Introduction to Category Theory and Monads

Overview:
- Basic constructions
- Diagrams
- Functors, Natural Transformations, and Monads
- Modeling state, output, and non-determinism
- Monad transformers

References:
Categories

- Category theory studies “objects” and “morphisms” between them.

- Category theory was originated, in part, as a technique for seeing a collection of similar results as instances of a more general theory of mathematical structures and the relationships between structures.

- Category theory provides a very abstract framework to reason about objects and their relationships. Any immediate access to the internal structure of objects is prevented: all properties of objects must be specified by properties of morphisms (existence of morphisms, their unicity, validity of some equations among them, and so on). This is quite similar to considering objects as “abstract data types”, i.e. data specifications that are independent of any particular implementation.

- Category theory is a tool for studying the semantics of programming languages and in many cases the ‘categorical viewpoint’ matches much better with the basic motivations of computer science than the alternative foundational theories.
A category $C$ comprises:

- A collection of objects,
- A collection of arrows (often called morphisms),
- Two operations $\text{dom}$, $\text{cod}$ assigning to each arrow $f$ two objects respectively called domain (source) and codomain (target) of $f$. We write $f : A \to B$ or $A \longrightarrow B$ to show that $\text{dom } f = A$ and $\text{cod } f = B$. The collection of all arrows with domain $A$ and codomain $B$ is written $C(A, B)$.
- An operation “$\circ$” (composition) assigning to each pair $f, g$ of arrows with $\text{dom } (f) = \text{cod } (g)$ an arrow $f \circ g$ such that $\text{dom } (f \circ g) = \text{dom } (g)$, $\text{cod } (f \circ g) = \text{cod } (f)$ satisfying the following associative law:

  for any arrow $f : A \to B$, $g : B \to C$, and $h : C \to D$ (with $A$, $B$, $C$, and $D$ not necessarily distinct):
  $$h \circ (g \circ f) = (h \circ g) \circ f$$

- An operation $\text{id}$ assigning to each object $b$ a morphism $\text{id}_b : A \to A$ (the identity of $b$) such that $\text{dom } (\text{id}_b) = \text{cod } (\text{id}_b) = b$ satisfying the following identity law:

  for any arrow $f : A \to B$, $\text{id}_b \circ f = f$ and $f \circ \text{id}_b = f$
Category **Set**

- The category **Set** has sets as objects and total functions between sets as arrows. Composition of arrows is the set-theoretic function composition. Identity arrows are identity functions.

- In order to see that **Set** is a category, we restate its definition in the categorical format and check that the laws hold:
  - An object in **Set** is a set,
  - An arrow \( f : A \to B \) in **Set** is a total function from set \( A \) into the set \( B \),
  - For each total function \( f \) with domain \( A \) and codomain \( B \), we have \( \text{dom } f = A \), \( \text{cod } f = B \), and \( f \in \text{Set}(A, B) \),
  - The composition of a total function \( f : A \to B \) with another total function \( g : B \to C \) is the total function from \( A \) to \( C \) mapping each element \( a \in A \) to \( g(f(a)) \in C \). Composition of total functions on set is associative: for any functions \( f : A \to B \), \( g : B \to C \), and \( h : C \to D \), we have \( h \circ (g \circ f) = (h \circ g) \circ f \),
  - For each set \( A \), the identity function \( \text{id}_A \) is a total function with domain and codomain \( A \). For any function \( f : A \to B \), the identity functions on \( A \) and \( B \) satisfy the identity law: \( \text{id}_B \circ f = f \) and \( f \circ \text{id}_A = f \) [Note: for a set \( A \), we have \( \text{id}_A = f \circ f^{-1} \)]
Category **Poset**

- An object in **Poset** is a partially-ordered set \((A, \leq_A)\) and an arrow is an order-preserving total function \(f: (A, \leq_A) \to (B, \leq_B)\), such that if \(a \leq_A a'\) then \(f(a) \leq_B f(a')\).

