A monad is just a monoid in the category of endofunctors, what's the problem?

Who first said

> A monad is just a monoid in the category of endofunctors, what's the problem?

and on a less important note is this true and if so could you give an explanation (hopefully one that can be understood by someone who doesn't have much haskell experience).

haskell quotes monads category-theory monoids

edited Dec 24 '10 at 15:43

asked Oct 6 '10 at 6:55
Roman A. Taycher
3,122 ●13 ●42 ●90

50 "hopefully one that can be understood by someone who doesn't have much haskell experience" Whether or not someone can understand this quote has little to do with his Haskell experience and a lot with his maths experience/knowledge about category theory. Also understanding this quote will tell you nothing about how monads in Haskell work and how to use them. So if that's your intention, you should not use this quote as a starting point (and probably forget that monads come from category theory altogether unless you want to understand why they are named monads). – sepp2k Oct 6 '10 at 8:03

38 IOW: it's a joke. – luqui Oct 6 '10 at 13:19

8 See "Categories for the Working Mathematician" – Don Stewart Oct 6 '10 at 15:27

10 You don't need to understand this to use monads in Haskell. From a practical perspective they are just a clever way to pass around "state" through some underground plumbing. – starblue Oct 7 '10 at 18:00

18 Actually, really understanding this quote *did* help me understand monads in a deeper sense, as well as monoids and functors. It only requires you to know other concepts which you *should* know anyway, to truly understand those concepts. And when you do, it nicely brings the concept to a single mental point. So ignore the stupid unconstructive comments above. All one needs, is a *proper* explanation of those concepts, before reading that quote. Then it't exactly the right thing to say. Which is the whole joke behind it. (That people don't know those concepts.) – Evi1M4chine Mar 3 '13 at 16:29

4 Answers

That particularly phrasing is by James Iry, from his highly entertaining *Brief, Incomplete and Mostly Wrong History of Programming Languages*, in which he fictionally attributes it to Philip Wadler.

The original quote is from Saunders Mac Lane in *Categories for the Working Mathematician*, one of the foundational texts of Category Theory. Here it is in context, which is probably the best place to learn exactly what it means.

But, I'll take a stab. The original sentence is this:

> All told, a monad in X is just a monoid in the category of endofunctors of X, with product × replaced by composition of endofunctors and unit set by the identity endofunctor.

X here is a category. Endofunctors are functors from a category to itself (which is usually *all* `Functor` s as far as functional programmers are concerned, since they're mostly dealing with just one category; the category of types--but I digress). But you could imagine another category which is the category of "endofunctors on *X*". This is a category in which the objects are endofunctors and the morphisms are natural transformations.

And of those endofunctors, some of them might be monads. Which ones are monads? Just exactly the ones which are *monoidal* in a particular sense. Instead of spelling out the exact mapping from monads to monoids (since Mac Lane does that far better than I could hope to), I'll just put their respective definitions side by side and let you compare:

A monoid is...

- A set, $\boldsymbol{S}$
- An operation, $\boldsymbol{\bullet : S \times S \rightarrow S}$
- An element of S, $\boldsymbol{e : 1 \rightarrow S}$

...satisfying these laws:

- $(a \bullet b) \bullet c = a \bullet (b \bullet c)$, for all a, b and c in S
- $e \bullet a = a = a \bullet e$, for all a in S

A monad is...

- An endofunctor, $\boldsymbol{T : X \rightarrow X}$ (in Haskell, a type constructor of kind `* -> *` with a `Functor` instance)
- A natural transformation, $\boldsymbol{\mu : T \times T \rightarrow T}$, where × means functor composition (also known as `join` in Haskell)
- A natural transformation, $\boldsymbol{\eta : I \rightarrow T}$, where I is the identity endofunctor on X (also known as `return` in Haskell)

...satisfying these laws:

- $\mu(\mu(T \times T) \times T)) = \mu(T \times \mu(T \times T))$
- $\mu(\eta(T)) = T = \mu(T(\eta))$

With a bit of squinting you can probably see that both of these definitions are instances of the same abstract concept (I think category theorists would say "monoid" is the abstract term, and my definition of "monoid" above is overly specific since it mentions sets and elements).

edited Aug 31 '15 at 0:48 answered Oct 6 '10 at 7:35
Tom Crockett
17.8k ●3 ●47 ●63

9 thanks for the explanation and thanks for the Brief, Incomplete and Mostly Wrong History of Programming Languages article. I thought it might be from there. Truly one of the greatest pieces of programming humor. – Roman A. Taycher Oct 6 '10 at 13:39

