# A Monad Rosetta Stone

for mathematicians and functional programmers

Ed Morehouse

Wesleyan University

# why monads?

In mathematics, monads arise most naturally from adjunctions, which are generalizations of the Galois correspondences that may be familiar from algebra or abstract interpretation.

But monads are also useful in themselves. While adjunctions emphasize relations *between* categories, monads emphasize relations *within* a category.

Many familiar mathematical structures are monads, giving us useful theorems about them for free.

# why monads?

In mathematics, monads arise most naturally from adjunctions, which are generalizations of the Galois correspondences that may be familiar from algebra or abstract interpretation.

But monads are also useful in themselves. While adjunctions emphasize relations *between* categories, monads emphasize relations *within* a category.

Many familiar mathematical structures are monads, giving us useful theorems about them for free.

# why monads?

In mathematics, monads arise most naturally from adjunctions, which are generalizations of the Galois correspondences that may be familiar from algebra or abstract interpretation.

But monads are also useful in themselves. While adjunctions emphasize relations *between* categories, monads emphasize relations *within* a category.

Many familiar mathematical structures are monads, giving us useful theorems about them for free.

# why monads?

In functional programming, *pure* languages have several desirable properties:

- program text close to program meaning
- no model of memory or global state required to understand program behavior

This makes programs simpler, clearer and easier to debug and optimize.

But purity comes at a cost: pure programs can't produce or consume side effects, don't have mutable state, and can require complicated or error-prone solutions to some simple problems (e.g. keeping a counter).

Functional programmers long for the virtues of purity without having to give up their impure indulgences.

# why monads?

In functional programming, *pure* languages have several desirable properties:

- program text close to program meaning
- no model of memory or global state required to understand program behavior

This makes programs simpler, clearer and easier to debug and optimize.

But purity comes at a cost: pure programs can't produce or consume side effects, don't have mutable state, and can require complicated or error-prone solutions to some simple problems (e.g. keeping a counter).

Functional programmers long for the virtues of purity without having to give up their impure indulgences.

# why monads?

In functional programming, *pure* languages have several desirable properties:

- program text close to program meaning
- no model of memory or global state required to understand program behavior

This makes programs simpler, clearer and easier to debug and optimize.

But purity comes at a cost: pure programs can't produce or consume side effects, don't have mutable state, and can require complicated or error-prone solutions to some simple problems (e.g. keeping a counter).

Functional programmers long for the virtues of purity without having to give up their impure indulgences.

# why monads?

Monads have been used with great success to add impure features to *pure* languages *conservatively* (i.e. without interfering with the ability to reason about program behavior).

Monads also provide a abstract mechanism to reduce the “plumbing” overhead of writing functional programs by encapsulating it into an abstract type and dealing with it in a uniform way.

# why monads?

Monads have been used with great success to add impure features to *pure* languages *conservatively* (i.e. without interfering with the ability to reason about program behavior).

Monads also provide a abstract mechanism to reduce the "plumbing" overhead of writing functional programs by encapsulating it into an abstract type and dealing with it in a uniform way.

# what monads?

But our topic today is not the *“why?”* of monads, but rather the *“what?”*

Literature on monads in mathematics and functional programming presents them quite differently.

- Different definitions.
- Different laws.

We will see how these concepts actually coincide and express the correspondence directly in a functional language so that we can use whichever perspective best suits our needs.

Along the way we will encounter a third perspective that, like the *Demotic* script of the Rosetta stone, will help us understand the connection between the other two.

# what monads?

But our topic today is not the *"why?"* of monads, but rather the *"what?"*

Literature on monads in mathematics and functional programming presents them quite differently.

- Different definitions.
- Different laws.

We will see how these concepts actually coincide and express the correspondence directly in a functional language so that we can use whichever perspective best suits our needs.

Along the way we will encounter a third perspective that, like the *Demotic* script of the Rosetta stone, will help us understand the connection between the other two.

# what monads?

But our topic today is not the *"why?"* of monads, but rather the *"what?"*

Literature on monads in mathematics and functional programming presents them quite differently.

- Different definitions.
- Different laws.

We will see how these concepts actually coincide and express the correspondence directly in a functional language so that we can use whichever perspective best suits our needs.

Along the way we will encounter a third perspective that, like the *Demotic* script of the Rosetta stone, will help us understand the connection between the other two.

# a simple plan:

1. The Mathematician's Perspective
2. The Functional Programmer's Perspective
3. Equivalence of the Perspectives

# category

A **category** $\mathbb{C}$ is determined by:

- a collection of **objects**:
  $\{C : \mathbb{C}\}$
- a collection of **arrows** between pairs of objects:
  $\forall\, A, B : \mathbb{C} \,.\, \exists\, \mathbb{C}\,(A \to B)$
- an associative **composition** operation on arrows:
  $f : \mathbb{C}\,(A \to B) \wedge g : \mathbb{C}\,(B \to C) \Longrightarrow f \cdot g : \mathbb{C}\,(A \to C)$
- an **identity arrow** for each object, which is a *unit* under composition:
  $\forall\, C : \mathbb{C} \,.\, \exists\, \mathrm{id}_C : \mathbb{C}\,(C \to C) \,.\, (f \cdot \mathrm{id}_C = f) \wedge (\mathrm{id}_C \cdot g = g)$

# some categories

In general, a category is any collection of "*things*" and structure-preserving, composable maps between them. For example:

| category | objects | arrows |
|---|---|---|
| directed graphs | vertices | paths in edges |
| partial orders | elements | $\leq$ |
| $\textsc{Set}$ | sets | functions |
| $\textsc{Grp}$ | groups | group homomorphisms |
| $\textsc{Top}$ | topological spaces | continuous maps |
| $\textsc{Cat}$ | *small* categories | ??? |

# functor

A **functor** $\mathrm{F} : \mathbb{A} \longrightarrow \mathbb{B}$ is an structure-preserving map between categories. That is, it must:

