Chapter 09: Monadic Onions

Pointy Functor Factory

Before we go any further, I have a confession to make: I haven't been fully honest about that `of` method we've placed on each of our types. Turns out, it is not there to avoid the `new` keyword, but rather to place values in what's called a *default minimal context*. Yes, `of` does not actually take the place of a constructor - it is part of an important interface we call *Pointed*.

> A *pointed functor* is a functor with an `of` method

What's important here is the ability to drop any value in our type and start mapping away.

```
IO.of('tetris').map(concat(' master'));
// IO('tetris master')

Maybe.of(1336).map(add(1));
// Maybe(1337)

Task.of([{ id: 2 }, { id: 3 }]).map(map(prop('id')));
// Task([2,3])

Either.of('The past, present and future walk into a bar...').map(concat('it was ten
// Right('The past, present and future walk into a bar...it was tense.')
```

If you recall, `IO` and `Task`'s constructors expect a function as their argument, but `Maybe` and `Either` do not. The motivation for this interface is a common, consistent way to place a value into our functor without the complexities and specific demands of constructors. The term "default minimal context" lacks precision, yet captures the idea well: we'd like to lift any value in our type and `map` away per usual with the expected behaviour of whichever functor.

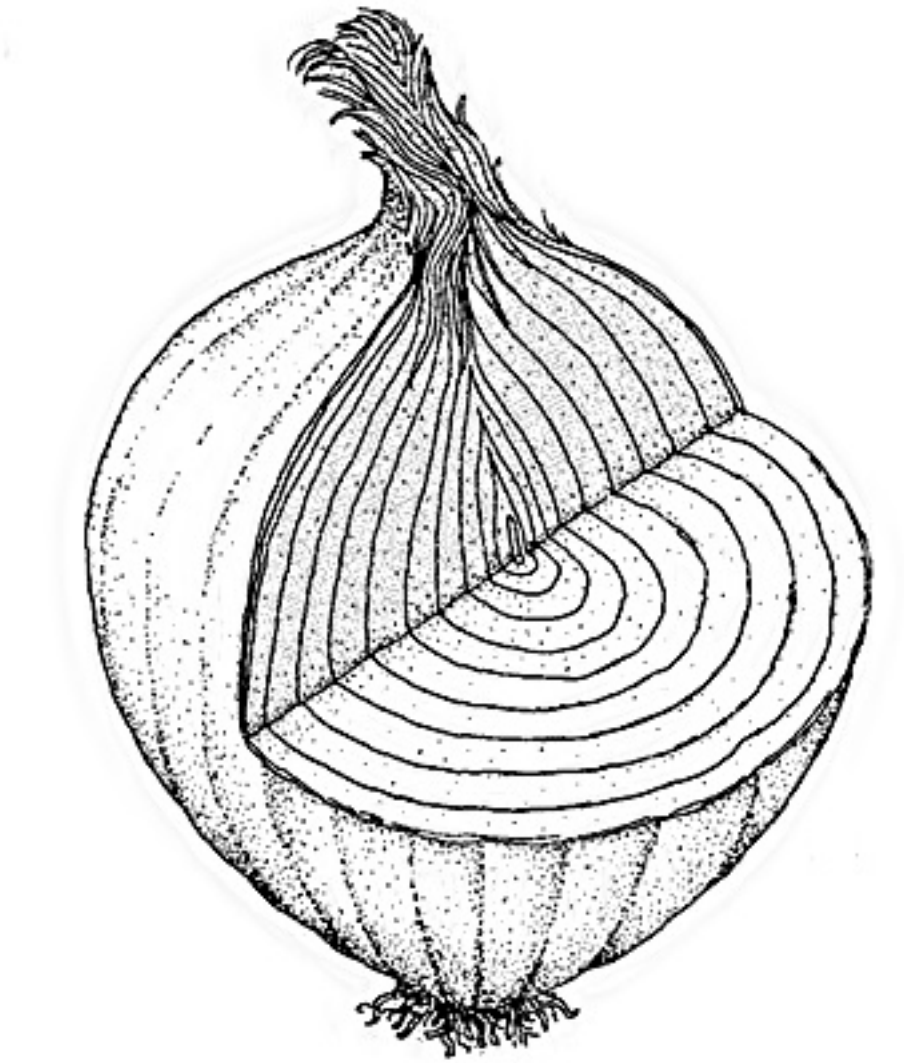
One important correction I must make at this point, pun intended, is that `Left.of` doesn't make any sense. Each functor must have one way to place a value inside it and with `Either`, that's `new Right(x)`. We define `of` using `Right` because if our type *can* `map`, it *should* `map`. Looking at the examples above, we should have an intuition about how `of` will usually work and `Left` breaks that mold.