- Verification of the composition operation:
The composition of two order-preserving total functions \(f: A \to B\) and \(g: B \to C\) is the total function \(g \circ f\) from \(A\) to \(C\). Furthermore, if \(a \leq_A a'\) it holds that \(f\) preserves \(A\)'s ordering, \(f(a) \leq_B f(a')\), and since \(g\) preserves \(B\)'s ordering, we have \(g(f(a)) \leq_C g(f(a'))\). Therefore, \(g \circ f\) is order-preserving. Composition of order-preserving functions is associative because each order-preserving function on partially-ordered sets is just a function on sets and composition of functions on sets is associative.
Category $FPL$

- **Types:**
  - Int
  - Real
  - Bool
  - Unit

- **Built-in functions:**
  - $\text{IsZero} : \text{Int} \to \text{Bool}$
  - $\text{Not} : \text{Bool} \to \text{Bool}$
  - $\text{SuccInt} : \text{Int} \to \text{Int}$
  - $\text{SuccReal} : \text{Int} \to \text{Int}$
  - $\text{ToReal} : \text{Int} \to \text{Real}$

- **Constants:**
  - $\text{Zero} : \text{Int}$
  - $\text{True} : \text{Bool}$
  - $\text{False} : \text{Bool}$
  - $\text{Unit} : \text{Unit}$

The corresponding category $FPL$ is built by:

- Taking Int, Real, Bool, and Unit to be objects,
- Taking $\text{IsZero}$, $\text{Not}$, $\text{SuccInt}$, $\text{SuccReal}$, and $\text{ToReal}$ to be arrows,
- Taking the constants Zero, True, False, and Unit to be arrows from the Unit object to Int, Real, Bool, and Unit, respectively, which map the single element of Unit to the appropriate elements of these types,
- Adding arrows for the identity functions at each type,
- For every composable pair of arrows, adding an arrow for the function formed by composing them,
- Equating certain arrows, such as $\text{False} = \text{Not} \circ \text{True}$ and $\text{IsZero} \circ \text{Zero} = \text{True}$, that represent the same functions according to the semantics of the language.
Leaving out the identities and the composites, the category FPL looks like this:
An important tool in the practice of category theory is the use of **diagrams**, a graphical style for representing equations.

**Definition:** A diagram in a category $\mathcal{C}$ is a collection of vertices and directed edges, consistently labeled with objects and arrows of $\mathcal{C}$, where “consistently” means that if an edge in the diagram is labeled with an arrow $f$ and $f$ has domain $A$ and codomain $B$, then the endpoints of this edge must be labeled with $A$ and $B$.

**Definition:** A diagram in a category $\mathcal{C}$ is said to commute if, for every pair of vertices $X$ and $Y$, all paths in the diagram from $X$ to $Y$ are equal, in the sense that each path in the diagram determines an arrow and these arrows are equal in $\mathcal{C}$.
Saying that “the diagram \( \text{commutes} \) is exactly the same as saying \( f \circ g' = g \circ f' \).
When a specific property is stated in terms of a commuting diagram, then proofs involving that property can be given “visually”.

Example:

**Proposition:** If both inner squares of the following diagram commute, then so does the outer rectangle.

\[
\begin{array}{ccc}
A & \xrightarrow{f} & B & \xrightarrow{f'} & C \\
\downarrow{a} & & \downarrow{b} & & \downarrow{c} \\
A' & \xrightarrow{g} & B' & \xrightarrow{g'} & C'
\end{array}
\]

**Proof:**

\[
(g' \circ g) \circ a = g' \circ (g \circ a) \quad \text{(associativity)}
\]

\[
= g' \circ (b \circ f) \quad \text{(commutativity of first square)}
\]

\[
= (g' \circ b) \circ f \quad \text{(associativity)}
\]

\[
= (c \circ f') \circ f \quad \text{(commutativity of second square)}
\]

\[
= c \circ (f' \circ f) \quad \text{(associativity)}
\]

(End of Proof)
If a programming language is described as a category, we can use commutative diagrams to assert the validity of program transformations in which the order of operations is permuted.

Example *FPL*:
Dual Category

Definition:

The dual category $C^{\text{op}}$ of a category $C$ has the same objects and the same morphisms of $C$, $\text{id}^{\text{op}}_b = \text{id}_b$, for all $f$, $f^{\text{op}} \text{ dom } (f) = \text{ cod } (f^{\text{op}})$, $\text{ cod } (f) = \text{ dom } (f^{\text{op}})$, and $f^{\text{op}} \circ g^{\text{op}} = g \circ f$.

- Duality is a very powerful technique of category theory. If $P$ is a generic proposition expressed in the language of category theory, the dual of $P$ ($P^{\text{op}}$) is the statement obtained by replacing the word “dom” by “cod”, “cod” by “dom”, and “$g \circ f$” by “$f^{\text{op}} \circ g^{\text{op}}$”.

