odot \times \mathsf{id}) \circ \alpha \circ (\mathsf{id} \times f) : M \times X \to M \times Y$ for any $f : X \to M \times Y$. Prove that T_M is indeed a monad by diagram chasing.

   You can use without a proof that the following diagram commutes:

   Hint: reformulate the definition of the monad structure in terms of monad multiplication, using the definitions from the lecture.

2. Implement M-module monads in Haskell as a type class `MonoidModule m` parametrized by a monoid `m` and make it an instance of the standard type classes `Functor`, `Monad`, consistently with the previous clauses 1), 2), by completing the following declaration:

```
instance Functor (MonoidModule m)
instance Monoid m => Monad (MonoidModule m)
```

   (note that `m` need not be a monoid in the first declaration.) You can use the standard implementation of monoids in Haskell [2].

*Here $f \times g$ with $f : A_1 \to B_1$, $g : A_2 \to B_2$ denotes $\langle f \circ \mathsf{pr}_1, g \circ \mathsf{pr}_2 \rangle : A_1 \times A_2 \to B_1 \times B_2$.

Exercise 2 Counting Monads (10 Points)

1. Declare a Haskell type class

```
class Monad m => CountMonad m where
    inc :: ()  -> m ()
```

   where `inc` is a function determining the value used for incrementing the internal counter.

2. Make `MonoidModule n` from Excessice 1 an instance of `CountMonad` by completing the declaration

```
instance (Num n, Monoid n) => CountMonad (MonoidModule n) where
```

   Here we use `n` both as a monoid type and as a number type to specify a counter. The function `inc` must set the counter to 1.

3. Implement a recursive binary search function

```
findFirst :: (a -> Bool) -> Tree a -> IntCountMonad (Maybe a)
```

   where `IntCountMonad` is an alias for `MonoidModule (Sum Int)`, over the following data structures:

```
data Tree a = Leaf a | Node (Tree a) (Tree a) deriving Show
```

   searching for the first occurrence of the number satisfying the given predicate. Make sure to run `inc` before every recursive call of `findFirst`, e.g. as follows:

```
inc () >> findFirst n t
```

4. How can you interpret the value of the counter returned by `findFirst`?

References

[1] https://www.haskell.org/onlinereport/complex.html.

[2] https://hackage.haskell.org/package/base-4.7.0.1/docs/Data-Monoid.html