Maybe, it's a Monad!

Paul Zhu

School of Software,
Tsinghua University

April 20, 2019

Tunight Tonight

1. Monads in Action
   - Motivation
   - Monad is an “Interface”
   - Monads for Probabilities
2. Monads, Categorically
   - Preliminary
   - Road to Monads
3. Beyond Monads: PL Today

Contents

Contents

Queries in Scala

Suppose we have a book database, represented as a list of books:

```
case class Book(title: String, authors: List[String])
val books: List[Book] = /* data */
```

To find all the books which have the word “Monad” in the title:

```
for (b <- books if b.title indexOf "Monad" >= 0) yield b.title
```

To find the names of all authors who have written at least two books present in the database:

```
{ for { b1 <- books
        b2 <- books
        if b1.title < b2.title
        a1 <- b1.authors
        a2 <- b2.authors
        if a1 == a2 } yield a1 }.distinct
```

For-Expressions are Just Functions

The Scala compiler expresses for-expressions in terms of `map`, `flatMap` and a *lazy* variant of `filter`, namely `withFilter`. Translation scheme:

- A for-expression with only one *generator*

  ```
  for (x <- e1) yield e2
  ```

  is translated to

  ```
  e1.map(x => e2)
  ```

- A for-expression with a filter `f`

  ```
  for (x <- e1 if f; rest) yield e2
  ```

  is translated to

  ```
  for (x <- e1.withFilter(x => f); rest) yield e2
  ```

  where `rest` denotes a (possibly empty) sequence of remaining generators and filters.

For-Expressions are Just Functions (Cont.)

- A for-expression with multiple generators

  ```
  for (x <- e1; y <- e2; rest) yield e3
  ```

  is translated to

  ```
  e1.flatMap(x => for (y <- e2; rest) yield e3)
  ```

Example: the following for-expression

```
for { i <- 1 until n
      j <- 1 until i
      if isPrime(i + j) } yield (i, j)
```

is translated into

```
(1 until n).flatMap { i =>
    (1 until i).withFilter { j => isPrime(i+j) }
               .map { j => (i, j) }
}
```

Flatmap, Map and then Flatten

```
> def neighbors(x: Int) = List(x - 1, x, x + 1)
> val xs = List(1, 2, 3)
> xs.flatMap(neighbors)
List(0, 1, 2, 1, 2, 3, 2, 3, 4)
> xs.map(neighbors).flatten
List(0, 1, 2, 1, 2, 3, 2, 3, 4)
```

In Haskell, this is called `concatMap`:

```
concatMap :: Foldable t => (a -> [b]) -> t a -> [b]
```

Interestingly, `map` can be defined in terms of `concatMap`:

```
map f = concatMap (\x -> [f x])
```

By swapping the first and second parameters of `concatMap` and generalizing `[b]` as `m b` for some type constructor `m`, we obtain the signature of the well-known "bind" function in *monad*:

```
(>>=) :: m a -> (a -> m b) -> m b
```

Secret of Box

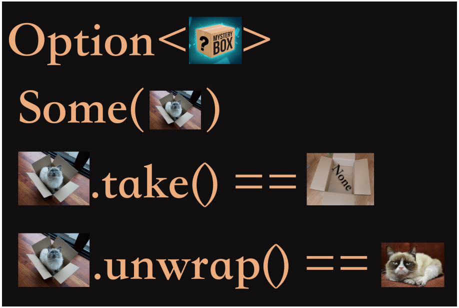

Figure: Option type, from Rust community.

```
($)   :: (a -> b) -> a -> b
fmap  :: (a -> b) -> f a -> f b
(>>=) :: m a -> (a -> m b) -> m b
```

Contents

Monad, an "Interface"

- In Haskell, `Monad` is a typeclass which requires `(>>=)` and `return`:

```
class Monad (m :: * -> *) where
  (>>=) :: m a -> (a -> m b) -> m b
  return :: a -> m a
```

- We see that

```
map f m = m >>= (return . f)
```

- In literature, `(>>=)` is called "bind" and `return` is called "unit".
- In Scala, `Monad` is a trait which also requires the above two methods:

```
trait M[T] {
  def flatMap[U](f: T => M[U]): M[U]
}
def unit[T](x: T): M[T]
```

All code presented in this talk may be different from the implementations in the standard library.

