UNWRAPPING MONADS & FRIENDS

Shining Light on Functional Programming's Scariest Concepts

KYLE SIMPSON

Unwrapping Monads & Friends

Shining Light on Functional Programming's Scariest Concepts

Kyle Simpson

This book is available at https://leanpub.com/monads-and-friends

This version was published on 2025-04-10

Tweet This Book!

Please help Kyle Simpson by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I'm not scared anymore! I'm finally going to learn what monads are all about, by reading "Unwrapping Monads + Friends" from @getifyX.
https://leanpub.com/monads-and-friends

The suggested hashtag for this book is #MonadsAndFriends.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#MonadsAndFriends

Contents

Intro

Yeah, I know. *Another* monad tutorial. It's a rite of passage in the functional programming world—every developer eventually writes one.

But here's the thing: despite the countless explanations already floating around out there, I never quite found the one that made it all click for me in a way that was both practical and conceptually grounded. So I wrote the one I wish I'd had.

This guide isn't trying to be comprehensive, academic, or authoritative in a formal sense. It's not about wading through dense category theory or mimicking Haskell idioms in JavaScript. Instead, it's a conversational exploration meant to demystify monads and their related concepts—things like functors, applicatives, and foldables—by connecting abstract ideas to actual working code, step by step.

We'll work through identity monads, maybe monads, IO, state, and more, but always through the lens of: what does this mean? Why should you care? How can this help you write more predictable, composable, and testable code? And if it ever feels like we're slipping into unnecessary abstraction, don't worry—I'll pull us back to Earth and back to JavaScript.

You don't need to be a math wizard or already fluent in FP to get something out of this. If you've ever written a map or filter in JS and thought "this feels… different," then you're already on the path. We'll fill in some of the conceptual gaps, build up your intuition, and hopefully—eventually—you'll have that "Ohhh… that's what a monad is" moment.

Throughout, I'll reference tools and libraries (like Monio) that can help you experiment and apply these ideas. But this guide isn't about any particular tool—it's about seeing the patterns that underlie them, and learning to reason about your code in a more structured way.

So if you're curious, confused, or cautiously optimistic about FP and monads, you're in the right place. Let's unwrap these ideas—gently, one layer at a time.

Note

The contents of this ebook are available to read for free online, as the "Expansive Intro to Monads" guide.

Chapter 2: What is a monad?

Monad is a funny word, let's admit it. It's a mathematical term more than a programming term. It's not even close to what most of us would reach for if we were coming up with a name for a powerful thing we hoped everyone around us would adopt!

Before we get to a more technical definition, let me try to ease you in. It may end up seeming a bit anti-climatic once you've read the next several sections of this guide, because you may be expecting it to be a big, complex topic. And in some ways it is. You may spend months or years revisiting *monads* (and related topics) and building a deeper understanding. I certainly am still learning them, several years on.

But at the outset, I hope you feel somewhat comfortable with the basics of this topic in *just a few more minutes* of reading here. I hope your reaction is, "Oh, is that all a monad is?!"

The formalism, the terminology, the mathematical notation, it can really start to overblow the concepts to an intimidating level. I stayed away from *monads* for a long time. But now I'm hooked. And I hope you get there, too.

What Is Monad?

Here's the simplest way I know how to describe what a monad is without getting into code. A monad isn't actually a specific *thing* so much as it is a *pattern*. In our programs, we can use *things* that behave according to this pattern (called "Monad", capitalized). These *things* in our programs are often referred to as "monads" (lowercased), but that's a bit informal. Instead, it might be more appropriate to think of those *things* in your program as instances of a *monad type*, kind of like the number `42` is a specific instance (value) of the number type.

The point of the monad pattern is to describe some behaviors we can expect with these instances when we interact with them. It's to give them a predictability, much like we know that the number `39` and the number `3` can be added to make the number `42`.

But it goes beyond that. Monad can perhaps more accurately be described as a pattern for a group of other patterns, like a higher-level type or a meta-type.

