

Contents

Functors

Just What Is A Functor,
Really?

Applicatives

Monads

Conclusion

Translations

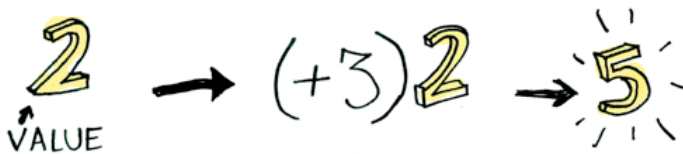
Functors, Applicatives, And Monads In Pictures

WRITTEN APRIL 17, 2013
UPDATED MAY 20, 2013

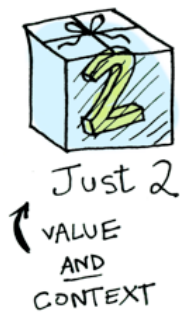
Here's a simple value:



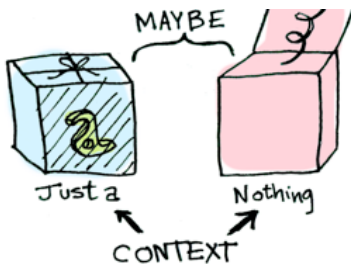
And we know how to apply a function to this value:



Simple enough. Lets extend this by saying that any value can be in a context. For now you can think of a context as a box that you can put a value in:



Now when you apply a function to this value, you'll get different results **depending on the context**. This is the idea that Functors, Applicatives, Monads, Arrows etc are all based on. The **Maybe** data type defines two related contexts:

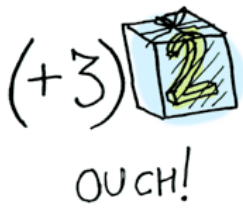


```
data Maybe a = Nothing | Just a
```

In a second we'll see how function application is different when something is a **Just a** versus a **Nothing**. First let's talk about Functors!

Functors

When a value is wrapped in a context, you can't apply a normal function to it:



This is where `fmap` comes in. `fmap` is from the street, `fmap` is hip to contexts. `fmap` knows how to apply functions to values that are wrapped in a context. For example, suppose you want to apply `(+3)` to `Just 2`. Use `fmap` :

```
> fmap (+3) (Just 2)
Just 5
```



Bam! `fmap` shows us how it's done! But how does `fmap` know how to apply the function?

Just what is a Functor, really?

Functor is a [typeclass](#). Here's the definition:

1. TO MAKE A DATA TYPE f
A FUNCTOR,

class Functor f where

$\rightarrow \text{fmap} :: (a \rightarrow b) \rightarrow f a \rightarrow f b$

2. THAT DATA TYPE
NEEDS TO DEFINE
HOW `fmap` WILL
WORK WITH IT.

A **Functor** is any data type that defines how `fmap` applies to it. Here's how `fmap` works:

$\text{fmap} :: (a \rightarrow b) \rightarrow f a \rightarrow f b$

1. `fmap` TAKES A
FUNCTION
(LIKE `(+3)`)

2. AND A
FUNCTOR
(LIKE `Just 2`)

3. AND RETURNS
A NEW FUNCTOR
(LIKE `Just 5`)

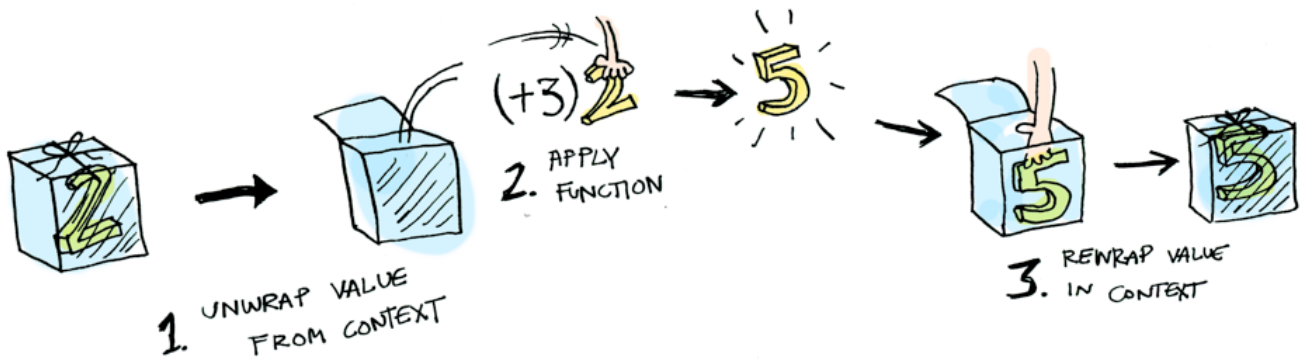
So we can do this:

```
> fmap (+3) (Just 2)
Just 5
```

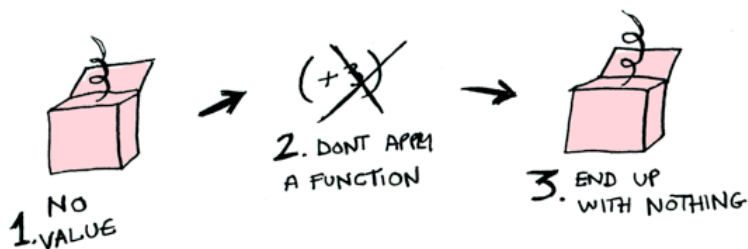
And `fmap` magically applies this function, because **Maybe** is a Functor. It specifies how `fmap` applies to **Just** s and **Nothing** s:

```
instance Functor Maybe where
  fmap func (Just val) = Just (func val)
  fmap func Nothing = Nothing
```