Instances

```haskell
instance Monad [] where
  return x = [x]
  (>>=) = concatMap

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

instance Monad (Either e) where
  return = Right
  Left  l >>= _ = Left l
  Right r >>= k = k r
```

`Right` means right!

Monad Laws

To qualify as a monad, a type has to satisfy three laws:

- Left unit:

```
(return e >>= f) = f e
```

- Right unit:

```
(m >>= return) = m
```

- Associativity:

```
((m >>= f) >>= g) = m >>= (\x -> (f x >>= g))
```

Maybe is a Monad

Let's check the monad laws for `Maybe`:

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

- Left unit: we can show

  ```
  (Just x >>= f) = f x
  ```

  by definition.

- Right unit: need to show

  ```
  (m >>= Just) = m
  ```

  by case analysis:

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

Maybe is a Monad (Cont.)

- Associativity: need to show

```
((m >>= f) >>= g) = m >>= (\x -> (f x >>= g))
```

by case analysis:

```
((Just x >>= f) >>= g) = f x >>= g
                       = Just x >>= (\x -> (f x >>= g))
((Nothing >>= f) >>= g) = Nothing >>= g = Nothing
                        = Nothing >>= (\x -> (f x >>= g))
```

Is Try a Monad?

```scala
abstract class Try[+T] {
  def flatMap[U](f: T => Try[U]): Try[U] = this match {
    case Success(x) =>
      try f(x) catch { case NonFatal(ex) => Failure(ex) }
    case fail: Failure => fail
  }
}
case class Success[T](x: T) extends Try[T]
case class Failure(ex: Exception) extends Try[Nothing]
```

Q: Does `Try` follows the monad laws?

- The left unit law fails:

```scala
Try(expr) flatMap f != f(expr)
```

- The left-hand side will never raise a non-fatal exception, whereas the right-hand side will raise any exception thrown by `expr` or `f`.

Make Monad Laws More Reasonable

Alteratively, we could use the monad composition operator (aka *Kleisli composition*)

```
(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> a -> m c
(m >=> n) x = do { y <- m x
                   n y }
```

to rewrite the monad laws:

- Left unit:

  ```
  (return >=> f) = f
  ```

- Right unit:

  ```
  (f >=> return) = f
  ```

- Associativity:

  ```
  (f >=> g) >=> h = f >=> (g >=> h)
  ```

“Monadic” Pipes

We could extend the F# pipe function (|>) to support options (marking failures) and lists, for fun:

```
let (?|>) (input: 't option) (next: 't -> 'v) : 'v option
let (?|>?) (input: 't option) (next: 't -> 'v option) : 'v option
let rec (||>) (inputs: 't list) (next: 't -> 'v) : 'v list
let rec (||>?) (inputs: 't list) (next: 't -> 'v option)
             : 'v list option
let (?||>?) (inputs: 't list option) (next: 't -> 'v option)
          : 'v list option
let (?||>) (inputs: 't list option) (next: 't -> 'v)
         : 'v list option
let assert (errMsg: string) (test: 't -> bool) (input : 't option)
         : 't option
let rec flatMapOption (f: 't -> 'v list option) (xs: 't list)
                    : 'v list option
```

Contents

Probabilistic Choice

- Type `Prob`: probabilities, e.g. a real between 0 and 1.
- Type `Dist a`: probability distributions.
- Probabilistic choice:

```
choice :: Prob -> a -> a -> Dist a
```

`choice p x y =` $\begin{pmatrix} x & y \\ p & 1-p \end{pmatrix}$

Example: a biased coin:

```
biasedCoin :: Prob -> Dist Bool
biasedCoin p = choice p False True
```

Composition Problem

Given two functions that creates distributions:

```
f :: a -> Dist b
g :: b -> Dist c
```

The naive function composition `g.f` is ill-typed!