- send objects of $\mathbb{A}$ to objects of $\mathbb{B}$:
  $\mathrm{A} : \mathbb{A} \implies \mathrm{F}(\mathrm{A}) : \mathbb{B}$
- send arrows of $\mathbb{A}$ to arrows of $\mathbb{B}$, preserving their domains and codomains:
  $f : \mathbb{A}\,(\mathrm{A} \to \mathrm{B}) \implies \mathrm{F}(f) : \mathbb{B}\,(\mathrm{F}(\mathrm{A}) \to \mathrm{F}(\mathrm{B}))$
- preserve compositions:
  $\mathrm{F}(f \cdot g) = \mathrm{F}(f) \cdot \mathrm{F}(g)$
- preserve identity arrows:
  $\mathrm{F}(\mathrm{id}_\mathrm{A}) = \mathrm{id}_{\mathrm{F}(\mathrm{A})}$

# natural transformation

A **natural transformation** $\varphi : F \longrightarrow G$ is a structure-preserving map between *parallel* functors.

$$\begin{tikzcd} \mathbb{A} \arrow[r, bend left, "F"] \arrow[r, bend right, "G"'] \arrow[r, phantom, "\Downarrow \varphi"] & \mathbb{B} \end{tikzcd}$$

Where for each $A : \mathbb{A}$, there is a $\varphi(A) : \mathbb{B}\,(F(A) \to G(A))$, called the **component** of $\varphi$ at $A$, such that the diagram in $\mathbb{B}$ **commutes**:

$$\mathbb{A}) \quad \begin{tikzcd} A \arrow[d, "f"'] \\ B \end{tikzcd} \qquad \mathbb{B}) \quad \begin{tikzcd} F(A) \arrow[r, "\varphi(A)"] \arrow[d, "F(f)"'] & G(A) \arrow[d, "G(f)"] \\ F(B) \arrow[r, "\varphi(B)"'] & G(B) \end{tikzcd}$$

# natural transformation

A **natural transformation** $\varphi : F \longrightarrow G$ is a structure-preserving map between *parallel* functors.

$$\begin{tikzcd} \mathbb{A} \arrow[r, bend left, "F"] \arrow[r, bend right, "G"'] \arrow[r, phantom, "\Downarrow \varphi"] & \mathbb{B} \end{tikzcd}$$

Where for each $A : \mathbb{A}$, there is a $\varphi(A) : \mathbb{B}\,(F(A) \to G(A))$, called the **component** of $\varphi$ at $A$, such that the diagram in $\mathbb{B}$ **commutes**:

$$\mathbb{A})\quad \begin{tikzcd} A \arrow[d, "f"'] \\ B \end{tikzcd} \qquad\qquad \mathbb{B})\quad \begin{tikzcd} F(A) \arrow[r, "\varphi(A)"] \arrow[d, "F(f)"'] & G(A) \arrow[d, "G(f)"] \\ F(B) \arrow[r, "\varphi(B)"'] & G(B) \end{tikzcd}$$

## compositions of natural transformations

natural transformations can be composed in one of two ways:

head-to-tail ("vertically"): $\alpha \cdot \beta \quad : \quad F \longrightarrow H$

$$
\begin{tikzcd}
\mathbb{A} \arrow[rr, bend left=50, "F"] \arrow[rr, "G" description] \arrow[rr, bend right=50, "H"'] & \arrow[d, phantom, near start, "\Downarrow\alpha"] \arrow[d, phantom, near end, "\Downarrow\beta"] & \mathbb{B} \\
& &
\end{tikzcd}
$$

side-by-side ("horizontally"): $\alpha \cdot\cdot \gamma \quad : \quad F \cdot I \longrightarrow G \cdot J$

$$
\begin{tikzcd}
\mathbb{A} \arrow[r, bend left=50, "F"] \arrow[r, bend right=50, "G"'] \arrow[r, phantom, "\Downarrow\alpha"] & \mathbb{B} \arrow[r, bend left=50, "I"] \arrow[r, bend right=50, "J"'] \arrow[r, phantom, "\Downarrow\gamma"] & \mathbb{C}
\end{tikzcd}
$$

# monad

Given a category $\mathbb{A}$, a **monad** $\mathcal{M}$ on $\mathbb{A}$ is a triple $(T\,,\eta\,,\mu)$, where,

$$\begin{array}{rcrcll} T & : & \mathbb{A} & \longrightarrow & \mathbb{A} & \text{(base functor)} \\ \eta & : & \mathrm{id}_{\mathbb{A}} & \longrightarrow & T & \text{(unit)} \\ \mu & : & T^2 & \longrightarrow & T & \text{(operator)} \end{array}$$

such that the following relations hold:

$$(T \cdot \cdot \eta) \cdot \mu = \mathrm{id}_T = (\eta \cdot \cdot T) \cdot \mu$$

(unit law)

and

$$(T \cdot \cdot \mu) \cdot \mu = (\mu \cdot \cdot T) \cdot \mu$$

(associative law)

notational shorthand: a functor in the place of a natural transformation indicates the identity natural transformation on that functor.

# monad

Given a category $\mathbb{A}$, a **monad** $\mathcal{M}$ on $\mathbb{A}$ is a triple $(\mathrm{T}, \eta, \mu)$, where,

$$\mathrm{T}^0 \xrightarrow{\eta} \mathrm{T} \xleftarrow{\mu} \mathrm{T}^2$$

such that the following relations hold:

$$\begin{tikzcd} \mathrm{T} \arrow[r, "\mathrm{T}\cdot\cdot\eta"] \arrow[d, "\eta\cdot\cdot\mathrm{T}"'] \arrow[dr, equal] & \mathrm{T}^2 \arrow[d, "\mu"] \\ \mathrm{T}^2 \arrow[r, "\mu"'] & \mathrm{T} \end{tikzcd} \quad \text{and} \quad \begin{tikzcd} \mathrm{T}^3 \arrow[r, "\mathrm{T}\cdot\cdot\mu"] \arrow[d, "\mu\cdot\cdot\mathrm{T}"'] & \mathrm{T}^2 \arrow[d, "\mu"] \\ \mathrm{T}^2 \arrow[r, "\mu"'] & \mathrm{T} \end{tikzcd}$$