Here's what is happening behind the scenes when we write `fmap (+3) (Just 2)` :



So then you're like, alright `fmap`, please apply `(+3)` to a `Nothing` ?



```
> fmap (+3) Nothing
Nothing
```



Bill O'Reilly being totally ignorant about the Maybe functor

Like Morpheus in the Matrix, `fmap` knows just what to do; you start with `Nothing`, and you end up with `Nothing` ! `fmap` is zen. Now it makes sense why the `Maybe` data type exists. For example, here's how you work with a database record in a language without `Maybe` :

```
post = Post.find_by_id(1)
if post
  return post.title
else
  return nil
end
```

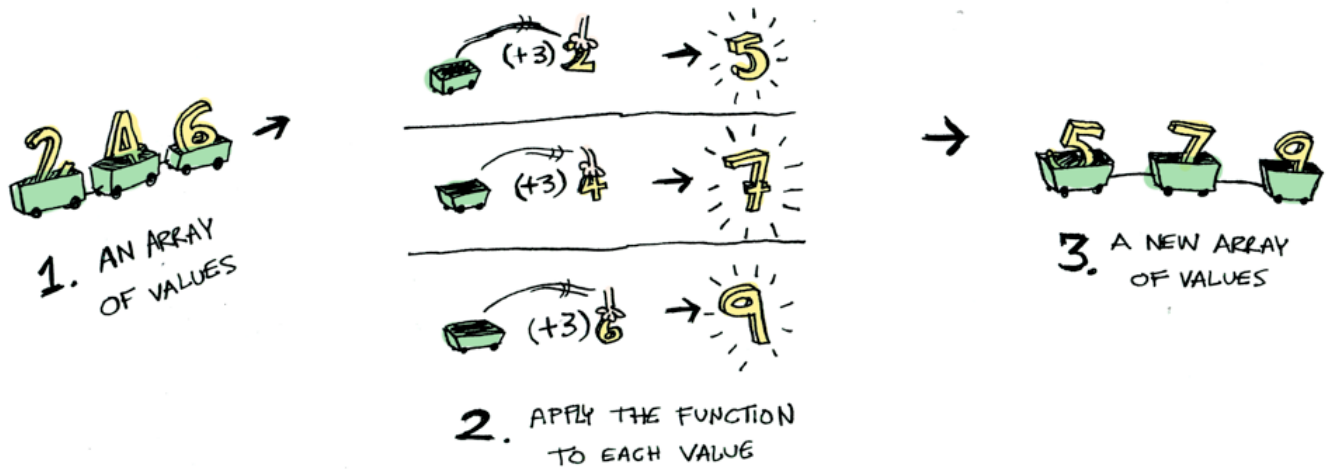
But in Haskell:

```
fmap (getPostTitle) (findPost 1)
```

If `findPost` returns a post, we will get the title with `getPostTitle` . If it returns `Nothing` , we will return `Nothing` ! Pretty neat, huh? `<$>` is the infix version of `fmap` , so you will often see this instead:

```
getPostTitle <$> (findPost 1)
```

Here's another example: what happens when you apply a function to a list?



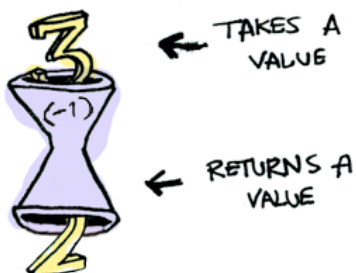
Lists are functors too! Here's the definition:

```
instance Functor [] where  
  fmap = map
```

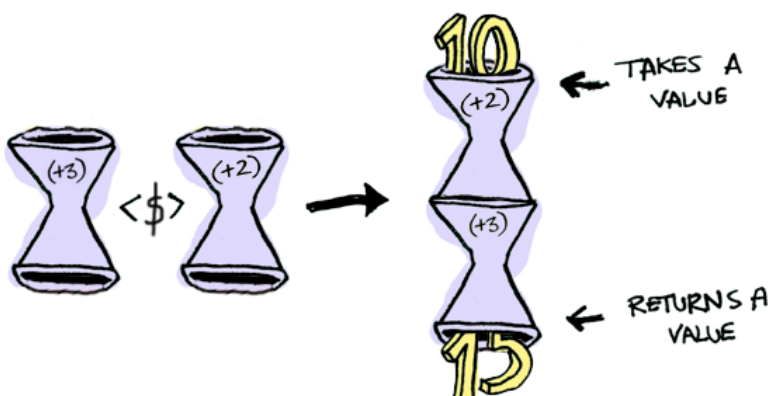
Okay, okay, one last example: what happens when you apply a function to another function?

```
fmap (+3) (+1)
```

Here's a function:



Here's a function applied to another function:



The result is just another function!

```
> import Control.Applicative
> let foo = fmap (+3) (+2)
> foo 10
15
```

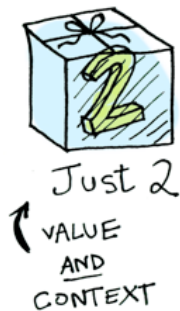
So functions are Functors too!

```
instance Functor ((->) r) where
  fmap f g = f . g
```

When you use `fmap` on a function, you're just doing function composition!