You may have heard of functions such as `pure`, `point`, `unit`, and `return`. These are various monikers for our `of` method, international function of mystery. `of` will become important when we start using monads because, as we will see, it's our responsibility to place

values back into the type manually.

To avoid the `new` keyword, there are several standard JavaScript tricks or libraries so let's use them and use `of` like a responsible adult from here on out. I recommend using functor instances from `folktale`, `ramda` or `fantasy-land` as they provide the correct `of` method as well as nice constructors that don't rely on `new`.

Mixing Metaphors



You see, in addition to space burritos (if you've heard the rumors), monads are like onions. Allow me to demonstrate with a common situation:

```
const fs = require('fs');

// readFile :: String -> IO String
const readFile = filename => new IO(() => fs.readFileSync(filename, 'utf-8'));
```

```
// print :: String -> IO String
const print = x => new IO(() => {
  console.log(x);
  return x;
});

// cat :: String -> IO (IO String)
const cat = compose(map(print), readFile);

cat('.git/config');
// IO(IO('[core]\nrepositoryformatversion = 0\n'))
```

What we've got here is an `IO` trapped inside another `IO` because `print` introduced a second `IO` during our `map`. To continue working with our string, we must `map(map(f))` and to observe the effect, we must `unsafePerformIO().unsafePerformIO()`.

```
// cat :: String -> IO (IO String)
const cat = compose(map(print), readFile);

// catFirstChar :: String -> IO (IO String)
const catFirstChar = compose(map(map(head)), cat);

catFirstChar('.git/config');
// IO(IO('['))
```

While it is nice to see that we have two effects packaged up and ready to go in our application, it feels a bit like working in two hazmat suits and we end up with an uncomfortably awkward API. Let's look at another situation:

```
// safeProp :: Key -> {Key: a} -> Maybe a
const safeProp = curry((x, obj) => Maybe.of(obj[x]));

// safeHead :: [a] -> Maybe a
const safeHead = safeProp(0);

// firstAddressStreet :: User -> Maybe (Maybe (Maybe Street))
const firstAddressStreet = compose(
  map(map(safeProp('street'))),
  map(safeHead),
  safeProp('addresses'),
);

firstAddressStreet({
  addresses: [{ street: { name: 'Mulburry', number: 8402 }, postcode: 'WC2N' }],
});
// Maybe(Maybe(Maybe({name: 'Mulburry', number: 8402})))
```

Again, we see this nested functor situation where it's neat to see there are three possible failures in our function, but it's a little presumptuous to expect a caller to `map` three times to get at the value - we'd only just met. This pattern will arise time and time again and it is the primary situation where we'll need to shine the mighty monad symbol into the night sky.

I said monads are like onions because tears well up as we peel back each layer of the nested functor with `map` to get at the inner value. We can dry our eyes, take a deep breath, and use a method called `join`.

```
const mmo = Maybe.of(Maybe.of('nunchucks'));
// Maybe(Maybe('nunchucks'))

mmo.join();
// Maybe('nunchucks')

const ioio = IO.of(IO.of('pizza'));
// IO(IO('pizza'))

ioio.join();
// IO('pizza')

const ttt = Task.of(Task.of(Task.of('sewers')));
// Task(Task(Task('sewers')));

ttt.join();
// Task(Task('sewers'))
```

If we have two layers of the same type, we can smash them together with `join`. This ability to join together, this functor matrimony, is what makes a monad a monad. Let's inch toward the full definition with something a little more accurate:

> Monads are pointed functors that can flatten

Any functor which defines a `join` method, has an `of` method, and obeys a few laws is a monad. Defining `join` is not too difficult so let's do so for `Maybe`:

```
Maybe.prototype.join = function join() {
  return this.isNothing() ? Maybe.of(null) : this.$value;
};
```

There, simple as consuming one's twin in the womb. If we have a `Maybe(Maybe(x))` then `.$value` will just remove the unnecessary extra layer and we can safely `map` from there. Otherwise, we'll just have the one `Maybe` as nothing would have been mapped in the first place.

