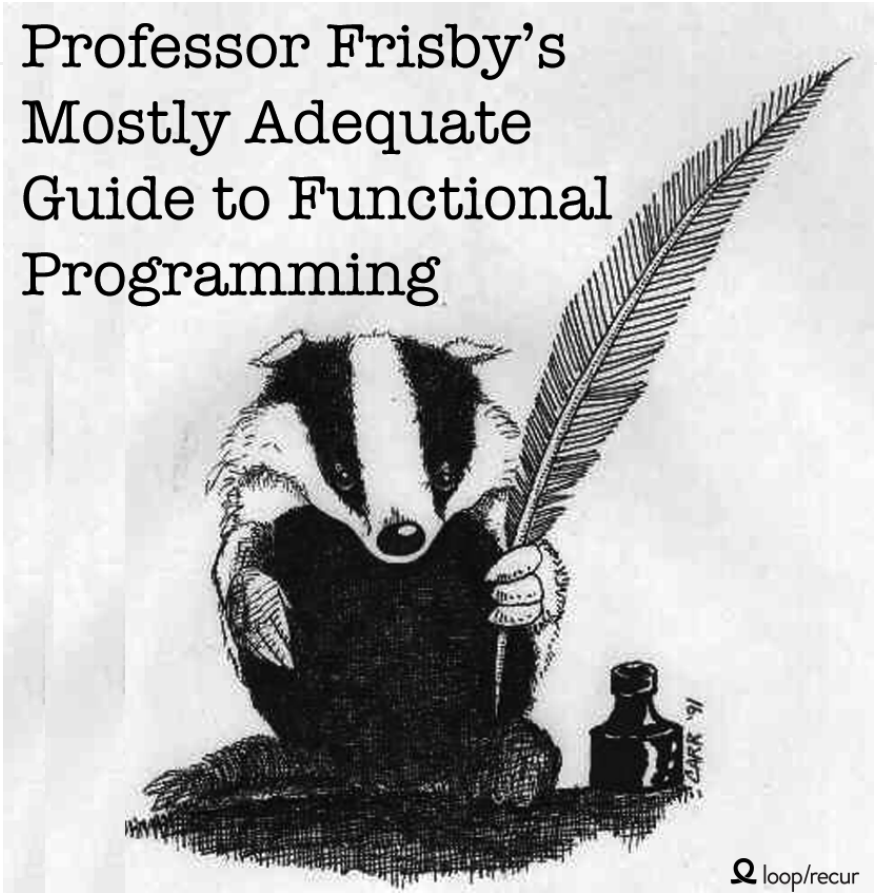
Professor Frisby's Mostly Adequate Guide to Functional Programming

loop/recur

Table of Contents



About this book

This is a book on the functional paradigm in general. We'll use the world's most popular functional programming language: JavaScript. Some may feel this is a poor choice as it's against the grain of the current culture which, at the moment, feels predominately imperative. However, I believe it is the best way to learn FP for several reasons:

- **You likely use it every day at work.**

  This makes it possible to practice and apply your acquired knowledge each day on real world programs rather than pet projects on nights and weekends in an esoteric FP language.

- **We don't have to learn everything up front to start writing programs.**

  In a pure functional language, you cannot log a variable or read a DOM node without using monads. Here we can cheat a little as we learn to purify our codebase. It's also easier to get started in this language since it's mixed paradigm and you can fall back on your current practices while there are gaps in your knowledge.

- **The language is fully capable of writing top notch functional code.**

> We have all the features we need to mimic a language like Scala or Haskell with the help of a tiny library or two. Object-oriented programming currently dominates the industry, but it's clearly awkward in JavaScript. It's akin to camping off of a highway or tap dancing in galoshes. We have to `bind` all over the place lest `this` change out from under us, we have various work arounds for the quirky behavior when the `new` keyword is forgotten, private members are only available via closures. To a lot of us, FP feels more natural anyways.

That said, typed functional languages will, without a doubt, be the best place to code in the style presented by this book. JavaScript will be our means of learning a paradigm, where you apply it is up to you. Luckily, the interfaces are mathematical and, as such, ubiquitous. You'll find yourself at home with Swiftz, Scalaz, Haskell, PureScript, and other mathematically inclined environments.

Read it Online

For a best reading experience, read it online via Gitbook.

- Quick-access side-bar
- In-browser exercises
- In-depth examples

Play Around with Code

To make the training efficient and not get too bored while I am telling you another story, make sure to play around with the concepts introduced in this book. Some can be tricky to catch at first and are better understood by getting your hands dirty. All functions and algebraic data-structures presented in the book are gathered in the appendixes. The corresponding code is also available as an npm module:

```
$ npm i @mostly-adequate/support
```

Alternatively, exercises of each chapter are runnable and can be completed in your editor! For example, complete the `exercise_*.js` in `exercises/ch04` and then run:

```
$ npm run ch04
```

Download it

Find pre-generated **PDF** and **EPUB** as build artifacts of the latest release.

Do it yourself

> ⚠️ This project setup is now a bit old and thus, you may run into various issues when building this locally. We recommend to use node v10.22.1 and the latest version of Calibre if possible.

```
git clone https://github.com/MostlyAdequate/mostly-adequate-guide.git
cd mostly-adequate-guide/
npm install
npm run setup
npm run generate-pdf
npm run generate-epub
```

> Note! To generate the ebook version you will need to install `ebook-convert`. Installation instructions.

Table of Contents

See SUMMARY.md

Contributing

See CONTRIBUTING.md

Translations

See TRANSLATIONS.md

FAQ

See FAQ.md

Plans for the future

- **Part 1** (chapters 1-7) is a guide to the basics. I'm updating as I find errors since this is the initial draft. Feel free to help!
- **Part 2** (chapters 8-13) address type classes like functors and monads all the way through to traversable. I hope to squeeze in transformers and a pure application.
- **Part 3** (chapters 14+) will start to dance the fine line between practical programming and academic absurdity. We'll look at comonads, f-algebras, free monads, yoneda, and other categorical constructs.

Chapter 01: What Ever Are We Doing?

Introductions

Hi there! I'm Professor Franklin Frisby. Pleased to make your acquaintance. We'll be spending some time together, as I'm supposed to teach you a bit about functional programming. But enough about me, what about you? I'm hoping that you're at least a bit familiar with the JavaScript language, have a teensy bit of Object-Oriented experience, and fancy yourself a working class programmer. You don't need to have a PhD in Entomology, you just need to know how to find and kill some bugs.

I won't assume that you have any previous functional programming knowledge, because we both know what happens when you assume. I will, however, expect you to have run into some of the unfavorable situations that arise when working with mutable state, unrestricted side effects, and unprincipled design. Now that we've been properly introduced, let's get on with it.

The purpose of this chapter is to give you a feel for what we're after when we write functional programs. In order to be able to understand the following chapters, we must have some idea about what makes a program *functional*. Otherwise we'll find ourselves scribbling aimlessly, avoiding objects at all costs - a clumsy endeavor indeed. We need a clear bullseye to hurl our code at, some celestial compass for when the waters get rough.

Now, there are some general programming principles - various acronymic credos that guide us through the dark tunnels of any application: DRY (don't repeat yourself), YAGNI (ya ain't gonna need it), loose coupling high cohesion, the principle of least surprise, single responsibility, and so on.

I won't belabor you by listing each and every guideline I've heard throughout the years... The point of the matter is that they hold up in a functional setting, although they're merely tangential to our ultimate goal. What I'd like you to get a feel for now, before we get any further, is our intention when we poke and prod at the keyboard; our functional Xanadu.

A Brief Encounter

Let's start with a touch of insanity. Here is a seagull application. When flocks conjoin they become a larger flock, and when they breed, they increase by the number of seagulls with whom they're breeding. Now, this is not intended to be good Object-Oriented code, mind you, it is here to highlight the perils of our modern, assignment based approach. Behold:

```
class Flock {
  constructor(n) {
    this.seagulls = n;
  }

  conjoin(other) {
    this.seagulls += other.seagulls;
    return this;
  }

  breed(other) {
    this.seagulls = this.seagulls * other.seagulls;
    return this;
  }
}

const flockA = new Flock(4);
const flockB = new Flock(2);
const flockC = new Flock(0);
const result = flockA
  .conjoin(flockC)
  .breed(flockB)
  .conjoin(flockA.breed(flockB))
  .seagulls;
// 32
```

Who on earth would craft such a ghastly abomination? It is unreasonably difficult to keep track of the mutating internal state. And, good heavens, the answer is even incorrect! It should have been `16`, but `flockA` wound up permanently altered in the process. Poor `flockA`. This is anarchy in the I.T.! This is wild animal arithmetic!

If you don't understand this program, it's okay, neither do I. The point to remember here is that state and mutable values are hard to follow, even in such a small example.

Let's try again, this time using a more functional approach:

```
const conjoin = (flockX, flockY) => flockX + flockY;
const breed = (flockX, flockY) => flockX * flockY;

const flockA = 4;
const flockB = 2;
const flockC = 0;
const result =
    conjoin(breed(flockB, conjoin(flockA, flockC)), breed(flockA, flockB));
// 16
```

Well, this time we got the right answer. With much less code. The function nesting is a tad confusing... (we'll remedy this situation in ch5). It's better, but let's dig a little bit deeper. There are benefits to calling a spade a spade. Had we scrutinized our custom functions more closely, we would have discovered that we're just working with simple addition (`conjoin`) and multiplication (`breed`).

There's really nothing special at all about these two functions other than their names. Let's rename our custom functions to `multiply` and `add` in order to reveal their true identities.

```
const add = (x, y) => x + y;
const multiply = (x, y) => x * y;

const flockA = 4;
const flockB = 2;
const flockC = 0;
const result =
    add(multiply(flockB, add(flockA, flockC)), multiply(flockA, flockB));
// 16
```

And with that, we gain the knowledge of the ancients:

```
// associative
add(add(x, y), z) === add(x, add(y, z));

// commutative
add(x, y) === add(y, x);

// identity
add(x, 0) === x;

// distributive
multiply(x, add(y,z)) === add(multiply(x, y), multiply(x, z));
```

Ah yes, those old faithful mathematical properties should come in handy. Don't worry if you didn't know them right off the top of your head. For a lot of us, it's been a while since we learned about these laws of arithmetic. Let's see if we can use these properties to simplify our little seagull program.

```
// Original line
add(multiply(flockB, add(flockA, flockC)), multiply(flockA, flockB));

// Apply the identity property to remove the extra add
// (add(flockA, flockC) == flockA)
add(multiply(flockB, flockA), multiply(flockA, flockB));

// Apply distributive property to achieve our result
multiply(flockB, add(flockA, flockA));
```

Brilliant! We didn't have to write a lick of custom code other than our calling function. We include `add` and `multiply` definitions here for completeness, but there is really no need to write them - we surely have an `add` and `multiply` provided by some existing library.

You may be thinking "how very strawman of you to put such a mathy example up front". Or "real programs are not this simple and cannot be reasoned about in such a way." I've chosen this example because most of us already know about addition and multiplication, so it's easy to see how math is very useful for us here.

Don't despair - throughout this book, we'll sprinkle in some category theory, set theory, and lambda calculus and write real world examples that achieve the same elegant simplicity and results as our flock of seagulls example. You needn't be a mathematician either. It will feel natural and easy, just like you were using a "normal" framework or API.

It may come as a surprise to hear that we can write full, everyday applications along the lines of the functional analog above. Programs that have sound properties. Programs that are terse, yet easy to reason about. Programs that don't reinvent the wheel at every turn. Lawlessness is good if you're a criminal, but in this book, we'll want to acknowledge and obey the laws of math.

We'll want to use a theory where every piece tends to fit together so politely. We'll want to represent our specific problem in terms of generic, composable bits and then exploit their properties for our own selfish benefit. It will take a bit more discipline than the "anything goes" approach of imperative programming (we'll go over the precise definition of "imperative" later in the book, but for now consider it anything other than functional programming). The payoff of working within a principled, mathematical framework will truly astound you.

We've seen a flicker of our functional northern star, but there are a few concrete concepts to grasp before we can really begin our journey.

Chapter 02: First Class Functions

Chapter 02: First Class Functions

A Quick Review

When we say functions are "first class", we mean they are just like everyone else... so in other words a normal class. We can treat functions like any other data type and there is nothing particularly special about them - they may be stored in arrays, passed around as function parameters, assigned to variables, and what have you.

That is JavaScript 101, but worth mentioning since a quick code search on github will reveal the collective evasion, or perhaps widespread ignorance of this concept. Shall we go for a feigned example? We shall.

```
const hi = name => `Hi ${name}`;
const greeting = name => hi(name);
```

Here, the function wrapper around `hi` in `greeting` is completely redundant. Why? Because functions are *callable* in JavaScript. When `hi` has the `()` at the end it will run and return a value. When it does not, it simply returns the function stored in the variable. Just to be sure, have a look yourself:

```
hi; // name => `Hi ${name}`
hi("jonas"); // "Hi jonas"
```

Since `greeting` is merely in turn calling `hi` with the very same argument, we could simply write:

```
const greeting = hi;
greeting("times"); // "Hi times"
```

In other words, `hi` is already a function that expects one argument, why place another function around it that simply calls `hi` with the same bloody argument? It doesn't make any damn sense. It's like donning your heaviest parka in the dead of July just to blast the air and demand an ice lolly.

It is obnoxiously verbose and, as it happens, bad practice to surround a function with another function merely to delay evaluation (we'll see why in a moment, but it has to do with maintenance)

A solid understanding of this is critical before moving on, so let's examine a few more fun examples excavated from the library of npm packages.

```
// ignorant
const getServerStuff = callback => ajaxCall(json => callback(json));

// enlightened
const getServerStuff = ajaxCall;
```

The world is littered with ajax code exactly like this. Here is the reason both are equivalent:

```
// this line
ajaxCall(json => callback(json));

// is the same as this line
ajaxCall(callback);

// so refactor getServerStuff
const getServerStuff = callback => ajaxCall(callback);

// ...which is equivalent to this
const getServerStuff = ajaxCall; // <-- look mum, no ()'s
```

And that, folks, is how it is done. Once more so that we understand why I'm being so persistent.

```
const BlogController = {
  index(posts) { return Views.index(posts); },
  show(post) { return Views.show(post); },
  create(attrs) { return Db.create(attrs); },
  update(post, attrs) { return Db.update(post, attrs); },
  destroy(post) { return Db.destroy(post); },
};
```

This ridiculous controller is 99% fluff. We could either rewrite it as:

```
const BlogController = {
  index: Views.index,
  show: Views.show,
  create: Db.create,
  update: Db.update,
  destroy: Db.destroy,
};
```

... or scrap it altogether since it does nothing more than just bundle our Views and Db together.

Why Favor First Class?

Okay, let's get down to the reasons to favor first class functions. As we saw in the `getServerStuff` and `BlogController` examples, it's easy to add layers of indirection that provide no added value and only increase the amount of redundant code to maintain and search through.

In addition, if such a needlessly wrapped function must be changed, we must also need to change our wrapper function as well.

```
httpGet('/post/2', json => renderPost(json));
```

If `httpGet` were to change to send a possible `err`, we would need to go back and change the "glue".

```
// go back to every httpGet call in the application and explicitly pass err alo
httpGet('/post/2', (json, err) => renderPost(json, err));
```

Had we written it as a first class function, much less would need to change:

```
// renderPost is called from within httpGet with however many arguments it want
httpGet('/post/2', renderPost);
```

Besides the removal of unnecessary functions, we must name and reference arguments. Names are a bit of an issue, you see. We have potential misnomers - especially as the codebase ages and requirements change.

