Crash Course in Monads

Vlad Patryshev

Introduction

Monads in programming seem to be the most mysterious notion of the century. I find two reasons for this:

- lack of familiarity with category theory;
- many authors carefully bypass any mention of categories.

It's like talking about electricity without using calculus. Good enough to replace a fuse, not good enough to design an amplifier.

This crash course starts with an easy introduction to categories and functors, then we define a monad, then give some basic examples of monads in categories, then present monadic terminology as used in programming languages.

I am sure that if you approach the topic from categorical point of view, everything will look almost elementary.

Vlad Patryshev, 3/7/2006 - 2/12/2007

Category

A *category* consists of *objects* and *morphisms* between objects. The term "morphism" is a little bit misleading (they are not required to morph anything); so morphisms are frequently called *"arrows"*, to stress their abstract nature. I'll use the term "arrow" except when an arrow represent some kind of function, in which case I'll call it a "morphism". But it's still just an arrow to me.

We do not care about the nature of object and arrows; all we need are the following properties:

An arrow starts at an object and ends at another (may be the same object); this is denoted in the following way: $f: a \to b$, where f is an arrow, and a and b are objects;

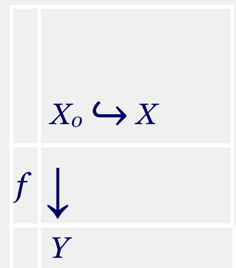
1. For arrows $f:a \to b$ and $g:b \to c$ there is an arrow $h: a \to c$ that is called their *composition*: $h = g \circ f$.
2. For each object a there is a *unit arrow*, $id_a: a \to a$, such that for any $f: a \to b$ the following is true: $f \circ id_a = f$, and for any $g: c \to a$ we have $id_a \circ g = g$.
3. Composition is associative: $f \circ (g \circ h) = (f \circ g) \circ h$.

Note. Due to an extremely abstract nature of the notion, we cannot even expect "all objects" to form a set, or "all arrows from *a* to *b*" form a set. Categories where these are sets are called *"small"* and *"locally small"*.

Examples of Categories

The following examples are "classic" categories.

1. ***Set*** - a category of all sets. Its objects are all sets; its morphisms are set functions.
2. ***Setf*** - a category of all finite sets and functions between them.
3. ***Rel*** - a category where objects are all sets, and binary relationships play the role of morphisms. Composition is defined via inner join.
4. ***Part*** - a category of all sets and partial functions as morphisms. A partial function from X to Y is a function from a subset $X_o \subset X$ to Y:



5. ***Top*** - a category of all topological spaces and continuous functions between them.

More Examples of Categories

There are more categories in the world than just general theories.

1. Any group can be considered a category: group elements are morphisms over one single object. Id is the group's neutral element. Composition is multiplication.
2. A partially ordered set can be represented as a category. The set's elements are objects. Add a single arrow $a \to b$ for each pair a, b such that $a < b$, and unit arrow $a \to a$ for each a.
   For each pair of objects there's no more than one arrow, and since partial order is transitive, we have composition ($a<b$, $b<c$ => $a<c$), and there is no need to worry about its associativity.
3. As a special case of the previous example, a segment of integers, $[N..M]$ can be thought of as a category.
4. Take any oriented graph. We can turn it into a category by treating its paths as arrows. An empty path is a unit morphism; path composition is concatenation.
5. Natural numbers as objects, matrices as morphisms. Matrix multiplication would play the role of composition; a unit N×N matrix is a unit morphism $N \to N$.

(You can **skip** the next page)

Extra Material

It is easy to define an isomorphism in a category: it is the one that has an inverse.

That is, if we have $f: a \rightarrow b$ and $g: b \rightarrow a$, and $f \circ g = id_b$ and $g \circ f = id_a$.

We will need this notion later on. A monomorphism and an epimorphism could be also defined, but it takes more efforts, and we are not going to cover them here.

Remember $[0..N]$ objects from the previous page? There are two special categories, $\mathbf{1} = [0]$, and $\mathbf{2} = [0..1]$. The first one has just one object and one morphism; the second one has two objects and three morphisms.

Do categories themselves form a category? They would, but we need to define arrows between categories. That's the second order arrows, and they are called functors.