Now that we have a `join` method, let's sprinkle some magic monad dust over the `firstAddressStreet` example and see it in action:

```
// join :: Monad m => m (m a) -> m a
const join = mma => mma.join();

// firstAddressStreet :: User -> Maybe Street
const firstAddressStreet = compose(
  join,
  map(safeProp('street')),
  join,
  map(safeHead), safeProp('addresses'),
);

firstAddressStreet({
  addresses: [{ street: { name: 'Mulburry', number: 8402 }, postcode: 'WC2N' }],
});
// Maybe({name: 'Mulburry', number: 8402})
```

We added `join` wherever we encountered the nested `Maybe`'s to keep them from getting out of hand. Let's do the same with `IO` to give us a feel for that.

```
IO.prototype.join = () => this.unsafePerformIO();
```

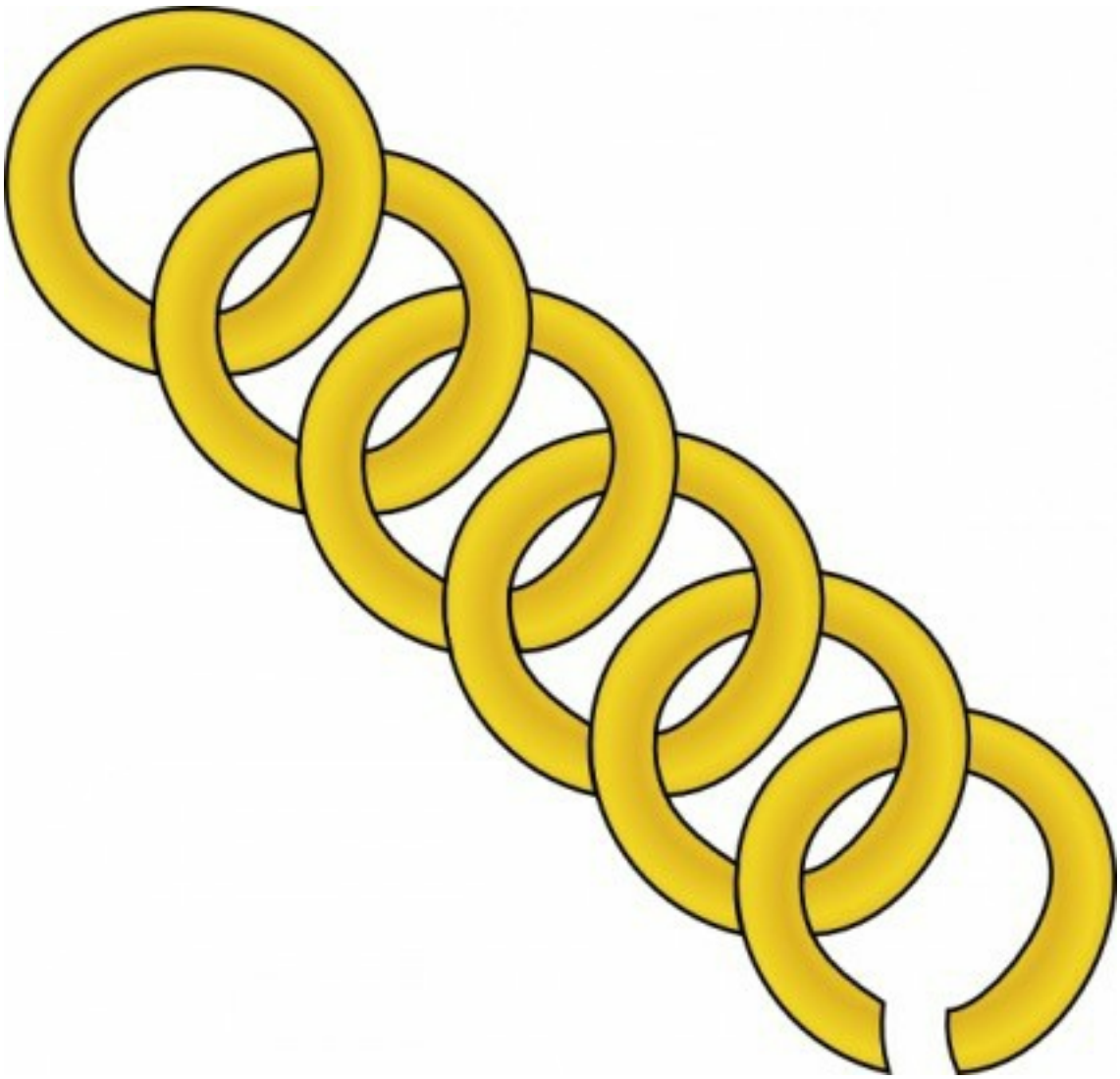
Again, we simply remove one layer. Mind you, we have not thrown out purity, but merely removed one layer of excess shrink wrap.

```
// log :: a -> IO a
const log = x => IO.of(() => {
  console.log(x);
  return x;
});

// setStyle :: Selector -> CSSProps -> IO DOM
const setStyle =
  curry((sel, props) => new IO(() => jQuery(sel).css(props)));

// getItem :: String -> IO String
const getItem = key => new IO(() => localStorage.getItem(key));

// applyPreferences :: String -> IO DOM
const applyPreferences = compose(
  join,
  map(setStyle('#main')),
  join,
  map(log),
```

```
    map(JSON.parse),
    getItem,
  );

  applyPreferences('preferences').unsafePerformIO();
  // Object {backgroundColor: "green"}
  // <div style="background-color: 'green'"/>
```

