Functional and Logic Programming — Monads
Fall 2022

1 History

These notes
- were initially created for CPSC 720 in the summer of 2013.
- were revised in Fall 2017 for CPSC 370;
- were revised again in Fall 2018 for CPSC 370;
- are being used in conjunction with the teaching of CPSC 370 in the Winter 2022 term at the University of Northern British Columbia.

Contents

2 Category Theory

2.1 Category Theory definitions

A category $\mathcal{C}$ is a collection of *objects* and *arrows*. Before going any further, here are some examples of categories

Examples

- the category of vector spaces: the objects are vector spaces, the arrows are linear transformations between vector space;
- the category of sets: the objects are sets, and the arrows are arbitrary functions between sets;
- the category of groups: the objects are groups, and the arrows are group homomorphisms between groups;
- the category of topological spaces: the objects are topological spaces, and the arrows are *continuous* functions.
- a very small category (whose name I forget) that consists of one object and one arrow that goes from that object back to itself.
- the category of Haskell types. Here the objects are Haskell type names, and the arrows are Haskell functions.

Notation

Mathematicians typically use curly letters ($\mathcal{C}$, $\mathcal{D}$, $\mathcal{P}$, $\mathcal{G}$, and so on) to stand for entire (unspecific) categories. Specific categories we give names like **Set**. We use capital letters like C and D for objects, and lower case letters like f and g for arrows. To further help distinguish objects and arrows, we denote membership slightly differently, and write $C \in \mathcal{D}$, but f in $\mathcal{D}$.

An arrow goes from one object to another (possibly the same object). To say that f is an arrow from C to D we write: $f : C \to D$, or sometimes $C \xrightarrow{f} D$. Since every arrow has exactly one starting point and one ending point, we can define *domain* (dom) and *codomain* (cod) operators. for

$$f : C \to D \qquad \text{we have} \qquad \operatorname{dom} f = C \quad \text{and} \quad \operatorname{cod} f = D.$$

Furthermore, each object C has an arrow $\mathrm{id}_C : C \to C$.

Arrow Composition

Every category has a partial operation on arrows, $\circ$, that behaves like function composition.

For a pair of arrows f and g such that f ends where g starts, that is, $\operatorname{cod} f = \operatorname{dom} g$, there is an arrow $g \circ f$ from the domain of f to the codomain of g.

It may help to name some things: For a pair of arrows $f : C \to D$ and $g : D \to E$ there is a unique arrow $g \circ f : C \to E$. We can capture this in a *commuting diagram*. See Figure 1.

The operation $\circ$ is associative:

$$h \circ (g \circ f) = (h \circ g) \circ f. \tag{1}$$

and composing with an identity arrow does nothing. For any arrow $f : C \to D$ we have

$$f \circ \mathrm{id}_C = \mathrm{id}_D \circ f = f \tag{2}$$

A commuting diagram for this is shown in Figure 2.

Summary

1. A category $\mathcal{C}$ consists of objects and arrows.
2. For every object $D \in \mathcal{C}$ there is an arrow id_D in $\mathcal{C}$ from D to D.

$$\begin{tikzcd}
D \arrow[r, "g"] & E \\
C \arrow[u, "f"] \arrow[ur, dotted, "g \circ f"] &
\end{tikzcd}$$

Figure 1: A diagram for arrow composition

$$\begin{tikzcd}
C \arrow[r, "f"] \arrow[d, "\mathrm{id}_C"'] \arrow[dr, "f"] & D \arrow[d, "\mathrm{id}_D"] \\
C \arrow[r, "f"] & D
\end{tikzcd}$$