Having multiple names for the same concept is a common source of confusion in projects. There is also the issue of generic code. For instance, these two functions do exactly the same thing, but one feels infinitely more general and reusable:

```
// specific to our current blog
const validArticles = articles =>
  articles.filter(article => article !== null && article !== undefined),

// vastly more relevant for future projects
const compact = xs => xs.filter(x => x !== null && x !== undefined);
```

By using specific naming, we've seemingly tied ourselves to specific data (in this case `articles`). This happens quite a bit and is a source of much reinvention.

I must mention that, just like with Object-Oriented code, you must be aware of `this` coming to bite you in the jugular. If an underlying function uses `this` and we call it first class, we are subject to this leaky abstraction's wrath.

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

// scary
fs.readFile('freaky_friday.txt', Db.save);

// less so
fs.readFile('freaky_friday.txt', Db.save.bind(Db));
```

Having been bound to itself, the `Db` is free to access its prototypical garbage code. I avoid using `this` like a dirty nappy. There's really no need when writing functional code. However, when interfacing with other libraries, you might have to acquiesce to the mad world around us.

Some will argue that `this` is necessary for optimizing speed. If you are the micro-optimization sort, please close this book. If you cannot get your money back, perhaps you can exchange it for something more fiddly.

And with that, we're ready to move on.

Chapter 03: Pure Happiness with Pure Functions

Chapter 03: Pure Happiness with Pure Functions

Oh to Be Pure Again

One thing we need to get straight is the idea of a pure function.

> A pure function is a function that, given the same input, will always return the same output and does not have any observable side effect.

Take `slice` and `splice`. They are two functions that do the exact same thing - in a vastly different way, mind you, but the same thing nonetheless. We say `slice` is *pure* because it returns the same output per input every time, guaranteed. `splice`, however, will chew up its array and spit it back out forever changed which is an observable effect.

```
const xs = [1,2,3,4,5];

// pure
xs.slice(0,3); // [1,2,3]

xs.slice(0,3); // [1,2,3]

xs.slice(0,3); // [1,2,3]


// impure
xs.splice(0,3); // [1,2,3]

xs.splice(0,3); // [4,5]

xs.splice(0,3); // []
```

In functional programming, we dislike unwieldy functions like `splice` that *mutate* data. This will never do as we're striving for reliable functions that return the same result every time, not functions that leave a mess in their wake like `splice`.

Let's look at another example.

```
// impure
let minimum = 21;
const checkAge = age => age >= minimum;

// pure
const checkAge = (age) => {
  const minimum = 21;
  return age >= minimum;
};
```

In the impure portion, `checkAge` depends on the mutable variable `minimum` to determine the result. In other words, it depends on system state which is disappointing because it increases the cognitive load by introducing an external environment.

It might not seem like a lot in this example, but this reliance upon state is one of the largest contributors to system complexity (http://curtclifton.net/papers/MoseleyMarks06a.pdf). This `checkAge` may return different results depending on factors external to input, which not only disqualifies it from being pure, but also puts our minds through the wringer each time we're reasoning about the software.

Its pure form, on the other hand, is completely self sufficient. We can also make `minimum` immutable, which preserves the purity as the state will never change. To do this, we must create an object to freeze.

```
const immutableState = Object.freeze({ minimum: 21 });
```

Side Effects May Include...

Let's look more at these "side effects" to improve our intuition. So what is this undoubtedly nefarious *side effect* mentioned in the definition of *pure function*? We'll be referring to *effect* as anything that occurs in our computation other than the calculation of a result.

There's nothing intrinsically bad about effects and we'll be using them all over the place in the chapters to come. It's that *side* part that bears the negative connotation. Water alone is not an inherent larvae incubator, it's the *stagnant* part that yields the swarms, and I assure you, *side* effects are a similar breeding ground in your own programs.

> A *side effect* is a change of system state or *observable interaction* with the outside world that occurs during the calculation of a result.

Side effects may include, but are not limited to

- changing the file system
- inserting a record into a database
- making an http call

- mutations
- printing to the screen / logging
- obtaining user input
- querying the DOM
- accessing system state

And the list goes on and on. Any interaction with the world outside of a function is a side effect, which is a fact that may prompt you to suspect the practicality of programming without them. The philosophy of functional programming postulates that side effects are a primary cause of incorrect behavior.

It is not that we're forbidden to use them, rather we want to contain them and run them in a controlled way. We'll learn how to do this when we get to functors and monads in later chapters, but for now, let's try to keep these insidious functions separate from our pure ones.

Side effects disqualify a function from being *pure*. And it makes sense: pure functions, by definition, must always return the same output given the same input, which is not possible to guarantee when dealing with matters outside our local function.

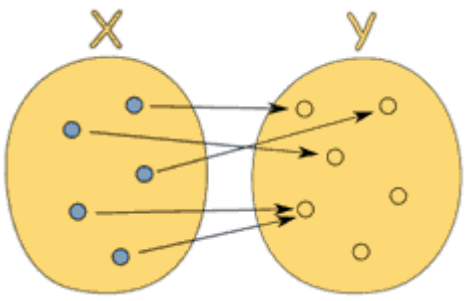
Let's take a closer look at why we insist on the same output per input. Pop your collars, we're going to look at some 8th grade math.

8th Grade Math

From mathisfun.com:

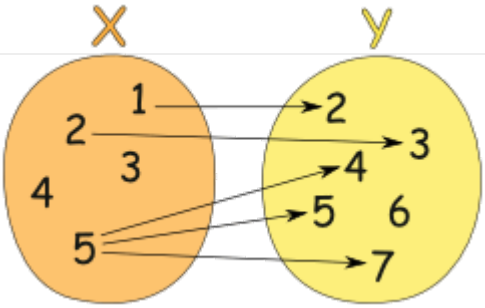
> A function is a special relationship between values: Each of its input values gives back exactly one output value.

In other words, it's just a relation between two values: the input and the output. Though each input has exactly one output, that output doesn't necessarily have to be unique per input. Below shows a diagram of a perfectly valid function from `x` to `y`;



(https://www.mathsisfun.com/sets/function.html)

To contrast, the following diagram shows a relation that is *not* a function since the input value `5` points to several outputs:



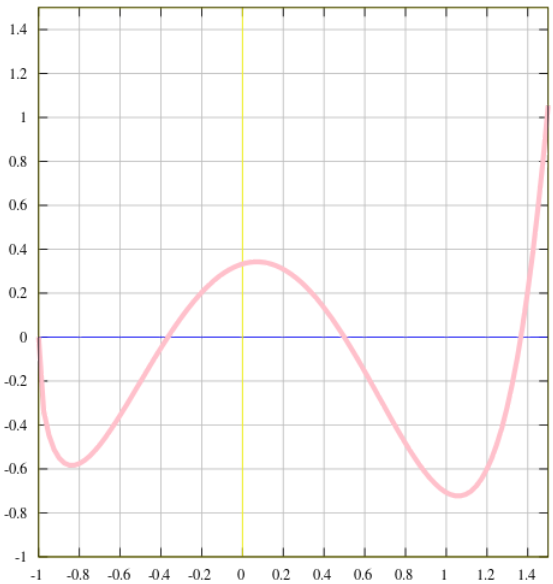
(https://www.mathsisfun.com/sets/function.html)

Functions can be described as a set of pairs with the position (input, output): `[(1,2), (3,6), (5,10)]` (It appears this function doubles its input).

Or perhaps a table:

Input	Output
1	2
2	4
3	6

Or even as a graph with `x` as the input and `y` as the output:



There's no need for implementation details if the input dictates the output. Since functions are simply mappings of input to output, one could simply jot down object literals and run them with `[]` instead of `()`.

```
const toLowerCase = {
  A: 'a',
  B: 'b',
  C: 'c',
  D: 'd',
  E: 'e',
  F: 'f',
};
toLowerCase['C']; // 'c'

const isPrime = {
  1: false,
  2: true,
  3: true,
  4: false,
  5: true,
  6: false,
};
isPrime[3]; // true
```

Of course, you might want to calculate instead of hand writing things out, but this illustrates a different way to think about functions. (You may be thinking "what about functions with multiple arguments?". Indeed, that presents a bit of an inconvenience when thinking in terms of mathematics. For now, we can bundle them up in an array or just think of the `arguments` object as the input. When we learn about *currying*, we'll see how we can directly model the mathematical definition of a function.)

Here comes the dramatic reveal: Pure functions *are* mathematical functions and they're what functional programming is all about. Programming with these little angels can provide huge benefits. Let's look at some reasons why we're willing to go to great lengths to preserve purity.

The Case for Purity

Cacheable

For starters, pure functions can always be cached by input. This is typically done using a technique called memoization:

```
const squareNumber = memoize(x => x * x);

squareNumber(4); // 16

squareNumber(4); // 16, returns cache for input 4

squareNumber(5); // 25

squareNumber(5); // 25, returns cache for input 5
```

Here is a simplified implementation, though there are plenty of more robust versions available.

```
const memoize = (f) => {
  const cache = {};

  return (...args) => {
    const argStr = JSON.stringify(args);
    cache[argStr] = cache[argStr] || f(...args);
    return cache[argStr];
  };
};
```

Something to note is that you can transform some impure functions into pure ones by delaying evaluation:

```
const pureHttpCall = memoize((url, params) => () => $.getJSON(url, params));
```

The interesting thing here is that we don't actually make the http call - we instead return a function that will do so when called. This function is pure because it will always return the same output given the same input: the function that will make the particular http call given the `url` and `params`.

Our `memoize` function works just fine, though it doesn't cache the results of the http call, rather it caches the generated function.

This is not very useful yet, but we'll soon learn some tricks that will make it so. The takeaway is that we can cache every function no matter how destructive they seem.

Portable / Self-documenting

Pure functions are completely self contained. Everything the function needs is handed to it on a silver platter. Ponder this for a moment... How might this be beneficial? For starters, a function's dependencies are explicit and therefore easier to see and understand - no funny business going on under the hood.

```
// impure
const signUp = (attrs) => {
  const user = saveUser(attrs);
  welcomeUser(user);
};

// pure
const signUp = (Db, Email, attrs) => () => {
  const user = saveUser(Db, attrs);
  welcomeUser(Email, user);
};
```

The example here demonstrates that the pure function must be honest about its dependencies and, as such, tell us exactly what it's up to. Just from its signature, we know that it will use a `Db`, `Email`, and `attrs` which should be telling to say the least.

We'll learn how to make functions like this pure without merely deferring evaluation, but the point should be clear that the pure form is much more informative than its sneaky impure counterpart which is up to who knows what.

Something else to notice is that we're forced to "inject" dependencies, or pass them in as arguments, which makes our app much more flexible because we've parameterized our database or mail client or what have you (don't worry, we'll see a way to make this less tedious than it sounds). Should we choose to use a different Db we need only to call our function with it. Should we find ourselves writing a new application in which we'd like to reuse this reliable function, we simply give this function whatever `Db` and `Email` we have at the time.

In a JavaScript setting, portability could mean serializing and sending functions over a socket. It could mean running all our app code in web workers. Portability is a powerful trait.

Contrary to "typical" methods and procedures in imperative programming rooted deep in their environment via state, dependencies, and available effects, pure functions can be run anywhere our hearts desire.

When was the last time you copied a method into a new app? One of my favorite quotes comes from Erlang creator, Joe Armstrong: "The problem with object-oriented languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana... and the entire jungle".

Testable

Next, we come to realize pure functions make testing much easier. We don't have to mock a "real" payment gateway or setup and assert the state of the world after each test. We simply give the function input and assert output.

In fact, we find the functional community pioneering new test tools that can blast our functions with generated input and assert that properties hold on the output. It's beyond the scope of this book, but I strongly encourage you to search for and try *Quickcheck* - a testing tool that is tailored for a purely functional environment.

Reasonable

Many believe the biggest win when working with pure functions is *referential transparency*. A spot of code is referentially transparent when it can be substituted for its evaluated value without changing the behavior of the program.

Since pure functions don't have side effects, they can only influence the behavior of a program through their output values. Furthermore, since their output values can reliably be calculated using only their input values, pure functions will always preserve referential transparency. Let's see an example.

```
const { Map } = require('immutable');

// Aliases: p = player, a = attacker, t = target
const jobe = Map({ name: 'Jobe', hp: 20, team: 'red' });
const michael = Map({ name: 'Michael', hp: 20, team: 'green' });
const decrementHP = p => p.set('hp', p.get('hp') - 1);
const isSameTeam = (p1, p2) => p1.get('team') === p2.get('team');
const punch = (a, t) => (isSameTeam(a, t) ? t : decrementHP(t));

punch(jobe, michael); // Map({name:'Michael', hp:19, team: 'green'})
```

`decrementHP`, `isSameTeam` and `punch` are all pure and therefore referentially transparent. We can use a technique called *equational reasoning* wherein one substitutes "equals for equals" to reason about code. It's a bit like manually evaluating the code without taking into account the quirks of programmatic evaluation. Using referential transparency, let's play with this code a bit.

First we'll inline the function `isSameTeam`.

```
const punch = (a, t) => (a.get('team') === t.get('team') ? t : decrementHP(t));
```

Since our data is immutable, we can simply replace the teams with their actual value

```
const punch = (a, t) => ('red' === 'green' ? t : decrementHP(t));
```

We see that it is false in this case so we can remove the entire if branch

```
const punch = (a, t) => decrementHP(t);
```

And if we inline `decrementHP`, we see that, in this case, punch becomes a call to decrement the `hp` by 1.

```
const punch = (a, t) => t.set('hp', t.get('hp') - 1);
```

This ability to reason about code is terrific for refactoring and understanding code in general. In fact, we used this technique to refactor our flock of seagulls program. We used equational reasoning to harness the properties of addition and multiplication. Indeed, we'll be using these techniques throughout the book.

Parallel Code

Finally, and here's the coup de grâce, we can run any pure function in parallel since it does not need access to shared memory and it cannot, by definition, have a race condition due to some side effect.

This is very much possible in a server side js environment with threads as well as in the browser with web workers though current culture seems to avoid it due to complexity when dealing with impure functions.

In Summary

We've seen what pure functions are and why we, as functional programmers, believe they are the cat's evening wear. From this point on, we'll strive to write all our functions in a pure way. We'll require some extra tools to help us do so, but in the meantime, we'll try to separate the impure functions from the rest of the pure code.

Writing programs with pure functions is a tad laborious without some extra tools in our belt. We have to juggle data by passing arguments all over the place, we're forbidden to use state, not to mention effects. How does one go about writing these masochistic programs? Let's acquire a new tool called curry.

Chapter 04: Currying

Chapter 04: Currying

Can't Live If Livin' Is without You

My Dad once explained how there are certain things one can live without until one acquires them. A microwave is one such thing. Smart phones, another. The older folks among us will remember a fulfilling life sans internet. For me, currying is on this list.

The concept is simple: You can call a function with fewer arguments than it expects. It returns a function that takes the remaining arguments.

You can choose to call it all at once or simply feed in each argument piecemeal.

```
const add = x => y => x + y;
const increment = add(1);
const addTen = add(10);

increment(2); // 3
addTen(2); // 12
```

Here we've made a function `add` that takes one argument and returns a function. By calling it, the returned function remembers the first argument from then on via the closure. Calling it with both arguments all at once is a bit of a pain, however, so we can use a special helper function called `curry` to make defining and calling functions like this easier.

Let's set up a few curried functions for our enjoyment. From now on, we'll summon our `curry` function defined in the Appendix A - Essential Function Support.

```
const match = curry((what, s) => s.match(what));
const replace = curry((what, replacement, s) => s.replace(what, replacement));
const filter = curry((f, xs) => xs.filter(f));
const map = curry((f, xs) => xs.map(f));
```

The pattern I've followed is a simple, but important one. I've strategically positioned the data we're operating on (String, Array) as the last argument. It will become clear as to why upon use.

(The syntax `/r/g` is a regular expression that means *match every letter 'r'*. Read more about regular expressions if you like.)