`getItem` returns an `IO String` so we `map` to parse it. Both `log` and `setStyle` return `IO`'s themselves so we must `join` to keep our nesting under control.

My Chain Hits My Chest



You might have noticed a pattern. We often end up calling `join` right after a `map`. Let's abstract this into a function called `chain`.

```
// chain :: Monad m => (a -> m b) -> m a -> m b
const chain = curry((f, m) => m.map(f).join());

// or

// chain :: Monad m => (a -> m b) -> m a -> m b
const chain = f => compose(join, map(f));
```

We'll just bundle up this map/join combo into a single function. If you've read about monads previously, you might have seen `chain` called `>>=` (pronounced bind) or `flatMap` which are all aliases for the same concept. I personally think `flatMap` is the most accurate name, but we'll stick with `chain` as it's the widely accepted name in JS. Let's refactor the two examples above with `chain`:

```
// map/join
const firstAddressStreet = compose(
  join,
  map(safeProp('street')),
  join,
  map(safeHead),
  safeProp('addresses'),
);

// chain
const firstAddressStreet = compose(
  chain(safeProp('street')),
  chain(safeHead),
  safeProp('addresses'),
);

// map/join
const applyPreferences = compose(
  join,
  map(setStyle('#main')),
  join,
  map(log),
  map(JSON.parse),
  getItem,
);

// chain
const applyPreferences = compose(
  chain(setStyle('#main')),
  chain(log),
  map(JSON.parse),
  getItem,
);
```

I swapped out any `map/join` with our new `chain` function to tidy things up a bit. Cleanliness is nice and all, but there's more to `chain` than meets the eye - it's more of a tornado than a vacuum. Because `chain` effortlessly nests effects, we can capture both *sequence* and *variable assignment* in a purely functional way.

```
// getJSON :: Url -> Params -> Task JSON
getJSON('/authenticate', { username: 'stale', password: 'crackers' })
  .chain(user => getJSON('/friends', { user_id: user.id }));
// Task([{name: 'Seimith', id: 14}, {name: 'Ric', id: 39}]);

// querySelector :: Selector -> IO DOM
querySelector('input.username')
  .chain(({ value: uname }) => querySelector('input.email')
  .chain(({ value: email }) => IO.of(`Welcome ${uname} prepare for spam at ${email}
// IO('Welcome Olivia prepare for spam at olivia@tremorcontrol.net');

Maybe.of(3)
  .chain(three => Maybe.of(2).map(add(three)));
// Maybe(5);

Maybe.of(null)
  .chain(safeProp('address'))
  .chain(safeProp('street'));
// Maybe(null);
```