# monad (in pictures)

unit law:

$$
\begin{tikzcd}
\mathbb{A} \arrow[r, bend left, "T"] \arrow[r, bend right, "T"'] \arrow[r, phantom, "\Downarrow \mathrm{id}"] \arrow[rr, bend right=70, "T"'] & \mathbb{A} \arrow[r, bend left, "\mathrm{id}"] \arrow[r, bend right, "T"'] \arrow[r, phantom, "\Downarrow \eta"] \arrow[d, phantom, "\Downarrow \mu"] & \mathbb{A} \\
 & \phantom{X} &
\end{tikzcd}
=
\begin{tikzcd}
\mathbb{A} \arrow[r, bend left, "T"] \arrow[r, bend right, "T"'] \arrow[r, phantom, "\Downarrow \mathrm{id}"] & \mathbb{A}
\end{tikzcd}
=
\begin{tikzcd}
\mathbb{A} \arrow[r, bend left, "\mathrm{id}"] \arrow[r, bend right, "T"'] \arrow[r, phantom, "\Downarrow \eta"] \arrow[rr, bend right=70, "T"'] & \mathbb{A} \arrow[r, bend left, "T"] \arrow[r, bend right, "T"'] \arrow[r, phantom, "\Downarrow \mathrm{id}"] \arrow[d, phantom, "\Downarrow \mu"] & \mathbb{A} \\
 & \phantom{X} &
\end{tikzcd}
$$

associative law:

$$
\begin{tikzcd}
 & & \mathbb{A} \arrow[ddr, "T"] & \\
 & \mathbb{A} \arrow[ur, "T"] \arrow[drr, "T"] \arrow[r, phantom, "\Downarrow\mu"] & & \\
\mathbb{A} \arrow[ur, "T"] \arrow[rrr, bend right, "T"'] \arrow[r, phantom, "\Downarrow\mu"] & & & \mathbb{A}
\end{tikzcd}
=
\begin{tikzcd}
 & \mathbb{A} \arrow[dr, "T"] & & \\
 & & \mathbb{A} \arrow[dr, "T"] \arrow[l, phantom, "\Downarrow\mu"] & \\
\mathbb{A} \arrow[uur, "T"] \arrow[urr, "T"] \arrow[rrr, bend right, "T"'] & & & \mathbb{A} \arrow[l, phantom, "\Downarrow\mu"]
\end{tikzcd}
$$

## example: closure operations

Let $\mathbb{P}$ be a partially ordered set *as category* and $\mathrm{T}$ be an endofunctor on $\mathbb{P}$ (i.e. $\mathrm{X} \leq \mathrm{Y} \implies \mathrm{T}(\mathrm{X}) \leq \mathrm{T}(\mathrm{Y})$).

Suppose $\mathcal{M} = (\mathrm{T}, \eta, \mu)$ is a monad on $\mathbb{P}$.
Then the existence of $\eta : \mathrm{id}_{\mathbb{P}} \longrightarrow \mathrm{T}$ implies $\forall\, \mathrm{X} : \mathbb{P} \,.\, \mathrm{X} \leq \mathrm{T}(\mathrm{X})$.
And the existence of $\mu : \mathrm{T}^2 \longrightarrow \mathrm{T}$ implies $\forall\, \mathrm{X} : \mathbb{P} \,.\, \mathrm{T}^2(\mathrm{X}) \leq \mathrm{T}(\mathrm{X})$.

By the *functoriality* of $\mathrm{T}$,

$$\forall\, \mathrm{X} : \mathbb{P} \,.\, \mathrm{X} \leq \mathrm{T}(\mathrm{X}) \qquad \implies \qquad \forall\, \mathrm{X} : \mathbb{P} \,.\, \mathrm{T}(\mathrm{X}) \leq \mathrm{T}^2(\mathrm{X})$$

This implies $\forall\, \mathrm{X} : \mathbb{P} \,.\, \mathrm{T}^2(\mathrm{X}) = \mathrm{T}(\mathrm{X})$.

So $\mathrm{T}$ is *expansive* and *idempotent*, thus a **closure operation** on $\mathbb{P}$.

# example: closure operations

Let $\mathbb{P}$ be a partially ordered set *as category* and $\mathrm{T}$ be an endofunctor on $\mathbb{P}$ (i.e. $X \leq Y \Longrightarrow \mathrm{T}(X) \leq \mathrm{T}(Y)$).

Suppose $\mathcal{M} = (\mathrm{T}, \eta, \mu)$ is a monad on $\mathbb{P}$.
Then the existence of $\eta : \mathrm{id}_{\mathbb{P}} \longrightarrow \mathrm{T}$ implies $\forall\, X : \mathbb{P} \,.\, X \leq \mathrm{T}(X)$.
And the existence of $\mu : \mathrm{T}^2 \longrightarrow \mathrm{T}$ implies $\forall\, X : \mathbb{P} \,.\, \mathrm{T}^2(X) \leq \mathrm{T}(X)$.

By the *functoriality* of $\mathrm{T}$,

$$\forall\, X : \mathbb{P} \,.\, X \leq \mathrm{T}(X) \qquad \Longrightarrow \qquad \forall\, X : \mathbb{P} \,.\, \mathrm{T}(X) \leq \mathrm{T}^2(X)$$

This implies $\forall\, X : \mathbb{P} \,.\, \mathrm{T}^2(X) = \mathrm{T}(X)$.

So $\mathrm{T}$ is *expansive* and *idempotent*, thus a **closure operation** on $\mathbb{P}$.

## example: closure operations

Let $\mathbb{P}$ be a partially ordered set *as category* and $\mathrm{T}$ be an endofunctor on $\mathbb{P}$ (i.e. $\mathrm{X} \leq \mathrm{Y} \Longrightarrow \mathrm{T}(\mathrm{X}) \leq \mathrm{T}(\mathrm{Y})$).