Applicatives

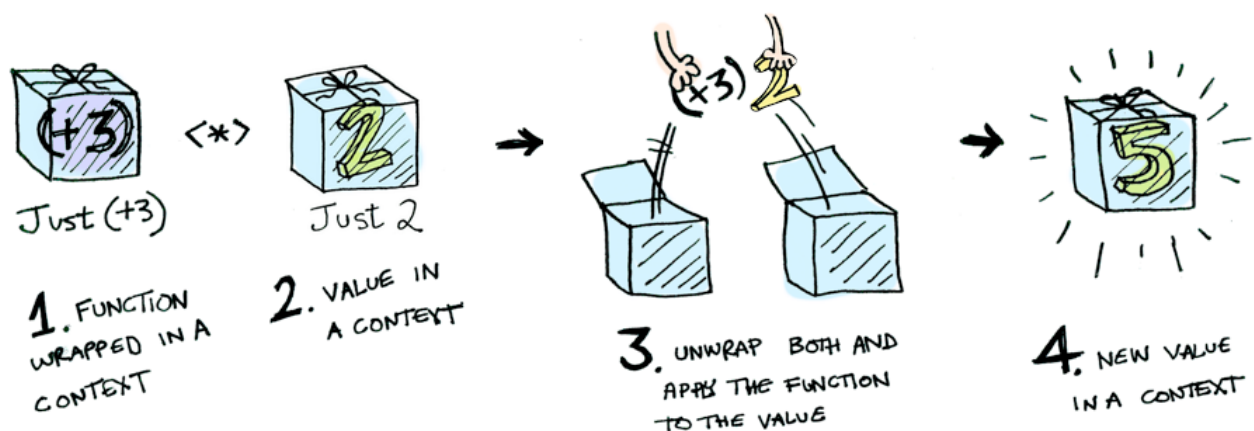
Applicatives take it to the next level. With an applicative, our values are wrapped in a context, just like Functors:



But our functions are wrapped in a context too!



Yeah. Let that sink in. Applicatives don't kid around. `Control.Applicative` defines `<*>`, which knows how to apply a function *wrapped in a context* to a value *wrapped in a context*:

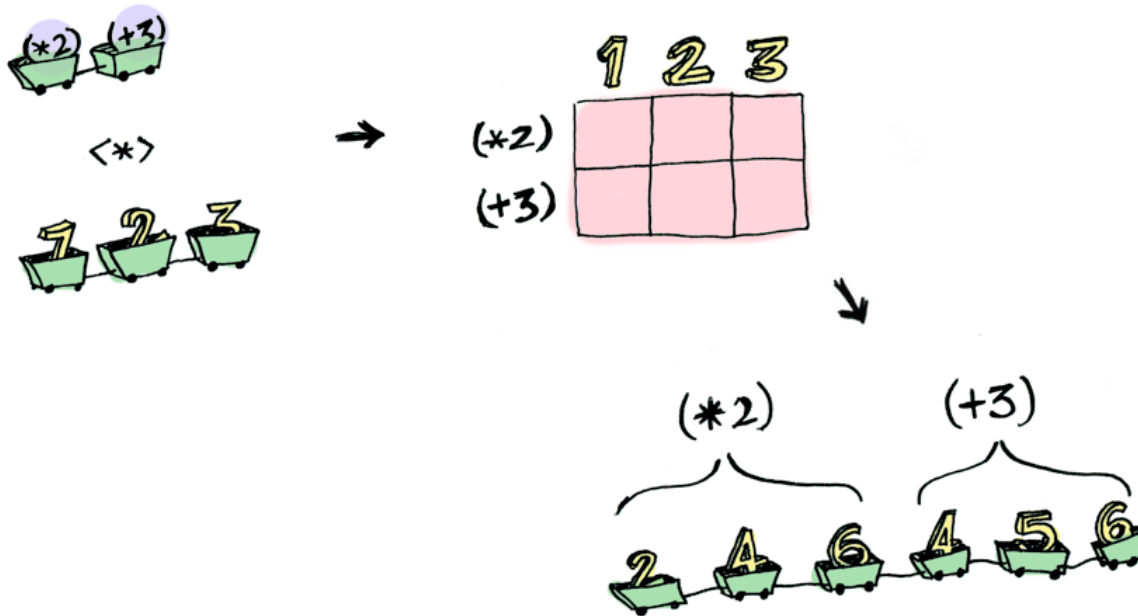


i.e:

```
Just (+3) <*> Just 2 == Just 5
```

Using `<*>` can lead to some interesting situations. For example:

```
> [(*2), (+3)] <*> [1, 2, 3]
[2, 4, 6, 4, 5, 6]
```



Here's something you can do with Applicatives that you can't do with Functors. How do you apply a function that takes two arguments to two wrapped values?

```
> (+) <$> (Just 5)
Just (+5)
> Just (+5) <$> (Just 4)
ERROR ??? WHAT DOES THIS EVEN MEAN WHY IS THE FUNCTION WRAPPED IN A JUST
```

Applicatives:

```
> (+) <$> (Just 5)
Just (+5)
> Just (+5) <*> (Just 3)
Just 8
```

Applicative pushes **Functor** aside. "Big boys can use functions with any number of arguments," it says. "Armed `<$>` and `<*>`, I can take any function that expects any number of unwrapped values. Then I pass it all wrapped values, and I get a wrapped value out! AHAHAHAHAH!"

```
> (*) <$> Just 5 <*> Just 3
Just 15
```

And hey! There's a function called `liftA2` that does the same thing:

```
> liftA2 (*) (Just 5) (Just 3)
Just 15
```

Monads

How to learn about Monads:

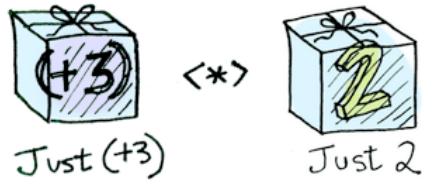
1. Get a PhD in computer science.
2. Throw it away because you don't need it for this section!