- If $P$ is true in a category $C$, then $P^{\text{op}}$ is true in $C^{\text{op}}$; if $P$ is true in every category, then also $P^{\text{op}}$ is, since every category is the dual of its dual.

- Duality may be applied to diagrams as well; given a diagram in a category $C$, the dual diagram in $C^{\text{op}}$ is obtained by simply inverting the arrows. A dual diagram commutes, however, if and only if the original one does.
Com S 541

Category $\mathbf{PX}$ and $\mathbf{PX}^{\text{op}}$

- $X = \{1, 2, 3\}$
- Arrows in $\mathbf{PX}$ are inclusions, while those in $\mathbf{PX}^{\text{op}}$ are the reverse of inclusion

The categories are shown without composites and identities. Furthermore, the categories are isomorphic, i.e. they differ only in the names of the objects and arrows.

Com S 541
Operator Representation in Categories

- Categories provide the basis for explaining a variety of familiar operators in a general way.

- Examples:
  - The product $A \times B$ of sets $A$ and $B$ is the set of ordered pairs $(a, b)$ where $a \in A$ and $b \in B$. In particular, there are projections $\text{fst}: A \times B \to A$ and $\text{snd}: A \times B \to B$, which map a pair $(a, b)$ to the first and second component $a$ and $b$, respectively. Moreover, a pair is uniquely determined by its projections in the sense that, for any $x \in A \times B$, we have $x = (\text{fst}(x), \text{snd}(x))$.

  - The idea of a product of structure is even more general. The product of posets $(A, \leq_A)$ and $(B, \leq_B)$ is the set product $A \times B$ of the underlying sets $A$ and $B$ with an ordering $\leq_{A \times B}$ given by $(a, b) \leq_{A \times B} (a', b')$ iff $a \leq_A a'$ and $b \leq_B b'$. The projections are defined as they are for sets.

- We can use categories as a language for expressing the essential common structures of operators. However, in a category-theoretic definition, everything must be defined in terms of the morphisms (arrows) of the category.
Definition:
A product of two objects $A$ and $B$ is an object $A \times B$, together with two projection arrows $\pi_1 : A \times B \rightarrow A$ and $\pi_2 : A \times B \rightarrow B$, such that for any object $C$ and pair of arrows $f : C \rightarrow A$ and $g : C \rightarrow B$ there is exactly one mediating arrow $\langle f, g \rangle : C \rightarrow A \times B$ making the diagram commute, i.e. $\pi_1 \circ \langle f, g \rangle = f$ and $\pi_2 \circ \langle f, g \rangle = g$.

(Dashed arrows are used to represent arrows that are asserted to exist when the rest of the diagram is filled in appropriately.)
Each pair of objects in the category $PX$ has a product. Consider subsets {1,2} and {1,2,3} of X. Then their product is

The product of two objects $A$ and $B$ in $PX$ is their intersections – the greatest lower bound of the two objects (infimum).

- The product in $Set$ is $A \times B$.
- Note: Not all categories have a product.
Each pair of objects in the category $\mathbf{PX}^{\text{op}}$ has a product. Consider subsets \{1,2\} and \{1,2,3\} of $X$. Then their product is

$$\langle f, g \rangle$$

The product of two objects $A$ and $B$ in $\mathbf{PX}^{\text{op}}$ is their union – the least upper bound of the two objects (supremum).
The dual notion of product, coproduct, corresponds to the set-theoretic disjoint union:

**Definition:**

A **coproduct** of two objects $A$ and $B$ is an object $A + B$, together with two injection arrows $\iota_1 : A \rightarrow A + B$ and $\iota_2 : B \rightarrow A + B$, such that for any object $C$ and pair of arrows $f : A \rightarrow C$ and $g : B \rightarrow C$ there is exactly one arrow $[fg] : A + B \rightarrow C$ making the following diagram commute:
Each pair of objects in the category \( \textbf{PX} \) has a coproduct (sum). Consider subsets \{1,2\} and \{1,2,3\} of \( X \). Then their coproduct is

\[
\begin{align*}
\{1,2\} & \xrightarrow{l_1} \{1,2\} \cup \{2,3\} & \xleftarrow{l_2} \{2,3\} \\
\downarrow f & & \downarrow g \\
\{1,2,3\} & &
\end{align*}
\]

The coproduct of two objects \( A \) and \( B \) in \( \textbf{PX} \) is their union – the least upper bound of the two objects (supremum).