Suppose $\mathcal{M} = (\mathrm{T}\,, \eta\,, \mu)$ is a monad on $\mathbb{P}$.
Then the existence of $\eta : \mathrm{id}_{\mathbb{P}} \longrightarrow \mathrm{T}$ implies $\forall\, \mathrm{X} : \mathbb{P}\,.\, \mathrm{X} \leq \mathrm{T}(\mathrm{X})$.
And the existence of $\mu : \mathrm{T}^2 \longrightarrow \mathrm{T}$ implies $\forall\, \mathrm{X} : \mathbb{P}\,.\, \mathrm{T}^2(\mathrm{X}) \leq \mathrm{T}(\mathrm{X})$.

By the *functoriality* of $\mathrm{T}$,

$$\forall\, \mathrm{X} : \mathbb{P}\,.\, \mathrm{X} \leq \mathrm{T}(\mathrm{X}) \qquad \Longrightarrow \qquad \forall\, \mathrm{X} : \mathbb{P}\,.\, \mathrm{T}(\mathrm{X}) \leq \mathrm{T}^2(\mathrm{X})$$

This implies $\forall\, \mathrm{X} : \mathbb{P}\,.\, \mathrm{T}^2(\mathrm{X}) = \mathrm{T}(\mathrm{X})$.

So $\mathrm{T}$ is *expansive* and *idempotent*, thus a **closure operation** on $\mathbb{P}$.

# example: closure operations

Let $\mathbb{P}$ be a partially ordered set *as category* and $\mathrm{T}$ be an endofunctor on $\mathbb{P}$ (i.e. $\mathrm{X} \leq \mathrm{Y} \Longrightarrow \mathrm{T}(\mathrm{X}) \leq \mathrm{T}(\mathrm{Y})$).

Suppose $\mathcal{M} = (\mathrm{T}\,,\eta\,,\mu)$ is a monad on $\mathbb{P}$.
Then the existence of $\eta : \mathrm{id}_{\mathbb{P}} \longrightarrow \mathrm{T}$ implies $\forall\, \mathrm{X} : \mathbb{P}\,.\, \mathrm{X} \leq \mathrm{T}(\mathrm{X})$.
And the existence of $\mu : \mathrm{T}^2 \longrightarrow \mathrm{T}$ implies $\forall\, \mathrm{X} : \mathbb{P}\,.\, \mathrm{T}^2(\mathrm{X}) \leq \mathrm{T}(\mathrm{X})$.

By the *functoriality* of $\mathrm{T}$,

$$\forall\, \mathrm{X} : \mathbb{P}\,.\, \mathrm{X} \leq \mathrm{T}(\mathrm{X}) \qquad \Longrightarrow \qquad \forall\, \mathrm{X} : \mathbb{P}\,.\, \mathrm{T}(\mathrm{X}) \leq \mathrm{T}^2(\mathrm{X})$$

This implies $\forall\, \mathrm{X} : \mathbb{P}\,.\, \mathrm{T}^2(\mathrm{X}) = \mathrm{T}(\mathrm{X})$.

So $\mathrm{T}$ is *expansive* and *idempotent*, thus a **closure operation** on $\mathbb{P}$.

# example: monoid actions

Let $\mathrm{M}$ be a set and $\mathrm{T} : \textsc{Set} \longrightarrow \textsc{Set}$ be the functor $\mathrm{T}(\mathrm{X}) = \mathrm{M} \times \mathrm{X}$.

Let $\eta(\mathrm{X}) : \mathrm{X} \longrightarrow \mathrm{M} \times \mathrm{X}$ be given by $\eta(\mathrm{X})(x) = (e\,, x)$,
and $\mu(\mathrm{X}) : \mathrm{M} \times (\mathrm{M} \times \mathrm{X}) \longrightarrow \mathrm{M} \times \mathrm{X}$ by
$\mu(\mathrm{X})(m\,, (n\,, x)) = (m * n\,, x)$.

And suppose $\mathcal{M} = (\mathrm{T}\,, \eta\,, \mu)$ is a monad on $\textsc{Set}$.

# example: monoid actions

Let $\mathrm{M}$ be a set and $\mathrm{T} : \textsc{Set} \longrightarrow \textsc{Set}$ be the functor $\mathrm{T}(\mathrm{X}) = \mathrm{M} \times \mathrm{X}$.

Let $\eta(\mathrm{X}) : \mathrm{X} \longrightarrow \mathrm{M} \times \mathrm{X}$ be given by $\eta(\mathrm{X})(x) = (e\,,x)$,
and $\mu(\mathrm{X}) : \mathrm{M} \times (\mathrm{M} \times \mathrm{X}) \longrightarrow \mathrm{M} \times \mathrm{X}$ by
$\mu(\mathrm{X})(m\,,(n\,,x)) = (m * n\,,x)$.

And suppose $\mathcal{M} = (\mathrm{T}\,,\eta\,,\mu)$ is a monad on $\textsc{Set}$.
The *unit law* implies:

$$(m\,,x) \xmapsto{(\mathrm{T}\cdot\cdot\eta)(\mathrm{X})} (e\,,(m\,,x)) \xmapsto{\mu(\mathrm{X})} (e * m\,,x) = (m\,,x)$$

$$(m\,,x) \xmapsto{(\eta\cdot\cdot\mathrm{T})(\mathrm{X})} (m\,,(e\,,x)) \xmapsto{\mu(\mathrm{X})} (m * e\,,x) = (m\,,x)$$

So that $e$ is a two-sided unit for the operation $*$.

# example: monoid actions

Let $\mathrm{M}$ be a set and $\mathrm{T} : \textsc{Set} \longrightarrow \textsc{Set}$ be the functor $\mathrm{T}(\mathrm{X}) = \mathrm{M} \times \mathrm{X}$.