Suppose that `Dist` is a monad, we could use Kleisli composition

```
(<=<) :: Monad m => (b -> m c) -> (a -> m b) -> (a -> m c)
```

and simply compose them in the following way:

```
g <=< f
```

Composing Distributions

```
f, g :: Int -> Dist Int
f x = choice 0.5 x (x + 1)
g x = choice 0.5 (x - 1) x

h = g <=< f
```

h x = $\begin{pmatrix} x-1 & x+1 & x \\ 0.25 & 0.25 & 0.5 \end{pmatrix}$

Remark: monad laws hold if we define the unit function as follows:

```
return :: a -> Dist a
return x = choice 0.5 x x
```

pGCL

A Probabilistic Guarded Command Language (pGCL), invented by Kozen, McIver and Morgan:

$$
\begin{aligned}
P ::= &\ \mathsf{skip} \mid \mathsf{abort} \mid x := E \mid P_1 ; P_2 \\
&\mid \mathsf{if}\ (G)\ P_1\ \mathsf{else}\ P_2 \mid \mathsf{while}\ (G)\ P \\
&\mid P_1\ [p]\ P_2 \\
&\mid \mathsf{observe}\ G
\end{aligned}
$$

where P denotes a program, x denotes a variable, E denotes an expression, G denotes a guard (condition), and p denotes a probability value.

We could use Markov Chains to model the operational semantics for pGCL.

pGCL: Example

$$
\begin{array}{l}
c := \mathsf{true}; \\
i := 0; \\
\mathsf{while}\ (c)\ \{ \\
\quad i := i + 1; \\
\quad c := \mathsf{false}\ [p]\ c := \mathsf{true}; \\
\} \\
\mathsf{observe}\ odd(i);
\end{array}
$$

The feasible program runs have a probability

$$\sum_{N \geqslant 0} (1 - p)^{2N} \cdot p = \frac{1}{2 - p}.$$

Monads are Everywhere

Monads here,

Monads there,

Monads are everywhere!

This poem is motivated by: Stefan Monnier, David Haguenauer. Singleton Types Here, Singleton Types There, Singleton Types Everywhere. PLPV'10.

Understanding Monad?

Q: How can I understand monad?

In fact, the question is problematic:

- Which "monad" do you mean?
- What makes you "understand" a new concept?

Contents

A Well-known Saying

A monad in $\mathcal{C}$ is just a monoid in the category of endofunctors of $\mathcal{C}$, with product replaced by composition of endofunctors and unit set by the identity endofunctor.

– Saunders Mac Lane, Categories for the Working Mathematician

Contents

Category

Definition

A category $\mathcal{C}$ comprises $\mathcal{C}$-objects (typically notated by $A, B, C, \ldots$) and $\mathcal{C}$-arrows (typically notated by $f, g, h, \ldots$), which are governed by the following axioms:

(i) For each arrow f, there are unique associated objects $src(f)$ and $tar(f)$, respectively the source and target of f, *not necessarily distinct*. We write $f : A \to B$ to denote f is an arrow with $src(f) = A$ and $tar(f) = B$.

(ii) For any two arrows $f : A \to B$ and $g : B \to C$ s.t. $tar(f) = src(g)$, there exists an arrow $g \circ f : A \to C$, namely the composition of f with g.

(iii) For any object A, there exists an arrow $1_A : A \to A$ called the identity arrow of A.

Category (Cont.)

Definition

(Cont.) Further, the arrow compositions are *associated*:

$$h \circ (g \circ f) = (h \circ g) \circ f$$

for any $f : A \to B$, $g : B \to C$, $h : C \to D$, and identity arrows behave as *identities*:

$$f \circ 1_A = f = 1_B \circ f$$

for any $f : A \to B$.

Theorem

Identity arrows on a given object are unique.