The coproduct in \( \textbf{Set} \) is the disjoint union of \( A \) and \( B \).
Each pair of objects in the category $\mathcal{P}X^{\text{op}}$ has a coproduct (sum). Consider subsets $\{1, 2\}$ and $\{1, 2, 3\}$ of $X$. Then their coproduct is

$$\{1, 2\} \xrightarrow{l_1} \{1, 2\} \cap \{1, 2, 3\} \xrightarrow{l_2} \{1, 2, 3\}$$

$$f \quad [fg] \quad g$$

$$\{1, 2\} \quad \{1, 2\} \quad \{1, 2, 3\}$$

The coproduct of two objects $A$ and $B$ in $\mathcal{P}X^{\text{op}}$ is their intersections – the greatest lower bound of the two objects (infimum).
Initial and Terminal Objects

An object $0$ in category $\mathbf{C}$ is called initial object, if for every object $A$ in $\mathbf{C}$, there is exactly one arrow $0 \to A$.

- $\emptyset$ is the initial object in $P \times$.
- $\{1,2,3\}$ is the initial object in $P \times^{\text{op}}$.

An object $1$ in category $\mathbf{C}$ is called terminal object, if for every object $A$ in $\mathbf{C}$, there is exactly one arrow $A \to 1$.

- $\emptyset$ is the terminal object in $P \times^{\text{op}}$.
- $\{1,2,3\}$ is the terminal object in $P \times$. 
Category $P\{1\}$

- $P\{1\}$:
  - $\emptyset$ is the initial object.
  - $\{1\}$ is the terminal object.
  - $P\{1\}$ has a product.
  - $P\{1\}$ has a coproduct.
In some categories, the collection of arrows from an object $A$ to an object $B$ can be reflected as itself an object $B^A$ of the category. For example, in $\textbf{Set}$ the functions from a set $A$ to a set $B$ is itself a set $B^A = \{ f | f : A \rightarrow B \}$.

The categorical characterization of the product is identified by finding the basic operations for products (pairing and projections) and their equational properties.

What are the analogous basic operations for functions?

One of these is application. For example, given a function $f : A \rightarrow B$ and an element $a \in A$, there is a binary operation between $f$ and $a$ which has as its value the result in $B$ of applying $a$ to $f$ (compare with domain $\textbf{Store}$).

The other is currying.
Definition:

Let $\mathbf{C}$ be a category with product $\times$. An exponent of two objects $A$ and $B$ is an object $B^A$ and an arrow $\text{apply}$, such that for any object $C$ and arrow $f: C \times A \to B$, there is a unique arrow $\text{curry}(f): C \to B^A$ making the diagram commute, i.e. unique $\text{curry}(f)$ such that

$$\text{apply} \circ (\text{curry}(f) \times \text{id}_A) = f$$
Functors

**Definition:**

Let $C$ and $D$ be categories. A functor $F : C \to D$ is a map taking each $C$-object $A$ to a $D$-object $F(A)$ and each $C$-arrow $f : A \to B$ to a $D$-arrow $F(f) : F(A) \to F(B)$, such that for all $C$-objects $A$ and composable $C$-arrows $f$ and $g$:

1. $F(id_A) = id_{F(A)}$
2. $F(g \circ f) = F(g) \circ F(f)$

![Diagram](image-url)
class Functor f where
    fmap :: (a -> b) -> f a -> f b

instance Functor Tree where
    fmap f (Leaf x)      = Leaf (f x)
    fmap f (Branch l r) = Branch (fmap f l) (fmap f r)
Natural Transformation

Definition:

Let $C$ and $D$ be categories and $F$ and $G$ be functors from $C$ to $D$. A **natural transformation** $\tau$ from $F$ to $G$, written $\tau : F \rightarrow G$, is a function that assigns to every $C$-object $A$ to a $D$-arrow $\tau_A : F(A) \rightarrow G(A)$ such that for any $C$-arrow $f : A \rightarrow B$ the following commutes in $D$.