Figure 2: A commuting diagram for identity arrows

3. Every arrow f in $\mathcal{C}$ starts at an object $\operatorname{dom} f \in \mathcal{C}$ and ends at an object $\operatorname{cod} f \in \mathcal{C}$.
4. For a pair of arrows f, g in $\mathcal{C}$ where $\operatorname{cod} f = \operatorname{dom} g$ there is an arrow $g \circ f : \operatorname{dom} f \to \operatorname{cod} g$.
5. Arrow composition is associative: $h \circ (g \circ f) = (h \circ g) \circ f$.
6. Identity arrows compose as expected:

$$h \circ \mathrm{id}_{\operatorname{dom} h} = h \text{ and } \mathrm{id}_{\operatorname{cod} h} \circ h = h.$$

The Haskell category

Of particular interest to CPSC 370 students is the category of Haskell types and functions. For convenience we will denote this category **Hask**. The objects of **Hask** are Haskell types, and the arrows are functions.

Fact 1. *The* id *operator in the category of* HASKELL *types is the function* "`id`" (*defined by*

```
id :: a -> a
id x = x
```

). *Because* Haskell *allows type polymorphism* "`id`" *works for all types.*

Fact 2. *The* $\circ$ *operator in the category of Haskell types is just function composition and is written* "`.`".

We can define our own function composition if we want:

```
infixr 9 !!!
(g !!! f) x = g (f x)
```

2.2 Diagrams

Frequently we draw pictures to say these things. For instance, Figure 1 illustrates arrow composition, and Figure 2 illustrates (2).

If every arrow path through a diagram from one point to another composes to the same arrow, we say that we have a *commuting diagram*. A lot of the utility of category theory comes from being able to understand complicated ideas in terms of pictures (commuting diagrams).

Figure 3 shows a commuting diagram in the category of Haskell types.

The **Hask** category has *types* as its objects, and *functions* as its arrows. We can draw commuting diagrams as shown in Figure 3.

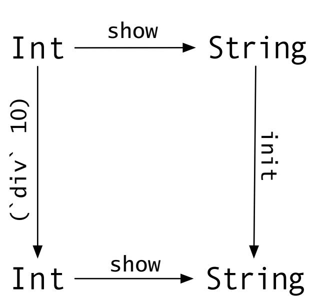

Figure 3: A commuting diagram in the *Haskell* category

3 Functors and the Category of Categories

Yes, there is a category of all categories, which we denote **Cat**. The objects in **Cat** are categories, the arrows are *functors*.

Loosely speaking a functor is something that copies diagrams from one category (its domain) to another category (its codomain).

A *functor* F is a function between two cateories (say $\mathcal{C}$ and $\mathcal{D}$). To be a functor F must satisfy the following rules:

- F maps objects to objects. If $C \in \mathcal{C}$ is an object in $\mathcal{C}$ then $F(C)$ is an object in $\mathcal{D}$.
- F maps arrows to arrows as follows. If $f : A \to B$ in $\mathcal{C}$ is an arrow in $\mathcal{C}$ then $F(f) : F(A) \to F(B)$ in $\mathcal{D}$ is an arrow in $\mathcal{D}$.
- F preserves domains, co-domains, and compositions of arrows. More precisely:
$$\operatorname{dom}(Ff) = F(\operatorname{dom} f) \tag{3}$$
$$\operatorname{cod}(Ff) = F(\operatorname{cod} f) \tag{4}$$
$$F(g \circ f) = F(g) \circ F(f). \tag{5}$$
- F preserves identiy arrows:
$$F(\mathrm{id}_A) = \mathrm{id}_{F(A)}. \tag{6}$$

To recapitulate, a functor is an arrow between two categories that consists of two functions: one that maps objects to objects, and one that maps arrows to arrows.

An Example

For instance there is a functor, called the forgetful functor from the category of vector spaces to the category of sets. The objects of the category of vector spaces are vector spaces. They get mapped to the objects of the category of sets, namely sets. The arrows of the category of vector spaces, namely linear transforms, get mapped to the arrows of the category of sets, namely plain old functions. Since every vector space is a set, and every linear transform is a function on the underlying set of vectors, this works.

3.1 Examples of functors in Haskell

The Haskell category has *types* as its objects, and *functions* as its arrows.