```
match(/r/g, 'hello world'); // [ 'r' ]

const hasLetterR = match(/r/g); // x => x.match(/r/g)
hasLetterR('hello world'); // [ 'r' ]
hasLetterR('just j and s and t etc'); // null

filter(hasLetterR, ['rock and roll', 'smooth jazz']); // ['rock and roll']

const removeStringsWithoutRs = filter(hasLetterR); // xs => xs.filter(x => x.ma
removeStringsWithoutRs(['rock and roll', 'smooth jazz', 'drum circle']); // ['r

const noVowels = replace(/[aeiou]/ig); // (r,x) => x.replace(/[aeiou]/ig, r)
const censored = noVowels('*'); // x => x.replace(/[aeiou]/ig, '*')
censored('Chocolate Rain'); // 'Ch*c*l*t* R**n'
```

What's demonstrated here is the ability to "pre-load" a function with an argument or two in order to receive a new function that remembers those arguments.

I encourage you to clone the Mostly Adequate repository (`git clone https://github.com/MostlyAdequate/mostly-adequate-guide.git`), copy the code above and have a go at it in the REPL. The curry function, as well as actually anything defined in the appendixes, are available in the `support/index.js` module.

Alternatively, have a look at a published version on `npm` :

```
npm install @mostly-adequate/support
```

More Than a Pun / Special Sauce

Currying is useful for many things. We can make new functions just by giving our base functions some arguments as seen in `hasLetterR` , `removeStringsWithoutRs` , and `censored` .

We also have the ability to transform any function that works on single elements into a function that works on arrays simply by wrapping it with `map` :

```
const getChildren = x => x.childNodes;
const allTheChildren = map(getChildren);
```

Giving a function fewer arguments than it expects is typically called *partial application*. Partially applying a function can remove a lot of boiler plate code. Consider what the above `allTheChildren` function would be with the uncurried `map` from lodash (note the arguments are in a different order):

```
const allTheChildren = elements => map(elements, getChildren);
```

We typically don't define functions that work on arrays, because we can just call `map(getChildren)` inline. Same with `sort`, `filter`, and other higher order functions (a *higher order function* is a function that takes or returns a function).

When we spoke about *pure functions*, we said they take 1 input to 1 output. Currying does exactly this: each single argument returns a new function expecting the remaining arguments. That, old sport, is 1 input to 1 output.

No matter if the output is another function - it qualifies as pure. We do allow more than one argument at a time, but this is seen as merely removing the extra `()`'s for convenience.

In Summary

Currying is handy and I very much enjoy working with curried functions on a daily basis. It is a tool for the belt that makes functional programming less verbose and tedious.

We can make new, useful functions on the fly simply by passing in a few arguments and as a bonus, we've retained the mathematical function definition despite multiple arguments.

Let's acquire another essential tool called `compose`.

Chapter 05: Coding by Composing

Exercises

Note about Exercises

Throughout the book, you might encounter an 'Exercises' section like this one. Exercises can be done directly in-browser provided you're reading from gitbook (recommended).

Note that, for all exercises of the book, you always have a handful of helper functions available in the global scope. Hence, anything that is defined in Appendix A, Appendix B and Appendix C is available for you! And, as if it wasn't enough, some exercises will also define functions specific to the problem they present; as a matter of fact, consider them available as well.

> Hint: you can submit your solution by doing `Ctrl + Enter` in the embedded editor!

Running Exercises on Your Machine (optional)

Should you prefer to do exercises directly in files using your own editor:

- clone the repository (`git clone git@github.com:MostlyAdequate/mostly-adequate-guide.git`)
- go in the *exercises* section (`cd mostly-adequate-guide/exercises`)
- install the necessary plumbing using npm (`npm install`)
- complete answers by modifying the files named *exercise_** in the corresponding chapter's folder
- run the correction with npm (e.g. `npm run ch04`)

Unit tests will run against your answers and provide hints in case of mistake. By the by, the answers to the exercises are available in files named *solution_**.

Let's Practice!

Exercise

Refactor to remove all arguments by partially applying the function.

```
// words :: String -> [String]
const words = str => split(' ', str);
```

Exercise

Refactor to remove all arguments by partially applying the functions.

```
// filterQs :: [String] -> [String]
const filterQs = xs => filter(x => x.match(/q/i), xs);
```

Considering the following function:

```
const keepHighest = (x, y) => (x >= y ? x : y);
```

Exercise

Refactor `max` to not reference any arguments using the helper function `keepHighest`.

```
// max :: [Number] -> Number
const max = xs => reduce((acc, x) => (x >= acc ? x : acc), -Infinity, xs);
```

Chapter 05: Coding by Composing

Functional Husbandry

Here's `compose`:

```
const compose = (...fns) => (...args) => fns.reduceRight((res, fn) => [fn.call(
```

... Don't be scared! This is the level-9000-super-Saiyan-form of *compose*. For the sake of reasoning, let's drop the variadic implementation and consider a simpler form that can compose two functions together. Once you get your head around that, you can push the abstraction further and consider it simply works for any number of functions (we could even prove that)! Here's a more friendly *compose* for you my dear readers:

```
const compose2 = (f, g) => x => f(g(x));
```

`f` and `g` are functions and `x` is the value being "piped" through them.

Composition feels like function husbandry. You, breeder of functions, select two with traits you'd like to combine and mash them together to spawn a brand new one. Usage is as follows:

```
const toUpperCase = x => x.toUpperCase();
const exclaim = x => `${x}!`;
const shout = compose(exclaim, toUpperCase);

shout('send in the clowns'); // "SEND IN THE CLOWNS!"
```

The composition of two functions returns a new function. This makes perfect sense: composing two units of some type (in this case function) should yield a new unit of that very type. You don't plug two legos together and get a lincoln log. There is a theory here, some underlying law that we will discover in due time.

In our definition of `compose`, the `g` will run before the `f`, creating a right to left flow of data. This is much more readable than nesting a bunch of function calls. Without compose, the above would read:

```
const shout = x => exclaim(toUpperCase(x));
```

Instead of inside to outside, we run right to left, which I suppose is a step in the left direction (boo!). Let's look at an example where sequence matters:

```
const head = x => x[0];
const reverse = reduce((acc, x) => [x, ...acc], []);
const last = compose(head, reverse);

last(['jumpkick', 'roundhouse', 'uppercut']); // 'uppercut'
```

`reverse` will turn the list around while `head` grabs the initial item. This results in an effective, albeit inefficient, `last` function. The sequence of functions in the composition should be apparent here. We could define a left to right version, however, we mirror the mathematical version much more closely as it stands. That's right, composition is straight from the math books. In fact, perhaps it's time to look at a property that holds for any composition.

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

Composition is associative, meaning it doesn't matter how you group two of them. So, should we choose to uppercase the string, we can write:

```
compose(toUpperCase, compose(head, reverse));
// or
compose(compose(toUpperCase, head), reverse);
```

Since it doesn't matter how we group our calls to `compose`, the result will be the same. That allows us to write a variadic compose and use it as follows:

```
// previously we'd have to write two composes, but since it's associative,
// we can give compose as many fn's as we like and let it decide how to group t
const arg = ['jumpkick', 'roundhouse', 'uppercut'];
const lastUpper = compose(toUpperCase, head, reverse);
const loudLastUpper = compose(exclaim, toUpperCase, head, reverse);

lastUpper(arg); // 'UPPERCUT'
loudLastUpper(arg); // 'UPPERCUT!'
```

Applying the associative property gives us this flexibility and peace of mind that the result will be equivalent. The slightly more complicated variadic definition is included with the support libraries for this book and is the normal definition you'll find in libraries like lodash, underscore, and ramda.

One pleasant benefit of associativity is that any group of functions can be extracted and bundled together in their very own composition. Let's play with refactoring our previous example:

```
const loudLastUpper = compose(exclaim, toUpperCase, head, reverse);

// -- or ----------------------------------------------------------------

const last = compose(head, reverse);
const loudLastUpper = compose(exclaim, toUpperCase, last);

// -- or ----------------------------------------------------------------

const last = compose(head, reverse);
const angry = compose(exclaim, toUpperCase);
const loudLastUpper = compose(angry, last);

// more variations...
```

There's no right or wrong answers - we're just plugging our legos together in whatever way we please. Usually it's best to group things in a reusable way like `last` and `angry`. If familiar with Fowler's "Refactoring", one might recognize this process as "extract function"...except without all the object state to worry about.

Pointfree

Pointfree style means never having to say your data. Excuse me. It means functions that never mention the data upon which they operate. First class functions, currying, and composition all play well together to create this style.

> Hint: Pointfree versions of `replace` & `toLowerCase` are defined in the Appendix C - Pointfree Utilities. Do not hesitate to have a peek!

```
// not pointfree because we mention the data: word
const snakeCase = word => word.toLowerCase().replace(/\s+/ig, '_');

// pointfree
const snakeCase = compose(replace(/\s+/ig, '_'), toLowerCase);
```

See how we partially applied `replace`? What we're doing is piping our data through each function of 1 argument. Currying allows us to prepare each function to just take its data, operate on it, and pass it along. Something else to notice is how we don't need the data to construct our function in the pointfree version, whereas in the pointful one, we must have our `word` available before anything else.

Let's look at another example.

```
// not pointfree because we mention the data: name
const initials = name => name.split(' ').map(compose(toUpperCase, head)).join(

// pointfree
// NOTE: we use 'intercalate' from the appendix instead of 'join' introduced ir
const initials = compose(intercalate('. '), map(compose(toUpperCase, head)), sp

initials('hunter stockton thompson'); // 'H. S. T'
```

Pointfree code can again, help us remove needless names and keep us concise and generic. Pointfree is a good litmus test for functional code as it lets us know we've got small functions that take input to output. One can't compose a while loop, for instance. Be warned, however, pointfree is a double-edged sword and can sometimes obfuscate intention. Not all functional code is pointfree and that is O.K. We'll shoot for it where we can and stick with normal functions otherwise.

Debugging

A common mistake is to compose something like `map`, a function of two arguments, without first partially applying it.

```
// wrong - we end up giving angry an array and we partially applied map with wh
const latin = compose(map, angry, reverse);

latin(['frog', 'eyes']); // error

// right - each function expects 1 argument.
const latin = compose(map(angry), reverse);

latin(['frog', 'eyes']); // ['EYES!', 'FROG!'])
```

If you are having trouble debugging a composition, we can use this helpful, but impure trace function to see what's going on.

```
const trace = curry((tag, x) => {
  console.log(tag, x);
  return x;
});

const dasherize = compose(
  intercalate('-'),
  toLower,
  split(' '),
  replace(/\s{2,}/ig, ' '),
);

dasherize('The world is a vampire');
// TypeError: Cannot read property 'apply' of undefined
```

Something is wrong here, let's `trace`

```
const dasherize = compose(
  intercalate('-'),
  toLower,
  trace('after split'),
  split(' '),
  replace(/\s{2,}/ig, ' '),
);

dasherize('The world is a vampire');
// after split [ 'The', 'world', 'is', 'a', 'vampire' ]
```

Ah! We need to `map` this `toLower` since it's working on an array.

```
const dasherize = compose(
  intercalate('-'),
  map(toLower),
  split(' '),
  replace(/\s{2,}/ig, ' '),
);

dasherize('The world is a vampire'); // 'the-world-is-a-vampire'
```

The `trace` function allows us to view the data at a certain point for debugging purposes. Languages like Haskell and PureScript have similar functions for ease of development.

Composition will be our tool for constructing programs and, as luck would have it, is backed by a powerful theory that ensures things will work out for us. Let's examine this theory.

Category Theory

Category theory is an abstract branch of mathematics that can formalize concepts from several different branches such as set theory, type theory, group theory, logic, and more. It primarily deals with objects, morphisms, and transformations, which mirrors programming quite closely. Here is a chart of the same concepts as viewed from each separate theory.

Types	Logic	Sets	Homotopy
A	proposition	set	space
$a : A$	proof	element	point
$B(x)$	predicate	family of sets	fibration
$b(x) : B(x)$	conditional proof	family of elements	section
$\mathbf{0}, \mathbf{1}$	$\bot, \top$	$\emptyset, \{\emptyset\}$	$\emptyset, *$
$A + B$	$A \vee B$	disjoint union	coproduct
$A \times B$	$A \wedge B$	set of pairs	product space
$A \to B$	$A \Rightarrow B$	set of functions	function space
$\sum_{(x:A)} B(x)$	$\exists_{x:A} B(x)$	disjoint sum	total space
$\prod_{(x:A)} B(x)$	$\forall_{x:A} B(x)$	product	space of sections
Id_A	equality $=$	$\{ (x, x) \mid x \in A \}$	path space A^I

Sorry, I didn't mean to frighten you. I don't expect you to be intimately familiar with all these concepts. My point is to show you how much duplication we have so you can see why category theory aims to unify these things.

In category theory, we have something called... a category. It is defined as a collection with the following components:

- A collection of objects
- A collection of morphisms
- A notion of composition on the morphisms
- A distinguished morphism called identity

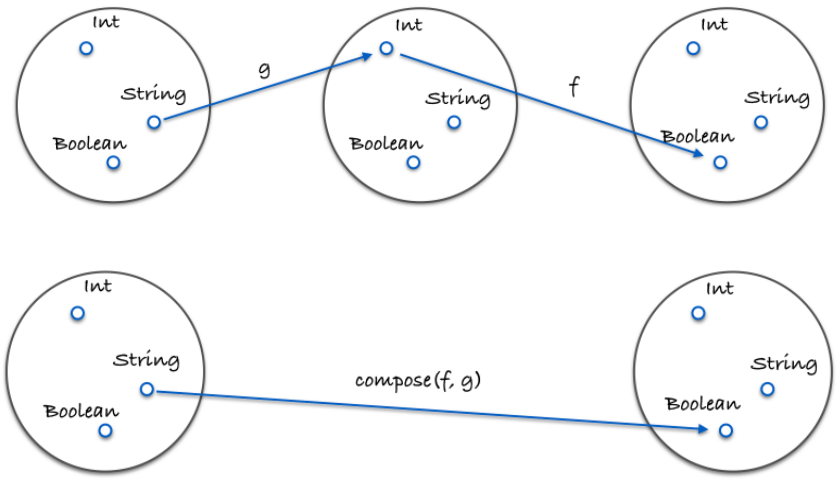
Category theory is abstract enough to model many things, but let's apply this to types and functions, which is what we care about at the moment.

A collection of objects The objects will be data types. For instance, `String`, `Boolean`, `Number`, `Object`, etc. We often view data types as sets of all the possible values. One could look at `Boolean` as the set of `[true, false]` and `Number` as the set of all possible numeric values. Treating types as sets is useful because we can use set theory to work with them.

A collection of morphisms The morphisms will be our standard every day pure functions.

A notion of composition on the morphisms This, as you may have guessed, is our brand new toy - `compose`. We've discussed that our `compose` function is associative which is no coincidence as it is a property that must hold for any composition in category theory.

Here is an image demonstrating composition:



Here is a concrete example in code:

```
const g = x => x.length;
const f = x => x === 4;
const isFourLetterWord = compose(f, g);
```

A distinguished morphism called identity Let's introduce a useful function called `id`. This function simply takes some input and spits it back at you. Take a look:

```
const id = x => x;
```

You might ask yourself "What in the bloody hell is that useful for?". We'll make extensive use of this function in the following chapters, but for now think of it as a function that can stand in for our value - a function masquerading as every day data.

`id` must play nicely with compose. Here is a property that always holds for every unary (unary: a one-argument function) function f:

```
// identity
compose(id, f) === compose(f, id) === f;
// true
```

Hey, it's just like the identity property on numbers! If that's not immediately clear, take some time with it. Understand the futility. We'll be seeing `id` used all over the place soon, but for now we see it's a function that acts as a stand in for a given value. This is quite useful when writing pointfree code.

So there you have it, a category of types and functions. If this is your first introduction, I imagine you're still a little fuzzy on what a category is and why it's useful. We will build upon this knowledge throughout the book. As of right now, in this chapter, on this line, you can at least see it as providing us with some wisdom regarding composition - namely, the associativity and identity properties.

What are some other categories, you ask? Well, we can define one for directed graphs with nodes being objects, edges being morphisms, and composition just being path concatenation. We can define with Numbers as objects and `>=` as morphisms (actually any partial or total order can be a category). There are heaps of categories, but for the purposes of this book, we'll only concern ourselves with the one defined above. We have sufficiently skimmed the surface and must move on.

In Summary

Composition connects our functions together like a series of pipes. Data will flow through our application as it must - pure functions are input to output after all, so breaking this chain would disregard output, rendering our software useless.

We hold composition as a design principle above all others. This is because it keeps our app simple and reasonable. Category theory will play a big part in app architecture, modelling side effects, and ensuring correctness.

We are now at a point where it would serve us well to see some of this in practice. Let's make an example application.

Chapter 06: Example Application

Exercises

In each following exercise, we'll consider Car objects with the following shape:

```
{
  name: 'Aston Martin One-77',
  horsepower: 750,
  dollar_value: 1850000,
  in_stock: true,
}
```

Exercise

Use `compose()` to rewrite the function below.