Tip: A simplified way of illustrating such a higher-level type / meta-type concept: in JS, several different *types* of values (numbers, strings, booleans, etc) all are described as "primitive" values (aka, not the "object" value type). Each individual type has its own respective behaviors, but all primitive values (of any primitive type) also share a common set of behavior; specifically, all JS primitive values held by-value and assigned/passed by-value-copy – as opposed to object value types being held by-reference and assigned/passed by-reference-copy.

So *Monads* (plural) would refer collectively to a variety of different *types* of monads – which, again, we can define instances of in our programs – each of which has their own unique individual pattern (behavior, etc). But all of these different Monads – aka, "monad types", "monad subtypes", "monad kinds", or however works best for your brain – *also* conform to the core Monad pattern (with its specific behaviors).

Formally, *Monad* is a part (Type) of a broad mathematical concept called "Category Theory". You could briefly and incompletely describe Category Theory as a way to categorize/group ideas based on how they behave with respect to composition and transformation, including as you mix them with each other.

The Monad type, as it appears in programming, is a way to represent a value or operation that associates the required specific behaviors with/around that (underlying) value/operation. These additional behaviors augment (i.e., improve!) the original value/operation with some "guarantees" about how it will interact predictably with other monad-represented values/operations in the program.

Your head may already be swimming with the abstractiveness of all that. Take a few minutes to let it sink in, then let's move on to illustrating them with some JS code.

Simplest JS Illustration

What's the most stripped-down way we could implement the Monad type in JS? How about this:

```js
function Identity(v) {
  return { val: v };
}

function chain(m,fn) {
  return fn(m.val);
}
```

That's it, that's a monad at its most basic. In particular, it's the "Identity" monad, which means that it will merely hold onto a value, and let you use that value, untouched, whenever you want to.

```
const myAge = Identity(41);   // { val: 41 }
```

We put a value inside an object container here only so we could recognize the value as *being monadic* (behaving according the Monad type). This “container”ness is one convenient way of implementing a monad, but it’s not actually required.

The `chain(..)` function provides a minimum basic capability to interact with our monad instance. For example, imagine we wanted to take the monadic representation of `41` and produce another monad instance where `41` had been incremented to `42`?

```
const myAge = Identity(41);   // { val: 41 }

const myNextAge = chain( myAge, v => Identity(v + 1) );   // { val: 42 }
```

We have two distinct, concrete values (held in `myAge` and `myNextAge`), both of which are instances of the “Identity” monad (as represented by this `Identity(..)` function and its `chain(..)` function). Again, people often call these instances “monads” (plural), but it’s better to think of them as instances of a single type of Monad, “Identity”.

It’s important to note that even though I use the function names `Identity` and `chain` here, those are simply just artistic choices. There’s nothing explicitly required by the concept of *Monad* in terms of what we name these things. But like any good programming discipline, if we use names that others have regularly chosen, it helps create a familiarity that improves our communications.

That `chain(..)` function looks pretty basic, but it’s really important (whatever it’s called). We’ll dig more into it much more in a bit.

But for now, I’m sure that code snippet seems pretty underwhelming to most readers. Why not just stick with `41` and `42` instead of `{ val: 41 }` and `{ val: 42 }`? The WHY of monads is likely not at all apparent yet. You’ll have to hang with me for a bit to start to uncover the WHY.

Hopefully I’ve at least shown you that down at the very core, a monad is not a mystical or complex *thing*.

Perhaps you just had that “Oh, is that it!?” moment.

Building Up Monads

Monads have somewhat (in)famously been described with a variety of silly-sounding metaphors, like “burritos”. Others call monads “wrappers” or “boxes”, or “data structures" or… the truth is, all these ways of describing a monad are partial descriptions. It's like looking at a Rubik's Cube. You can look at one face of the cube, then turn it around and look at a different face, and get more of the whole thing.

A complete understanding requires being familiar with all sides. But complete understanding is not a single atomic event. It's often built up by lots of smaller bits of understanding, like looking at each face of the cube one at a time.

For now, I just want you to focus on the idea that you could take a value like `42`, or an operation like `console.log("Hello, friend!")`, and attach/associate additional behaviors to them which will give them super powers. That's what the *Monad* type/pattern will do.