Functor

A functor maps one category to another. To do this, we have to map objects of the first category to objects of the second category, and arrows (morphisms) of the first category to arrows of the second category, in a consistent manner.

What kind of consistency do we expect? Let X and Y be two categories, and let's start defining a functor $F: X \to Y$. We have to map objects of X to Y, having for each a in X an object $F(a)$ in Y, and for an arrow f in X we have to have an arrow $F(f)$ in Y.
For consistency we will need the following:

- for $f: a \to b$ have $F(f): F(a) \to F(b)$ - domain and codomain preservation;
- for $id_a: a \to a$ have $F(id_a) = id_{F(a)}: a \to a$ - unit preservation;
- for $f:a \to b$ and $g:b \to c$ $F(g \circ f) = F(g) \circ F(f)$ - composition preservation.

Functor composition is defined in an obvious way: apply one functor, then another.

Examples of Functors

1. Identity functor: for a category X an identity $X \to X$ keeps objects and arrows intact. Still it is a functor.

2. ***Setf*** $\hookrightarrow$ ***Set*** - a functor that includes ***Setf*** into ***Set***, that is, maps each finite set to itself, and the same with functions. Note that this is not an identity functor.
3. ***Set*** $\to$ ***Top*** - similar to the previous example, this functor makes ***Set*** a part of ***Top***. Each set is mapped to a discrete topological space.
4. For any set A we can define the following functor: $(- \times A)$: ***Set*** $\to$ ***Set*** - it will map any set X to a Cartesian product, $X \times A$.
5. For any set A we can define a functor P_A: ***Set*** $\to$ ***Set***; it maps any set X to X^A, a set of functions from A to X.

6. ***Set*** $\hookrightarrow$ ***Part*** embeds sets to sets with partial functions: it maps sets and functions to themselves.
7. An opposite to 6. is a functor $+Null$: ***Part*** $\to$ ***Set*** -this functor adds an "extension" $Null$ to each set: $X \mapsto (X+Null)$, so that a partial function X → Y maps to a function $(X+Null) \to (Y+Null)$. (Exercise. Define such an extension for partial functions.)

Again, let's take a look at small categories and their functors.

1. If we consider groups as categories, what would be their functors? A functor must preserve unit morphism and composition. Hence, a functor is just a group homomorphism.
2. Any order-preserving (a.k.a. monotonous) function between two partially ordered sets is a functor.
3. Take a pair of oriented graphs and a ma that preserve edges. We can extend this map to a function that maps path from one graph to paths of another graph. This function, by definition, preserves concatenation and empty paths, thus it is a functor from one graph-generated category to another.
4. Remember category $\mathbf{1}$? Now, how would a functor from $\mathbf{1}$ to a category $\mathbf{C}$ look like? $\mathbf{1}$ has just one object, and an identity morphism. So, to specify such a functor is the same as to select an object in $\mathbf{C}$ - and vice versa, for any object X in category $\mathbf{C}$ we can specify a functor $Point_X$: $\mathbf{1} \to \mathbf{C}$

Natural Transformation

This is probably the most difficult part of this presentation... Suppose we have two functors, $F,G\colon \mathbf{X} \to \mathbf{Y}$. A natural transformation $\varphi\colon F \to G$ is defined when for each object $x \in \mathbf{X}$ there is an arrow $\varphi(x)\colon F(x) \to G(x)$ in $\mathbf{Y}$, and we have the following property:

- for all $f\colon a \to b$ the equality is true: $\varphi(a) \circ G(f) = F(f) \circ \varphi(b)$.

$$\begin{array}{ccc} F(a) & \xrightarrow{F(f)} & F(b) \\ \varphi(a)\downarrow & & \downarrow\varphi(b) \\ G(a) & \xrightarrow[G(f)]{} & G(b) \end{array}$$

That's why it is called 'natural' - it acts consistently with the functors actions on arrows.