Monads add a new twist.

Functors apply a function to a wrapped value:

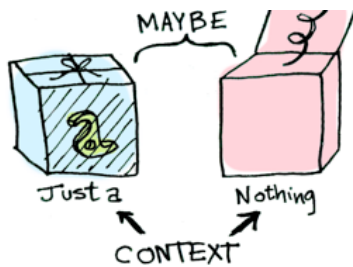


Applicatives apply a wrapped function to a wrapped value:



Monads apply a function **that returns a wrapped value** to a wrapped value. Monads have a function `>>=` (pronounced “bind”) to do this.

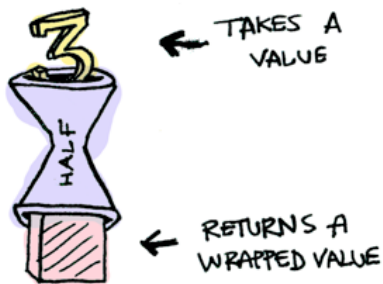
Let's see an example. Good ol' **Maybe** is a monad:



Just a monad hanging out

Suppose `half` is a function that only works on even numbers:

```
half x = if even x
         then Just (x `div` 2)
         else Nothing
```



What if we feed it a wrapped value?



We need to use `>>=` to shove our wrapped value into the function. Here's a photo of `>>=` :



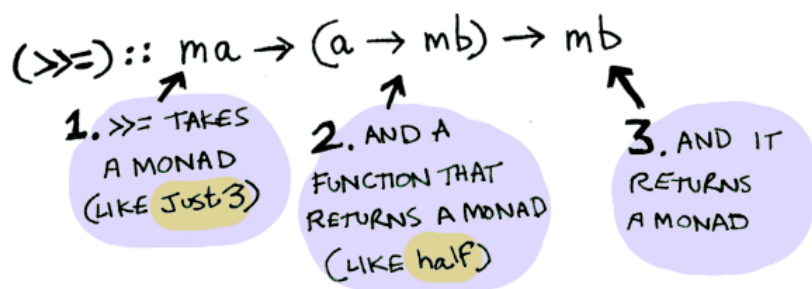
Here's how it works:

```
> Just 3 >= half
Nothing
> Just 4 >= half
Just 2
> Nothing >= half
Nothing
```

What's happening inside? **Monad** is another typeclass. Here's a partial definition:

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

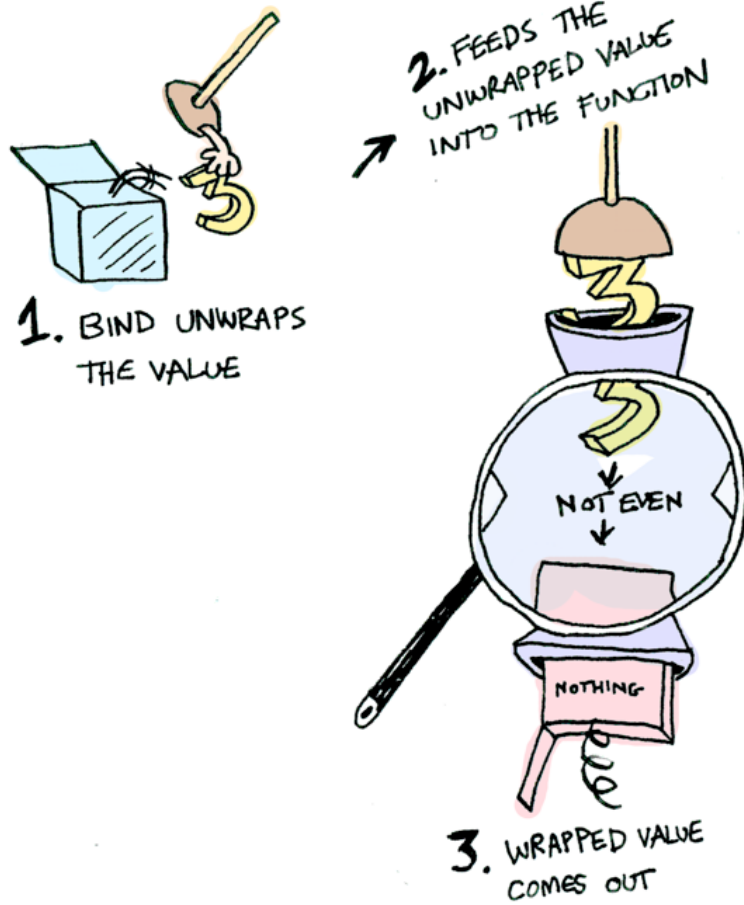
Where `>=` is:



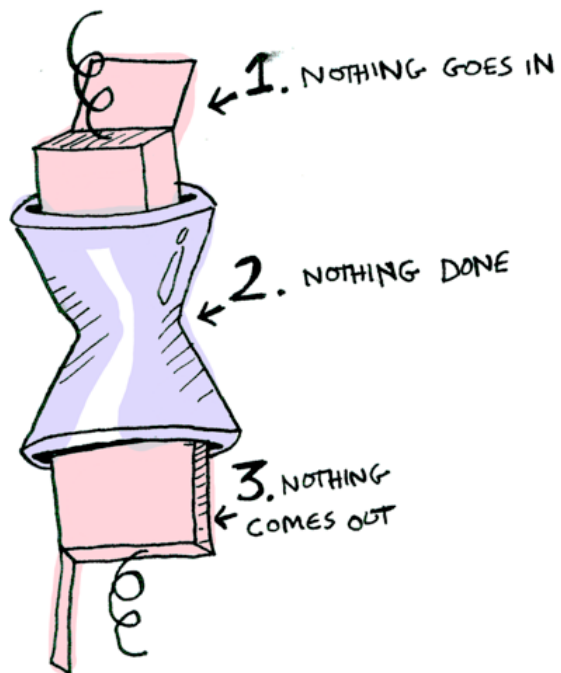
So **Maybe** is a Monad:

```
instance Monad Maybe where
  Nothing >= func = Nothing
  Just val >= func = func val
```

Here it is in action with a **Just 3** !