\[
\begin{array}{ccc}
F(A) & \xrightarrow{\tau_A} & G(A) \\
\downarrow F(f) & & \downarrow G(f) \\
F(B) & \xrightarrow{\tau_B} & G(B)
\end{array}
\]
Sliding Between Categories
Natural Transformation reverse

reverse :: [a] -> [a]
reverse [] = []
reverse (x:xs) = (reverse xs) : x

? reverse [1,2,3,4,5]
[5,4,3,2,1]
A common pictorial representation of a natural transformation is a diagram of the form:

\[ \begin{array}{ccc}
C & \xrightarrow{\tau} & D \\
\downarrow{G} & & \downarrow{\tau} \\
F & & G
\end{array} \]

which can be read as the two functors \( F, G : C \rightarrow D \), with the natural transformation \( \tau \) that provides a “translation” between the two.
Natural transformations can be composed either “vertically” or “horizontally”. Given two natural transformations $\tau$ and $\sigma$, the diagram below shows a vertical composition:
Functor Category

For every object $A$ in the category $A$, the functors $F, G, H : A \to B$, and the natural transformations $\tau$ and $\sigma$ the following commuting diagram in the category $B$ can be drawn:

![Commuting Diagram]

Category $A$

The commuting diagram on the right can be used as the basis for a new category $F$ that has functors as objects and natural transformations as its morphisms – *the functor category.*
The horizontal composition of two natural transformations $\tau$ and $\sigma$ can be expressed by the following diagram:

$$
\begin{array}{c}
\text{A} \\
\downarrow G \\
\text{B} \\
\downarrow \tau \\
\text{C} \\
\end{array}
\quad
\begin{array}{c}
\text{F} \\
\downarrow \sigma \\
\text{H} \\
\downarrow K \\
\text{G} \\
\end{array}
$$
From Category $\mathcal{A}$ to Category $\mathcal{C}$

For an object $A$ in the category $\mathcal{A}$, the functors $F, G : \mathcal{A} \to \mathcal{B}$, functors $H, K : \mathcal{B} \to \mathcal{C}$, and the natural transformations $\tau$ and $\sigma$ the following commuting diagram in the category $\mathcal{C}$ can be drawn:
The underlying idea of monads in computer science is the distinction between simple data-valued functions and functions that perform computations:

- A data-valued function is one in which the returned value is solely determined by the values of its arguments (no side-effects).

- A function that performs a \textit{computation} can encompass ideas such as state or non-determinism. Moreover, as a consequence of its application, a function that performs a computation produce implicitly more results than the explicitly returned value.
Monads in Category Theory

Definition:
A monad over a category $\mathcal{C}$ is a triple $(T, \eta, \mu)$, where $T$ is the endofunctor $T : \mathcal{C} \to \mathcal{C}$ (a function with a mapping to and from the same category), and $\eta$ and $\mu$ are two natural transformations.

The natural transformations of the monad are defined as $\eta : id_{\mathcal{C}} \to T$ and $\mu : T^2 \to T$ ($T^2 = TT = T \circ T$, the composition of functor $T$)
The endofunctor $T$ is a mapping between all objects of the category $\mathcal{C} - \text{Obj}(\mathcal{C})$, which can be viewed as the set of all values of type $\tau$, to a corresponding set of objects $T(\text{Obj}(\mathcal{C}))$, which are interpreted as the set of computations of type $\tau$.

The natural transformation $\eta$ can be thought of as an operator that includes values into computations.

The natural transformation $\mu$ “flattens” a computation of computations into a single computation.
In order to call the triple formed by $T$, $\eta$, and $\mu$ a monad, the associative law of a monad, and the left and right identity laws must hold:

\[
\begin{align*}
\mu_T(A) & \quad T\mu_A \quad T^2(A) \\
\mu_A & \quad \mu_A \quad T^2(A) \\
\mu_A & \quad T(A) \quad T(A)
\end{align*}
\]

\[
\begin{align*}
\mu \circ (\mu \circ T) &= \mu \circ (\mu \circ A) \\
\mu \circ (T \circ \eta) &= \text{id}_A \equiv \mu \circ (\eta \circ T)
\end{align*}
\]
Com S 541

State Transformer

Computations with side effects may be described by the functor

\[ T(A) = (A \times S)^S \]

where \( (A \times S)^S = \{ f \mid f : S \to (A \times S) \} \), an exponential object that represents all morphisms from \( S \), a set of states, to a pair \( (A \times S) \), the product of an object \( A \) and a (updated) set of states.