What are functors here then? More specifically what functors are there that go from the Haskell category back to itself? First we need some that maps objects (that is, types) onto objects. We already know about several of these! `Maybe`, `Either String`, and we can create our own, for instance,

```
type Triple x = (x, (x,x)) -- our own weird definition
```

However, an object map by itself is not enough. We also need an associated arrow map, that is a map from functions to functions.

For instance, for the `Maybe` type function, we need a map of type `(a -> b) -> (Maybe a -> Maybe b)`. Here is one possible definition

```
maybeMap :: (a -> b) -> (Maybe a -> Maybe b)
maybeMap f Nothing    = Nothing
maybeMap f (Just x)   = Just (f x)
```

Not every definition will do! We need to check that `maybeMap (g . f)` equals `maybeMap g . maybeMap f`, and that `maybeMap (id :: a-> a)` equals `id :: Maybe a-> Maybe a`.

similarly, we can define

```
tripleMap :: (a -> b) -> (Triple a -> Triple b)
tripleMap f (u,(v,w))  = (f u, (f v, f w))
```

and check that it also obeys the appropriate functor laws.

To tell Haskell that a type function is part of a functor, we say that it is an instance of the class `Functor`. For instance, we can write

```
instance Functor Triple where
  fmap f (u,(v,w)) =  (f u, (f v, f w))
  -- {- or -} fmap = tripleMap
```

In other words, the `Functor class` requires one function, `fmap`. Many Haskell type functions (such as `Maybe`) have already been declared to be instances of the `Functor` class.

4 Endo-functors and Monads

An *endo-functor* is a functor from a category back to itself.

Fact 3. *There are several interesting endo-functors on the category of* HASKELL *types. One of these relates to* HASKELL *lists. On objects, it maps a type* `T` *to* `[T]` *(lists of* `T`*). On functions, it maps* `f` *to* `map f`*.*

Recall that `map` is defined on lists by

```
map f [] = []
map f (x : xs) = f x : map f xs
```

\+ **Question 4.** Verify that `[]` and `map` indeed give an endo-functor on the category of HASKELL types.

\+ **Question 5.** Show that map `T` $\mapsto$ `Maybe T` is the object part of an endo-functor on the category of HASKELL types. What is the corresponding map on arrows (here functions)?

A *monad* is a special kind of endo-functor. Mathematicians usually define a monad as a triple (T, η, μ) where T is an endo-functor, and η and μ are natural transformations (whatever that means[1]) Computer scientists often use an equivalent definition involving *Kleisli triples*. The triple $(T, \eta, ^*)$ is a Kleisli triple if

- T is an endo-functor.
- η_C is an arrow from C to TC.
- For $f : C \to TD$, f^* is an arrow $f^* : TC \to TD$.
- For $f : C \to TD$,

$$f = f^* \circ \eta_C. \tag{7}$$

Fact 6. *The endo-functor on the category of* HASKELL *types give by* `T` $\mapsto$ `[T]` *and* `f` $\mapsto$ `map f` *is also a monad. Here the* η *function is given by*

```
singleton :: a -> [a]
singleton x = [x]
```

[1]something like a function between functors. See Section A.

$$\begin{tikzcd} C \arrow[r, "f"] \arrow[d, "\eta_C"'] & TD \\ TC \arrow[ur, "f^*"] \arrow[r, "Tf"'] & TTD \arrow[u, "\mu"'] \end{tikzcd}$$

Figure 4: Chasing arrows for f^*

and the * *function is given by*

```
star :: (a -> [b]) -> ([a] -> [b])
star f aList = (concat . map f) aList
```

.

Fact 7. *The endo-functor on the category of* HASKELL *types give by* `T` $\mapsto$ `Maybe T` *and the function from Question 5 also form a monad.*

\+ **Question 8.** What are the η function and the * function for the (`Maybe`,η, *) monad?