Discrete Categories

- **0** has neither objects nor arrows.
- **1** has exactly one object with the identity arrow:

$$\begin{tikzcd} \bigstar \arrow[loop right] \end{tikzcd}$$

- **2** has two objects, the necessary identity arrows, plus one further arrow between them:

$$\begin{tikzcd} \bigstar \arrow[loop left] \arrow[r] & \blacksquare \arrow[loop right] \end{tikzcd}$$

- The above "graphs" are called *diagrams*. Informally, a diagram represents some of the objects and arrows of a category as nodes and edges, in a directed graph. The identity arrows can be omitted.

More Categories

- **Grp**: objects are groups, and arrows are group homomorphisms.
- A monoid itself forms a category.
- **Pos**: objects are partially-ordered collections, and arrows are order-preserving maps.
- A pre-ordered set $(N, \leqslant)$ induces a category whose objects are the elements of N, and $A \to B$ forms an arrow if $A \leqslant B, A, B \in N$.
- **Vect**$_k$: objects are vector spaces over the field k, and arrows are linear maps between the spaces.
- **Set**: objects are all sets, and arrows are (total set) functions between them.

Lift the Arrows, First Attempt

Anything could be the objects of a category, even arrows, which derives the concept of arrow category $\mathcal{C}^{\to}$, when given any category $\mathcal{C}$, with

- objects are all $\mathcal{C}$-arrows, and
- arrows have the form $f_1 \to f_2$ given two $\mathcal{C}^{\to}$-objects $f_1 : X_1 \to Y_1$ and $f_2 : X_2 \to Y_2$, where there exists a pair of $\mathcal{C}$-arrows (j, k) s.t.

$$k \circ f_1 = f_2 \circ j \quad (*)$$

Alteratively, the property $(*)$ could be expressed as "the following diagram *commutes*:

$$\begin{tikzcd} X_1 \arrow[r, "j"] \arrow[d, "f_1"] & X_2 \arrow[d, "f_2"] \\ Y_1 \arrow[r, "k"] & Y_2 \end{tikzcd}$$

".

Lift the Arrows, Over Again

- Mathematicians not only study the *structures*, but also the *structures of structures*.
- A category is such a structure, and the arrows of the category connects the objects inside.
- It is natural to lift the arrow one level up, so that we can study the arrows between two categories.

Functors

Definition

Given two categories $\mathcal{C}$ and $\mathcal{D}$, a (covariant) functor $F : \mathcal{C} \to \mathcal{D}$

- maps every $\mathcal{C}$-object A into a $\mathcal{D}$-object (i.e. FA), and
- maps every $\mathcal{C}$-arrow $f : A \to B$ into a $\mathcal{D}$-arrow (i.e. $Ff : FA \to FB$).

Further, it must

(i) respect identity, i.e. $F1_A = 1_{FA}$, and

(ii) respect composition, i.e. $F(g \circ f) = Fg \circ Ff$.

Examples:

- The *forgetful functor* U : **Grp** $\to$ **Set** sends groups to their underlying carrier sets and sends group homorphisms to themselves as set functions, forgetting about the group structure.
- The *powerset functor* P : **Set** $\to$ **Set** maps a set X to its powerset $\mathcal{P}(X)$ and maps a set-function $f : X \to Y$ to the function which sends $Z \in \mathcal{P}(X)$ to its f-image $f[Z] = \{f(x) \mid x \in Z\} \in \mathcal{P}(Y)$.

Diagrams, Formally

Definition

Given categories $\mathcal{C}$ and $\mathcal{J}$, we say that a functor $D : \mathcal{J} \to \mathcal{C}$, is a diagram (of shape $\mathcal{J}$) in $\mathcal{C}$.

Remarks:

- A diagram usually contains a small part of the full category.
- Although being partial, a diagram must be a category.
- In a diagram, we could name the objects/arrows as we wish, as long as they have a one-one correspondence to the original ones in the category.

