A link: Erik Meyer @ FooCafé

https://www.youtube.com/watch?v=JMP6gI5mLHc

A category $C$ consists of the following three entities:
- A class $ob(C)$ of objects;
- A class $hom(C)$ of morphisms (also called maps or arrows). Each morphism $f$ has a unique source object $a$ and target object $b$. The expression $f : a \rightarrow b$ is read “$f$ is a morphism from $a$ to $b$”. $hom(a, b)$ denotes the class of all morphisms from $a$ to $b$;
- morphism composition (see next slide).
CATEGORIES

A category $C$ consists of the following three entities:
- objects (see previous slide);
- morphisms (see previous slide);
- A binary operation $\circ$, called composition of morphisms such that the following axioms hold:

  **Associativity:** If $f : a \to b, g : b \to c, h : c \to d$ then $h \circ (g \circ f) = (h \circ g) \circ f$, and

  **Identity:** For every object $x$ there exists a morphism $1_x : x \to x$ called the identity morphism for $x$, such that for every morphism $f : a \to b$ we have $1_b \circ f = f = f \circ 1_a$.

FUNCTORS

Functors are structure-preserving maps between categories:

A (covariant) functor $F$ from a category $C$ to a category $D$, written $F : C \to D$, consists of:
- for each object $x$ in $C$, an object $F(x)$ in $D$;
- for each morphism $f : x \to y$ in $C$, a morphism $F(f) : F(x) \to F(y)$,

such that the following two properties hold:
- For every object $x$ in $C$, $F(1_x) = 1_{F(x)}$;
- For all morphisms $f : x \to y$ and $g : y \to z$, $F(g \circ f) = F(g) \circ F(f)$.

Informally: a contravariant functor is like covariant, except that it reverses all morphisms (arrows).

THE FUNCTOR CLASS

Consider the following class:

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

The `fmap` function generalizes the `map` function used previously.

```haskell
instance Functor [] where
    fmap = map
```

FUNCTOR AXIOMS

Functor laws (not enforced by Haskell but necessary to ensure correctness):

- $fmap id = id$
- $fmap (f . g) = (fmap f) . (fmap g)$

The laws means that `fmap` does not alter the structure of the functor.
**A Functor example**

Another instance:

```haskell
data Tree a = Leaf a | Branch (Tree a) (Tree a)
instance Functor Tree where
  fmap f (Leaf x) = Leaf (f x)
  fmap f (Branch t1 t2) = Branch (fmap f t1) (fmap f t2)
```

**Applicative functors**

```haskell
class (Functor f) => Applicative f where
  pure :: a -> f a
  (<>*) :: f (a -> b) -> f a -> f b
instance Applicative Maybe where
  pure = Just
  Nothing <>* _ = Nothing
  (Just f) <>* something = fmap f something
```

**The Monad class**

```haskell
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  (>>) :: m a -> m b -> m b
  return :: a -> m a
  fail :: String -> m a

Minimal complete definition requires only >>= and return, as the other two have the default definition:

```haskell
m >>= k = m >>= \_ -> k
fail s = error s
```

**Re: definition**

Kleisli triple

- A **type construction** for type `a` create `Ma`
- A **unit function** `a -> Ma` (return in Haskell)
- A **binding operation** of polymorphic type
  
  ```haskell
  Ma -> (a -> Mb) -> Mb.
  ```

  Four stages (informally):
  
  - The monad-related structure on the first argument is "pierced" to expose any number of values in the underlying type `a`.
  - The given function is applied to all of those values to obtain values of type `(M b)`.
  - The monad-related structure on those values is also pierced, exposing values of type `b`.
  - Finally, the monad-related structure is reassembled over all of the results, giving a single value of type `(M b)`.
Categorical view on Haskell monads

Instead of \textit{return} and \textit{bind}, we can define a monad by \textit{return} (or \textit{pure}), \textit{fmap} and \textit{join}:

\[
\begin{align*}
\text{fmap} & : (a \to b) \to m a \to m b \\
\text{join} & : m (m a) \to m a
\end{align*}
\]

with the mutual relations as follows:

\[
(f \text{map } f) t = t >>= (\lambda x \to \text{return } (f x))
\]

\[
\text{join } n = n >>= \text{id}
\]

The first one is sometimes written as:

\[
\text{fmap } f t = t >>= (\text{return } . f)
\]

More laws

For \textit{pointed functors} in the same category:

\[
\text{return } . f = \text{fmap } f . \text{return}
\]

What is a \textit{pointed functor} then?

\[
\begin{align*}
\text{class } \text{Pointed } f \text{ where} \\
\text{return} & : a \to f a \\
\text{point} & : a \to f a
\end{align*}
\]

Monad laws expressed with \textit{join}:

\[
\begin{align*}
\text{join } . \text{fmap } \text{join} &= \text{join } . \text{join} \\
\text{join } . \text{fmap } \text{return} &= \text{join } . \text{return } = \text{id} \\
\text{join } . \text{fmap } (f \text{map } f) &= \text{fmap } f . \text{join}
\end{align*}
\]

Yet another variant

In the monad context we define sometimes:

\[
\begin{align*}
\text{lift}^M & : (\text{Monad } m) \Rightarrow (a \to b) \to (m a \to m b) \\
\text{lift}^M f & = \{ x \to do \{ x' \leftarrow x; \text{return } (f x') \} \}
\end{align*}
\]

So that e.g. \text{lift}^M \sin (\text{Just } 0) evaluates to \text{Just } 0.0

Relation to category theory

Intuitively:

“A Haskell monad corresponds to a \textit{strong monad} in a \textit{cartesian closed category}. A category is cartesian closed if it has enough structure to interpret \textit{$\lambda$}-calculus. In particular, associated with any pair of objects (types) $x$ and $y$ there is an object $[x \to y]$ representing the space of all functions from $x$ to $y$. $M$ is a functor if there exists (for any arrow $f$) the arrow $f \text{map } f$ obeying the functor laws. A functor is \textit{strong} if it itself is represented by a single arrow $f \text{map}$.”

P. Wadler (1990)

“A monad is a monoid in the category of endofunctors.”
The List monad

instance Functor [] where
    fmap = map

instance Monad [] where
    return x = [x]
    xs >>= f = concat (map f xs)
    fail s = []

instance MonadPlus [] where
    mzero = []
    mplus = (++)

The Maybe monad

instance Functor Maybe where
    fmap f (Just x) = Just (f x)
    fmap f Nothing = Nothing

instance Monad Maybe where
    return x = Just x
    Just x >>= f = f x
    Nothing >>= f = Nothing

instance MonadPlus Maybe where
    mzero = Nothing
    Nothing 'mplus' ys = ys
    xs 'mplus' ys = xs

Re: definition

Monad axioms:

1. \( \text{return} \) acts as a neutral element of \( >>= \).

\[(\text{return } x) >>= f \leftrightarrow f x\]
\[m >>= \text{return} \leftrightarrow m\]

2. Binding two functions in succession is the same as binding one function that can be determined from them.

\[(m >>= f) >>= g \leftrightarrow m >>= (\lambda x. (f x >>= g))\]

We can restate it in a cleaner way (Thomson). Define

\[\text{>=>} :: \text{Monad } m \Rightarrow (a \to m \, b) \to (b \to m \, c) \to (a \to m \, c)\]

\[f \text{>=>} g = \lambda x. (f x) \text{>=>} g\]

Now, the monad axioms may be written as:

\[\text{return} \text{>=>} f = f\]
\[f \text{>=>} \text{return} = f\]

\[(f \text{>=>} g) \text{>=>} h = f \text{>=>} (g \text{>=>} h)\]

Note: In Thomson's book \(\text{>=>}\) is \(\emptyset\).

Note 2: It is so called Kleisli composition.
Lambda calculus

- Introduced by Alonzo Church (1933)
- A set of \( \lambda \text{-terms} \) and rules to manipulate them
- Origin of functional programming (LISP, 1960)
- Equivalent expressivity to recursive functions (Gödel) and Turing Machines

\[ \lambda x. E(x) \]