Suppose that $(T, \eta, ^*)$ is a Kleisli triple. Consider the expression $(\eta^*)^*$. Take an object $C \in \mathcal{C}$. Then η_C is an arrow $\eta_C : C \to TC$, so we can apply star to get an arrow $\eta_C^* : TC \to TC$. When we apply * again we get an arrow $(\eta_C^*)^* : T(TC) \to TC$. This is usually where mathematicians start. What we call η^{**} they usually call μ.

We can also define * in terms of μ. If $f : C \to TD$ is an arrow, then so is $\mu \circ Tf : TC \to TD$, and in fact this is f^*. (To reason about these kinds of things, mathematicians often draw diagrams like Figure 4.) The conclusion is that it doesn't matter whether we start with * or with μ.

Example 9. Let us work out what μ is for the list monad. Chasing definitions we get that $\mu =$

```
concat . map  $  concat . map  $  \ x -> [x]
```

This looks pretty intimidating, but let's work out

```
concat ( map  (\ x -> [x]) [1,3,2] )
```

This is `concat [[1],[3],[2]]` or `[1,3,2]`. In fact, in general, "`concat . map $ \ x -> [x]`" is "`id`" on any list type. So μ = `concat . map $ id` = `concat . map id` = `concat . id` = `concat`.

+ **Question 10.** What is μ in the `Maybe` monad?

5 More about Haskell types

Haskell has *ad hoc* type classes. The closest JAVA idea is an `interface`.

The syntax of a `class` definition uses the keywords class and where and is

```
class ClassName typeParm where
  decls...
```

For instance,

```
class PluralizableClass c where
  many :: c -> [c]
  singular :: [c] -> c
```

says that a class `c` is a `PluralizableClass` if it has functions `many` and `singular` with the appropriate signatures.

To say that a particular type is an instace of a class (translation to JAVA: "*to say that a particular class implements an interface*") we use the keywords instance and where.

For instance, suppose that we have our own data type

```
  data Some a = One a | Two a a | Lots [a]
```

we can write

```
  instance PluralizableClass (Some a) where
    many (One x)      = [x]
    many (Two a b)    = [a,b]
    many (Lots xs)    = xs
    singular [x]      = One x
    singular [x,y]    = Two x y
    singular xs       = Lots xs
```

5.1 Haskell classes for Category Theory

There are three Haskell classes relevant to monads. They are `Functor`, `Applicative`, and `Monad`. Every `Monad` is an `Applicative`, and every `Applicative` is a `Functor`.

Slightly simplified class definitions are

```
class Functor t where
  fmap :: (a -> b)  -> (t a -> t b)

class (Functor t) => Applicative t where
  pure :: a -> t a
  <*>  :: t (a -> b) -> t a -> t b

class (Applicative t) => Monad t where
  return :: a -> t a
  >>=    :: t a -> (a -> t b) -> t b
```

The `Functor` class corresponds directly to mathematical functors, with `fmap` being the arrow (function) map that corresponds to the object (type) map `t`.

The `Applicative` class is slightly odd. Its "`pure`" function is like "`return`" in monads. The `<*>` function says that lifted functions can be applied to lifted objects. Not all `Applicative` t are monadic, but for those that are, `star` defined in the monad as

```
star :: Monad m :: m (a -> b) -> m a -> m b
star ff xs = do
    f <- ff
    x <- xx
    return (f x)
```

is equivalent to `<*>`.

The `Monad` class is a fairly direct translation of Kleisli triples. Its "`return`" function is polymorphic and corresponds to η. The Haskell "`m >>= f`" is the Kleisli $f^*(m)$.

5.2 Doing the classes upside down

Modern Haskell requres that every `Monad` is an `Applicative` and every `Applicative` is a `Functor`. However, Haskell does not require us to define them in that order!

For instance, let's make `Some` into a monad.

```
instance Monad Some where
  return x = One x
  xs >>= f = singular $ many xs >>= f
```

(the definition of `>>=` makes use of the corresponding function for lists.)