```
// isLastInStock :: [Car] -> Boolean
const isLastInStock = (cars) => {
  const lastCar = last(cars);
  return prop('in_stock', lastCar);
};
```

Considering the following function:

```
const average = xs => reduce(add, 0, xs) / xs.length;
```

Exercise

Use the helper function `average` to refactor `averageDollarValue` as a composition.

```
// averageDollarValue :: [Car] -> Int
const averageDollarValue = (cars) => {
  const dollarValues = map(c => c.dollar_value, cars);
  return average(dollarValues);
};
```

Exercise

Refactor `fastestCar` using `compose()` and other functions in pointfree-style. Hint, the `append` function may come in handy.

```
const fastestCar = (cars) => {
  const sorted = sortBy(car => car.horsepower, cars);
  const fastest = last(sorted);
  return concat(fastest.name, ' is the fastest');
};
```

Chapter 06: Example Application

Declarative Coding

We are going to switch our mindset. From here on out, we'll stop telling the computer how to do its job and instead write a specification of what we'd like as a result. I'm sure you'll find it much less stressful than trying to micromanage everything all the time.

Declarative, as opposed to imperative, means that we will write expressions, as opposed to step by step instructions.

Think of SQL. There is no "first do this, then do that". There is one expression that specifies what we'd like from the database. We don't decide how to do the work, it does. When the database is upgraded and the SQL engine optimized, we don't have to change our query. This is because there are many ways to interpret our specification and achieve the same result.

For some folks, myself included, it's hard to grasp the concept of declarative coding at first so let's point out a few examples to get a feel for it.

```
// imperative
const makes = [];
for (let i = 0; i < cars.length; i += 1) {
  makes.push(cars[i].make);
}

// declarative
const makes = cars.map(car => car.make);
```

The imperative loop must first instantiate the array. The interpreter must evaluate this statement before moving on. Then it directly iterates through the list of cars, manually increasing a counter and showing its bits and pieces to us in a vulgar display of explicit iteration.

The `map` version is one expression. It does not require any order of evaluation. There is much freedom here for how the map function iterates and how the returned array may be assembled. It specifies *what*, not *how*. Thus, it wears the shiny declarative sash.

In addition to being clearer and more concise, the map function may be optimized at will and our precious application code needn't change.

For those of you who are thinking "Yes, but it's much faster to do the imperative loop", I suggest you educate yourself on how the JIT optimizes your code. Here's a terrific video that may shed some light

Here is another example.

```
// imperative
const authenticate = (form) => {
  const user = toUser(form);
  return logIn(user);
};

// declarative
const authenticate = compose(logIn, toUser);
```

Though there's nothing necessarily wrong with the imperative version, there is still an encoded step-by-step evaluation baked in. The `compose` expression simply states a fact: Authentication is the composition of `toUser` and `logIn`. Again, this leaves wiggle room for support code changes and results in our application code being a high level specification.

In the example above, the order of evaluation is specified (`toUser` must be called before `logIn`), but there are many scenarios where the order is not important, and this is easily specified with declarative coding (more on this later).

Because we don't have to encode the order of evaluation, declarative coding lends itself to parallel computing. This coupled with pure functions is why FP is a good option for the parallel future - we don't really need to do anything special to achieve parallel/concurrent systems.

A Flickr of Functional Programming

We will now build an example application in a declarative, composable way. We'll still cheat and use side effects for now, but we'll keep them minimal and separate from our pure codebase. We are going to build a browser widget that sucks in flickr images and displays them. Let's start by scaffolding the app. Here's the html:

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Flickr App</title>
  </head>
  <body>
    <main id="js-main" class="main"></main>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/require.js/2.2.0/requi
    <script src="main.js"></script>
  </body>
</html>
```

And here's the main.js skeleton:

```
const CDN = s => `https://cdnjs.cloudflare.com/ajax/libs/${s}`;
const ramda = CDN('ramda/0.21.0/ramda.min');
const jquery = CDN('jquery/3.0.0-rc1/jquery.min');

requirejs.config({ paths: { ramda, jquery } });
requirejs(['jquery', 'ramda'], ($, { compose, curry, map, prop }) => {
  // app goes here
});
```

We're pulling in ramda instead of lodash or some other utility library. It includes `compose`, `curry`, and more. I've used requirejs, which may seem like overkill, but we'll be using it throughout the book and consistency is key.

Now that that's out of the way, on to the spec. Our app will do 4 things.

1. Construct a url for our particular search term
2. Make the flickr api call
3. Transform the resulting json into html images
4. Place them on the screen

There are 2 impure actions mentioned above. Do you see them? Those bits about getting data from the flickr api and placing it on the screen. Let's define those first so we can quarantine them. Also, I'll add our nice `trace` function for easy debugging.

```
const Impure = {
  getJSON: curry((callback, url) => $.getJSON(url, callback)),
  setHtml: curry((sel, html) => $(sel).html(html)),
  trace: curry((tag, x) => { console.log(tag, x); return x; }),
};
```

Here we've simply wrapped jQuery's methods to be curried and we've swapped the arguments to a more favorable position. I've namespaced them with `Impure` so we know these are dangerous functions. In a future example, we will make these two functions pure.

Next we must construct a url to pass to our `Impure.getJSON` function.

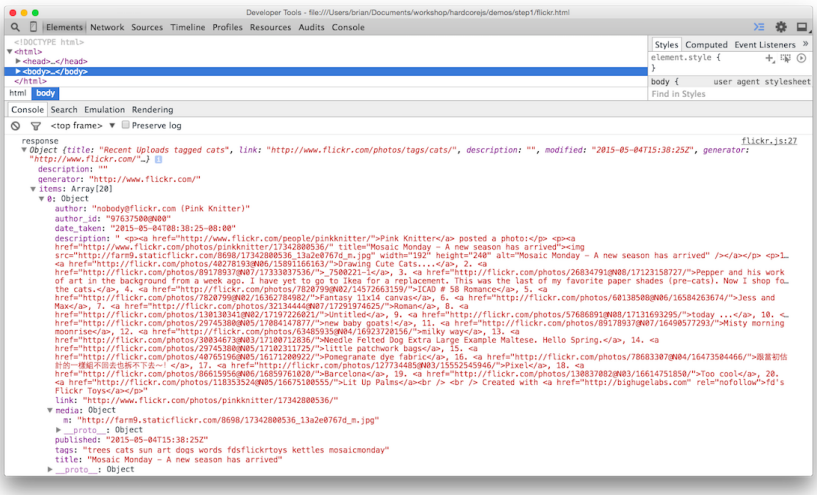
```
const host = 'api.flickr.com';
const path = '/services/feeds/photos_public.gne';
const query = t => `?tags=${t}&format=json&jsoncallback=?`;
const url = t => `https://${host}${path}${query(t)}`;
```

There are fancy and overly complex ways of writing `url` pointfree using monoids(we'll learn about these later) or combinators. We've chosen to stick with a readable version and assemble this string in the normal pointful fashion.

Let's write an app function that makes the call and places the contents on the screen.

```
const app = compose(Impure.getJSON(Impure.trace('response')), url);
app('cats');
```

This calls our `url` function, then passes the string to our `getJSON` function, which has been partially applied with `trace`. Loading the app will show the response from the api call in the console.



We'd like to construct images out of this json. It looks like the `mediaUrls` are buried in `items` then each `media`'s `m` property.

Anyhow, to get at these nested properties we can use a nice universal getter function from ramda called `prop`. Here's a homegrown version so you can see what's happening:

```
const prop = curry((property, object) => object[property]);
```

It's quite dull actually. We just use `[]` syntax to access a property on whatever object. Let's use this to get at our `mediaUrls`.

```
const mediaUrl = compose(prop('m'), prop('media'));
const mediaUrls = compose(map(mediaUrl), prop('items'));
```

Once we gather the `items`, we must `map` over them to extract each media url. This results in a nice array of `mediaUrls`. Let's hook this up to our app and print them on the screen.

```
const render = compose(Impure.setHtml('#js-main'), mediaUrls);
const app = compose(Impure.getJSON(render), url);
```

All we've done is make a new composition that will call our `mediaUrls` and set the `<main>` html with them. We've replaced the `trace` call with `render` now that we have something to render besides raw json. This will crudely display our

`mediaUrls` within the body.

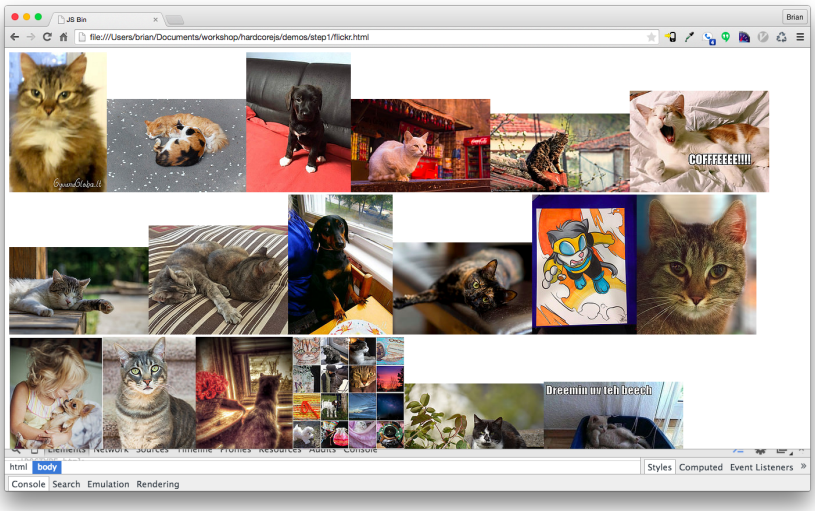
Our final step is to turn these `mediaUrls` into bonafide `images`. In a bigger application, we'd use a template/dom library like Handlebars or React. For this application though, we only need an img tag so let's stick with jQuery.

```
const img = src => $('<img />', { src });
```

jQuery's `html` method will accept an array of tags. We only have to transform our mediaUrls into images and send them along to `setHtml`.

```
const images = compose(map(img), mediaUrls);
const render = compose(Impure.setHtml('#js-main'), images);
const app = compose(Impure.getJSON(render), url);
```

And we're done!



Here is the finished script:

```
const CDN = s => `https://cdnjs.cloudflare.com/ajax/libs/${s}`;
const ramda = CDN('ramda/0.21.0/ramda.min');
const jquery = CDN('jquery/3.0.0-rc1/jquery.min');

requirejs.config({ paths: { ramda, jquery } });
require(['jquery', 'ramda'], ($, { compose, curry, map, prop }) => {
  // -- Utils ------------------------------------------------------
  const Impure = {
    trace: curry((tag, x) => { console.log(tag, x); return x; }), // eslint-dis
    getJSON: curry((callback, url) => $.getJSON(url, callback)),
    setHtml: curry((sel, html) => $(sel).html(html)),
  };

  // -- Pure -------------------------------------------------------
  const host = 'api.flickr.com';
  const path = '/services/feeds/photos_public.gne';
  const query = t => `?tags=${t}&format=json&jsoncallback=?`;
  const url = t => `https://${host}${path}${query(t)}`;

  const img = src => $('<img />', { src });
  const mediaUrl = compose(prop('m'), prop('media'));
  const mediaUrls = compose(map(mediaUrl), prop('items'));
  const images = compose(map(img), mediaUrls);

  // -- Impure -----------------------------------------------------
  const render = compose(Impure.setHtml('#js-main'), images);
  const app = compose(Impure.getJSON(render), url);

  app('cats');
});
```

Now look at that. A beautifully declarative specification of what things are, not how they come to be. We now view each line as an equation with properties that hold. We can use these properties to reason about our application and refactor.

A Principled Refactor

There is an optimization available - we map over each item to turn it into a media url, then we map again over those mediaUrls to turn them into img tags. There is a law regarding map and composition:

```
// map's composition law
compose(map(f), map(g)) === map(compose(f, g));
```

We can use this property to optimize our code. Let's have a principled refactor.

```
// original code
const mediaUrl = compose(prop('m'), prop('media'));
const mediaUrls = compose(map(mediaUrl), prop('items'));
const images = compose(map(img), mediaUrls);
```

Let's line up our maps. We can inline the call to `mediaUrls` in `images` thanks to equational reasoning and purity.

```
const mediaUrl = compose(prop('m'), prop('media'));
const images = compose(map(img), map(mediaUrl), prop('items'));
```

Now that we've lined up our `map`s we can apply the composition law.

```
/*
compose(map(f), map(g)) === map(compose(f, g));
compose(map(img), map(mediaUrl)) === map(compose(img, mediaUrl));
*/

const mediaUrl = compose(prop('m'), prop('media'));
const images = compose(map(compose(img, mediaUrl)), prop('items'));
```

Now the bugger will only loop once while turning each item into an img. Let's just make it a little more readable by extracting the function out.

```
const mediaUrl = compose(prop('m'), prop('media'));
const mediaToImg = compose(img, mediaUrl);
const images = compose(map(mediaToImg), prop('items'));
```

In Summary

We have seen how to put our new skills into use with a small, but real world app. We've used our mathematical framework to reason about and refactor our code. But what about error handling and code branching? How can we make the whole application pure instead of merely namespacing destructive functions? How can we make our app safer and more expressive? These are the questions we will tackle in part 2.

Chapter 07: Hindley-Milner and Me

Chapter 07: Hindley-Milner and Me

What's Your Type?

If you're new to the functional world, it won't be long before you find yourself knee deep in type signatures. Types are the meta language that enables people from all different backgrounds to communicate succinctly and effectively. For the most part, they're written with a system called "Hindley-Milner", which we'll be examining together in this chapter.

When working with pure functions, type signatures have an expressive power to which the English language cannot hold a candle. These signatures whisper in your ear the intimate secrets of a function. In a single, compact line, they expose behaviour and intention. We can derive "free theorems" from them. Types can be inferred so there's no need for explicit type annotations. They can be tuned to fine point precision or left general and abstract. They are not only useful for compile time checks, but also turn out to be the best possible documentation available. Type signatures thus play an important part in functional programming - much more than you might first expect.

JavaScript is a dynamic language, but that does not mean we avoid types all together. We're still working with strings, numbers, booleans, and so on. It's just that there isn't any language level integration so we hold this information in our heads. Not to worry, since we're using signatures for documentation, we can use comments to serve our purpose.

There are type checking tools available for JavaScript such as Flow or the typed dialect, TypeScript. The aim of this book is to equip one with the tools to write functional code so we'll stick with the standard type system used across FP languages.