4 @Jonathan: In the classical formulation of a monoid, × means the cartesian product of sets. You can read more about that here: en.wikipedia.org/wiki/Cartesian_product, but the basic idea is that an element of $S \times T$ is a pair (s, t), where $s \in S$ and $t \in T$. So the signature of the monoidal product $\bullet : S \times S \text{ -> } S$ in this context simply means a function that takes 2 elements of S as input and produces another element of S as an output. – Tom Crockett Oct 20 '10 at 8:19

8 I have to memorize this definition, to show off :p – Aivar Sep 14 '11 at 19:47

7 @TahirHassan - In the generality of category theory, we deal with opaque "objects" instead of sets, and so there is no a priori notion of "elements". But if you think about the category **Set** where the objects are sets and the arrows are functions, the elements of any set S are in one-to-one correspondence with the functions from any one-element set to S. That is, for any element **e** of **S**, there is exactly one function **f : 1 -> S**, where **1** is any one-element set... (cont'd) – Tom Crockett Nov 1 '12 at 23:22

8 @TahirHassan 1-element sets are themselves specializations of the more general category-theoretic notion of "terminal objects": a terminal object is any object of a category for which there is exactly one arrow from any other object to it (you can check that this is true of 1-element sets in **Set**). In category theory terminal objects are simply referred to as **1**; they are unique up to isomorphism so there is no point distinguishing them. So now we have a purely category-theoretical description of "elements of **S**" for any **S**: they are just the arrows from **1** to **S**! – Tom Crockett Nov 1 '12 at 23:26

Intuitively, I think that what the fancy math vocabulary is saying is that:

Monoid

A **monoid** is a set of objects, and a method of combining them. Well known monoids are:

- numbers you can add
- lists you can concatenate
- sets you can union

There are more complex examples also.

Further, *every* monoid has an **identity**, which is that "no-op" element that has no effect when you combine it with something else:

- 0 + 7 **==** 7 + 0 **==** 7
- [] ++ [1,2,3] **==** [1,2,3] ++ [] **==** [1,2,3]
- {} *union* {apple} **==** {apple} *union* {} **==** {apple}

Finally, a monoid must be **associative**. (you can reduce a long string of combinations anyway you want, as long as you don't change the left-to-right-order of objects) Addition is OK ((5+3)+1 == 5+(3+1)), but subtraction isn't ((5-3)-1 != 5-(3-1)).

Monad

Now, let's consider a special kind of set and a special way of combining objects.

Objects

Suppose your set contains objects of a special kind: **functions**. And these functions have an interesting signature: They don't carry numbers to numbers or strings to strings. Instead, each function carries a number to a list of numbers in a two-step process.

1. Compute 0 or more results
2. Combine those results unto a single answer somehow.

Examples:

- 1 -> [1] (just wrap the input)
- 1 -> [] (discard the input, wrap the nothingness in a list)
- 1 -> [2] (add 1 to the input, and wrap the result)
- 3 -> [4, 6] (add 1 to input, and multiply input by 2, and wrap the *multiple results*)

Combining Objects

Also, our way of combining functions is special. A simple way to combine function is *composition*: Let's take our examples above, and compose each function with itself:

- 1 -> [1] -> [[1]] (wrap the input, twice)
- 1 -> [] -> [] (discard the input, wrap the nothingness in a list, twice)
- 1 -> [2] -> [UH-OH!] (we can't "add 1" to a list!")
- 3 -> [4, 6] -> [UH-OH!] (we can't add 1 a list!)

Without getting too much into type theory, the point is that you can combine two integers to get an integer, but you can't always compose two functions and get a function of the same type. (Functions with type *a -> a* will compose, but *a-> [a]* won't.)

So, let's define a different way of combining functions. When we combine two of these functions, we don't want to "double-wrap" the results.

Here is what we do. When we want to combine two functions F and G, we follow this process (called *binding*):

1. Compute the "results" from F but don't combine them.
2. Compute the results from applying G to each of F's results separately, yielding a collection of collection of results.
3. Flatten the 2-level collection and combine all the results.

Back to our examples, let's combine (bind) a function with itself using this new way of "binding" functions:

- 1 -> [1] -> [1] (wrap the input, twice)
- 1 -> [] -> [] (discard the input, wrap the nothingness in a list, twice)
- 1 -> [2] -> [3] (add 1, then add 1 again, and wrap the result.)
- 3 -> [4,6] -> [5,8,7,12] (add 1 to input, and also multiply input by 2, keeping both results, then do it all again to both results, and then wrap the final results in a list.)