Here's another possible way of expressing monad instances in JS, using capabilities provided by the **Monio** library:

```
const myAge = Just(41);
```

Monio Reference: `Just`

The above code shows a function called `Just(..)`, which is pretty similar to the `Identity(..)` function shown previously. It acts as a constructor (aka, "unit") of the `Just` monad.

And also…

```
const printGreeting = IO(() => console.log("Hello, friend!"));
```

Here we see another **Monio** function called `IO(..)`, which acts as a constructor for the `IO` monad (which holds functions).

Thinking of our sketch in the previous section, you could sort of think of `myAge` as `{ val: 41 }` and `printGreeting` as `{ val: () => console.log("Hello, friend!") }`. **Monio**'s representation is actually a bit more sophisticated than just an object like that. But under the covers, it's not *that far* different.

I'm going to use **Monio** throughout the rest of the guide, so that we don't have to keep inventing all our own monad implementations. The convenient affordances are nice to use, and easier to illustrate with.

Keep in mind, however, that under all the trappings, we could be doing something as straight-forward as defining an object like `{ val: 41 }`.

Digging Into Map

Instances of `Just(..)` as shown above come with some methods on them, namely `chain(..)` (like we saw earlier) and also `map(..)`, which we'll look at now.

Consider the notion of an array's `map(..)` method. Its job is to apply a mapping (value translation) operation against all the contents of the associated array.

```
[ 1, 2, 3 ].map(v => v * 2);    // [ 2, 4, 6 ]
```

Note: the technical term for this capability is Functor. In fact, all monads are Functors, but don't worry too much about that term for now. Just file in the back of your head.

We started with one array (`[ 1, 2, 3 ]`) and produced a new distinct array (`[ 2, 4, 6 ]`) by mapping each element in the original array to a new value that was doubled.

This mapping on arrays of course works even if our array has a single element, right?

```
[ 41 ].map(v => v + 1);    // [ 42 ]
```

An extremely important detail there, that's easy to miss, is that the `map(..)` function didn't just give us `42` (a number) but gave us `[ 42 ]` (array holding a number). Why? Because `map(..)`'s job is to produce a new instance of the same type of "container" it was invoked against. In other words, if you use array's `map(..)`, you're going to always get back an array.

But what if our "container" is a monad instance, and what if there's only one underlying value, like `41` in it? Since the monad is also a functor (able to be "mapped"), we should still expect the same kind of outcome, right?

```
const myAge = Just(41);

const myNextAge = myAge.map(v => v + 1);    // Just(42)
```

Hopefully it makes intuitive sense here that `myNextAge` should be another `Just` instance, representing the underlying number `42`.

Recall this bare-bones example from the previous section?

```
// assumed: function Identity(val) { .. }
// assumed: myAge ==> { val: 41 }

const myNextAge = chain( myAge, v => Identity(v + 1) );   // { val: 42 }
```

Substituting **Monio**'s implementation, that looks like:

```
const myNextAge = myAge.chain(v => Just(v + 1));
```

So what's the relationship here between the `map(..)` and `chain(..)`? Let's line the operations up next to each other, to see *it*:

```
myAge.map(    v =>       v + 1  );     // Just(42)
myAge.chain( v => Just(v + 1) );     // Just(42)
```

Now do you see *it*? `map(..)` assumes that its returned value needs to be automatically "wrapped up" in an instance of the "container", whereas `chain(..)` expects the return value to already be "wrapped up" in the right kind of "container".

The `map(..)` function doesn't at all have to be named that to satisfy the functor'ness of the monad instance. In fact, you don't even strictly *need* a `map(..)` function at all, if you have `chain(..)`, because `map(..)` can be implemented with `chain(..)`:

```
function JustMap(m,fn) { return m.chain(v => Just(fn(v))); }

fortyOne.map(     v => v + 1);    // Just(42)
JustMap(fortyOne,v => v + 1);    // Just(42)
```

Having `map(..)` (or whatever it's called) available is a convenience over using just the `chain(..)` by itself; but it's not strictly required.