Tales from the Cryptic

From the dusty pages of math books, across the vast sea of white papers, amongst casual Saturday morning blog posts, down into the source code itself, we find Hindley-Milner type signatures. The system is quite simple, but warrants a quick explanation and some practice to fully absorb the little language.

```
// capitalize :: String -> String
const capitalize = s => toUpperCase(head(s)) + toLowerCase(tail(s));


capitalize('smurf'); // 'Smurf'
```

Here, `capitalize` takes a `String` and returns a `String`. Never mind the implementation, it's the type signature we're interested in.

In HM, functions are written as `a -> b` where `a` and `b` are variables for any type. So the signatures for `capitalize` can be read as "a function from `String` to `String`". In other words, it takes a `String` as its input and returns a `String` as its output.

Let's look at some more function signatures:

```
// strLength :: String -> Number
const strLength = s => s.length;

// join :: String -> [String] -> String
const join = curry((what, xs) => xs.join(what));

// match :: Regex -> String -> [String]
const match = curry((reg, s) => s.match(reg));

// replace :: Regex -> String -> String -> String
const replace = curry((reg, sub, s) => s.replace(reg, sub));
```

`strLength` is the same idea as before: we take a `String` and return you a `Number`.

The others might perplex you at first glance. Without fully understanding the details, you could always just view the last type as the return value. So for `match` you can interpret as: It takes a `Regex` and a `String` and returns you `[String]`. But an interesting thing is going on here that I'd like to take a moment to explain if I may.

For `match` we are free to group the signature like so:

```
// match :: Regex -> (String -> [String])
const match = curry((reg, s) => s.match(reg));
```

Ah yes, grouping the last part in parenthesis reveals more information. Now it is seen as a function that takes a `Regex` and returns us a function from `String` to `[String]`. Because of currying, this is indeed the case: give it a `Regex` and we get a function back waiting for its `String` argument. Of course, we don't have to think of it this way, but it is good to understand why the last type is the one returned.

```
// match :: Regex -> (String -> [String])
// onHoliday :: String -> [String]
const onHoliday = match(/holiday/ig);
```

Each argument pops one type off the front of the signature. `onHoliday` is `match` that already has a `Regex`.

```
// replace :: Regex -> (String -> (String -> String))
const replace = curry((reg, sub, s) => s.replace(reg, sub));
```

As you can see with the full parenthesis on `replace`, the extra notation can get a little noisy and redundant so we simply omit them. We can give all the arguments at once if we choose so it's easier to just think of it as: `replace` takes a `Regex`, a `String`, another `String` and returns you a `String`.

A few last things here:

```
// id :: a -> a
const id = x => x;

// map :: (a -> b) -> [a] -> [b]
const map = curry((f, xs) => xs.map(f));
```

The `id` function takes any old type `a` and returns something of the same type `a`. We're able to use variables in types just like in code. Variable names like `a` and `b` are convention, but they are arbitrary and can be replaced with whatever name you'd like. If they are the same variable, they have to be the same type. That's an important rule so let's reiterate: `a -> b` can be any type `a` to any type `b`, but `a -> a` means it has to be the same type. For example, `id` may be `String -> String` or `Number -> Number`, but not `String -> Bool`.

`map` similarly uses type variables, but this time we introduce `b` which may or may not be the same type as `a`. We can read it as: `map` takes a function from any type `a` to the same or different type `b`, then takes an array of `a`'s and results in an array of `b`'s.

Hopefully, you've been overcome by the expressive beauty in this type signature. It literally tells us what the function does almost word for word. It's given a function from `a` to `b`, an array of `a`, and it delivers us an array of `b`. The only sensible thing for it to do is call the bloody function on each `a`. Anything else would be a bold face lie.

Being able to reason about types and their implications is a skill that will take you far in the functional world. Not only will papers, blogs, docs, etc, become more digestible, but the signature itself will practically lecture you on its functionality. It takes practice to become a fluent reader, but if you stick with it, heaps of information will become available to you sans RTFMing.

Here's a few more just to see if you can decipher them on your own.

```
// head :: [a] -> a
const head = xs => xs[0];

// filter :: (a -> Bool) -> [a] -> [a]
const filter = curry((f, xs) => xs.filter(f));

// reduce :: ((b, a) -> b) -> b -> [a] -> b
const reduce = curry((f, x, xs) => xs.reduce(f, x));
```

`reduce` is perhaps, the most expressive of all. It's a tricky one, however, so don't feel inadequate should you struggle with it. For the curious, I'll try to explain in English though working through the signature on your own is much more instructive.

Ahem, here goes nothing....looking at the signature, we see the first argument is a function that expects `b` and `a`, and produces a `b`. Where might it get these `a` s and `b` s? Well, the following arguments in the signature are a `b` and an array of `a` s so we can only assume that the `b` and each of those `a` s will be fed in. We also see that the result of the function is a `b` so the thinking here is our final incantation of the passed in function will be our output value. Knowing what reduce does, we can state that the above investigation is accurate.

Narrowing the Possibility

Once a type variable is introduced, there emerges a curious property called *parametricity*. This property states that a function will *act on all types in a uniform manner*. Let's investigate:

```
// head :: [a] -> a
```

Looking at `head`, we see that it takes `[a]` to `a`. Besides the concrete type `array`, it has no other information available and, therefore, its functionality is limited to working on the array alone. What could it possibly do with the variable `a` if it knows nothing about it? In other words, `a` says it cannot be a *specific* type, which means it can be *any* type, which leaves us with a function that must work uniformly for *every* conceivable type. This is what *parametricity* is all about. Guessing at the implementation, the only reasonable assumptions are that it takes the first, last, or a random element from that array. The name `head` should tip us off.

Here's another one:

```
// reverse :: [a] -> [a]
```

From the type signature alone, what could `reverse` possibly be up to? Again, it cannot do anything specific to `a`. It cannot change `a` to a different type or we'd introduce a `b`. Can it sort? Well, no, it wouldn't have enough information to sort

every possible type. Can it re-arrange? Yes, I suppose it can do that, but it has to do so in exactly the same predictable way. Another possibility is that it may decide to remove or duplicate an element. In any case, the point is, the possible behaviour is massively narrowed by its polymorphic type.

This narrowing of possibility allows us to use type signature search engines like Hoogle to find a function we're after. The information packed tightly into a signature is quite powerful indeed.

Free as in Theorem

Besides deducing implementation possibilities, this sort of reasoning gains us *free theorems*. What follows are a few random example theorems lifted directly from Wadler's paper on the subject.

```
// head :: [a] -> a
compose(f, head) === compose(head, map(f));

// filter :: (a -> Bool) -> [a] -> [a]
compose(map(f), filter(compose(p, f))) === compose(filter(p), map(f));
```

You don't need any code to get these theorems, they follow directly from the types. The first one says that if we get the `head` of our array, then run some function `f` on it, that is equivalent to, and incidentally, much faster than, if we first `map(f)` over every element then take the `head` of the result.

You might think, well that's just common sense. But last I checked, computers don't have common sense. Indeed, they must have a formal way to automate these kind of code optimizations. Maths has a way of formalizing the intuitive, which is helpful amidst the rigid terrain of computer logic.

The `filter` theorem is similar. It says that if we compose `f` and `p` to check which should be filtered, then actually apply the `f` via `map` (remember `filter` will not transform the elements - its signature enforces that `a` will not be touched), it will always be equivalent to mapping our `f` then filtering the result with the `p` predicate.

These are just two examples, but you can apply this reasoning to any polymorphic type signature and it will always hold. In JavaScript, there are some tools available to declare rewrite rules. One might also do this via the `compose` function itself. The fruit is low hanging and the possibilities are endless.

Constraints

One last thing to note is that we can constrain types to an interface.

```
// sort :: Ord a => [a] -> [a]
```

What we see on the left side of our fat arrow here is the statement of a fact: `a` must be an `Ord`. Or in other words, `a` must implement the `Ord` interface. What is `Ord` and where did it come from? In a typed language it would be a defined interface that says we can order the values. This not only tells us more about the `a` and what our `sort` function is up to, but also restricts the domain. We call these interface declarations *type constraints*.

```
// assertEqual :: (Eq a, Show a) => a -> a -> Assertion
```

Here, we have two constraints: `Eq` and `Show`. Those will ensure that we can check equality of our `a`s and print the difference if they are not equal.

We'll see more examples of constraints and the idea should take more shape in later chapters.

In Summary

Hindley-Milner type signatures are ubiquitous in the functional world. Though they are simple to read and write, it takes time to master the technique of understanding programs through signatures alone. We will add type signatures to each line of code from here on out.

Chapter 08: Tupperware

Chapter 08: Tupperware

The Mighty Container

We've seen how to write programs which pipe data through a series of pure functions. They are declarative specifications of behaviour. But what about control flow, error handling, asynchronous actions, state and, dare I say, effects?! In this chapter, we will discover the foundation upon which all of these helpful abstractions are built.

First we will create a container. This container must hold any type of value; a ziplock that holds only tapioca pudding is rarely useful. It will be an object, but we will not give it properties and methods in the OO sense. No, we will treat it like a treasure chest - a special box that cradles our valuable data.

```
class Container {
  constructor(x) {
    this.$value = x;
  }

  static of(x) {
    return new Container(x);
  }
}
```

Here is our first container. We've thoughtfully named it `Container`. We will use `Container.of` as a constructor which saves us from having to write that awful `new` keyword all over the place. There's more to the `of` function than meets the eye, but for now, think of it as the proper way to place values into our container.

Let's examine our brand new box...

```
Container.of(3);
// Container(3)

Container.of('hotdogs');
// Container("hotdogs")

Container.of(Container.of({ name: 'yoda' }));
// Container(Container({ name: 'yoda' }))
```

If you are using node, you will see `{$value: x}` even though we've got ourselves a `Container(x)`. Chrome will output the type properly, but no matter; as long as we understand what a `Container` looks like, we'll be fine. In some environments you can overwrite the `inspect` method if you'd like, but we will not be so thorough. For this book, we will write the conceptual output as if we'd overwritten `inspect` as it's much more instructive than `{$value: x}` for pedagogical as well as aesthetic reasons.

Let's make a few things clear before we move on:

- `Container` is an object with one property. Lots of containers just hold one thing, though they aren't limited to one. We've arbitrarily named its property `$value`.
- The `$value` cannot be one specific type or our `Container` would hardly live up to the name.
- Once data goes into the `Container` it stays there. We *could* get it out by using `.$value`, but that would defeat the purpose.

The reasons we're doing this will become clear as a mason jar, but for now, bear with me.

My First Functor

Once our value, whatever it may be, is in the container, we'll need a way to run functions on it.

```
// (a -> b) -> Container a -> Container b
Container.prototype.map = function (f) {
  return Container.of(f(this.$value));
};
```

Why, it's just like Array's famous `map`, except we have `Container a` instead of `[a]`. And it works essentially the same way:

```
Container.of(2).map(two => two + 2);
// Container(4)

Container.of('flamethrowers').map(s => s.toUpperCase());
// Container('FLAMETHROWERS')

Container.of('bombs').map(append(' away')).map(prop('length'));
// Container(10)
```

We can work with our value without ever having to leave the `Container`. This is a remarkable thing. Our value in the `Container` is handed to the `map` function so we can fuss with it and afterward, returned to its `Container` for safe keeping. As a result of never leaving the `Container`, we can continue to `map` away, running functions as we please. We can even change the type as we go along as demonstrated in the latter of the three examples.

Wait a minute, if we keep calling `map`, it appears to be some sort of composition! What mathematical magic is at work here? Well chaps, we've just discovered *Functors*.

> A Functor is a type that implements `map` and obeys some laws

Yes, *Functor* is simply an interface with a contract. We could have just as easily named it *Mappable*, but now, where's the *fun* in that? Functors come from category theory and we'll look at the maths in detail toward the end of the chapter, but for now, let's work on intuition and practical uses for this bizarrely named interface.

What reason could we possibly have for bottling up a value and using `map` to get at it? The answer reveals itself if we choose a better question: What do we gain from asking our container to apply functions for us? Well, abstraction of function application. When we `map` a function, we ask the container type to run it for us. This is a very powerful concept, indeed.

Schrödinger's Maybe

`Container` is fairly boring. In fact, it is usually called `Identity` and has about the same impact as our `id` function (again there is a mathematical connection we'll look at when the time is right). However, there are other functors, that is, container-like types that have a proper `map` function, which can provide useful behaviour whilst mapping. Let's define one now.

> A complete implementation is given in the Appendix B

```
class Maybe {
  static of(x) {
    return new Maybe(x);
  }

  get isNothing() {
    return this.$value === null || this.$value === undefined;
  }

  constructor(x) {
    this.$value = x;
  }

  map(fn) {
    return this.isNothing ? this : Maybe.of(fn(this.$value));
  }

  inspect() {
    return this.isNothing ? 'Nothing' : `Just(${inspect(this.$value)})`;
  }
}
```

Now, `Maybe` looks a lot like `Container` with one minor change: it will first check to see if it has a value before calling the supplied function. This has the effect of side stepping those pesky nulls as we `map` (Note that this implementation is simplified for teaching).

```
Maybe.of('Malkovich Malkovich').map(match(/a/ig));
// Just(True)

Maybe.of(null).map(match(/a/ig));
// Nothing

Maybe.of({ name: 'Boris' }).map(prop('age')).map(add(10));
// Nothing

Maybe.of({ name: 'Dinah', age: 14 }).map(prop('age')).map(add(10));
// Just(24)
```

Notice our app doesn't explode with errors as we map functions over our null values. This is because `Maybe` will take care to check for a value each and every time it applies a function.

This dot syntax is perfectly fine and functional, but for reasons mentioned in Part 1, we'd like to maintain our pointfree style. As it happens, `map` is fully equipped to delegate to whatever functor it receives:

```
// map :: Functor f => (a -> b) -> f a -> f b
const map = curry((f, anyFunctor) => anyFunctor.map(f));
```

This is delightful as we can carry on with composition per usual and `map` will work as expected. This is the case with ramda's `map` as well. We'll use dot notation when it's instructive and the pointfree version when it's convenient. Did you notice that? I've sneakily introduced extra notation into our type signature. The `Functor f =>` tells us that `f` must be a Functor. Not that difficult, but I felt I should mention it.

Use Cases

In the wild, we'll typically see `Maybe` used in functions which might fail to return a result.

```
// safeHead :: [a] -> Maybe(a)
const safeHead = xs => Maybe.of(xs[0]);

// streetName :: Object -> Maybe String
const streetName = compose(map(prop('street')), safeHead, prop('addresses'));

streetName({ addresses: [] });
// Nothing

streetName({ addresses: [{ street: 'Shady Ln.', number: 4201 }] });
// Just('Shady Ln.')
```

`safeHead` is like our normal `head`, but with added type safety. A curious thing happens when `Maybe` is introduced into our code; we are forced to deal with those sneaky `null` values. The `safeHead` function is honest and up front about its possible failure - there's really nothing to be ashamed of - and so it returns a `Maybe` to inform us of this matter. We are more than merely *informed*, however, because we are forced to `map` to get at the value we want since it is tucked away inside the `Maybe` object. Essentially, this is a `null` check enforced by the `safeHead` function itself. We can now sleep better at night knowing a `null` value won't rear its ugly, decapitated head when we least expect it. APIs like this will upgrade a flimsy application from paper and tacks to wood and nails. They will guarantee safer software.