Let $\eta(\mathrm{X}) : \mathrm{X} \longrightarrow \mathrm{M} \times \mathrm{X}$ be given by $\eta(\mathrm{X})(x) = (e\,,x)$,
and $\mu(\mathrm{X}) : \mathrm{M} \times (\mathrm{M} \times \mathrm{X}) \longrightarrow \mathrm{M} \times \mathrm{X}$ by
$\mu(\mathrm{X})(m\,,(n\,,x)) = (m * n\,,x)$.

And suppose $\mathcal{M} = (\mathrm{T}\,,\eta\,,\mu)$ is a monad on $\textsc{Set}$.
The *associative law* implies:

$$\begin{array}{ccccc}
(l\,,(m\,,(n\,,x))) & \xmapsto{(\mathrm{T}\cdot\cdot\mu)(\mathrm{X})} & ((l * m)\,,(n\,,x)) & \xmapsto{\mu(\mathrm{X})} & ((l * m) * n\,,x) \\
 & & & & \| \\
(l\,,(m\,,(n\,,x))) & \xmapsto{(\mu\cdot\cdot\mathrm{T})(\mathrm{X})} & (l\,,(m * n\,,x)) & \xmapsto{\mu(\mathrm{X})} & (l * (m * n)\,,x)
\end{array}$$

So that the operation $*$ is associative.

# example: monoid actions

Let $\mathrm{M}$ be a set and $\mathrm{T} : \textsc{Set} \longrightarrow \textsc{Set}$ be the functor $\mathrm{T}(\mathrm{X}) = \mathrm{M} \times \mathrm{X}$.

Let $\eta(\mathrm{X}) : \mathrm{X} \longrightarrow \mathrm{M} \times \mathrm{X}$ be given by $\eta(\mathrm{X})(x) = (e , x)$,
and $\mu(\mathrm{X}) : \mathrm{M} \times (\mathrm{M} \times \mathrm{X}) \longrightarrow \mathrm{M} \times \mathrm{X}$ by
$\mu(\mathrm{X})(m , (n , x)) = (m * n , x)$.

And suppose $\mathcal{M} = (\mathrm{T} , \eta , \mu)$ is a monad on $\textsc{Set}$.

Then $(\mathrm{M} , * , e)$ is a **monoid** acting on $\mathrm{X}$.

Indeed, this is the motivating example from which the monad laws receive their names.

# a useful idiom

In functional programming, monads provide a principled, mathematical way of adding side effects such as input-output or mutable state to otherwise *pure* (referentially transparent) languages.

Moggi [Mog91] showed that monads could describe, in a uniform way, not only side effects, but a broad range of "*notions of computation*", including partiality, non-determinism, exceptions and continuations.

Shortly after this, Wadler [Wad92] showed how monads could be seen as a generalization of list comprehensions, that could characterize many useful data structures and language features.

# a useful idiom

In functional programming, monads provide a principled, mathematical way of adding side effects such as input-output or mutable state to otherwise *pure* (referentially transparent) languages.

Moggi [Mog91] showed that monads could describe, in a uniform way, not only side effects, but a broad range of "*notions of computation*", including partiality, non-determinism, exceptions and continuations.

Shortly after this, Wadler [Wad92] showed how monads could be seen as a generalization of list comprehensions, that could characterize many useful data structures and language features.

# a useful idiom

In functional programming, monads provide a principled, mathematical way of adding side effects such as input-output or mutable state to otherwise *pure* (referentially transparent) languages.

Moggi [Mog91] showed that monads could describe, in a uniform way, not only side effects, but a broad range of "*notions of computation*", including partiality, non-determinism, exceptions and continuations.

Shortly after this, Wadler [Wad92] showed how monads could be seen as a generalization of list comprehensions, that could characterize many useful data structures and language features.

# an implementation of monads

One of the first programming languages to incorporate monads was the lazy functional language, Haskell. Haskell's ad-hoc polymorphism (*typeclass*) mechanism was used to add monads without changing the core language.

The standard library was updated to encapsulate side-effect producing or consuming operations within monadic types and to make several common types monad instances, facilitating a form of *generic programming*.

A concrete syntax ("*do* notation") for monads allowed for programming in a nearly imperative style.

## an implementation of monads

One of the first programming languages to incorporate monads was the lazy functional language, Haskell. Haskell's ad-hoc polymorphism (*typeclass*) mechanism was used to add monads without changing the core language.

The standard library was updated to encapsulate side-effect producing or consuming operations within monadic types and to make several common types monad instances, facilitating a form of *generic programming*.

A concrete syntax ("*do* notation") for monads allowed for programming in a nearly imperative style.

# an implementation of monads

One of the first programming languages to incorporate monads was the lazy functional language, Haskell. Haskell's ad-hoc polymorphism (*typeclass*) mechanism was used to add monads without changing the core language.

The standard library was updated to encapsulate side-effect producing or consuming operations within monadic types and to make several common types monad instances, facilitating a form of *generic programming*.

A concrete syntax ("*do* notation") for monads allowed for programming in a nearly imperative style.

# monads in Haskell

In Haskell the `Monad` typeclass is defined as:

```
class Monad m  where
return :: ∀ a . a → m a
(>>=) :: ∀ a b . m a → (a → m b) → m b
...
```

(>>= is pronounced "`bind`")

In diagrams:

$$
\begin{tikzcd}
M(A) \\
A \arrow[u, dashed, "return"]
\end{tikzcd}
\qquad
\begin{tikzcd}
M(A) \arrow[r, dashed, "(>>= f)"] & M(B) \\
A \arrow[ur, "f"'] &
\end{tikzcd}
$$

# Haskell monad laws

Although not enforced by the compiler, monads are expected to obey the three **monad laws**:

"left identity": `(return a) >>= g = g a`

"right identity": `ma >>= return = ma`

"associativity": `(ma >>= f) >>= g = ma >>= (λ x → (f x >>= g))`

l-id:

$$\begin{tikzcd} M(A) \arrow[r, "(>>=g)"] & M(B) \\ A \arrow[u, "return"] \arrow[ur, "g"'] & \end{tikzcd}$$

r-id:

$$\begin{tikzcd} M(A) \arrow[r, equal, "(>>=return)"] & M(A) \\ A \arrow[ur, "return"'] & \end{tikzcd}$$

# Haskell monad laws

Although not enforced by the compiler, monads are expected to obey the three **monad laws**:

"left identity": `(return a) >>= g = g a`

"right identity": `ma >>= return = ma`

"associativity": `(ma >>= f) >>= g = ma >>= (λ x → (f x >>= g))`

assoc:

$$\begin{tikzcd} M(A) \arrow[r, "(>>= f)"] & M(B) \arrow[r, "(>>= g)"] & M(C) \\ A \arrow[ur, "f"'] & B \arrow[ur, "g"'] & C \end{tikzcd}$$

Notice `λ x → (f x >>= g)` is the composition `f · (>>= g)`.