We could have written these examples with `compose`, but we'd need a few helper functions and this style lends itself to explicit variable assignment via closure anyhow. Instead we're using the infix version of `chain` which, incidentally, can be derived from `map` and `join` for any type automatically: `t.prototype.chain = function(f) { return this.map(f).join(); }`. We can also define `chain` manually if we'd like a false sense of performance, though we must take care to maintain the correct functionality - that is, it must equal `map` followed by `join`. An interesting fact is that we can derive `map` for free if we've created `chain` simply by bottling the value back up when we're finished with `of`. With `chain`, we can also define `join` as `chain(id)`. It may feel like playing Texas Hold em' with a rhinestone magician in that I'm just pulling things out of my behind, but, as with most mathematics, all of these principled constructs are interrelated. Lots of these derivations are mentioned in the fantasyland repo, which is the official specification for algebraic data types in JavaScript.

Anyways, let's get to the examples above. In the first example, we see two `Task`'s chained in a sequence of asynchronous actions - first it retrieves the `user`, then it finds the friends with that user's id. We use `chain` to avoid a `Task(Task([Friend]))` situation.

Next, we use `querySelector` to find a few different inputs and create a welcoming message. Notice how we have access to both `uname` and `email` at the innermost function - this is

functional variable assignment at its finest. Since `IO` is graciously lending us its value, we are in charge of putting it back how we found it - we wouldn't want to break its trust (and our program). `IO.of` is the perfect tool for the job and it's why Pointed is an important prerequisite to the Monad interface. However, we could choose to `map` as that would also return the correct type:

```
querySelector('input.username').chain(({ value: uname }) =>
  querySelector('input.email').map(({ value: email }) =>
    `Welcome ${uname} prepare for spam at ${email}`));
// IO('Welcome Olivia prepare for spam at olivia@tremorcontrol.net');
```

Finally, we have two examples using `Maybe`. Since `chain` is mapping under the hood, if any value is `null`, we stop the computation dead in its tracks.

Don't worry if these examples are hard to grasp at first. Play with them. Poke them with a stick. Smash them to bits and reassemble. Remember to `map` when returning a "normal" value and `chain` when we're returning another functor. In the next chapter, we'll approach `Applicatives` and see nice tricks to make this kind of expressions nicer and highly readable.

As a reminder, this does not work with two different nested types. Functor composition and later, monad transformers, can help us in that situation.

Power Trip

Container style programming can be confusing at times. We sometimes find ourselves struggling to understand how many containers deep a value is or if we need `map` or `chain` (soon we'll see more container methods). We can greatly improve debugging with tricks like implementing `inspect` and we'll learn how to create a "stack" that can handle whatever effects we throw at it, but there are times when we question if it's worth the hassle.

I'd like to swing the fiery monadic sword for a moment to exhibit the power of programming this way.

Let's read a file, then upload it directly afterward:

```
// readFile :: Filename -> Either String (Task Error String)
// httpPost :: String -> Task Error JSON
// upload :: String -> Either String (Task Error JSON)
const upload = compose(map(chain(httpPost('/uploads'))), readFile);
```

Here, we are branching our code several times. Looking at the type signatures I can see that we protect against 3 errors - `readFile` uses `Either` to validate the input (perhaps ensuring the filename is present), `readFile` may error when accessing the file as expressed in the first type parameter of `Task`, and the upload may fail for whatever reason which is expressed by the `Error` in `httpPost`. We casually pull off two nested, sequential asynchronous actions with `chain`.

All of this is achieved in one linear left to right flow. This is all pure and declarative. It holds equational reasoning and reliable properties. We aren't forced to add needless and confusing variable names. Our `upload` function is written against generic interfaces and not specific one-off apis. It's one bloody line for goodness sake.