denotes a function that, given input \( x \), computes \( E(x) \). To apply this function, one substitutes the input for the variable and evaluates the body, e.g.:

\[ \lambda x. (x + 1) \]

is the successor function on natural numbers. To apply it for input 7 one performs substitution and then evaluates:

\[
(\lambda x. (x + 1))(7) \rightarrow (7 + 1) \rightarrow 8
\]

Note: curried form!

An example

A higher-order example:

\[ \lambda f. \lambda g. \lambda x. f(g(x)) \]

Applying it to the successor function \( \lambda x. (x + 1) \) twice yields:

\[
(\lambda f. \lambda g. \lambda x. f(g(x)))(\lambda y. (y + 1))(\lambda z. (z + 1)) \rightarrow (\lambda g. \lambda x. ((\lambda y. (y + 1))(g(x)))(\lambda z. (z + 1)) \rightarrow (\lambda x. ((\lambda y. (y + 1))(\lambda z. (z + 1))x)) \rightarrow \lambda x. ((\lambda y. (y + 1))(x + 1)) \rightarrow \lambda x. ((x + 1) + 1)
\]

Pure lambda calculus

In pure \( \lambda \)-calculus, there are only variables \( f, g, h, ..., x, y, z, ... \) and operators for \( \lambda \)-abstraction and application.

\( \lambda \text{-terms} \) are recursively created from these:

- any variable \( x \) is a \( \lambda \text{-term} \);
- if \( M \) and \( N \) are \( \lambda \text{-terms} \), then \( MN \) is a \( \lambda \text{-term} \) (functional application);
- if \( M \) is a \( \lambda \text{-term} \) and \( x \) is a variable, then \( \lambda x.M \) is a \( \lambda \text{-term} \) (functional abstraction).

Application is not associative, i.e. usually \( (MN)P \neq M(NP) \).
\( MNP \) is interpreted as \( (MN)P \).
Some interesting facts

- In pure $\lambda$-calculus, $\lambda$-terms serve both as functions and as data. (+1 above was informal!)
- The substitution rule above is called $\beta$-reduction.
- Renaming variables (e.g. $\lambda x.zx$ to $\lambda y.zy$) is called $\alpha$-reduction.
- Computations in $\lambda$-calculus is performed by $\beta$-reducing terms whenever possible and as long as possible.
- Theorem (Church-Rosser): the order of reductions does not matter (as there will always be some common final reduction).
- A term is in normal form if no $\beta$-reductions apply (halting state of TM).
- There are terms with no normal form (corresponding to non-halting computations of TMs). E.g. $(\lambda x.xx)(\lambda x.xx)$.

Church numerals

$0 \overset{df}{=} \lambda f.\lambda x.x$
$1 \overset{df}{=} \lambda f.\lambda x.fx$
$2 \overset{df}{=} \lambda f.\lambda x.f(fx)$
$3 \overset{df}{=} \lambda f.\lambda x.f(f(fx))$
$
\vdots$
$n \overset{df}{=} \lambda f.\lambda x.f^n x$
$
\vdots$

Then successor may be defined as:

$\lambda m.\lambda f.\lambda x.f(mfx)$

Recursive functions

- **Successor**: the function $s : N \to N$ given by $s(x) = x + 1$ is computable;
- **Zero**: the function $z : N^0 \to N$ given by $z(0) = 0$ is computable;
- **Projections**: The functions $\pi^k_n : N^n \to N$ given by
  $\pi^k_n(x_1, \ldots, x_n) = x_k$, for $1 \leq k \leq n$ is computable;
- **Composition**: If $f : N^k \to N$ and $g_1, \ldots, g_k : N^n \to N$ are computable, then so is the function $f \circ (g_1, \ldots, g_k) : N^n \to N$ that on input $\vec{x} = x_1, \ldots, x_n$ gives $f(g_1(\vec{x}), \ldots, g_k(\vec{x}))$.

**Primitive recursion**: If $h_i : N^{n-1} \to N$ and $g_i : N^{n+k} \to N$ are computable, $1 \leq i \leq k$, then so are functions $f_i : N^n \to N$, $1 \leq i \leq k$, defined by mutual induction as follows:

$f_i(0, \vec{x}) \overset{df}{=} h_i(\vec{x}),$

$f_i(x + 1, \vec{x}) \overset{df}{=} g_i(x, \vec{x}, f_i(1, \vec{x}), \ldots, f_i(k, \vec{x})),$

where $\vec{x} = x_2, \ldots, x_n$.

**Unbounded minimization**: If $g : N^{n+1} \to N$ is computable, then so is the function $f : N^n \to N$ that on input $\vec{x} = x_1, \ldots, x_n$ gives the least $y$ such that $g(z, \vec{x})$ is defined for all $z \leq y$ and $g(y, \vec{x}) = 0$ if such a $y$ exists and is undefined otherwise. We denote this by

$f(\vec{x}) = \mu y. (g(y, \vec{x}) = 0)$. 

Recursive functions
Recursive functions

- primitive recursive functions obey (1) – (5)
- μ-recursive functions obey (1) – (6)
- There exists a non-primitive (total) recursive function
  (Ackermann’s function)

\[
A(0, y) = y + 1,
\]
\[
A(x + 1, 0) = A(x, 1),
\]
\[
A(x + 1, y + 1) = A(x, A(x + 1, y)).
\]
- Primitive recursive functions are total, μ-recursive may be partial.
- Recursive functions correspond to Turing Machines.

Turing Machine

- a tape
- an alphabet (with a “blank”)
- a head over the tape
- read or write operation
- left or right tape movement
- state (finitely many)
- transition function (may be partial)

\[
\delta : (Q \setminus F) \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}
\]
- Universal Turing Machines!