Functors Compose

- Given a category $\mathcal{C}$, a functor $F : \mathcal{C} \to \mathcal{C}$ is called an endofunctor of $\mathcal{C}$.
- There exists a trivial endofunctor which sends objects and arrows alike to themselves, namely the identity functor $1_{\mathcal{C}} : \mathcal{C} \to \mathcal{C}$.
- Like arrows, functors can also be composed:

Theorem

For any two functors $F : \mathcal{C} \to \mathcal{D}$ and $G : \mathcal{D} \to \mathcal{E}$, there exists a functor $G \circ F : \mathcal{C} \to \mathcal{E}$, or GF for short, namely the composition of G with F, which

- *maps a $\mathcal{C}$-object A to an $\mathcal{E}$-object (i.e. GFA), and*
- *maps a $\mathcal{C}$-arrow $f : A \to B$ to an $\mathcal{E}$-arrow (i.e. $GFf : GFA \to GFB$).*

As a shorthand, we write F^n $(n > 1)$ for $F^{n-1} \circ F$.

"Functors" in Haskell

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

- The signature of `fmap` looks like it maps a $\mathcal{C}$-arrow $f : A \to B$ into a $\mathcal{D}$-arrow $Ff : FA \to FB$.
- Once there were some Haskell people said that **Hask** is a category, with all Haskell types as objects, and function types as arrows. In fact, **Hask** is even NOT a category, and is NOT *Cartesian closed*.[3]
- However, some subset of Haskell where types do not have *bottom values* might form a real category.

[3] https://wiki.haskell.org/Hask

Categories of Categories

- Trivially, there is a category of categories whose sole object is some category $\mathcal{C}$ and whose sole arrow is the identity functor $1_{\mathcal{C}}$.
- As an extension, there is a category whose objects are all finite categories, and whose arrows are all the functors between them.
- Unsurprisingly, there is no *universal category*, i.e. a category $\mathcal{U}$ such that every category is an object of $\mathcal{U}$.
- Q: How many categories can we "put into" a category?

A "Small Cat"

Definition

- A category $\mathcal{C}$ is small iff it has overall only a "set's worth" of arrows – i.e. the arrows can be put into one-one correspondence with the members of some set.
- A category $\mathcal{C}$ is locally small iff for every pair of $\mathcal{C}$ objects (C, D), there is only a "set's worth" of arrows from C and D.

- **Cat** is the category whose objects are small categories and whose arrows are the functors between them.
- **Cat*** is the category whose objects are locally small categories and whose arrows are the functors between them.

Q: Can we, again, lift functors one more level up?

Natural Transformation

Definition

Given two categories $\mathcal{C}$ and $\mathcal{D}$. Let $F, G : \mathcal{C} \to \mathcal{D}$ be two functors. Suppose that for each $\mathcal{C}$-object C there is a $\mathcal{D}$-arrow $\alpha_C : FC \to GC$. Then α, the family of arrows α_C, is a natural transformation between F and G, written $\alpha : F \Rightarrow G$, iff for every $\mathcal{C}$-arrow $f : A \to B$,

$$\alpha_B \circ Ff = Gf \circ \alpha_A.$$

That is, the following diagram commutes:

$$\begin{tikzcd}
FA \arrow[r, "Ff"] \arrow[d, "\alpha_A"] & FB \arrow[d, "\alpha_B"] \\
GA \arrow[r, "Gf"] & GB
\end{tikzcd}$$

In sum, α sends an F-image of (some or all of) $\mathcal{C}$ to its G-image in a way which, at least, preserves composition.

Functor Categories

Definition

The functor category from a category $\mathcal{C}$ to a category $\mathcal{D}$, denoted $[\mathcal{C}, \mathcal{D}]$, is the category whose objects are all the (covariant) functors $F : \mathcal{C} \to \mathcal{D}$, with the natural transformations between them as arrows.

Remarks:

- Especially, $[\mathcal{C}, \mathcal{C}]$ is called the endofunctor category of $\mathcal{C}$.
- The identity arrow of a $[\mathcal{C}, \mathcal{D}]$-object F is trivially the identity natural transformation $1_F : F \Rightarrow F$, whose components 1_{FC} are identity arrows.
- In a functor category, natural transformations are just normal arrows, and thus we use the standard arrow notion $\to$ instead of $\Rightarrow$. In particular, we draw $\to$ in diagrams.