This more sophisticated way of combining functions *is* associative (following from how function composition is associative when you aren't doing the fancy wrapping stuff).

Tying it all together,

- a monad is a structure that defines a way to combine (the results of) functions,
- analogously to how a monoid is a structure that defines a way to combine objects,
- where the method of combination is associative,
- and where there is a special 'No-op' that can be combined with any *something* to result in *something* unchanged.

Notes

There are lots of ways to "wrap" results. You can make a list, or a set, or discard all but the first result while noting if there are no results, attach a sidecar of state, print a log message, etc, etc.

I've played a bit loose with the definitions in hopes of getting the essential idea across intuitively.

I've simplified things a bit by insisting that our monad operates on functions of type *a -> [a]*. In fact, monads work on functions of type *a -> m b*, but the generalization is kind of a technical detail that isn't the main insight.

edited Nov 30 '14 at 0:53 community wiki
9 revs, 3 users 91%

misterbee

19 Best explanation I've read. I finally think I'm starting to get this, after 3 years pottering with Haskell every few months. – chrisdew Oct 20 '11 at 8:46

9 This is a nice explanation of how every monad constitutes a *category* (the Kleisli category is what you're demonstrating--there is also the Eilenberg-Moore category). But due to the fact that you can't compose *any* two Kleisli arrows `a -> [b]` and `c -> [d]` (you can only do this if `b = c`), this doesn't quite describe a monoid. It's actually the flattening operation you described, rather than function composition, which is the "monoid operator". – Tom Crockett Dec 10 '11 at 19:35

4 I wish I could vote this up twice. – jwg Feb 6 '13 at 17:08

2 On the last note, it helps to remember, that a -> [a] is just a -> [] a. ([] is just type constructor too.) And so it can not only be seen as a -> m b, but [] is indeed an instance of the Monad class. – Evi1M4chine Mar 3 '13 at 17:34

2 This is the best and most grokkable explanation of monads and their mathematical background of monoids I have come across in literally weeks. This is what should be printed in every Haskell book when it comes to monads, hands down. UPVOTE! Maybe further get the piece of information, that monads are realized as parameterized typeclass instances wrapping whatever put in them in haskell, into the post. (At least that is how I understood them by now. Correct me if I am wrong. See haskell.org/haskellwiki/What_a_Monad_is_not) – sjas Dec 2 '13 at 19:20

This is an old question, but I feel there's a way to make the answer a bit more concrete with some code. At least, I'm better at Haskell than I am at category theory, so I find it easier to understand it this way :-P.

First, the extensions and libraries that we're going to use:

```
{-# LANGUAGE RankNTypes, TypeOperators #-}

import Control.Monad (join)
```

Of these, `RankNTypes` is the only one that's absolutely essential to the below. I once wrote an explanation of `RankNTypes` that some people seem to have found useful, so I'll refer to that.

Quoting Tom Crockett's excellent answer, we have:

> ## A monad is...
>
> - An endofunctor, ***T : X -> X***
> - A natural transformation, ***μ : T × T -> T***, where × means functor composition
> - A natural transformation, ***η : I -> T***, where *I* is the identity endofunctor on *X*
>
> ## ...satisfying these laws:
>
> - *μ(μ(T × T) × T)) = μ(T × μ(T × T))*
> - *μ(η(T)) = T = μ(T(η))*

How do we translate this to Haskell code? Well, let's start with the notion of a **natural transformation**:

```
-- | A natural transformations between two 'Functor' instances.  Law:
--
-- > fmap f . eta g == eta g . fmap f
--
-- Neat fact: the type system actually guarantees this law.
--
newtype f :-> g =
    Natural { eta :: forall x. f x -> g x }
```

A type of the form `f :-> g` is analogous to a function type, but instead of thinking of it as a *function* between two *types* (of kind `*`), think of it as a **morphism** between two **functors** (each of kind `* -> *`). Examples:

```
listToMaybe :: [] :-> Maybe
listToMaybe = Natural go
    where go [] = Nothing
          go (x:_) = Just x

maybeToList :: Maybe :-> []
maybeToList = Natural go
    where go Nothing = []
          go (Just x) = [x]

reverse' :: [] :-> []
reverse' = Natural reverse
```

Basically, in Haskell, natural transformations are functions from some type `f x` to another type `g x` such that the `x` type variable is "inaccessible" to the caller. So for example, `sort :: Ord a => [a] -> [a]` cannot be made into a natural transformation, because it's "picky" about which types we may instantiate for `a`. One intuitive way I often use to think of this is the following:

- A functor is a way of operating on the *content* of something without touching the *structure*.
- A natural transformation is a way of operating on the *structure* of something without touching or looking at the *content*.