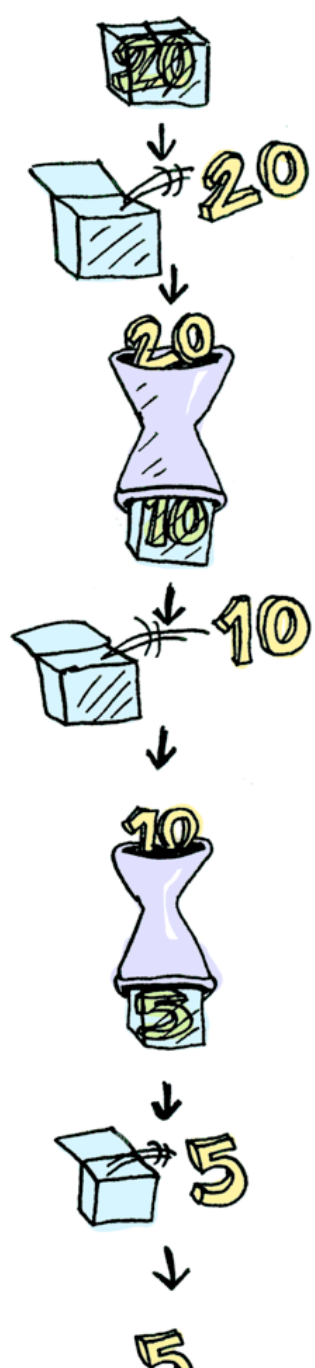


And if you pass in a **Nothing** it's even simpler:



You can also chain these calls:

```
> Just 20 >>= half >>= half >>= half
Nothing
```





Cool stuff! So now we know that `Maybe` is a `Functor`, an `Applicative`, and a `Monad`.
Now let's mosey on over to another example: the `IO` monad:



Specifically three functions. `getLine` takes no arguments and gets user input:



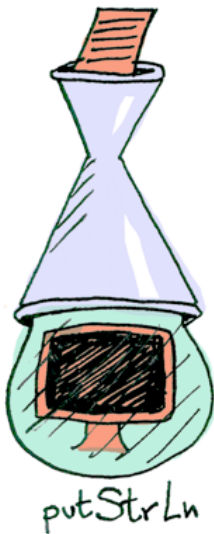
```
getLine :: IO String
```

`readFile` takes a string (a filename) and returns that file's contents:



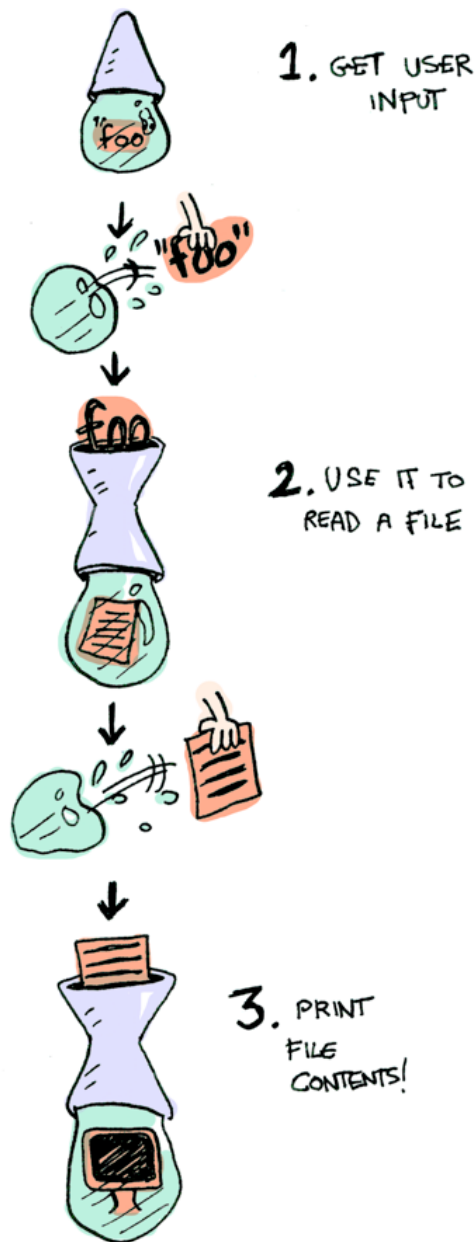
```
readFile :: FilePath -> IO String
```

`putStrLn` takes a string and prints it:



```
putStrLn :: String -> IO ()
```

All three functions take a regular value (or no value) and return a wrapped value. We can chain all of these using `>>=` !



```
getLine >>= readFile >>= putStrLn
```

Aw yeah! Front row seats to the monad show!

Haskell also provides us with some syntactical sugar for monads, called `do` notation:

```
foo = do
  filename <- getLine
  contents <- readFile filename
  putStrLn contents
```

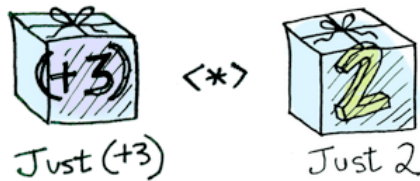
Conclusion

1. A functor is a data type that implements the `Functor` typeclass.
2. An applicative is a data type that implements the `Applicative` typeclass.
3. A monad is a data type that implements the `Monad` typeclass.
4. A `Maybe` implements all three, so it is a functor, an applicative, and a monad.