Old Wine in New Bottles

Consider the functor category $[\mathbf{2}, \mathcal{C}]$.

- An object in this category is a functor $F : \mathbf{2} \to \mathcal{C}$, where
  - $F\bigstar = X$ and $F\blacksquare = Y$ for some $\mathcal{C}$-object X and Y;
  - $F1_{\bigstar} = 1_X$, $F1_{\blacksquare} = 1_Y$, and $F(\bigstar \to \blacksquare) = f : X \to Y$.

  There is a bijection between the objects of $[\mathbf{2}, \mathcal{C}]$ and the arrows of $\mathcal{C}$.
- An arrow is a natural transformation between two functors $F, G : \mathbf{2} \to \mathcal{C}$, involving any $\mathcal{C}$-arrows j and k as components, which makes the square commute:

$$\begin{tikzcd} F\bigstar \arrow[r, "F(\bigstar\to\blacksquare)"] \arrow[d, "j"] & F\blacksquare \arrow[d, "k"] \\ G\bigstar \arrow[r, "G(\bigstar\to\blacksquare)"] & G\blacksquare \end{tikzcd}$$

  There is a bijection between the natural transformations and the pairs of $\mathcal{C}$-arrows.

In sum, $[\mathbf{2}, \mathcal{C}]$ is (*isomorphic* to) the arrow category $\mathcal{C}^{\to}$.

Contents

Adjoint Functors

Definition

Given two categories $\mathcal{C}$ and $\mathcal{D}$. Let $F : \mathcal{C} \to \mathcal{D}$ and $G : \mathcal{D} \to \mathcal{C}$ be two functors. Then F is left adjoint to G and G is right adjoint to F, notated $F \dashv G$, iff

(i) there are natural transformations $\eta : 1_{\mathcal{C}} \Rightarrow GF$, called the unit, and $\varepsilon : FG \Rightarrow 1_{\mathcal{D}}$ called the counit, such that

(ii) for every $\mathcal{C}$-object A, $\varepsilon_{FA} \circ F\eta_A = 1_{FA}$, and for every $\mathcal{D}$-object B, $G\varepsilon_B \circ \eta_{GB} = 1_{GB}$.

Equivalently, (ii) means the following diagrams commutes:

$$\begin{tikzcd} FA \arrow[r, "F\eta_A"] \arrow[dr, "1_{FA}"'] & FGFA \arrow[d, "\varepsilon_{FA}"] \\ & FA \end{tikzcd} \qquad \begin{tikzcd} GB \arrow[r, "\eta_{GB}"] \arrow[dr, "1_{GB}"'] & GFGB \arrow[d, "G\varepsilon_B"] \\ & GB \end{tikzcd}$$

Putting Triangles Together

$$
\begin{tikzcd}
GB \arrow[r, "\eta_{GB}"] \arrow[dr, "1_{GB}"'] & GFGB \arrow[d, "G\varepsilon_B"] \\
& GB
\end{tikzcd}
\qquad
\begin{tikzcd}
FA \arrow[r, "F\eta_A"] \arrow[dr, "1_{FA}"'] & FGFA \arrow[d, "\varepsilon_{FA}"] \\
& FA
\end{tikzcd}
$$

In the above two triangles, let $B = FA$, and apply G to the right diagram so that it also becomes a diagram on $\mathcal{C}$. Now, we obtain the following commutative diagram:

$$
\begin{tikzcd}
GFA \arrow[r, "\eta_{GFA}"] \arrow[dr, "1_{GFA}"'] & GFGFA \arrow[d, "G\varepsilon_{FA}"] & GFA \arrow[l, "GF\eta_A"'] \arrow[dl, "1_{GFA}"] \\
& GFA &
\end{tikzcd}
$$

Example: $U \vdash F$ where $U : \mathbf{Grp} \to \mathbf{Set}$ is the forgetful functor, and $F : \mathbf{Set} \to \mathbf{Grp}$ is a functor which sends a set to the *free group* on that set.