A computation takes a state and returns a value together with the modified state. Then a monad is obtained by setting, for each type (or object) \( A \)

\[ \eta_A(a) = \lambda s. (a, s) \quad \text{and} \quad \mu_A(f) = \lambda s. \text{apply}(fs) \]

For every \( f \in (A \times S)^S \), \( \mu_A(f) \) is the computation that, given a state \( s \), first computes the pair (computation,state) given by \( (f \times s') = fs \); then evaluates \( f \) applied to \( s' \) (\( \text{apply} \)) and returns a new pair (value,state). \( \eta_A(a) \) maps \( a \) into a computation.
The Kleisli Category

In the application of monads the idea is to go from objects, or types, to the image by a functor of an object. More precisely, programs take values in $A$ to computations in $T(B)$.

Definition:

Given a monad $(T, \eta, \mu)$ over category $C$, the Kleisli category $C_T$ is the category whose objects are those of $C$; the set $C_T(A, B)$ of morphisms from $A$ to $B$ in $C_T$ is $C(A, T(B))$; the identity $C_T(A, A)$ is $\eta: A \to T(A)$. Moreover, the composition of $f \in C_T(A, B)$ and $g \in C_T(B, C)$ in $C_T$ is

$$g \circ f = \mu_C \circ Tg \circ f : A \to \{ -T(B) \to T^2(C) - \} \to T(C)$$

The expression $\mu_C \circ Tg \circ f$ is interpreted as applying $f$ to some value $a$ to produce some computation $fa$; this computation is evaluated to produce some value $b$, and $g$ is applied to $b$ to produce a computation as a result.
The Kleisli Triple

Definition:
The Kleisli triple \((T_{\text{obj}}, \eta, \_\star)\) can be constructed from a monad \((T, \eta, \mu)\), where \(T_{\text{obj}}\) is the restriction of the endofunctor \(T\) to object and and for any function \(f: A \rightarrow T(A)\) we have \(f^\star : T(A) \rightarrow T(B) \equiv \mu_B \circ T(f)\).

- Kleisli triples can be thought of as a different syntactic presentation of a monad, as there is a one-to-one correspondence between Kleisli triple and a monad.
  - Saunders Mac Lane, *Categories for working mathematician*, Springer-Verlag, 1971, page 143
Philip Wadler adapted the ideas of using monads to structure the semantics of computations into a tool for structuring functional programs.

Given a monad \((\mathbb{T}, \eta, \mu)\), the functor \(\mathbb{T}\) and the two natural transformations are modeled in a functional programming language by a type constructor, a function \(\text{map}\) parameterized on the type constructor, and two polymorphic functions.

Example – State Monad:

\[
\begin{align*}
\text{type } S \ a & = \text{State} \to (a, \text{State}) & \text{- functor } \mathbb{T} \text{ to map objects} \\
\text{mapS} : (a \to b) \to (S \ a \to S \ b) & \text{- functor } \mathbb{T} \text{ to map morphisms} \\
\text{unitS} : a \to S \ a & \text{- } \eta \\
\text{joinS} : S (S \ a) \to S \ a & \text{- } \mu
\end{align*}
\]
Monad Laws

\[
\begin{align*}
\text{map } \text{id} & \quad = \quad \text{id} \\
\text{map } (f \circ g) & \quad = \quad \text{map } f \circ \text{map } g \\
\text{map } f \circ \text{unit} & \quad = \quad \text{unit} \circ f \\
\text{map } f \circ \text{join} & \quad = \quad \text{join} \circ \text{map } (\text{map } f) \\
\text{join} \circ \text{unit} & \quad = \quad \text{id} \\
\text{join} \circ \text{map } \text{unit} & \quad = \quad \text{id} \\
\text{join} \circ \text{map } \text{join} & \quad = \quad \text{join} \circ \text{join}
\end{align*}
\]
Monad Laws Illustrated – $T(A) = [A]$ 

**map id = id:**
- $(\text{map id}) [3]$
- $\text{id} [3]$

**map $(f \circ g) = \text{map } f \circ \text{map } g$:**
- $(\text{map } ((\lambda a \to a + 1) \circ \text{ord}) ['A']) = [66]$
- $(\text{map } ((\lambda a \to a + 1) \circ \text{map ord}) ['A']) = (\text{map } ((\lambda a \to a + 1)) [65]) = [66]$

**map $f \circ \text{unit} = \text{unit} \circ f$:**
- $(\text{map } \text{chr } \circ \text{unit}) 65 = (\text{map } \text{chr}) [65]) = ['A']$
- $(\text{unit } \circ \text{chr}) 65 = \text{unit} 'A' = ['A']$

**join \circ \text{unit} = \text{id}:**
- $(\text{join } \circ \text{unit}) [3] = \text{join } [[3]] = [3]$
- $\text{id} [3]$

**join \circ \text{map unit} = \text{id}:**
- $(\text{join } \circ \text{map unit}) [3] = \text{join } [[3]] = [3]$
- $\text{id} [3]$

**join \circ \text{map join} = \text{join } \circ \text{join}:**
- $(\text{join } \circ \text{map join}) [[[3]]] = \text{join } [[3]] = [3]$
- $(\text{join } \circ \text{join}) [[[3]]] = \text{join } [[3]] = [3]$
In more recent developments, the current use of monads bears a closer resemblance to Kleisli triples.