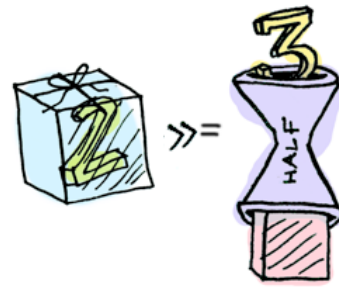
What is the difference between the three?



Functor



Applicative



Monad

- **functors:** you apply a function to a wrapped value using `fmap` or `<$>`
- **applicatives:** you apply a wrapped function to a wrapped value using `<*>` or `liftA`
- **monads:** you apply a function that returns a wrapped value, to a wrapped value using `>>=` or `liftM`

So, dear friend (I think we are friends by this point), I think we both agree that monads are easy and a SMART IDEA(tm). Now that you've wet your whistle on this guide, why not pull a Mel Gibson and grab the whole bottle. Check out LYAH's [section on Monads](#). There's a lot of things I've glossed over because Miran does a great job going in-depth with this stuff.

Translations

This post has been translated into:

- [Russian](#)
- [Japanese](#)
- [Korean](#)
- [Chinese](#)
- [French](#)
- [Python](#)
- [Swift](#)
- [Spanish](#)

If you translate this post, [send me an email](#) and I'll add it to this list!

For more monads and pictures, check out [three useful monads](#).

120 Comments [adit.io](#)

[Login](#)

[Recommend](#) 63 [Share](#)

[Sort by Best](#)



Join the discussion...



Nick • 2 years ago

wow - no donate or buy me a coffee link. I feel like I've stolen from you. Absolutely fantastic way of explaining the essence of monads.

70 ^ | v • [Reply](#) • [Share](#)



rickasaurus • 2 years ago

Great Job! Now do this for arrows! :)

53 ^ | v • [Reply](#) • [Share](#)



Ionuț G. Stan → rickasaurus • 2 years ago

I think the pictures here, although not as pretty as the ones in this blog post, explain arrows pretty well.

<http://en.wikibooks.org/wiki/H...>

9 ^ | v • [Reply](#) • [Share](#)



Adit Mod → rickasaurus • 2 years ago

It seems like the Haskell community is moving away from Arrows, so I don't know how useful that would be. I want to do one for monad transformers though.

3 ^ | v • [Reply](#) • [Share](#)



Karol Mieczysław Marcjan → Adit • 2 years ago

For one thing, Yampa and a few other FRP libraries are based on Arrows.

2 ^ | v • [Reply](#) • [Share](#)



Holandez → Adit · a year ago

Where did you get this idea?

1 ^ | v · Reply · Share ›



Nimrod · 2 years ago

Might be a good idea to say something about operator precedence / associativity. Or at least write

(*) <\$> Just 5 <*> Just 3 = Just (*5) <*> Just 3 = Just 15

11 ^ | v · Reply · Share ›



Matthew Pettis → Nimrod · 2 years ago

I'd like to second Nimrod's point here. In `_Learn You a Haskell_`, there is a section on functions as applicatives. I was trying to comprehend the example in there (mind you my copy may be out of date), but my biggest problem was that I assumed '`<*>`' had higher precedence than '`<$>`', and I couldn't work out how the example was working. After I read this and Nimrod's comment here, I applied precedence/associativity correctly, and it all worked out in my head. So I strongly recommend spending some words discussing that. But I couldn't have gotten anywhere without this tutorial, so thanks for that!

2 ^ | v · Reply · Share ›



Anonymous · 2 years ago

Simply the best article on topic I've ever read. No doubts we are friends by this point :)

I'm even a little disappointed how simple it actually is. I've suspected that there's some clever mechanism in IO and do-syntax that divides side-effect-rich stuff from lazy, immutable, easy-to-parallel semantics. But eventually there's no magic at all and now I'm wondering what all these "mutithreading on comiler-side" fantasies I've heard of were all about.

8 ^ | v · Reply · Share ›



Adit Mod → Anonymous · 2 years ago

Haha thanks!

^ | v · Reply · Share ›



exim · 2 years ago

Waiting for monad transformers pictures.

8 ^ | v · Reply · Share ›



Kristopher Micinski · 2 years ago

I've often found that people say, "why the box?" The answer is laziness, which really gets at the heart of Haskell, period. Unfortunately, I've found the hardest part about learning monads (not as true with applicative functors) is learning about all the individual monad *instances*. E.g., how `'Cont'`, `'Reader'`, or `'State'` work. Monads just give you a way to chain things together, to actually understand individual instances of monads, you have to scratch your head a little and draw out some examples.

7 ^ | v · Reply · Share ›



Justin Le → Kristopher Micinski · 2 years ago

Well...that's because bind is just a design pattern. There is nothing 'conceptually' that links `Cont`, `Reader`, `State`; just like, say, implementing mergesort and implementing hashtables are not conceptually linked. Sharing environment, passing state, and dealing with continuations are all completely different concepts (both diferent from eachother and different from monads) that have nothing to do with monad-ness at all. :) If you understand this article, you understand all there is to "know" about monads :) Like how the hardest part about learning 'for loops' or 'semicolons' is not learning about A* Pathfinding.