Using the monad defintions we can now give instances for `Applicative` and `Functor`

```
instance Applicative Some where
  pure = return
  fs <*> xs = fs >>= (\ f -> xs >>= (\ x -> return (f x)))
```

(We will see in the next section that we can also write
`fs <*> xs = do { f <- fs; x <- xs ; return (f x) }`)

We also have

```
instance Functor Some where
  fmap f xs = (xs >>= (return . f))
```

Note that these definitions are completely generic and will work for any monad.

6 "do" notation and Monads in Haskell

Consider the HASKELL code

```
do
 x <- [1,2,3]
 y <- ["a", "b"]
 return (x,y)
```

This computes the list

```
[(1,"a"), (1,"b"), (2,"a"), (2,"b"), (3,"a"), (3,"b")] .
```

How does this work? First of all, “do” blocks always involve monads. Here the monad is the list monad. The result is always in the monad (in this example, “is a list”). HASKELL converts the code above to

```
[1,2,3] >>= ( \ x -> ["a", "b"] >>= ( \ y -> return (x,y)) )
```

This is generic. Now, in the list monad this becomes,

```
[1,2,3] >>= ( \ x -> ["a", "b"] >>= ( \ y -> [(x,y)] ) )
```

or

```
concat . map(\x-> ["a", "b"] >>= (\y-> [(x,y)] ) $ [1,2,3]
```

or

```
concat . map(\x-> concat . map(\y-> [(x,y)]) $ ["a", "b"]) $ [1,2,3]
```

Most people find the “do” notation easier to read and understand intuitively even if they’ve never heard of an endo-functor.

More formally the rules for a `do` expression are

- `do { pat <- expr; ... }` means the same as `expr >>= (\pat -> do {...})`
- `do { expr; ... }`. means the same as `expr >>= (\_ -> do {...})`.
- `do {expr}` means the same as `expr`.

+ **Question 11.** What does

```
do
 x <- Just 5
 y <- Just 3
 return $ x*y
```

compute?[2] Check it out!

+ **Question 12.** What does

[2]Beware! `return x * y` generates a strange error because it parses as `(return x) * y`. Be very aware that `return` is *not* a control structure.

```
do
 x <- Just 5
 return $ x * x
 y <- Just 3
 return $ y*y
```

compute? Check it out!

+ **Question 13.** Suppose that we define

```
data Array a = Leaf a | Tree Integer (Array a) (Array b)
-- ...
append a b = Tree (size a + size b) a b
catamorph leaf tree x = case x of
  Leaf a -> leaf a
  Tree n a b -> tree (cata leaf tree a) (cata leaf tree b)
```

1. What does "`catamorph`" do? In particular,
2. What does "`catamorph Leaf append`" do?
3. What does "`catamorph (const 1) (+)`" do?
4. What is the type of "`catamorph`" ?
5. What is the type of "`catamorph id append`" ?
6. Can you build a monad using `Array` as the object functor?

7 A "Useless" Monad

Consider the `Empty` type defined by

```
data Empty a = Void
```

This type raises some puzzling type questions, such as, what is the difference between "`Empty (Empty String)`" and "`Empty Integer`"?

Nonetheless, it gives rise to a monad.

+ **Question 14.** Determine what `return` and `>>=` are for the `Empty` monad.

8 Standard Haskell Monads

Here is a list of pre-built Haskell monads. To use, for instance, the `State` monad, import `Control.Monad.State.`

1. The `Identity` monad. The object (type) function maps `a` to `a`.
2. The `Reader` monad is paramaterized by an additional type parameter r. The object (type) function maps `a` to `r -> a`. "`return x = \ _ -> x`" or "`return = const`".

   This monad is hidden in undergraduate mathematics where we confuse the number 3 with the constant function on the real numbers $x \mapsto 3$.
3. The `State` monad is paramaterized by an additional type parameter s. The object (type) function maps `a` to `s -> (s,a)`. We have "`return x = \ s -> (s,x)`".

   The state monad corresponds to computing with memory.