So associativity says `(>>= f) · (>>= g) = (>>= (f · (>>= g)))`.

# a Haskell monad: Maybe

In Haskell, the type constructor `Maybe` creates *lifted* types and can be used to represent partial functions and simple exceptions:

```
data Maybe a :: * where
Nothing :: Maybe a
Just :: a → Maybe a
```

It is also an instance of `Monad`:

```
instance  Monad Maybe  where
return  =  Just

Nothing  >>= f  =  Nothing
(Just x) >>= f  =  f x
```

The monad laws are easily verified.

# unravelling Haskell's monad laws

The Haskell monad laws seem a bit strange because the left identity and associative laws don't exactly fit their names. Can we make better sense of this?

# a funny sort of composition

Within the `Monad` typeclass, we find the following curious definition:

```
(>=>) :: Monad m ⇒ (a → m b) → (b → m c) → (a → m c)
f >=> g  =  λ x → f x >>= g
```

whose typing looks like that of composition, except the types are "crooked". We'll call it "`komp`" for now.

But it's just our new friend, `f · (>>= g)`!

Setting `f` to be identity, we can also define >>= in terms of >=>:

```
(>>= g)  =  id >=> g
```

# a funny sort of composition

Within the `Monad` typeclass, we find the following curious definition:

```
(>=>) :: Monad m ⇒ (a → m b) → (b → m c) → (a → m c)
f >=> g  =  λ x → f x >>= g
```

whose typing looks like that of composition, except the types are "crooked". We'll call it "`komp`" for now.

But it's just our new friend, `f · (>>= g)`!

Setting `f` to be identity, we can also define >>= in terms of >=>:

```
(>>= g) =  id >=> g
```

# a funny sort of composition

Within the `Monad` typeclass, we find the following curious definition:

```
(>=>) :: Monad m ⇒ (a → m b) → (b → m c) → (a → m c)
f >=> g  =  λ x → f x >>= g
```

whose typing looks like that of composition, except the types are "crooked". We'll call it "`komp`" for now.

But it's just our new friend, `f · (>>= g)`!

Setting `f` to be identity, we can also define >>= in terms of >=>:

```
(>>= g) =  id >=> g
```

# Haskell's monad laws unravelled

Let's rewrite Haskell's monad laws using >=> rather that >>=:

# Haskell's monad laws unravelled

Let's rewrite Haskell's monad laws using >=> rather that >>=:

left identity:

$$
\begin{array}{ll}
 & \text{return } a >>= g \quad = \quad g(a) \\
\Rightarrow & [\beta\text{-expansion at } a] \\
 & (\lambda x \,.\, \text{return } x >>= g)(a) \quad = \quad g(a) \\
\Rightarrow & [\text{definition } >=>] \\
 & (\text{return } >=> g)(a) \quad = \quad g(a) \\
\Rightarrow & [\eta\text{-reduction}] \\
 & \text{return } >=> g \quad = \quad g
\end{array}
$$

# Haskell's monad laws unravelled

Let's rewrite Haskell's monad laws using >=> rather that >>=:

right identity:

$$
\begin{array}{ll}
& ma \mathbin{>\!\!>\!\!=} \text{return} \quad = \quad ma \\
\Rightarrow & [\text{specializing } ma \text{ to } f(a)] \\
& f(a) \mathbin{>\!\!>\!\!=} \text{return} \quad = \quad f(a) \\
\Rightarrow & [\beta\text{-expansion at } a] \\
& (\lambda x \,.\, f(x) \mathbin{>\!\!>\!\!=} \text{return})(a) \quad = \quad f(a) \\
\Rightarrow & [\text{definition} \mathbin{>\!=\!>}] \\
& (f \mathbin{>\!=\!>} \text{return})(a) \quad = \quad f(a) \\
\Rightarrow & [\eta\text{-reduction}] \\
& f \mathbin{>\!=\!>} \text{return} \quad = \quad f
\end{array}
$$

# Haskell's monad laws unravelled

Let's rewrite Haskell's monad laws using >=> rather that >>=:

associativity:

$$
\begin{array}{ll}
 & (ma >>= f) >>= g \quad = \quad ma >>= (\lambda x \,.\, (f x >>= g)) \\
\Rightarrow & [\text{specializing } ma \text{ to } e(y)] \\
 & (e(y) >>= f) >>= g \quad = \quad e(y) >>= (\lambda x \,.\, (f(x) >>= g)) \\
\Rightarrow & [\text{definition} >=>] \\
 & (e(y) >>= f) >>= g \quad = \quad e(y) >>= (f >=> g) \\
\Rightarrow & [\beta\text{-expansion at } y \text{ and definition} >=>] \\
 & ((e >=> f)(y)) >>= g \quad = \quad (e >=> (f >=> g))(y) \\
\Rightarrow & [\beta\text{-expansion at } y \text{ and definition} >=>] \\
 & ((e >=> f) >=> g)(y) \quad = \quad (e >=> (f >=> g))(y) \\
\Rightarrow & [\eta\text{-reduction}] \\
 & (e >=> f) >=> g \quad = \quad e >=> (f >=> g)
\end{array}
$$

# Haskell's monad laws unravelled

So Haskell's monad laws imply that the funny composition operator, >=>, is associative with two-sided identity `return`, which seems to better fit their names.