Sometimes a function might return a `Nothing` explicitly to signal failure. For instance:

```
// withdraw :: Number -> Account -> Maybe(Account)
const withdraw = curry((amount, { balance }) =>
  Maybe.of(balance >= amount ? { balance: balance - amount } : null));

// This function is hypothetical, not implemented here... nor anywhere else.
// updateLedger :: Account -> Account
const updateLedger = account => account;

// remainingBalance :: Account -> String
const remainingBalance = ({ balance }) => `Your balance is $${balance}`;

// finishTransaction :: Account -> String
const finishTransaction = compose(remainingBalance, updateLedger);



// getTwenty :: Account -> Maybe(String)
const getTwenty = compose(map(finishTransaction), withdraw(20));

getTwenty({ balance: 200.00 });
// Just('Your balance is $180')

getTwenty({ balance: 10.00 });
// Nothing
```

`withdraw` will tip its nose at us and return `Nothing` if we're short on cash. This function also communicates its fickleness and leaves us no choice, but to `map` everything afterwards. The difference is that the `null` was intentional here. Instead of a `Just('..')`, we get the `Nothing` back to signal failure and our application effectively halts in its tracks. This is important to note: if the `withdraw` fails, then `map` will sever the rest of our computation since it doesn't ever run the mapped functions, namely `finishTransaction`. This is precisely the intended behaviour as we'd prefer not to update our ledger or show a new balance if we hadn't successfully withdrawn funds.

Releasing the Value

One thing people often miss is that there will always be an end of the line; some effecting function that sends JSON along, or prints to the screen, or alters our filesystem, or what have you. We cannot deliver the output with `return`, we must run some function or another to send it out into the world. We can phrase it like a Zen Buddhist koan: "If a program has no observable effect, does it even run?". Does it run correctly for its own satisfaction? I suspect it merely burns some cycles and goes back to sleep...

Our application's job is to retrieve, transform, and carry that data along until it's time to say goodbye and the function which does so may be mapped, thus the value needn't leave the warm womb of its container. Indeed, a common error is to

try to remove the value from our `Maybe` one way or another as if the possible value inside will suddenly materialize and all will be forgiven. We must understand it may be a branch of code where our value is not around to live up to its destiny. Our code, much like Schrödinger's cat, is in two states at once and should maintain that fact until the final function. This gives our code a linear flow despite the logical branching.

There is, however, an escape hatch. If we would rather return a custom value and continue on, we can use a little helper called `maybe`.

```
// maybe :: b -> (a -> b) -> Maybe a -> b
const maybe = curry((v, f, m) => {
  if (m.isNothing) {
    return v;
  }

  return f(m.$value);
});

// getTwenty :: Account -> String
const getTwenty = compose(maybe('You\'re broke!', finishTransaction), withdraw(

getTwenty({ balance: 200.00 });
// 'Your balance is $180.00'

getTwenty({ balance: 10.00 });
// 'You\'re broke!'
```

We will now either return a static value (of the same type that `finishTransaction` returns) or continue on merrily finishing up the transaction sans `Maybe`. With `maybe`, we are witnessing the equivalent of an `if/else` statement whereas with `map`, the imperative analog would be: `if (x !== null) { return f(x) }`.

The introduction of `Maybe` can cause some initial discomfort. Users of Swift and Scala will know what I mean as it's baked right into the core libraries under the guise of `Option(al)`. When pushed to deal with `null` checks all the time (and there are times we know with absolute certainty the value exists), most people can't help but feel it's a tad laborious. However, with time, it will become second nature and you'll likely appreciate the safety. After all, most of the time it will prevent cut corners and save our hides.

Writing unsafe software is like taking care to paint each egg with pastels before hurling it into traffic; like building a retirement home with materials warned against by three little pigs. It will do us well to put some safety into our functions and `Maybe` helps us do just that.

I'd be remiss if I didn't mention that the "real" implementation will split `Maybe` into two types: one for something and the other for nothing. This allows us to obey parametricity in `map` so values like `null` and `undefined` can still be mapped

over and the universal qualification of the value in a functor will be respected. You'll often see types like `Some(x) / None` or `Just(x) / Nothing` instead of a `Maybe` that does a `null` check on its value.

Pure Error Handling

It may come as a shock, but `throw/catch` is not very pure. When an error is thrown, instead of returning an output value, we sound the alarms! The function attacks, spewing thousands of 0s and 1s like shields and spears in an electric battle against our intruding input. With our new friend `Either`, we can do better than to declare war on input, we can respond with a polite message. Let's take a look:

> A complete implementation is given in the Appendix B

```
class Either {
  static of(x) {
    return new Right(x);
  }

  constructor(x) {
    this.$value = x;
  }
}

class Left extends Either {
  map(f) {
    return this;
  }

  inspect() {
    return `Left(${inspect(this.$value)})`;
  }
}

class Right extends Either {
  map(f) {
    return Either.of(f(this.$value));
  }

  inspect() {
    return `Right(${inspect(this.$value)})`;
  }
}

const left = x => new Left(x);
```

`Left` and `Right` are two subclasses of an abstract type we call `Either`. I've skipped the ceremony of creating the `Either` superclass as we won't ever use it, but it's good to be aware. Now then, there's nothing new here besides the two types. Let's see how they act:

```
Either.of('rain').map(str => `b${str}`);
// Right('brain')

left('rain').map(str => `It's gonna ${str}, better bring your umbrella!`);
// Left('rain')

Either.of({ host: 'localhost', port: 80 }).map(prop('host'));
// Right('localhost')

left('rolls eyes...').map(prop('host'));
// Left('rolls eyes...')
```

`Left` is the teenagery sort and ignores our request to `map` over it. `Right` will work just like `Container` (a.k.a Identity). The power comes from the ability to embed an error message within the `Left`.

Suppose we have a function that might not succeed. How about we calculate an age from a birth date. We could use `Nothing` to signal failure and branch our program, however, that doesn't tell us much. Perhaps, we'd like to know why it failed. Let's write this using `Either`.

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

// getAge :: Date -> User -> Either(String, Number)
const getAge = curry((now, user) => {
  const birthDate = moment(user.birthDate, 'YYYY-MM-DD');

  return birthDate.isValid()
    ? Either.of(now.diff(birthDate, 'years'))
    : left('Birth date could not be parsed');
});

getAge(moment(), { birthDate: '2005-12-12' });
// Right(9)

getAge(moment(), { birthDate: 'July 4, 2001' });
// Left('Birth date could not be parsed')
```

Now, just like `Nothing`, we are short-circuiting our app when we return a `Left`. The difference, is now we have a clue as to why our program has derailed. Something to notice is that we return `Either(String, Number)`, which holds a `String` as its left value and a `Number` as its `Right`. This type signature is a bit informal as we haven't taken the time to define an actual `Either` superclass, however, we learn a lot from the type. It informs us that we're either getting an error message or the age back.

```
// fortune :: Number -> String
const fortune = compose(concat('If you survive, you will be '), toString, add(1

// zoltar :: User -> Either(String, _)
const zoltar = compose(map(console.log), map(fortune), getAge(moment()));

zoltar({ birthDate: '2005-12-12' });
// 'If you survive, you will be 10'
// Right(undefined)

zoltar({ birthDate: 'balloons!' });
// Left('Birth date could not be parsed')
```

When the `birthDate` is valid, the program outputs its mystical fortune to the screen for us to behold. Otherwise, we are handed a `Left` with the error message plain as day though still tucked away in its container. That acts just as if we'd thrown an error, but in a calm, mild manner fashion as opposed to losing its temper and screaming like a child when something goes wrong.

In this example, we are logically branching our control flow depending on the validity of the birth date, yet it reads as one linear motion from right to left rather than climbing through the curly braces of a conditional statement. Usually, we'd move the `console.log` out of our `zoltar` function and `map` it at the time of calling, but it's helpful to see how the `Right` branch differs. We use `_` in the right branch's type signature to indicate it's a value that should be ignored (In some browsers you have to use `console.log.bind(console)` to use it first class).

I'd like to take this opportunity to point out something you may have missed: `fortune`, despite its use with `Either` in this example, is completely ignorant of any functors milling about. This was also the case with `finishTransaction` in the previous example. At the time of calling, a function can be surrounded by `map`, which transforms it from a non-functory function to a functory one, in informal terms. We call this process *lifting*. Functions tend to be better off working with normal data types rather than container types, then *lifted* into the right container as deemed necessary. This leads to simpler, more reusable functions that can be altered to work with any functor on demand.

`Either` is great for casual errors like validation as well as more serious, stop the show errors like missing files or broken sockets. Try replacing some of the `Maybe` examples with `Either` to give better feedback.

Now, I can't help but feel I've done `Either` a disservice by introducing it as merely a container for error messages. It captures logical disjunction (a.k.a `||`) in a type. It also encodes the idea of a *Coproduct* from category theory, which won't be touched on in this book, but is well worth reading up on as there's properties to be exploited. It is the canonical sum type (or disjoint union of sets)

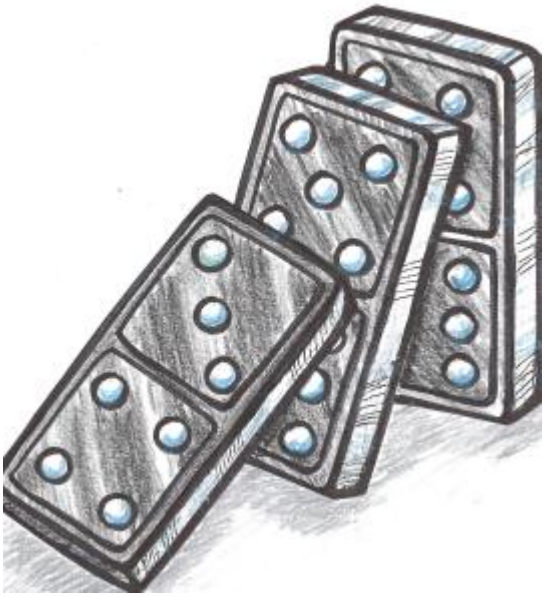
because its amount of possible inhabitants is the sum of the two contained types (I know that's a bit hand wavy so here's a great article). There are many things `Either` can be, but as a functor, it is used for its error handling.

Just like with `Maybe`, we have little `either`, which behaves similarly, but takes two functions instead of one and a static value. Each function should return the same type:

```
// either :: (a -> c) -> (b -> c) -> Either a b -> c
const either = curry((f, g, e) => {
  let result;

  switch (e.constructor) {
    case Left:
      result = f(e.$value);
      break;

    case Right:
      result = g(e.$value);
      break;

    // No Default
  }

  return result;
});

// zoltar :: User -> _
const zoltar = compose(console.log, either(id, fortune), getAge(moment()));

zoltar({ birthDate: '2005-12-12' });
// 'If you survive, you will be 10'
// undefined

zoltar({ birthDate: 'balloons!' });
// 'Birth date could not be parsed'
// undefined
```

Finally, a use for that mysterious `id` function. It simply parrots back the value in the `Left` to pass the error message to `console.log`. We've made our fortune-telling app more robust by enforcing error handling from within `getAge`. We either slap the user with a hard truth like a high five from a palm reader or we carry on with our process. And with that, we're ready to move on to an entirely different type of functor.

Old McDonald Had Effects...



In our chapter about purity we saw a peculiar example of a pure function. This function contained a side-effect, but we dubbed it pure by wrapping its action in another function. Here's another example of this:

```
// getFromStorage :: String -> (_ -> String)
const getFromStorage = key => () => localStorage[key];
```

Had we not surrounded its guts in another function, `getFromStorage` would vary its output depending on external circumstance. With the sturdy wrapper in place, we will always get the same output per input: a function that, when called, will retrieve a particular item from `localStorage`. And just like that (maybe throw in a few Hail Mary's) we've cleared our conscience and all is forgiven.

Except, this isn't particularly useful now is it. Like a collectible action figure in its original packaging, we can't actually play with it. If only there were a way to reach inside of the container and get at its contents... Enter `IO`.

```
class IO {
  static of(x) {
    return new IO(() => x);
  }

  constructor(fn) {
    this.$value = fn;
  }

  map(fn) {
    return new IO(compose(fn, this.$value));
  }

  inspect() {
    return `IO(${inspect(this.$value)})`;
  }
}
```

`IO` differs from the previous functors in that the `$value` is always a function. We don't think of its `$value` as a function, however - that is an implementation detail and we best ignore it. What is happening is exactly what we saw with the `getFromStorage` example: `IO` delays the impure action by capturing it in a function wrapper. As such, we think of `IO` as containing the return value of the wrapped action and not the wrapper itself. This is apparent in the `of` function: we have an `IO(x)`, the `IO(() => x)` is just necessary to avoid evaluation. Note that, to simplify reading, we'll show the hypothetical value contained in the `IO` as result; however in practice, you can't tell what this value is until you've actually unleashed the effects!

Let's see it in use:

```
// ioWindow :: IO Window
const ioWindow = new IO(() => window);

ioWindow.map(win => win.innerWidth);
// IO(1430)

ioWindow
  .map(prop('location'))
  .map(prop('href'))
  .map(split('/'));
// IO(['http:', '', 'localhost:8000', 'blog', 'posts'])



// $ :: String -> IO [DOM]
const $ = selector => new IO(() => document.querySelectorAll(selector));

$('#myDiv').map(head).map(div => div.innerHTML);
// IO('I am some inner html')
```

Here, `ioWindow` is an actual `IO` that we can `map` over straight away, whereas `$` is a function that returns an `IO` after it's called. I've written out the *conceptual* return values to better express the `IO`, though, in reality, it will always be `{ $value: [Function] }`. When we `map` over our `IO`, we stick that function at the end of a composition which, in turn, becomes the new `$value` and so on. Our mapped functions do not run, they get tacked on the end of a computation we're building up, function by function, like carefully placing dominoes that we don't dare tip over. The result is reminiscent of Gang of Four's command pattern or a queue.

Take a moment to channel your functor intuition. If we see past the implementation details, we should feel right at home mapping over any container no matter its quirks or idiosyncrasies. We have the functor laws, which we will explore toward the end of the chapter, to thank for this pseudo-psychic power. At any rate, we can finally play with impure values without sacrificing our precious purity.

Now, we've caged the beast, but we'll still have to set it free at some point. Mapping over our `IO` has built up a mighty impure computation and running it is surely going to disturb the peace. So where and when can we pull the trigger? Is it even possible to run our `IO` and still wear white at our wedding? The answer is yes, if we put the onus on the calling code. Our pure code, despite the nefarious plotting and scheming, maintains its innocence and it's the caller who gets burdened with the responsibility of actually running the effects. Let's see an example to make this concrete.

```
// url :: IO String
const url = new IO(() => window.location.href);

// toPairs :: String -> [[String]]
const toPairs = compose(map(split('=')), split('&'));

// params :: String -> [[String]]
const params = compose(toPairs, last, split('?'));

// findParam :: String -> IO Maybe [String]
const findParam = key => map(compose(Maybe.of, find(compose(eq(key), head)), pa

// -- Impure calling code ----------------------------------------------

// run it by calling $value()!
findParam('searchTerm').$value();
// Just(['searchTerm', 'wafflehouse'])
```

Our library keeps its hands clean by wrapping `url` in an `IO` and passing the buck to the caller. You might have also noticed that we have stacked our containers; it's perfectly reasonable to have a `IO(Maybe([x]))`, which is three functors deep (`Array` is most definitely a mappable container type) and exceptionally expressive.

There's something that's been bothering me and we should rectify it immediately: `IO`'s `$value` isn't really its contained value, nor is it a private property. It is the pin in the grenade and it is meant to be pulled by a caller in the most public of ways. Let's rename this property to `unsafePerformIO` to remind our users of its volatility.