8.1 The State Monad

Programming in C++ does not appear to be functional. An expression like "`c++`"
(a) depends on the current state of memory; and
(b) alters the current state of memory.

Thus, if we want to view the meaning of "`c++`" as a function it has to be a function of the form $M \to \mathbb{Z} \times M$ where M represents the current state of memory. Similarly, in general we might want to model real-valued C++ expressions by functions of the form $M \to \mathbb{R} \times M$.

This suggests that there is a useful functor (let's call it T lurking in the background, where $\mathrm{T}\,A = (A \times M)^M$ (or $\mathrm{T}\,A = (M \to A \times M)$ in more Haskell-ish notation). In fact, T gives rise to a monad called the State monad that we shall consider below.

8.2 Implementing the State monad

In this section, I shall give one explicit way to create a `State` monad. It is similar to what the standard Haskell library does, but likely differs in the details. To use the standard implementation

```
import Control.Monad.Trans.State
```

For syntax reasons we want an explicit, distinct, datatype associated with our state monad functor. One approach would be to write

```
data State s a = State ( s -> (a,s) )
```

However, Haskell has syntax specifically designed for this situation, and it is more idiomatic to write

```
newtype State s a = State { runState :: s -> (a,s) }
```

The monad instance for `State s` is

```
instance Monad (State s) where
  return x = State $ \ s -> (x,s)
  m >>= f  = State $ \ s -> let
                (b,s2) = runState m s
                in runState (f b) s2
```

Note how we keep using `runState` to convert back to a plain function type. The constructor `State` and the function `runState` are inverses: "`runState . State $ m`" is `m`.

The functor instance uses the same kind of logic:

```
instance Functor (State s) where
  fmap f m  = State $ \ s -> let
                (b,s2) = runState m s
                in (f b,s2)
```

Finally, the applicative intance can steal from the monad instance; here we spell that out

```
instance Applicative (State s) where
  pure x    = State $ \ s -> (x,s)
  ff <*> xx = State $ \ s1 -> let
                (f,s2) = runState ff s1
                (x,s3) = runState xx s2
                in (f x,s3)
```

In order for the `State` monad to be useful we need to be able access the state (and change it!). Typically we build access and modfication on the following two functions:

```
get :: State s s
get = State $ \s -> (s,s)

put :: s-> State s ()
put s1 = State $ \ _s0_ -> ((), s1)
```

Haskell provides another useful function related to `get`, defined by

```
gets :: (s->b) -> State s b
gets f = State $ \s -> (f s,s)
```

This is probably clearer in "`do`" notation:

```
gets f = do { s <- get ; return (f s) }
```

Another useful function `modify` applies a function `f` directly to the state using `get` and `put`

```
modify :: (s -> s) -> State s ()
modify f = do
  state <- get
  put (f state)
```

8.3 Using the state monad

What is the state monad useful for? The answer is, calculations where there is some kind of underlying object that is constantly being updated. We can explicitly code this by passing the object(s) being modified by as parameters, but it is conceptually cleaner to acknowledge the existence of state.

8.3.1 Example: Using a State stack

One example comes from constructing `fromList` functions for particular foldable types. Converting a foldable into a list is generic — `foldMap (:[])` will do — but going the other direction requires specifics of the foldable type.

The idea is to use `State [a]` as an implicit stack. To that end, let us write some functions:

```
popTree :: n -> State [q] (Tree q)
popTree 0 = return EmptyTree
popTree n = do
  let ellN = (n-1) ‘div‘ 2
  let arrN = n ‘div‘ 2
  -- assertion ellN + arrN + 1 == n
  -- the following builds in infix order, adjust to taste
  left <- popTree ellN
  root <- pop
  right <- popTree arrN
  return $! Node root left right
```

Figure 5: Building a Tree in the state monad

```
stackSize :: State [a] Int
stackisEmpty :: State [a] Bool
push :: a -> State [a] ()
pop :: State [a] a
```

with implementations

```
stackSize = gets length
stackisEmpty = gets null
push s = do { xs <- get ; put (x:xs) }
-- or push = modify (x:)
pop = do
  x <- gets head
  modify tail -- yes, this is real code.  What does it do?
  return x
```

(As a matter of style, we could separarate effect and value computations, and split `pop` into two parts: `top = gets head` and `popNoValue = modify tail`.)