Motivation

Q: Given any category $\mathcal{C}$ and endofunctor $T : \mathcal{C} \to \mathcal{C}$. When is $T = GF$ for some adjoint functor $F \dashv G$ to and from another category $\mathcal{D}$?

- Suppose we have $\mathcal{D}$ and $F \dashv G$, and $T = GF$. Then, we have a natural transformation $\eta : 1_{\mathcal{C}} \Rightarrow T$.
- For an arbitrary $\mathcal{C}$-object C, we have $\varepsilon_{FC} : FGFC \to FC$, and hence $G\varepsilon_{FC} : GFGFC \to GFC$, i.e. $T^2C \to TC$, yielding a natural transformation $\mu : T^2 \Rightarrow T$.
- In general, if T arises from an adjunction, then it should have such a structure (T, η, μ).
- What properties does (T, η, μ) have?

Motivation (Cont.)

Property

(Associativity) $\mu \circ \mu_T = \mu \circ T\mu$.

Proof.

For any $\mathcal{D}$-arrow $f : X \to Y$, the diagram
$$\begin{tikzcd} FGX \arrow[r, "FGf"] \arrow[d, "\varepsilon_X"] & FGY \arrow[d, "\varepsilon_Y"] \\ X \arrow[r, "f"] & Y \end{tikzcd}$$
commutes since ε is a natural transformation. Let $X = FGY$, $Y = FC$ (for every $\mathcal{C}$-object C), and $f = \varepsilon_Y$, applying G therefore gives the commutative diagram:

$$\begin{tikzcd} GFGFGFC \arrow[r, "GFG\varepsilon_{FC}"] \arrow[d, "G\varepsilon_{FGFC}"] & GFGFC \arrow[d, "G\varepsilon_{FC}"] \\ GFGFC \arrow[r, "G\varepsilon_{FC}"] & GFC \end{tikzcd} \equiv \begin{tikzcd} T^2(TC) \arrow[r, "T\mu_C"] \arrow[d, "\mu_{TC}"] & T^2C \arrow[d, "\mu_C"] \\ T(TC) \arrow[r, "\mu_C"] & TC \end{tikzcd} \equiv \begin{tikzcd} T^3 \arrow[r, "T\mu"] \arrow[d, "\mu_T"] & T^2 \arrow[d, "\mu"] \\ T^2 \arrow[r, "\mu"] & T \end{tikzcd}$$

Recall that $T = GF$, $\mu_C = G\varepsilon_{FC}$. $\square$

Motivation (Cont.)

Property

(Unit) $\mu \circ \eta_T = 1_T = \mu \circ T\eta$.

Proof.

The following diagram immediately commutes since $F \dashv G$:

$$
\begin{tikzcd}
GFC \arrow[r, "\eta_{GFC}"] \arrow[dr, "1_{GFC}"'] & GFGFC \arrow[d, "G\varepsilon_{FC}"] & GFC \arrow[l, "GF\eta_C"'] \arrow[dl, "1_{GFC}"] \\
& GFC &
\end{tikzcd}
\equiv
\begin{tikzcd}
T \arrow[r, "\eta_T"] \arrow[dr, "1_T"'] & T^2 \arrow[d, "\mu"] & T \arrow[l, "T\eta"'] \arrow[dl, "1_T"] \\
& T &
\end{tikzcd}
$$

Recall that $T = GF$, $\mu_C = G\varepsilon_{FC}$. □

Monad

Definition

A monad (T, η, μ) on a category $\mathcal{C}$ consists of an endofunctor $T : \mathcal{C} \to \mathcal{C}$, with natural transformations $\eta : 1_{\mathcal{C}} \Rightarrow T$ called the unit, and $\mu : T^2 \Rightarrow T$ called the multiplication, satisfying two axioms:

(Associativity) $\mu \circ \mu_T = \mu \circ T\mu$, and

(Unit) $\mu \circ \eta_T = 1_T = \mu \circ T\eta$.