The triple \((T, \text{unit}, \text{bind})\) forms a monad when the following laws hold:

\[
\begin{align*}
    m \text{ `bind` unit} &= m \\
    (\text{unit a} \text{ `bind` f}) &= f a \\
    (m \text{ `bind` f} \text{ `bind` g}) &= m \text{ `bind` (f `bind` g)}
\end{align*}
\]

The elements of the triple have to be read as \(T\) is a type constructor that maps a value into a computation, \(\text{unit}\) is the inclusion function (i.e. \(\eta\)), and \(\text{bind}\) is a composition operator, which, applied to some \(f\) and \(g\), denotes the same expression as \(\mu_c \circ Tg \circ f\) in the Kleisli category.
class Monad m where

    return :: a -> m a -- unit
    (>>=):: m a -> (a -> m b) -> m b -- bind
    (>>) :: m a -> m b -> m b -- bind (ignore left)
    fail :: String -> m a

-- Minimal complete definition: (>>=), return

    p >> q = p >>= \_ -> q
    fail s = error s
data StateMonad a = SM (State -> (a,State))

instance Monad StateMonad where
    -- define state propagation
    SM m1 >>= fm2 = SM \s0 -> let (r,s1) = m1 s0
                             SM m2 = fm2 r
                             in m2 s1
    return a = SM \s0 -> (a,s0)

    -- extract state from the monad
    readSM :: StateMonad State
    readSM = SM \s -> (s,s)

    -- update the state of the monad
    updateSM :: (State -> State) -> StateMonad ()
    updateSM f = SM \s -> (((), f s))

    -- run a computation in the SM monad
    runSM :: State -> StateMonad a -> (a,State)
    runSM s0 (SM c) = c s0

Com S 541
Monads Enhance Modularity

- Pure functional languages offer the power of lazy evaluation and the simplicity of equational reasoning.

- Impure functional languages offer features like state, exception handling, or continuations.

- Pure languages ease the change of programs, since operations do not have side-effects. But a small change may require a program in a pure language to be extensively restructured, when the use of an impure feature may obtain the same effect by merely altering a few lines of code.

- Monads enhance modularity. In order to change a program one has to redefine the monad and make a few local changes.
Abstract Syntax:

Statement ::= Name ‘=’ Term ‘;’ Statement
   | Term ‘;’ Statement

Term ::= Name
   | Number
   | Term ‘+’ Term
   | ‘function’ Name ‘.’ Term
   | Term Term
   | ‘(’ Term ‘)’
A Monadic Interpreter – Data Structures

```haskell
data Term = Variable Name
            | Constant Int
            | Add Term Term
            | Function Name Term
            | Application Term Term
            | Braced Term

data Value = Wrong
            | Num Int
            | Fun (Value -> M Value)

type Environment = [(Name, Value)]
```
The Identity Monad

data M a = I a  

instance Monad M where

  I v >>= fm = fm v  

  return v = I v  

showM :: Show a => M a -> String
showM (I v) = show v
Auxiliary Definitions

instance Show Value where
    show Wrong = "<wrong>"
    show (Num i) = show i
    show (Fun f) = "<function>"

lookupName :: Name -> Environment -> M Value
lookupName _ [] = return Wrong
lookupName n ((nx,vx):xs)
    | n == nx = return vx
    | otherwise = lookupName n xs
The Term Interpreter

interpreter :: Term -> Environment -> M Value
interpreter (Variable n) e = lookupName n e
interpreter (Constant i) e = return (Num i)
interpreter (Add t1 t2) e =
    interpreter t1 e >>= (\v1 -> interpreter t2 e >>= (\v2 -> addition v1 v2))
    where addition (Num i1) (Num i2) = return (Num (i1+i2))
         addition _ _ = return Wrong
interpreter (Function n t) e = return (Fun (\a -> interpreter t ((n,a):e) ))
interpreter (Application t1 t2) e =
    interpreter t1 e >>= (\f -> interpreter t2 e >>= (\a -> apply f a))
    where apply (Fun f) a = f a
         apply _ _ = return Wrong
interpreter (Braced t) e = interpreter t e