Examples of Natural Transformations

1. Remember point functors? Now, if in category $\mathbf{C}$ we have an arrow $f: a \to b$, this arrow defines a natural transformation from $Point_a$ to $Point_b$. There is a one-to-one match between transformations of Point functors and arrows of a category.
2. Let's take two sets, A, B, and a function $f: A \to B$. This function determines a natural tranformation between functors $(- \times A),(- \times B)$: $\mathbf{Set} \to \mathbf{Set}$.
   $(-\times f): (-\times A) \to (-\times B)$ by the following formula: $(x,a) \mapsto (x,f(a))$.

   I feel a temptation to write the definition like this:

   ```
   (define (cartesian f)
      (lambda (x a) (list x (f a))))
   ```

   Since for every set A there is a function A → (.), where (.) is a singleton set, we have a natural transformation $(-\times A) \to 1$ that for each object X is just a projection: $X \times A \to X$.

For each A in ***Set*** there is a natural transformation $1 \to P_A$. Take any X in ***Set***; we need a function from X to X^A. The natural choice would be the one that maps each element x of X to a constant function from A to X that returns x.

Again, I feel a temptation to write the definition like this:

```
(define (return x) (lambda (a) (x))
```

Monad

A monad in category $\mathbf{C}$ is an endofunctor $F\colon \mathbf{C} \to \mathbf{C}$ with two natural transformations:
$u\colon 1 \to F$ and $m\colon F \circ F \to F$.
Let's denote $F(u)\colon F \text{->} F \circ F$ the transformation that results in applying F to u, and $F(m) : (F \circ F) \circ F \to F \circ F$.
Having these, now we can write down the following two monad axioms:

1. $F(u) \circ m$ is identity $F \to F$

   :

	$F(u_X)$		$m_{F(X)}$	
$F(X)$	$\longrightarrow$	$F(F(X))$	$\longrightarrow$	$F(X)$

2. $F(m) \circ m$ is the same as $m \circ m$:

	$F(m_X)$	
$F(F(F(X)))$	$\longrightarrow$	$F(F(X))$
$m_{F(X)}\downarrow$		$\downarrow m_X$
$F(F(X))$	$\longrightarrow$	$F(X)$
	m_X	

Examples of Monads

1. For any category $\mathbf{C}$ an identity monad can be defined. It consists of an identity functor and identity morphisms.

2. Suppose we have a group G. Let's define a monad M_G in $\mathbf{Set}$. The monad functor will be like this:
   $X \mapsto X \times G$.
   $u(X) : X \to X \times G$ maps an element x to a pair (x,e), where e is the group's unit.
   $M_G(M_G(X)) = (id_X,m_G)$, where m_G is the group multiplication.

3. Lists in $\mathbf{Set}$. For a set X the result of applying the functor, let's call it $List$, is the set of all lists, $(x_1,x_2,x_3...)$, including the empty one, of elements of X. This functor becomes a popular monad if we add u and m. Let $u_X : X \to List(X)$ create a single-element list for each $x \in X$.
   And $m_X: List(List(X)) \to List(X)$ maps lists of lists to plain lists by flattening them

More Examples of Monads

Closure operation. Not related to closures in computer science.

Remember that we can treat partially-ordered sets and their order-preserving functions as categories and functors?

A monotonous (order-preserving) function $C: X \to X$ is called **closure** if $\forall x \in X \quad x <= C(x)$ and $C(C(x)) = C(x)$.

These two conditions are exactly what monad axioms turn into when applied to a partially ordered set - meaning that monads in partially ordered sets are just closures. We can also try to apply this to the partially ordered set of paths in a graph.

(**skip the next page if it is too difficult**)

Exception Monad

We are in ***Part*** category. Take an set *A*, and let's define the following functor:

PlusNull: X ↦ (X+Null)

We already talked about this functor, and last time it was from ***Part*** to ***Set***. This time we composed it with the inclusion of ***Set*** to ***Part***, thus getting an endofunctor.

Why is it a monad? We need $u_X: X \to (X+Null)$ and $m_X: ((X+Null) + Null) \to (X + Null)$.

The first one is a simple inclusion; the second one maps both *Null* singletons to *Null*. In Lisp it looks like this:

```
(define (ux x) x)

(define (mx x) x)
```

As you see, it is a monad (if you don't, prove it as an exercise).

We are in ***Set*** category. Take an set A, and let's define the following functor:

$X \mapsto (X{\times}A)^A$

We can think of A as a machine's set of states; then $(X \times A)^A$ consists of all state machines on X with output to X, that is, all functions $A \to (A \times X)$, the first component being transition, and the second an output to X.