Arguing in reverse and setting the specialized function to be the identity, we see that the implication is indeed an equivalence.

# Haskell's monad laws unravelled

So Haskell's monad laws imply that the funny composition operator, >=>, is associative with two-sided identity `return`, which seems to better fit their names.

Arguing in reverse and setting the specialized function to be the identity, we see that the implication is indeed an equivalence.

3 Equivalence of the Perspectives

- The Kleisli Category
- Extending Haskell's Monads
- Equivalence.hs

# the missing link

The equivalence of the two perspectives on monads depends on the idea of a **Kleisli category**.

In category theory a monad can be thought of as a structure derived from a more general construction, known as an *adjunction* (generalized Galois correspondence).

Through a certain canonical adjunction, each monad on a category determines another category, the Kleisli category of the monad.

# the missing link

The equivalence of the two perspectives on monads depends on the idea of a **Kleisli category**.

In category theory a monad can be thought of as a structure derived from a more general construction, known as an *adjunction* (generalized Galois correspondence).

Through a certain canonical adjunction, each monad on a category determines another category, the Kleisli category of the monad.

# the missing link

The equivalence of the two perspectives on monads depends on the idea of a **Kleisli category**.

In category theory a monad can be thought of as a structure derived from a more general construction, known as an *adjunction* (generalized Galois correspondence).

Through a certain canonical adjunction, each monad on a category determines another category, the Kleisli category of the monad.

# Kleisli category

For monad $\mathcal{M} = (T, \eta, \mu)$ on category $\mathbb{A}$, the **Kleisli category** of $\mathcal{M}$, $\mathbb{A}_T$, is the category with:

objects those of $\mathbb{A}$

arrows $\mathbb{A}_T\,(A \to B) = \mathbb{A}\,(A \to T(B))$

composition for $f : \mathbb{A}_T\,(A \to B)$, $g : \mathbb{A}_T\,(B \to C)$

$$(f \cdot g) : \mathbb{A}_T\,(A \to C) = (f \cdot T(g) \cdot \mu(C)) : \mathbb{A}\,(A \to T(C))$$

# Kleisli category

For monad $\mathcal{M} = (\mathrm{T}\,, \eta\,, \mu)$ on category $\mathbb{A}$, the **Kleisli category** of $\mathcal{M}$, $\mathbb{A}_{\mathrm{T}}$, is the category with:

objects those of $\mathbb{A}$

arrows $\mathbb{A}_{\mathrm{T}}\,(\mathrm{A} \to \mathrm{B}) = \mathbb{A}\,(\mathrm{A} \to \mathrm{T}(\mathrm{B}))$

composition for $f : \mathbb{A}_{\mathrm{T}}\,(\mathrm{A} \to \mathrm{B})\,, g : \mathbb{A}_{\mathrm{T}}\,(\mathrm{B} \to \mathrm{C})$

$$
\begin{tikzcd}
\mathbb{A}) & & \mathrm{T}^2(\mathrm{C}) \arrow[d, "\mu(\mathrm{C})"] \\
& \mathrm{T}(\mathrm{B}) \arrow[ur, "\mathrm{T}(g)"] & \mathrm{T}(\mathrm{C}) \\
\mathrm{A} \arrow[ur, "f"] & \mathrm{B} \arrow[ur, "g"] &
\end{tikzcd}
$$

The definition of Kleisli category tells us that the μ of a monad determines Kleisli composition.

But we can also use the definition to recover μ *from* Kleisli composition:

$$\mathbb{A}) \qquad \begin{array}{ccccc} T^2(A) & \xrightarrow{\mathrm{id}} & T^2(A) & \xrightarrow[=\mathrm{id}]{T(\mathrm{id})} & T^2(A) \\ & & & & \big\downarrow{\scriptstyle \mu(A)} \\ & & T(A) & \xrightarrow[\mathrm{id}]{} & T(A) \end{array}$$

$$\Longrightarrow \quad \mu(A) = \mathrm{id}_{T^2(A)} \cdot \mathrm{id}_{T(A)}$$

where the composition is interpreted in the Kleisli category.

(but beware: while the identity arrows from $\mathbb{A}$ shown *are* arrows in the Kleisli category, they are *not* identity arrows there.)

# Kleisli's the key

This *Kleisli composition* looks a lot like the *funny composition*, >=>, we saw earlier.

And indeed, that's what it is.

So the *identity laws* for monads in Haskell say that `return` is a two-sided identity for Kleisli composition. `return` has type `∀ a . monad m ⇒ a → m a`.

This looks suspiciously like the type of $\eta : \mathrm{id}_{\mathbb{A}} \longrightarrow \mathrm{T}$, where $\mathrm{T}$ is the endofuctor for monad $\mathcal{M}$.

Could it be that `return` is $\eta$?

# Kleisli's the key

This *Kleisli composition* looks a lot like the *funny composition*, >=>, we saw earlier.

And indeed, that's what it is.

So the *identity laws* for monads in Haskell say that `return` is a two-sided identity for Kleisli composition. `return` has type
`∀ a . monad m ⇒ a → m a`.

This looks suspiciously like the type of $\eta : \mathrm{id}_{\mathbb{A}} \longrightarrow \mathrm{T}$, where $\mathrm{T}$ is the endofuctor for monad $\mathcal{M}$.

Could it be that `return` is $\eta$?

# Kleisli's the key

This *Kleisli composition* looks a lot like the *funny composition*, >=>, we saw earlier.

And indeed, that's what it is.

So the *identity laws* for monads in Haskell say that `return` is a two-sided identity for Kleisli composition. `return` has type
`∀ a . monad m ⇒ a → m a`.

This looks suspiciously like the type of $\eta : \mathrm{id}_{\mathbb{A}} \longrightarrow \mathrm{T}$, where $\mathrm{T}$ is the endofuctor for monad $\mathcal{M}$.