Monadic Chain

`chain(..)` sometimes goes by other names (in other libraries or languages), like `flatMap(..)` or `bind(..)`. In **Monio**'s monads, all three methods names are aliased to each other, so pick whichever one you prefer.

The name `flatMap(..)` can help reinforce the relationship between it and `map(..)`.

```
Just(41).map(     v => Just(v + 1) );    // Just(Just(42)) -- oops!?

Just(41).flatMap( v => Just(v + 1) );    // Just(42) -- phew!
```

If we return a `Just` monad instance from `map(..)`, it still wraps that in another `Just`, so we end up with nesting. That is perfectly valid and sometimes desired, but often not. If we return the same form of value from `flatMap(..)` (again, aka `chain(..)`), there's no nesting. Essentially, the `flatMap(..)` flattens out the nesting by one level!

The `chain(..)` method is intended for the provided function to return a monad of the same kind (`Just`, `Maybe`, etc) as the one the method was invoked on. However, **Monio** does not generally perform explicit type enforcement, so there's nothing that strictly prevents such crossing of monad kinds (e.g., between `Just` and `Maybe`). It's up to the developer to follow (or not) the implied type characteristics of these mechanisms.

I've asserted `chain(..)` (or whatever we call it!) is pretty central to something being monadic. Yet even as simple as `chain` looks to implement (see earlier), it works in such a specific way that it provides some very important guarantees about how one monad instance can interact with another monad instance. Such interactions and transformations are critical to building up a program of monads without chaos.

Another side of the **Monad** Rubik's Cube is these guarantees; they're ensured by a set of "laws" that all conforming monad implementations must satisfy:

1. Left Identity
2. Right Identity
3. Associativity

The formality and mathematical importance of these laws is not super important to immerse in right now. But to illustrate them very simply with our trivial identity monad `Just` from **Monio**:

```
// helpers:
const inc = v => Just(v + 1);
const double = v => Just(v * 2);

// (1) "left identity" law
Just(41).chain(inc);                        // Just(42)

// (2) "right identity" law
Just(42).chain(Just);                       // Just(42)

// (3) "associativity" law
Just(20).chain(inc).chain(double);          // Just(42)
Just(20).chain(v => inc(v).chain(double));  // Just(42)
```

Here I used **Monio**'s `chain(..)` method; that's again merely for convenient illustration. The monad laws are stated in terms of a *chain* operation, regardless of what an implementation chooses to call it.

Back To The Core Of Monad

Boiling this all down: the *Monad* type only strictly requires two things:

1. a function (of any name) to construct an "instance" of the type (the unit constructor)
2. a function (of any name) to properly perform the "chain" operation, as shown in the 3 laws

Everything else you see in the code snippets in this guide, such as wrapper monad instances, specific method names, "friends of monads" behaviors, etc – that's all convenient affordance provided specifically by **Monio**.

But from that narrow perspective, a monad doesn't have to be a "container" (like a wrapping object or class instance) and there doesn't even have to be a concrete "value" (like `42`) involved. While a "container wrapping a value" is one potentially helpful side of the Rubik's Cube to look at, it's not *all* that a monad is or can be. Don't get too *wrapped up* in that way of thinking!

*Wrap*ping up!

We've now scratched the surface of monads (and several *friends*). That's by no means a complete exploration of the topic, but I hope you're starting to feel they're a little less mysterious or intimidating.

A monad is a narrow set of behavior (required by "laws") you associate with a value or operation. Category Theory yields other adjacent/related behaviors, such as Foldable and Concatable, that can augment the capabilities of this representation.

This set of behavior improves coordination/interoperation between other monad-and-friends-compliant values, such that results are more predictable. The behaviors also offer many opportunities to abstract (shift into the behavioral-definitions) certain logic that usually clutters up our imperative code, such as null'ish checks.

Monads certainly don't fix all the problems we may encounter in our code, but I think there's plenty of intriguing power to unlock by exploring them further. I hope this guide inspires you to keep digging, and perhaps in your explorations, you'll find the Monio library helpful.