For contrast, let's look at the standard imperative way to pull this off:

```
// upload :: String -> (String -> a) -> Void
const upload = (filename, callback) => {
  if (!filename) {
    throw new Error('You need a filename!');
  } else {
    readFile(filename, (errF, contents) => {
      if (errF) throw err;
      httpPost(contents, (errH, json) => {
        if (errH) throw errH;
        callback(json);
      });
    });
  }
};
```

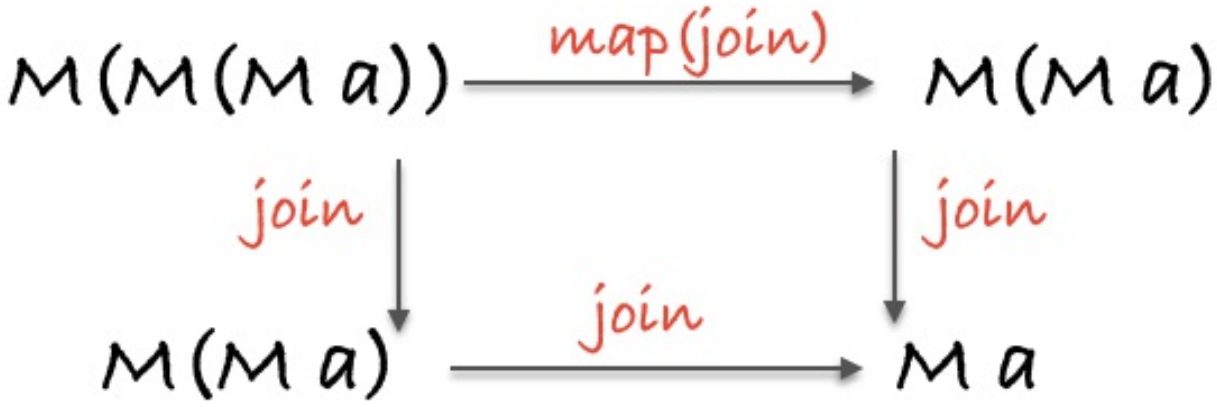
Well isn't that the devil's arithmetic. We're pinballed through a volatile maze of madness. Imagine if it were a typical app that also mutated variables along the way! We'd be in the tar pit indeed.

Theory

The first law we'll look at is associativity, but perhaps not in the way you're used to it.

```
// associativity
compose(join, map(join)) === compose(join, join);
```

These laws get at the nested nature of monads so associativity focuses on joining the inner or outer types first to achieve the same result. A picture might be more instructive:

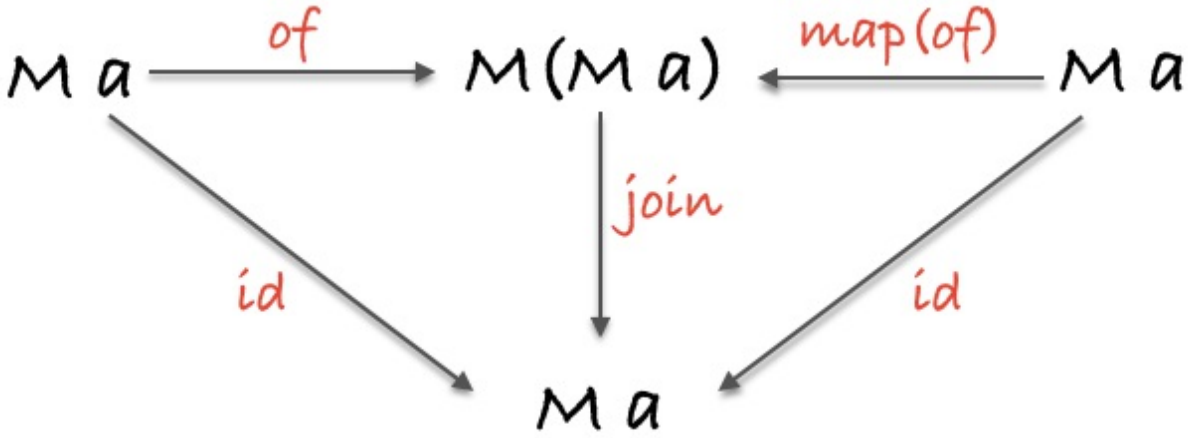


Starting with the top left moving downward, we can `join` the outer two `M`'s of `M(M(M a))` first then cruise over to our desired `M a` with another `join`. Alternatively, we can pop the hood and flatten the inner two `M`'s with `map(join)`. We end up with the same `M a` regardless of if we join the inner or outer `M`'s first and that's what associativity is all about. It's worth noting that `map(join) != join`. The intermediate steps can vary in value, but the end result of the last `join` will be the same.