Why is it a monad? $u_X : X \to (A \times X)^A$ maps any element $x \in X$ to a function that is identity on A and the constant x on X.
Let me express it in Lisp:

```
(define (ux x)
   (lambda (a) (list a x)))
```

How about $m_X : (A \times (A \times X)^A)^A \to (A \times X)^A$?

Yes, how about $m_X : (A \times (A \times X)^A)^A \to (A \times X)^A$?

We have another collection of state machines, $m_X : (A \times (A \times X)^A)^A$, which has A as a state set, and which outputs to another collection of state machines (this one too has A as a state set, and X as output set).How do we find for such a compound state machine a match in a state machine on A? Write it in Lisp:

```
(define (mx f)
  (let (tr1 out1)
       ((car f)(cadr f)));two components of f:A → (A × (A × X)^A
  (lambda (a)
   (let a1 (tr1 a));state after first transition
   (let f2 (out1 a)) ;mapped a state to another state machine
   (let (tr2 out2) ((car f2) (cadr f2))) ;second machine 
components
      (list (tr2 a1) (out2 a1))))
```

See what happens here: we have a function from A to $A{\times}(A{\times}X)^A$, which consists of transition $A \to A$ and output $A \to (A{\times}X)^A$, that is, for each a we have another state $a1$ and an output function; the resulting function from A to $A{\times}X$ should just apply that output function to the new state.

Boring Exercise: Prove that this is actually a monad.

Monads in Programming

From categorical point of view, functional programming consists of representing a program as a morphism in a category: $f\colon X \to Y$, where X is "input", and Y is "output". (The advantage of such a model is that we can immerse all our statement regarding programs into a very stable and sound category theory; we can vary the nature of objects and categories, change the logic of the underlying topos and still be 100% aware of what we are talking about. E.g. instead of "fuzzy logic" we can use intuitionist logic and Kripke-Joyal semantics.)

Certain programming activities seem not to fit into this model: e.g. exceptions and side effect.

To deal with exceptions, one solution is to introduce a NullObject, or a NaN numeric value. For instance, if our function is not defined on the whole domain X, we can extend it to taking values in $(Y+Null)$, as in **Exception Monad**.

To deal with stateful functions, we need to apply the notion of **State Machine Monad**.

Monads in Programming - references

See **http://en.wikipedia.org/wiki/Monads_in_functional_programming** as the primary source of references.

This link:
http://mikael.jansson.be/hacking/misc/export/83/tutors_and_papers/haskell/db-utwente-0000003696.pdf is a complicated, boring but comprehensive explanation of monadic terminology used in programming.

"Comprehending “ Comprehending Monads” ' by Frederik Eaton:
http://ofb.net/~frederik/comp2.pdf is a 3-page nice glossary of terms and notions.

In Haskell, *u* is called return, and *m* is called **combinator**, or '**join**'.

Haskell books mention *List* as an example of a monad, and have a rather special interpretation of State Machine Monad.

Another example of a monad is **Google's map/reduce**.

Haskell IO Monad

Haskell introduced IO monad as if to overcome the awkwardness of explanation, how come a function can have a side-effect.

Here is the trick they use in Haskell. Take the State Machine Monad, as we did before. Imagine that A, the state set, represents the whole outside world, and X is the set of function results.

In this model, every state machine becomes a function. A is further represented as a Cartesian product of two $String$ sets, the first component being "input", and the second - "output". Special transition functions are introduced, one, `getc`, pops a character from input; the other, `putc`, adds a character to output.

I am just curious, what exactly does this model try to achieve.

Contents

Introduction 2
Category 3
Examples of Categories 4
More Examples of Categories 5
Extra Material 6
Functor 7
Examples of Functors 8
More Examples of Functors 9
Natural Transformation 10
Examples of Natural Transformations 11
More Examples of Natural Transformations 12
Monad 13
Examples of Monads 14
More Examples of Monads 15
Exception Monad 16
State Machine Monad 17
State Machine (continued) 18
Monads in Programming 19
Monads in Programming - references 20
Haskell IO Monad 21