Now, let’s use this machinery to build a `Tree q` from a `[q]` .

The actual tree building is done inside the `State [q]` monad as shown in Figure 5. The outer function `treeFromList` sets up a monadic calculation and runs it:

```
treeFromList :: [q] -> Tree q
treeFromList xs = let
  n = length xs
  monad = popTree n
  (tree,_) = runState monad xs
  in tree
```

8.3.2 Example: Decorating a Traversable

In this example we take a traversable container `t q` and convert it into a `t (Int,q)`, giving each element a unique positive integer number. Removing a decoration is easier

```
undecorate :: (Functor t) => t (a,b)   -> t b
undecorate xs = fmap snd xs -- snd is \ (a,b) -> b
```

However, generating distinct integers requires memory, that is `State`. We start by implementing something like C++'s `++n`.

```
plusPlus :: State Int Int
plusPlus = do { modify (+1) ; get }
```

Question 15. Show how to mimic postfix `n++`.

Next we figure out how to decorate an individual element.

```
decorateElt :: a-> State Int (Int,a)
decorateElt x = do
   n <- plusPlus
   return (n,x)
```

Question 16. For `xs` of type `Traversable t=>(t q)`, what is the type of `traverse decorateElt xs`?

And then we put it all together

```
decorate :: Traversable t => t a-> t (Int,a)
decorate xs = runState (traverse decorateElt xs) 0
```

+ **Question 17.** What happens if change the `0` above to some other `Int`?

+ **Question 18.** Write a function `cumulative :: Traversable t => t Int -> t Int` that replaces each element with the cumulative sum traversed at that point.

For isntance, `cumulative [1,3,2]` is `[1,4,6]`.

... to be continued.

A Aside: Natural Transformations, mathematically speaking

We don't really need to know what a natural transformation is, but here goes. Suppose that we have two categories $\mathcal{C}$ and $\mathcal{D}$, and two functors F and G that go from $\mathcal{C}$ to $\mathcal{D}$.

$$\mathcal{C} \underset{G}{\overset{F}{\rightrightarrows}} \mathcal{D}$$

Then a natural transformation ϕ from F to G is a function from the objects of $\mathcal{C}$ to the arrows of $\mathcal{D}$ such that for every object $A \in \mathcal{C}$ there is an arrow $\phi_A : FA \to GA$ in $\mathcal{D}$. Furthermore ϕ must satisfy the rule that for every arrow $A \xrightarrow{f} B$ in $\mathcal{C}$ the digagram below commutes.

$$\begin{tikzcd}
A \arrow[d, "f"'] & \mathrm{F}\,A \arrow[r, "\phi_A"] \arrow[d, "\mathrm{F}\,f"] & \mathrm{G}\,A \arrow[d, "\mathrm{G}\,f"] \\
B & \mathrm{F}\,B \arrow[r, "\phi_B"] & \mathrm{G}\,B
\end{tikzcd}$$

A.1 With respect to Monads

For a monad (T, η, μ), η is a natural transformation from the identity functor to T. Translating the previous diagram into Haskell, we get the requrement that

$$\texttt{fmap f . return} \equiv \texttt{return . f} \tag{8}$$

Furthermore, μ is a natural transformation from $\mathrm{T} \circ \mathrm{T}$ to T. In Haskell, μ is usually called `join`. Translating the previous diagram into Haskell, we get the requrement that

$$\texttt{fmap f . join} \equiv \texttt{join . fmap (fmap f)} \tag{9}$$

Using the fact that `join` is just `(>>= id)` we get another version

```
fmap f (m >>= id) ≡ (fmap (fmap f) m) >>= id
```