```
class IO {
  constructor(io) {
    this.unsafePerformIO = io;
  }

  map(fn) {
    return new IO(compose(fn, this.unsafePerformIO));
  }
}
```

There, much better. Now our calling code becomes `findParam('searchTerm').unsafePerformIO()`, which is clear as day to users (and readers) of the application.

`IO` will be a loyal companion, helping us tame those feral impure actions. Next, we'll see a type similar in spirit, but having a drastically different use case.

Asynchronous Tasks

Callbacks are the narrowing spiral staircase to hell. They are control flow as designed by M.C. Escher. With each nested callback squeezed in between the jungle gym of curly braces and parenthesis, they feel like limbo in an oubliette (how low can we go?!). I'm getting claustrophobic chills just thinking about them. Not to worry, we have a much better way of dealing with asynchronous code and it starts with an "F".

The internals are a bit too complicated to spill out all over the page here so we will use `Data.Task` (previously `Data.Future`) from Quildreen Motta's fantastic Folktale. Behold some example usage:

```
// -- Node readFile example -------------------------------------

const fs = require('fs');

// readFile :: String -> Task Error String
const readFile = filename => new Task((reject, result) => {
  fs.readFile(filename, (err, data) => (err ? reject(err) : result(data)));
});

readFile('metamorphosis').map(split('\n')).map(head);
// Task('One morning, as Gregor Samsa was waking up from anxious dreams, he dis
// in bed he had been changed into a monstrous verminous bug.')



// -- jQuery getJSON example -------------------------------------

// getJSON :: String -> {} -> Task Error JSON
const getJSON = curry((url, params) => new Task((reject, result) => {
  $.getJSON(url, params, result).fail(reject);
}));

getJSON('/video', { id: 10 }).map(prop('title'));
// Task('Family Matters ep 15')



// -- Default Minimal Context -----------------------------------

// We can put normal, non futuristic values inside as well
Task.of(3).map(three => three + 1);
// Task(4)
```

The functions I'm calling `reject` and `result` are our error and success callbacks, respectively. As you can see, we simply `map` over the `Task` to work on the future value as if it was right there in our grasp. By now `map` should be old

hat.

If you're familiar with promises, you might recognize the function `map` as `then` with `Task` playing the role of our promise. Don't fret if you aren't familiar with promises, we won't be using them anyhow because they are not pure, but the analogy holds nonetheless.

Like `IO`, `Task` will patiently wait for us to give it the green light before running. In fact, because it waits for our command, `IO` is effectively subsumed by `Task` for all things asynchronous; `readFile` and `getJSON` don't require an extra `IO` container to be pure. What's more, `Task` works in a similar fashion when we `map` over it: we're placing instructions for the future like a chore chart in a time capsule - an act of sophisticated technological procrastination.

To run our `Task`, we must call the method `fork`. This works like `unsafePerformIO`, but as the name suggests, it will fork our process and evaluation continues on without blocking our thread. This can be implemented in numerous ways with threads and such, but here it acts as a normal async call would and the big wheel of the event loop keeps on turning. Let's look at `fork`:

```
// -- Pure application -------------------------------------------------
// blogPage :: Posts -> HTML
const blogPage = Handlebars.compile(blogTemplate);

// renderPage :: Posts -> HTML
const renderPage = compose(blogPage, sortBy(prop('date')));

// blog :: Params -> Task Error HTML
const blog = compose(map(renderPage), getJSON('/posts'));



// -- Impure calling code ----------------------------------------------
blog({}).fork(
  error => $('#error').html(error.message),
  page => $('#main').html(page),
);

$('#spinner').show();
```

Upon calling `fork`, the `Task` hurries off to find some posts and render the page. Meanwhile, we show a spinner since `fork` does not wait for a response. Finally, we will either display an error or render the page onto the screen depending if the `getJSON` call succeeded or not.

Take a moment to consider how linear the control flow is here. We just read bottom to top, right to left even though the program will actually jump around a bit during execution. This makes reading and reasoning about our application simpler than having to bounce between callbacks and error handling blocks.

Goodness, would you look at that, `Task` has also swallowed up `Either`! It must do so in order to handle futuristic failures since our normal control flow does not apply in the async world. This is all well and good as it provides sufficient and pure error handling out of the box.

Even with `Task`, our `IO` and `Either` functors are not out of a job. Bear with me on a quick example that leans toward the more complex and hypothetical side, but is useful for illustrative purposes.

```
// Postgres.connect :: Url -> IO DbConnection
// runQuery :: DbConnection -> ResultSet
// readFile :: String -> Task Error String

// -- Pure application -----------------------------------------------

// dbUrl :: Config -> Either Error Url
const dbUrl = ({ uname, pass, host, db }) => {
  if (uname && pass && host && db) {
    return Either.of(`db:pg://${uname}:${pass}@${host}5432/${db}`);
  }

  return left(Error('Invalid config!'));
};

// connectDb :: Config -> Either Error (IO DbConnection)
const connectDb = compose(map(Postgres.connect), dbUrl);

// getConfig :: Filename -> Task Error (Either Error (IO DbConnection))
const getConfig = compose(map(compose(connectDb, JSON.parse)), readFile);



// -- Impure calling code -----------------------------------------------

getConfig('db.json').fork(
  logErr('couldn\'t read file'),
  either(console.log, map(runQuery)),
);
```

In this example, we still make use of `Either` and `IO` from within the success branch of `readFile`. `Task` takes care of the impurities of reading a file asynchronously, but we still deal with validating the config with `Either` and wrangling the db connection with `IO`. So you see, we're still in business for all things synchronous.

I could go on, but that's all there is to it. Simple as `map`.

In practice, you'll likely have multiple asynchronous tasks in one workflow and we haven't yet acquired the full container apis to tackle this scenario. Not to worry, we'll look at monads and such soon, but first, we must examine the maths that

make this all possible.

A Spot of Theory

As mentioned before, functors come from category theory and satisfy a few laws. Let's first explore these useful properties.

```
// identity
map(id) === id;

// composition
compose(map(f), map(g)) === map(compose(f, g));
```

The *identity* law is simple, but important. These laws are runnable bits of code so we can try them on our own functors to validate their legitimacy.

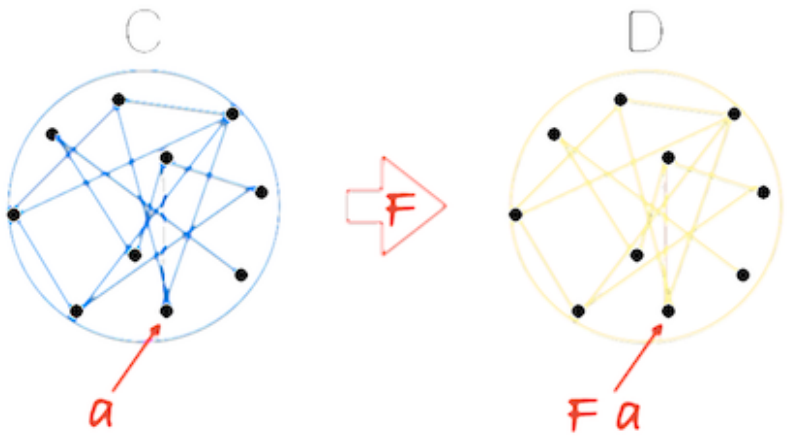
```
const idLaw1 = map(id);
const idLaw2 = id;

idLaw1(Container.of(2)); // Container(2)
idLaw2(Container.of(2)); // Container(2)
```

You see, they are equal. Next let's look at composition.

```
const compLaw1 = compose(map(append(' world')), map(append(' cruel')));
const compLaw2 = map(compose(append(' world'), append(' cruel')));

compLaw1(Container.of('Goodbye')); // Container('Goodbye cruel world')
compLaw2(Container.of('Goodbye')); // Container('Goodbye cruel world')
```

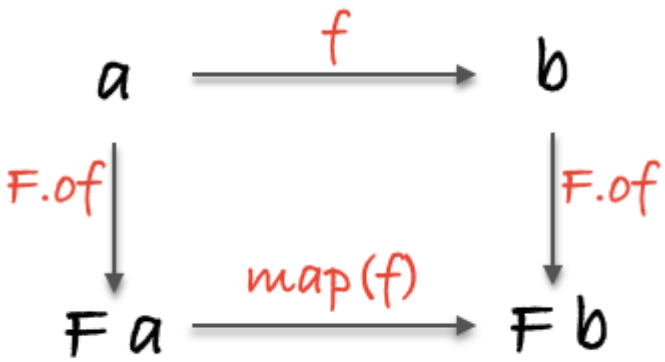
In category theory, functors take the objects and morphisms of a category and map them to a different category. By definition, this new category must have an identity and the ability to compose morphisms, but we needn't check because the aforementioned laws ensure these are preserved.

Perhaps our definition of a category is still a bit fuzzy. You can think of a category as a network of objects with morphisms that connect them. So a functor would map the one category to the other without breaking the network. If an object `a` is in our source category `C`, when we map it to category `D` with functor `F`, we refer to that object as `F a` (If you put it together what does that spell?!). Perhaps, it's better to look at a diagram:



For instance, `Maybe` maps our category of types and functions to a category where each object may not exist and each morphism has a `null` check. We accomplish this in code by surrounding each function with `map` and each type with our functor. We know that each of our normal types and functions will continue to compose in this new world. Technically, each functor in our code maps to a sub category of types and functions which makes all functors a particular brand called endofunctors, but for our purposes, we'll think of it as a different category.

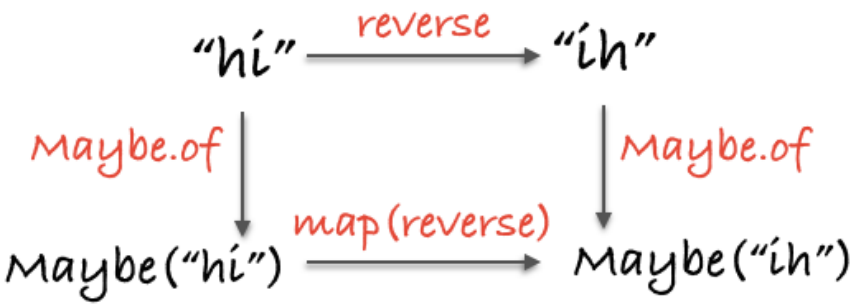
We can also visualize the mapping of a morphism and its corresponding objects with this diagram:



In addition to visualizing the mapped morphism from one category to another under the functor `F`, we see that the diagram commutes, which is to say, if you follow the arrows each route produces the same result. The different routes mean different behavior, but we always end at the same type. This formalism gives us principled ways to reason about our code - we can boldly apply formulas without having to parse and examine each individual scenario. Let's take a concrete example.

```
// topRoute :: String -> Maybe String
const topRoute = compose(Maybe.of, reverse);

// bottomRoute :: String -> Maybe String
const bottomRoute = compose(map(reverse), Maybe.of);

topRoute('hi'); // Just('ih')
bottomRoute('hi'); // Just('ih')
```

Or visually:



We can instantly see and refactor code based on properties held by all functors.

Functors can stack:

```
const nested = Task.of([Either.of('pillows'), left('no sleep for you')]);

map(map(map(toUpperCase)), nested);
// Task([Right('PILLOWS'), Left('no sleep for you')])
```

What we have here with `nested` is a future array of elements that might be errors. We `map` to peel back each layer and run our function on the elements. We see no callbacks, if/else's, or for loops; just an explicit context. We do, however, have to `map(map(map(f)))`. We can instead compose functors. You heard me correctly:

```
class Compose {
  constructor(fgx) {
    this.getCompose = fgx;
  }

  static of(fgx) {
    return new Compose(fgx);
  }

  map(fn) {
    return new Compose(map(map(fn), this.getCompose));
  }
}

const tmd = Task.of(Maybe.of('Rock over London'));

const ctmd = Compose.of(tmd);

const ctmd2 = map(append(', rock on, Chicago'), ctmd);
// Compose(Task(Just('Rock over London, rock on, Chicago')))

ctmd2.getCompose;
// Task(Just('Rock over London, rock on, Chicago'))
```

There, one `map`. Functor composition is associative and earlier, we defined `Container`, which is actually called the `Identity` functor. If we have identity and associative composition we have a category. This particular category has categories as objects and functors as morphisms, which is enough to make one's brain perspire. We won't delve too far into this, but it's nice to appreciate the architectural implications or even just the simple abstract beauty in the pattern.

In Summary

We've seen a few different functors, but there are infinitely many. Some notable omissions are iterable data structures like trees, lists, maps, pairs, you name it. Event streams and observables are both functors. Others can be for encapsulation or even just type modelling. Functors are all around us and we'll use them extensively throughout the book.

What about calling a function with multiple functor arguments? How about working with an order sequence of impure or async actions? We haven't yet acquired the full tool set for working in this boxed up world. Next, we'll cut right to the chase and look at monads.

Chapter 09: Monadic Onions

Exercises

Exercise

Use `add` and `map` to make a function that increments a value inside a functor.

```
// incrF :: Functor f => f Int -> f Int
const incrF = undefined;
```

Given the following User object:

```
const user = { id: 2, name: 'Albert', active: true };
```

Exercise

Use `safeProp` and `head` to find the first initial of the user.

```
// initial :: User -> Maybe String
const initial = undefined;
```

Given the following helper functions:

```
// showWelcome :: User -> String
const showWelcome = compose(concat('Welcome '), prop('name'));

// checkActive :: User -> Either String User
const checkActive = function checkActive(user) {
  return user.active
    ? Either.of(user)
    : left('Your account is not active');
};
```

Exercise

Write a function that uses `checkActive` and `showWelcome` to grant access or return the error.

```
// eitherWelcome :: User -> Either String String
const eitherWelcome = undefined;
```

We now consider the following functions:

```
// validateUser :: (User -> Either String ()) -> User -> Either String User
const validateUser = curry((validate, user) => validate(user).map(_ => user));

// save :: User -> IO User
const save = user => new IO(() => ({ ...user, saved: true }));
```

Exercise

Write a function `validateName` which checks whether a user has a name longer than 3 characters or return an error message. Then use `either`, `showWelcome` and `save` to write a `register` function to signup and welcome a user when the validation is ok. Remember either's two arguments must return the same type.