Proposition

Every adjoint functor pair $F \vdash G$ with $G : \mathcal{D} \to \mathcal{C}$, unit $\eta : GF \Rightarrow 1_{\mathcal{C}}$, and counit $\varepsilon : 1_{\mathcal{D}} \Rightarrow FG$ gives rise to a monad (T, η, μ) on $\mathcal{C}$ with

- $T = GF : \mathcal{C} \to \mathcal{C}$, *and*
- $\mu = G\varepsilon_F : T^2 \Rightarrow T$.

The Well-known Saying, Again

A monad in $\mathcal{C}$ is just a monoid in the category of endofunctors of $\mathcal{C}$, with product replaced by composition of endofunctors and unit set by the identity endofunctor.

– Saunders Mac Lane, Categories for the Working Mathematician

Given a monad (T, η, μ) on a category $\mathcal{C}$.

- Regard $\mu : T \circ T \Rightarrow T$ as multiplication, the axiom (Associativity) induces the associativity of the multiplication.
- Regard 1_T as the identity, the axiom (Unit) induces that η is a witness to the existence of 1_T.

Powerset Functor, Revisited

Consider the powerset functor $P : \mathbf{Set} \to \mathbf{Set}$.

- Let $\eta_X : X \to \mathcal{P}(X)$ be the singleton operation

$$\eta_X(x) = \{x\}.$$

- Let $\mu_X : \mathcal{P}(\mathcal{P}(X)) \to \mathcal{P}(X)$ be the union operation

$$\mu_X(S) = \bigcup S.$$

You can verify that (P, η, μ) is a monad.

Category Theory for Everyone

The audience ... This could include a motivated high school student who hasn't seen calculus yet but has loved reading a weird book on mathematical logic they found at the library. Or a machine-learning researcher who wants to understand what vector spaces, design theory, and dynamical systems could possibly have in common. Or a pure mathematician who wants to imagine what sorts of applications their work might have. Or a recently-retired programmer who's always had an eerie feeling that category theory is what they've been looking for to tie it all together, but who's found the usual books on the subject impenetrable.
– Brendan Fong, David I. Spivak. Seven Sketches in Compositionality: An Invitation to Applied Category Theory.

Link: `https://johncarlosbaez.wordpress.com/2018/03/26/seven-sketches-in-compositionality/`

Contents

Theory & Practice

Theory is when you know everything but nothing works. Practice is when everything works but no one knows why. In our lab, theory and practice are combined: nothing works and no one knows why.
– A proverb

- Programming Languages (PL) is one of the most theoretical topic in computer science, but
- it is also one of the most practical field targeting at software and system engineering.

Goals of PL?

- Working languages?
- Faster programs?
- Correctness and safety?
- Automatic code generation?
- etc.

I believe anything that could be expressed as a PL can be studied in the field of PL.

Future Topics

- Dependent types, gradual typing, session types, game semantics
- Dynamic languages
- Program synthesis, automatic programming
- Probabilistic programs, quantum programs
- Categories
- Big code, machine learning, linear algebra

References & Further Reading

- Coursera: Functional Program Design in Scala. [link]
- Richard Bird. Thinking Functionally with Haskell.
- Scibior et al. Practical Probabilistic Programming with Monads. Haskell'15.
- Peter Smith. Category Theory: A Gentle Introduction. [link]
- Steve Awodey. Category Theory.
- PL Conferences: POPL, PLDI, ICFP, OOPSLA, ECOOP, CPP, PEPM, etc.

Acknowledgement

I'd like to thank Feng Jiang et al. for the seminar/workshop on category theory, held in Capital Normal University, in which we learned and discussed a lot, and Joost-Pieter Katoen for his tutorial *Principles of Probabilistic Programming* in SSFM'18. Also, I appreciate TUNA for organizing Tunight, where speakers communicate weird topics with the audiences, and it could be even better if have some tuna for dim sum.