2 ^ | v · Reply · Share ›



swampwiz · 2 years ago

I am an unemployed "seasoned" (early Gen-X) C++ programmer, and I have absolutely no idea what is going on here. If the youngsters can figure this out, and this is actually useful to developing applications, I guess I deserve to be unemployed.

5 ^ | v · Reply · Share ›



Adit Mod → swampwiz · 2 years ago

You will find it easier to understand if you learn some haskell first: learnyouahaskell.com :)

6 ^ | v · Reply · Share ›



Jono → swampwiz · 2 years ago

swampwiz I feel your pain. I'm a *baby*boomer* C/C++ programmer... I feel a little more comfortable with this stuff having learned me some LISP and some Scheme and have even written a couple of little Schemes myself, one in C and one in C++. I heartily recommend that exercise! John McCarthy published the first paper on LISP in 1958, btw. I'd like to learn me some Haskell but ...^#@#&. When?

2 ^ | v · Reply · Share ›



Sébastien Lorber · 2 years ago


Could you do the same with monad transformers please? :)



5 ^ | v · Reply · Share ›




Bobby_Joe_Moe_Go_Bro_Toe · 2 years ago

None better. So much clarity comes with the proper visualization, and this is an example of using it to cut through a notoriously difficult



 None better. So much clarity comes with the proper visualization, and this is an example of using it to cut through a notoriously difficult subject like a hot knife through butter. This is how teaching needs to be done.




4  |  • Reply • Share ›

 **Igor Goncharenko** • 2 years ago



Looks really great!



By the way, what tool do you use to write such pretty illustrations/pictures ?

4  |  • Reply • Share ›


 **Adit**   Igor Goncharenko • 2 years ago


Thank you! I did them with a fountain pen and then colored them in Pixelmator (Gimp would work just fine too).

12  |  • Reply • Share ›



 **Tyler**  Adit • 2 years ago

pixelmator is the best photo app ever

3  |  • Reply • Share ›



 **zlitan** • a year ago


Okay - thank you very much :) This was really great writing, pics and explanation.

3  |  • Reply • Share ›



 **Chris Lynch** • a year ago

Simply outstanding. I hope a book is in order!

2  |  • Reply • Share ›

 **Ivsti** • a year ago

you are awesome. period.

2  |  • Reply • Share ›

 **EunPyoung Kim** • 2 years ago



thanks share the good post with pictures.



good for freshman :)

I tried to translate that into Korean.

<https://github.com/netpyoung/n...>



(my [github.io](https://github.com/netpyoung/n...) is underconstructing... is done, i will post this to [github.io](https://github.com/netpyoung/n...))

2  |  • Reply • Share ›

 **EunPyoung Kim**  EunPyoung Kim • 2 years ago

updated in personal [github.io](https://github.com/netpyoung/n...)


<http://netpyoung.github.io/ext...>

 |  • Reply • Share ›



 **bsunter** • 4 months ago



Really awesome. Thank you so much!

1  |  • Reply • Share ›



 **Roger Qiu** • a year ago

Awesome, this is the first time I'm starting to grok the concept of monads. Just wanted to clarify, the whole "wrapping box" is basically there to encapsulate non-determinism/side-effects in an otherwise pure functional language? That way you can pass that "uncertainty" around the place, while still looking pure with no-side effects. The uncertainty becomes certain when they get pushed out to IO right?

1  |  • Reply • Share ›

 **Bruce Richardson**  Roger Qiu • 7 months ago

No, the box is not about hiding side effects. *Some* monads are used to hide side effects but that is not a characteristic. Think of it more as separation of concerns. Imagine you have a collection of integers. Let's call it "ints". It doesn't matter whether the collection is a list or a set or a map or a Maybe, the Haskell code `fmap (* 2) ints` will double every integer inside the collection, without altering the container. So if ints is a list, order will be preserved, while if it is a set, uniqueness will still be guaranteed. If ints is a Maybe then if there was Nothing there, there will still be Nothing there, otherwise it will contain one doubled value. Why does this involve separation of concerns? Because if ints is a list but now you decide you want it to be a set (that is, you no longer care about order but you want unique values), the code which manipulates the values inside the collection does not change. Let's look at this with modulo arithmetic. If ints is a list, then `fmap (`mod` 2) ints` would change the list `[1,2,3,4]` into `[1,0,1,0]`. If I decide I want to switch to using sets, then the same code will change the set `[1,2,3,4]` into the set `[0,1]`.

 |  • Reply • Share ›

 **Bruce Richardson**  Bruce Richardson • 7 months ago

It is a common misunderstanding that Monads hide side effects, but this is because two of the most famous monads (IO

and ST) are specially designed to do this. They use the general property of Monads that functions and values can be wrapped inside them despite not having been designed for the specific context represented by the monad and that this context is preserved all the way through. What they add to the Monad concept is that some non-deterministic values cannot be extracted from them. In the case of the ST monad, no mutable values can be extracted. In the case of IO, nothing at all can come out. But lists, sets, maybes, eithers and all the regular monads can wrap anything and anything wrapped in them can be extracted from them again.

^ | v • Reply • Share ›



Roger Qiu → Bruce Richardson • 7 months ago

Thanks for the clarification. In the time since my first comment. I had a realisation that monads contain "context", whatever that context might be. But all the different monad patterns implement different kinds of context, or invariants of the contained data. What was interesting was that you said that in the IO monad, that nothing can come out. What do you mean precisely? That once I have created an IO monad, I can no longer unwrap in the code? What actually unwraps it then? The main function? This makes me think that the IO monad seems very similar to linear typed world/IO value that is passed around in other languages like Mercury.