Could it be that `return` is $\eta$?

# Kleisli's the key

Indeed, the components of $\eta$ are the identity arrows of the Kleisli category:

for $f : \mathbb{A}_T\,(A \to B)$,

# Kleisli's the key

Indeed, the components of $\eta$ are the identity arrows of the Kleisli category:

for $f : \mathbb{A}_T\,(A \to B)$,

$$
\begin{aligned}
& \eta(A) \cdot f \quad : \quad \mathbb{A}_T\,(A \to B) \\
= \quad & \text{[definition of Kleisli composition]} \\
& \eta(A) \cdot T(f) \cdot \mu(B) \quad : \quad \mathbb{A}\,(A \to T(B)) \\
= \quad & \text{[naturality of } \eta\text{]} \\
& f \cdot \eta(T(B)) \cdot \mu(B) \\
= \quad & \text{[monad unit law]} \\
& f \cdot \mathrm{id}_{T(B)} = f
\end{aligned}
$$

# Kleisli's the key

Indeed, the components of η are the identity arrows of the Kleisli category:

for $f : \mathbb{A}_T\,(A \to B)$,

$$
\begin{array}{rl}
& f \cdot \eta(B) \quad : \quad \mathbb{A}_T\,(A \to B) \\
= & \text{[definition of Kleisli composition]} \\
& f \cdot T(\eta(B)) \cdot \mu(B) \quad : \quad \mathbb{A}\,(A \to T(B)) \\
= & \text{[monad unit law]} \\
& f \cdot \mathrm{id}_{T(B)} = f
\end{array}
$$

# extending Haskell's monads

So we see that >>=, Kleisli composition (>=>), and the monad operator ($\mu$) are all interdefinable.

What's nice is that the type system of Haskell is rich enough to allow these definitions to be typed in Haskell itself.

By default, if you tell Haskell what >>= is for your monad, it will automatically compute >=> and $\mu$ (it's called "`join`" in the standard library) for you.

But if you want to give your monad in terms of one of the other two operators, you can do so using a well-typed function:

# extending Haskell's monads

So we see that >>=, Kleisli composition (>=>), and the monad operator ($\mu$) are all interdefinable.

What's nice is that the type system of Haskell is rich enough to allow these definitions to be typed in Haskell itself.

By default, if you tell Haskell what >>= is for your monad, it will automatically compute >=> and $\mu$ (it's called "`join`" in the standard library) for you.

But if you want to give your monad in terms of one of the other two operators, you can do so using a well-typed function:

# extending Haskell's monads

So we see that >>=, Kleisli composition (>=>), and the monad operator ($\mu$) are all interdefinable.

What's nice is that the type system of Haskell is rich enough to allow these definitions to be typed in Haskell itself.

By default, if you tell Haskell what >>= is for your monad, it will automatically compute >=> and $\mu$ (it's called "`join`" in the standard library) for you.

But if you want to give your monad in terms of one of the other two operators, you can do so using a well-typed function:

# the equivalence in Haskell

Recall our definition of >>= in terms of Kleisli composition.

We can write this in Haskell as:

```
komp_to_bind :: ∀ (m :: ∗ → ∗)
. Functor m
⇒ (∀ a b c . (a → m b) → (b → m c) → (a → m c))
→ (∀ a b . (m a → (a → m b) → m b))
komp_to_bind komp m_x f = (id ‘komp‘ f) m_x
```

# the equivalence in Haskell

Combining this with our definition of Kleisli composition in terms μ, we see that for $f : \mathbb{A}\,(A \to T(B))$,

$$\begin{aligned}(>\!>\!= f) &= id_A >\!=\!> f \\ &= id_A \cdot T(f) \cdot \mu(B) \\ &= T(f) \cdot \mu(B)\end{aligned}$$

which we can write in Haskell as:

```
join_to_bind :: ∀ (m :: * → *)
. Functor m
⇒ (∀ t . (m (m t) → m t))
→ (∀ a b . (m a → (a → m b) → m b))
join_to_bind mu m_x f = (fmap f · mu) m_x
```

# your monads, your way

Using relationships between μ, >=> and >>=, you can define a monad using whichever one has the simplest, most intuitive description.

For example, the description of the `List` monad in the Haskell standard library is

```
instance  Monad []  where
return x  =  [x]
m >>= k  =  foldr ((++) . k) [] m
...
```

but

```
(>>=)  =  join_to_bind concat
```

would be simpler.

## your monads, your way

Using relationships between $\mu$, >=> and >>=, you can define a monad using whichever one has the simplest, most intuitive description.

For example, the description of the `List` monad in the Haskell standard library is

```
instance  Monad []  where
return x  =  [x]
m >>= k  =  foldr ((++) . k) [] m
...
```

but

```
(>>=)  =  join_to_bind concat
```

would be simpler.

# summary

So we've seen that:

- the seemingly different concepts of monad from category theory and functional programming actually coincide.
- the link connecting the two concepts is the notion of a Kleisli category
- $\mu$, Kleisli composition and >>= are all interdefinable
- we can easily translate from one perspective to another within the language Haskell to describe monads in whichever way is easiest.

# References


[Web] *All About Monads*. URL: http://www.haskell.org/all_about_monads/.

[BW05] Michael Barr and Charles Wells. *Toposes, Triples and Theories*. 1.1. 12. Reprints in Theory and Applications of Categories, 2005. URL: http://www.tac.mta.ca/tac/reprints/articles/12/tr12abs.html.

[ML98] Saunders Mac Lane. *Categories for the Working Mathematician*. second. Graduate Texts in Mathematics. Springer, 1998.

[Mog91] Eugenio Moggi. "Notions of computation and monads". In: *Information And Computation* 91.1 (1991). URL: http://www.disi.unige.it/person/MoggiE/ftp/ic91.pdf.

[Wad92] Philip Wadler. "Comprehending Monads". In: *Mathematical Structures in Computer Science* 2 (1992).