Now, with that out of the way, let's tackle the clauses of the definition.

The first clause is "an endofunctor, ***T : X -> X***." Well, every `Functor` in Haskell is an endofunctor in what people call "the Hask category," whose objects are Haskell types (of kind `*`) and whose morphisms are Haskell functions. This sounds like a complicated statement, but it's actually a very trivial one. All it means is that that a `Functor f :: * -> *` gives you the means of constructing a type `f a :: *` for any `a :: *` and a function `fmap f :: f a -> f b` out of any `f :: a -> b`, and that these obey the functor laws.

Second clause: the `Identity` functor in Haskell (which comes with the Platform, so you can just import it) is defined this way:

```
newtype Identity a = Identity { runIdentity :: a }

instance Functor Identity where
    fmap f (Identity a) = Identity (f a)
```

So natural transformation ***η : I -> T*** from Tom Crockett's definition can be written this way for any `Monad` instance `t`:

```
return' :: Monad t => Identity :-> t
return' = Natural (return . runIdentity)
```

Third clause: the composition of two functors in Haskell can be defined this way (which also comes with the Platform):

```
newtype Compose f g a = Compose { getCompose :: f (g a) }

-- | The composition of two 'Functor's is also a 'Functor'.
instance (Functor f, Functor g) => Functor (Compose f g) where
    fmap f (Compose fga) = Compose (fmap (fmap f) fga)
```

So the natural transformation ***μ : T × T -> T*** from Tom Crockett's definition can be written like this:

```
join' :: Monad t => Compose t t :-> t
join' = Natural (join . getCompose)
```

The statement that this is a monoid in the category of endofunctors then means that `Compose` (partially applied to just its first two parameters) is associative, and that `Identity` is its identity element. I.e., that the following isomorphisms hold:

- `Compose f (Compose g h) ~= Compose (Compose f g) h`
- `Compose f Identity ~= f`
- `Compose Identity g ~= g`

These are very easy to prove because `Compose` and `Identity` are both defined as `newtype`, and the Haskell Reports define the semantics of `newtype` as an isomorphism between the type being defined and the type of the argument to the `newtype`'s data constructor. So for example, let's prove `Compose f Identity ~= f`:

```
Compose f Identity a
    ~= f (Identity a)                -- newtype Compose f g a = Compose (f (g a))
    ~= f a                           -- newtype Identity a = Identity a
Q.E.D.
```

edited Mar 21 '15 at 7:51 Lambda Fairy 4,444 ●2 ●20 ●40

answered May 2 '14 at 0:07 Luis Casillas 19.1k ●2 ●29 ●75

Your explanation is very clear and wonderful – Song Zhang Feb 1 '15 at 3:43

In the `Natural` newtype, I can't figure out what the `(Functor f, Functor g)` constraint is doing. Could you explain? – dfeuer Mar 20 '15 at 15:53

@dfeuer It's not really doing anything essential. – Luis Casillas Mar 20 '15 at 18:15

1 @LuisCasillas I've removed those `Functor` constraints since they don't seem necessary. If you disagree then feel free to add them back. – Lambda Fairy Mar 21 '15 at 8:03

Can you elaborate on what it means formally for the product of functors to be taken as composition? In particular, what are the projection morphisms for functor composition? My guess is that the product is only defined for a functor F against itself, F x F and only when `join` is defined. And that `join` is the projection morphism. But I'm not sure. – tksfz Apr 1 '15 at 21:54

It's quite possible that Iry had read From Monoids to Monads, a post in which Dan Piponi (sigfpe) derives monads from monoids in Haskell, with much discussion of category theory and explicit mention of "the category of endofunctors on Hask" . In any case, anyone who wonders what it means for a monad to be a monoid in the category of endofunctors might benefit from reading this derivation.

answered Sep 16 '15 at 6:58 hobbs 98.5k ●10 ●109 ●189

It's the other way round. I wrote that because I felt the need to explain Iry's comment. – sigfpe Nov 30 '15 at 22:15

1 @sigfpe darn. Well, thanks for dropping by to clear things up :) – hobbs Nov 30 '15 at 22:20