The second law is similar:

```
// identity for all (M a)
compose(join, of) === compose(join, map(of)) === id;
```

It states that, for any monad `M`, `of` and `join` amounts to `id`. We can also `map(of)` and attack it from the inside out. We call this "triangle identity" because it makes such a shape when visualized:



If we start at the top left heading right, we can see that `of` does indeed drop our `M a` in

another `M` container. Then if we move downward and `join` it, we get the same as if we just called `id` in the first place. Moving right to left, we see that if we sneak under the covers with `map` and call `of` of the plain `a`, we'll still end up with `M (M a)` and `join`ing will bring us back to square one.

I should mention that I've just written `of`, however, it must be the specific `M.of` for whatever monad we're using.

Now, I've seen these laws, identity and associativity, somewhere before… Hold on, I'm thinking…Yes of course! They are the laws for a category. But that would mean we need a composition function to complete the definition. Behold:

```
const mcompose = (f, g) => compose(chain(f), g);

// left identity
mcompose(M, f) === f;

// right identity
mcompose(f, M) === f;

// associativity
mcompose(mcompose(f, g), h) === mcompose(f, mcompose(g, h));
```

They are the category laws after all. Monads form a category called the "Kleisli category" where all objects are monads and morphisms are chained functions. I don't mean to taunt you with bits and bobs of category theory without much explanation of how the jigsaw fits together. The intention is to scratch the surface enough to show the relevance and spark some interest while focusing on the practical properties we can use each day.

In Summary

Monads let us drill downward into nested computations. We can assign variables, run sequential effects, perform asynchronous tasks, all without laying one brick in a pyramid of doom. They come to the rescue when a value finds itself jailed in multiple layers of the same type. With the help of the trusty sidekick "pointed", monads are able to lend us an unboxed value and know we'll be able to place it back in when we're done.

Yes, monads are very powerful, yet we still find ourselves needing some extra container functions. For instance, what if we wanted to run a list of api calls at once, then gather the results? We can accomplish this task with monads, but we'd have to wait for each one to finish before calling the next. What about combining several validations? We'd like to continue validating to gather the list of errors, but monads would stop the show after the first `Left`

entered the picture.

In the next chapter, we'll see how applicative functors fit into the container world and why we prefer them to monads in many cases.

Chapter 10: Applicative Functors

Exercises

Considering a User object as follow:

`js const user = { id: 1, name: 'Albert', address: { street: { number: 22, name: 'Walnut St', }, }, };`

{% exercise %}
Use `safeProp` and `map/join` or `chain` to safely get the street name when given a user

{% initial src="./exercises/ch09/exercise_a.js#L16;" %}
`js // getStreetName :: User -> Maybe String const getStreetName = undefined;`

{% solution src="./exercises/ch09/solution_a.js" %}
{% validation src="./exercises/ch09/validation_a.js" %}
{% context src="./exercises/support.js" %}
{% endexercise %}

We now consider the following functions

```
// getFile :: () -> IO String
const getFile = () => IO.of('/home/mostly-adequate/ch9.md');

// pureLog :: String -> IO ()
const pureLog = str => new IO(() => console.log(str));
```

{% exercise %}
Use getFile to get the filepath, remove the directory and keep only the basename, then purely log it. Hint: you may want to use `split` and `last` to obtain the basename from a filepath.

{% initial src="./exercises/ch09/exercise_b.js#L13;" %}
```js
```

```
// logFilename :: IO ()
const logFilename = undefined;
```

{% solution src="./exercises/ch09/solution_b.js" %}
{% validation src="./exercises/ch09/validation_b.js" %}
{% context src="./exercises/support.js" %}
{% endexercise %}

For this exercise, we consider helpers with the following signatures:

```
// validateEmail :: Email -> Either String Email
// addToMailingList :: Email -> IO([Email])
// emailBlast :: [Email] -> IO ()
```

{% exercise %}
Use `validateEmail`, `addToMailingList` and `emailBlast` to create a function which adds a new email to the mailing list if valid, and then notify the whole list.

{% initial src="./exercises/ch09/exercise_c.js#L11;" %}
`js // joinMailingList :: Email -> Either String (IO ()) const joinMailingList = undefined;`

{% solution src="./exercises/ch09/solution_c.js" %}
{% validation src="./exercises/ch09/validation_c.js" %}
{% context src="./exercises/support.js" %}
{% endexercise %}