^ | v • Reply • Share ›



Bruce Richardson → Roger Qiu • 7 months ago

Haskell is a strongly typed language. All IO functions return IO types; getChar returns "IO Char", getLine returns "IO Line". The Haskell compiler will only allow functions which return IO types to execute within the main function (which is itself an IO function). So pure functions cannot call IO functions or be passed IO types nor extract values from IO monads. However, IO functions *can* receive those values and extract them (only within the Main context, because that is the only place they can be called). IO functions can extract non-deterministic data and then pass it as input to pure functions. So you either have to define and call all of your code inside the Main context - in which case you might as well be using PHP - or (the recommended way) you write most of your code as pure functions, have the IO functions manage the non-deterministic data and call the pure functions to manipulate it.

Done properly, this means the vast majority of your code is provably correct and easily testable, with the problems of managing unpredictable, mutable state confined to one small

see more

1 ^ | v • Reply • Share ›



题叶 • 2 years ago

Great guide and great pictures!
Then why is this wrapper necessary?

1 ^ | v • Reply • Share ›



Justin Le → 题叶 • 2 years ago

It is not that wrappers are necessary; it is more that sometimes you have things that are wrapped. For example...a list. You have a bunch of numbers wrapped inside a list object. Or a Maybe, you have things wrapped inside a "Is there something or is there not?" context. I'm sure in your programming you have run into values that happen to be inside objects, or contexts. It's not as if we purposefully wrap things; sometimes we just have to deal with values inside containers/objects...because that's how we receive them.

2 ^ | v • Reply • Share ›



Adit **Mod** → 题叶 • 2 years ago

Which wrapper?

^ | v • Reply • Share ›



题叶 → Adit • 2 years ago

Did I misused English words here? >_<

It is said in the article that "Functors apply a function to a wrapped value", but why Haskell wraps value? Is there a reason that Haskell has to handle data like this?

--- Updated:

I tried to translate that into Chinese. Somehow I got clearer about your context.

<http://jiyinyiyong.github.io/m...>

So context might be the structure of data(like list), or more complicated.

2 ^ | v • Reply • Share ›



Adit **Mod** → 题叶 • 2 years ago

I updated the post. Maybe it will make more sense now.

^ | v • Reply • Share ›



题叶 · Adit · 2 years ago

Is there something like diff I may refer to? This article just looks the same as I read last night.

2 ^ | v · Reply · Share ›



Guest · 2 years ago

awesome :)

1 ^ | v · Reply · Share ›



Ertugrul Söylemez · 2 years ago

I'm loving this! Well done. =)

1 ^ | v · Reply · Share ›



shellington · 4 days ago

Absolute gold.

Thank you so much for writing this, it cleared a lot up.

^ | v · Reply · Share ›



Denis Kolodin · 20 days ago

Great! The best about monads.

^ | v · Reply · Share ›



Tab Atkins Jr. · 21 days ago

The explanation of bind is slightly off. The way you describe it, it sounds like bind is just an fmap that doesn't re-wrap its contents, letting the return value of the function stand on its own. That's misleading, because it actually *does* rewrap its contents, then smushes the wrappers together (or figures out how to combine them early, and then just wraps it once in the combined wrapper).

Without knowing this, the operation of the List monad, for example, is impossible to understand, let alone more complicated ones.

(I criticize out of love - I'm getting more and more pissed off at terrible Haskell tutorials that teach a broken understanding of monads and such; you produce some of the best tutorials I've found, and I want them to be as good as possible. ^_^)

^ | v · Reply · Share ›



Ariel Rodriguez · 22 days ago

Loved this... I was browsig js articles and end up here... I dont even use haskell but this was fun☐

^ | v · Reply · Share ›



Logan Collins · 2 months ago

Blessed be this above all other Monad tutorials. (In part because it explains Functors and Applicatives so well, also.)

^ | v · Reply · Share ›



marcosero · 2 months ago

This is the best explanations of functors, applicative and monads I've ever seen.

^ | v · Reply · Share ›



Binshuo Hu · 4 months ago

The best illustration I have read about monads.

For that, I guess I'll buy your book this Fall.

^ | v · Reply · Share ›



Amir Razmjou · 4 months ago

Great post. Wish you luck and more posts coming on. Thanks

^ | v · Reply · Share ›



Gilles Major · 4 months ago

Brilliant

^ | v · Reply · Share ›



Alessandro · 5 months ago

I've spent the day on my class' slides trying to figure out Monads, they I found this. This article is exactly what I needed, thank you so much!

^ | v · Reply · Share ›

Load more comments

ALSO ON ADIT.IO

[Making A Website With Haskell](#)

29 comments · 2 years ago

[How I Made My Ruby Project 10x Faster](#)

11 comments · 2 years ago

WHAT'S THIS?

nacengineer — Could you please post the source as a gist? Your walk through is hard to follow when you start decomposing files into modules as you kind of gloss over what text is going into ...

Locks, Actors, And STM In Pictures

16 comments • 2 years ago

Adit — 1. Ah, what I meant to say was, actors are never shared between multiple threads. It's fixed in the post now. 2. Good point, I actually ran into this yesterday! 3. I <3 futures but I ...

pangloss — You can further simplify the method redefinition - using `send` is unnecessary. Also, it is a good idea to include the line number so that your stack traces are friendlier. Here is the ...

The Dining Philosophers Problem With Ron Swanson

10 comments • 2 years ago

Mark Paulson — Excellent explanation, but... wouldn't chopsticks make way more sense?

 [Subscribe](#)

 [Add Disqus to your site](#)

 [Privacy](#)

DISQUS