? showM (interpreter (Add (Variable "n") (Constant 3)) [("n",(Num 4))] )
"7"
The Interpreter Loop

\[
\text{data Statements} = \text{S Statement Statements} | \text{Nil} \\
\text{data Statement} = \text{Assign Name Term} | \text{Eval Term}
\]

\[
\text{interpLoop :: Statements} \to \text{Environment} \to \text{M Value} \to \text{M Value} \\
\text{interpLoop Nil e v} = v \\
\text{interpLoop (S s ss) e v} = \text{let} (v',e') = \text{evalStatement s e v in interpLoop ss e' v'}
\]

\[
\text{evalStatement :: Statement} \to \text{Environment} \to \text{M Value} \to \text{(M Value, Environment)} \\
\text{evalStatement (Assign n t) e v} = \text{let} I v = \text{interpreter t e in} (I v,(n,v):e) \\
\text{evalStatement (Eval t) e v} = (\text{interpreter t e}, e)
\]

\[
\text{val} = (\text{S (Assign "d" (Function "f" (Add (Variable "f") (Constant 3))))}) \\
(S (\text{Eval (Application (Variable "d") (Constant 4))}) \text{ Nil}))
\]

\[
? \text{showM (interpLoop val [] (return Wrong))} \\
"7"
\]
To add error messages to the interpreter, we can define the following monad:

```haskell
data M a = Success a | Error String

instance Monad M where

  Success v >>= fm = fm v
  Error s >>= fm = Error s

  return v = Success v

returnError :: String -> M a
returnError s = Error s

showM :: Show a => M a -> String
showM (Success v) = "Success: " ++ show v
showM (Error s) = "Error: " ++ s
```
Towards the Error Monad

- Necessary chances:

  lookupName n [] = returnError ("undefined name: " ++ n)

  addition a b = returnError ("should be numbers: " ++ show a ++ ", " ++ show b)

  apply f _ = returnError ("should be a function: " ++ show f)

  evalStatement (Assign n t) e v = evalTerm (interpreter t e) n e
    where evalTerm (Success v) n e = (Success v, (n,v):e)
          evalTerm (Error s) _ e   = (Error s, e)
type State = Int

data M a = SM (State -> (a,State))

instance Monad M where

    SM sm >>= fm = SM (\s0 -> let (v1,s1) = sm s0
                      SM sm2 = fm v1
                      in sm2 s1 )

    return v = SM (\s0 -> (v,s0))

tickS :: M ()
tickS = SM (\s -> ((),s+1))

showM :: Show a => M a -> String
showM (SM m) = let (v, s) = m 0
               in "Value: " ++ show v ++ "; " ++
                 "Count: " ++ show s
Towards Application Count

To add a function application count, we have to change 2 additional lines in the original interpreter code:

```
addition (Num i1) (Num i2) = tickS >>= return (Num (i1+i2))
apply (Fun f) a = tickS >>= f a
val = (S (Assign "d" (Function "f" (Add (Variable "f")(Constant 3))))
      (S (Eval (Application (Variable "d") (Constant 4)) Nil))
? showM (interpLoop val [] (return Wrong))
“Value: 7; Count 1”
```
The DO Syntax

- The do syntax provides a simple shorthand for chains of monadic operations:

  \[
  \text{do } e_1; e_2 = e_1 >> e_2 \\
  \text{do } p \leftarrow e_1; e_2 = e_1 >>= \lambda p \rightarrow e_2
  \]

- The second form of do is refutable, pattern matching can fail. In this case the fail operation is called (complete translation):

  \[
  \text{do } \{ p \leftarrow e; \text{stm ts} \} = \text{let } \text{ok } p = \text{do } \{ \text{stm ts} \} \\
  \quad \text{ok } \_ = \text{fail "..."} \\
  \quad \text{in } e >>= \text{ok}
  \]

- The new application count definitions:

  \[
  \text{addition } (\text{Num } i_1) (\text{Num } i_2) = \text{do } \text{tickS}; \text{return } (\text{Num } (i_1+i_2)) \\
  \text{apply } (\text{Fun } f) a = \text{do } \text{tickS}; f \ a
  \]