```
// validateName :: User -> Either String ()
const validateName = undefined;

// register :: User -> IO String
const register = compose(undefined, validateUser(validateName));
```

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
// 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.

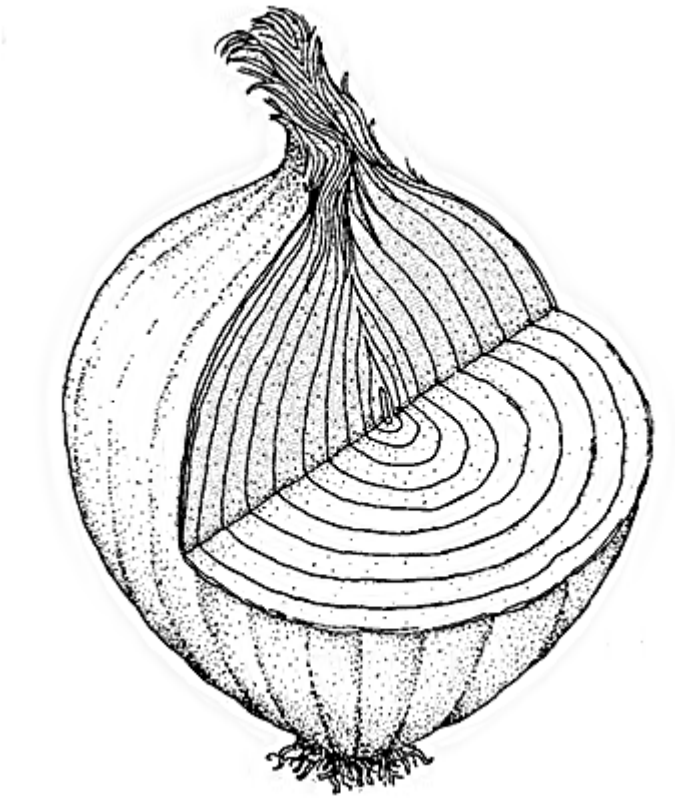
One important correction I must make at this point, pun intended, is that `Left.of` doesn't make any sense. Each functor must have one way to place a value inside it and with `Either`, that's `new Right(x)`. We define `of` using `Right` because if our type *can* `map`, it *should* `map`. Looking at the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

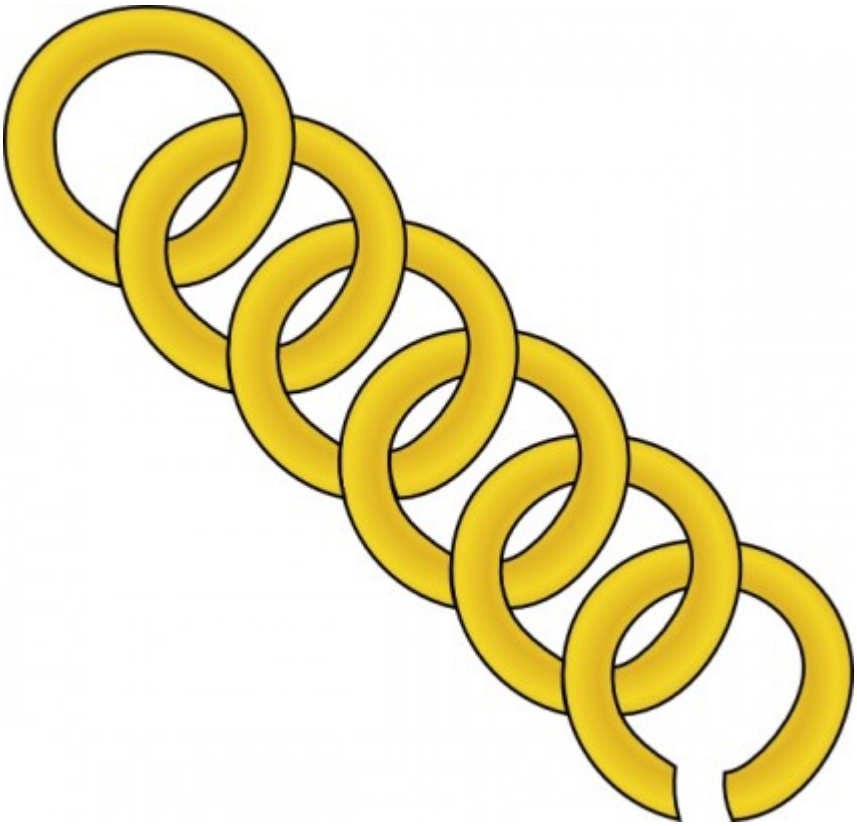
```
IO.prototype.join = function() {
  const $ = this;
  return new IO(() => $.unsafePerformIO().unsafePerformIO());
};
```

We simply bundle running the two layers of IO sequentially: outer then inner. Mind you, we have not thrown out purity, but merely repackaged the excessive two layers of shrink wrap into one easier-to-open package.

```
// log :: a -> IO a
const log = x => new IO(() => {
  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
  );
// 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 -> String -> Task Error JSON
// upload :: Filename -> 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 :: Filename -> (String -> a) -> Void
const upload = (filename, callback) => {
  if (!filename) {
    throw new Error('You need a filename!');
  } else {
    readFile(filename, (errF, contents) => {
      if (errF) throw errF;
      httpPost('/uploads', 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);
```

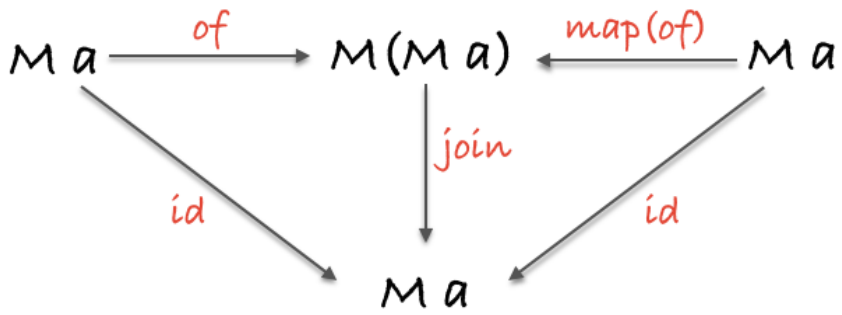
These laws get at the nested nature of monads so associativity focuses on joining the inner or outer types first to achieve the same result. A picture might be more instructive:

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:

```
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

```
// getStreetName :: User -> Maybe String
const getStreetName = undefined;
```

We now consider the following items:

```
// getFile :: IO String
const getFile = IO.of('/home/mostly-adequate/ch09.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.

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

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.

```
// joinMailingList :: Email -> Either String (IO ())
const joinMailingList = undefined;
```

Chapter 10: Applicative Functors

Applying Applicatives

The name **applicative functor** is pleasantly descriptive given its functional origins. Functional programmers are notorious for coming up with names like `mappend` or `liftA4`, which seem perfectly natural when viewed in the math lab, but hold the clarity of an indecisive Darth Vader at the drive thru in any other context.

Anyhow, the name should spill the beans on what this interface gives us: *the ability to apply functors to each other*.

Now, why would a normal, rational person such as yourself want such a thing? What does it even *mean* to apply one functor to another?

To answer these questions, we'll start with a situation you may have already encountered in your functional travels. Let's say, hypothetically, that we have two functors (of the same type) and we'd like to call a function with both of their values as arguments. Something simple like adding the values of two `Container`s.

```
// We can't do this because the numbers are bottled up.
add(Container.of(2), Container.of(3));
// NaN

// Let's use our trusty map
const containerOfAdd2 = map(add, Container.of(2));
// Container(add(2))
```

We have ourselves a `Container` with a partially applied function inside. More specifically, we have a `Container(add(2))` and we'd like to apply its `add(2)` to the `3` in `Container(3)` to complete the call. In other words, we'd like to apply one functor to another.

Now, it just so happens that we already have the tools to accomplish this task. We can `chain` and then `map` the partially applied `add(2)` like so:

```
Container.of(2).chain(two => Container.of(3).map(add(two)));
```

The issue here is that we are stuck in the sequential world of monads wherein nothing may be evaluated until the previous monad has finished its business. We have ourselves two strong, independent values and I should think it unnecessary to delay the creation of `Container(3)` merely to satisfy the monad's sequential demands.

In fact, it would be lovely if we could succinctly apply one functor's contents to another's value without these needless functions and variables should we find ourselves in this pickle jar.

Ships in Bottles

`ap` is a function that can apply the function contents of one functor to the value contents of another. Say that five times fast.

```
Container.of(add(2)).ap(Container.of(3));
// Container(5)


// all together now

Container.of(2).map(add).ap(Container.of(3));
// Container(5)
```

There we are, nice and neat. Good news for `Container(3)` as it's been set free from the jail of the nested monadic function. It's worth mentioning again that `add`, in this case, gets partially applied during the first `map` so this only works when `add` is curried.

We can define `ap` like so:

```
Container.prototype.ap = function (otherContainer) {
  return otherContainer.map(this.$value);
};
```

Remember, `this.$value` will be a function and we'll be accepting another functor so we need only `map` it. And with that we have our interface definition:

> An *applicative functor* is a pointed functor with an `ap` method

Note the dependence on **pointed**. The pointed interface is crucial here as we'll see throughout the following examples.

Now, I sense your skepticism (or perhaps confusion and horror), but keep an open mind; this `ap` character will prove useful. Before we get into it, let's explore a nice property.

```
F.of(x).map(f) === F.of(f).ap(F.of(x));
```

In proper English, mapping `f` is equivalent to `ap`ing a functor of `f`. Or in properer English, we can place `x` into our container and `map(f)` OR we can lift both `f` and `x` into our container and `ap` them. This allows us to write in a left-to-right fashion:

```
Maybe.of(add).ap(Maybe.of(2)).ap(Maybe.of(3));
// Maybe(5)

Task.of(add).ap(Task.of(2)).ap(Task.of(3));
// Task(5)
```

One might even recognise the vague shape of a normal function call if viewed mid squint. We'll look at the pointfree version later in the chapter, but for now, this is the preferred way to write such code. Using `of`, each value gets transported to the magical land of containers, this parallel universe where each application can be async or null or what have you and `ap` will apply functions within this fantastical place. It's like building a ship in a bottle.

Did you see there? We used `Task` in our example. This is a prime situation where applicative functors pull their weight. Let's look at a more in-depth example.

Coordination Motivation

Say we're building a travel site and we'd like to retrieve both a list of tourist destinations and local events. Each of these are separate, stand-alone api calls.

```
// Http.get :: String -> Task Error HTML

const renderPage = curry((destinations, events) => { /* render page */ });

Task.of(renderPage).ap(Http.get('/destinations')).ap(Http.get('/events'));
// Task("<div>some page with dest and events</div>")
```

Both `Http` calls will happen instantly and `renderPage` will be called when both are resolved. Contrast this with the monadic version where one `Task` must finish before the next fires off. Since we don't need the destinations to retrieve events, we are free from sequential evaluation.

Again, because we're using partial application to achieve this result, we must ensure `renderPage` is curried or it will not wait for both `Tasks` to finish. Incidentally, if you've ever had to do such a thing manually, you'll appreciate the

astonishing simplicity of this interface. This is the kind of beautiful code that takes us one step closer to the singularity.

Let's look at another example.

```
// $ :: String -> IO DOM
const $ = selector => new IO(() => document.querySelector(selector));

// getVal :: String -> IO String
const getVal = compose(map(prop('value')), $);

// signIn :: String -> String -> Bool -> User
const signIn = curry((username, password, rememberMe) => { /* signing in */ });

IO.of(signIn).ap(getVal('#email')).ap(getVal('#password')).ap(IO.of(false));
// IO({ id: 3, email: 'gg@allin.com' })
```

`signIn` is a curried function of 3 arguments so we have to `ap` accordingly. With each `ap`, `signIn` receives one more argument until it is complete and runs. We can continue this pattern with as many arguments as necessary. Another thing to note is that two arguments end up naturally in `IO` whereas the last one needs a little help from `of` to lift it into `IO` since `ap` expects the function and all its arguments to be in the same type.

Bro, Do You Even Lift?

Let's examine a pointfree way to write these applicative calls. Since we know `map` is equal to `of/ap`, we can write generic functions that will `ap` as many times as we specify:

```
const liftA2 = curry((g, f1, f2) => f1.map(g).ap(f2));

const liftA3 = curry((g, f1, f2, f3) => f1.map(g).ap(f2).ap(f3));

// liftA4, etc
```

`liftA2` is a strange name. It sounds like one of the finicky freight elevators in a rundown factory or a vanity plate for a cheap limo company. Once enlightened, however, it's self explanatory: lift these pieces into the applicative functor world.

When I first saw this 2-3-4 nonsense it struck me as ugly and unnecessary. After all, we can check the arity of functions in JavaScript and build this up dynamically. However, it is often useful to partially apply `liftA(N)` itself, so it cannot vary in argument length.

Let's see this in use:

```
// checkEmail :: User -> Either String Email
// checkName :: User -> Either String String

const user = {
  name: 'John Doe',
  email: 'blurp_blurp',
};

//  createUser :: Email -> String -> IO User
const createUser = curry((email, name) => { /* creating... */ });

Either.of(createUser).ap(checkEmail(user)).ap(checkName(user));
// Left('invalid email')

liftA2(createUser, checkEmail(user), checkName(user));
// Left('invalid email')
```

Since `createUser` takes two arguments, we use the corresponding `liftA2`. The two statements are equivalent, but the `liftA2` version has no mention of `Either`. This makes it more generic and flexible since we are no longer married to a specific type.

Let's see the previous examples written this way:

```
liftA2(add, Maybe.of(2), Maybe.of(3));
// Maybe(5)

liftA2(renderPage, Http.get('/destinations'), Http.get('/events'));
// Task('<div>some page with dest and events</div>')

liftA3(signIn, getVal('#email'), getVal('#password'), IO.of(false));
// IO({ id: 3, email: 'gg@allin.com' })
```

Operators

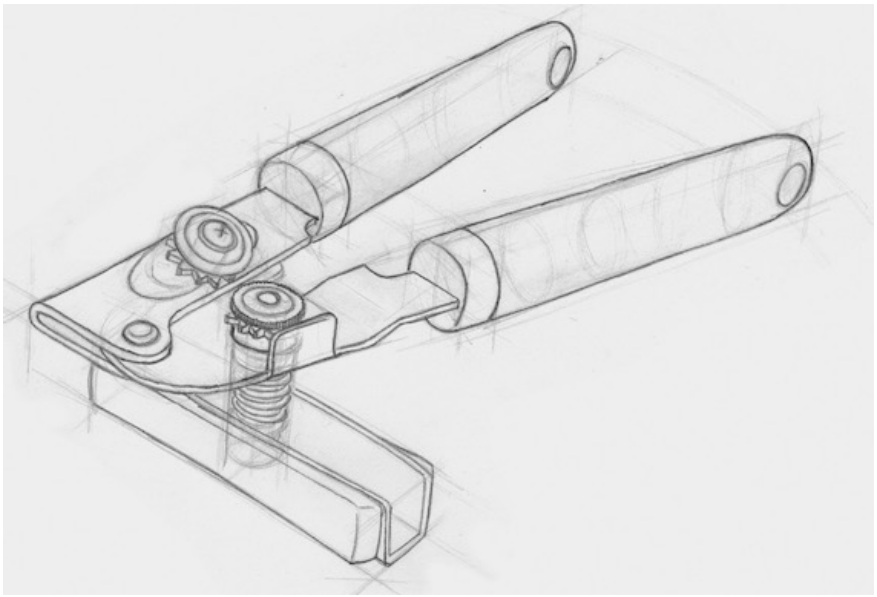
In languages like Haskell, Scala, PureScript, and Swift, where it is possible to create your own infix operators you may see syntax like this:

```
-- Haskell / PureScript
add <$> Right 2 <*> Right 3
```

```
// JavaScript
map(add, Right(2)).ap(Right(3));
```

It's helpful to know that `<$>` is `map` (aka `fmap`) and `<*>` is just `ap`. This allows for a more natural function application style and can help remove some parenthesis.

Free Can Openers



We haven't spoken much about derived functions. Seeing as all of these interfaces are built off of each other and obey a set of laws, we can define some weaker interfaces in terms of the stronger ones.

For instance, we know that an applicative is first a functor, so if we have an applicative instance, surely we can define a functor for our type.

This kind of perfect computational harmony is possible because we're working within a mathematical framework. Mozart couldn't have done better even if he had torrented Ableton as a child.

I mentioned earlier that `of/ap` is equivalent to `map`. We can use this knowledge to define `map` for free:

```
// map derived from of/ap
X.prototype.map = function map(f) {
  return this.constructor.of(f).ap(this);
};
```

Monads are at the top of the food chain, so to speak, so if we have `chain`, we get functor and applicative for free:

```
// map derived from chain
X.prototype.map = function map(f) {
  return this.chain(a => this.constructor.of(f(a)));
};

// ap derived from chain/map
X.prototype.ap = function ap(other) {
  return this.chain(f => other.map(f));
};
```

If we can define a monad, we can define both the applicative and functor interfaces. This is quite remarkable as we get all of these can openers for free. We can even examine a type and automate this process.

It should be pointed out that part of `ap` 's appeal is the ability to run things concurrently so defining it via `chain` is missing out on that optimization. Despite that, it's good to have an immediate working interface while one works out the best possible implementation.

Why not just use monads and be done with it, you ask? It's good practice to work with the level of power you need, no more, no less. This keeps cognitive load to a minimum by ruling out possible functionality. For this reason, it's good to favor applicatives over monads.

Monads have the unique ability to sequence computation, assign variables, and halt further execution all thanks to the downward nesting structure. When one sees applicatives in use, they needn't concern themselves with any of that business.

Now, on to the legalities ...

Laws

Like the other mathematical constructs we've explored, applicative functors hold some useful properties for us to rely on in our daily code. First off, you should know that applicatives are "closed under composition", meaning `ap` will never change container types on us (yet another reason to favor over monads). That's not to say we cannot have multiple different effects - we can stack our types knowing that they will remain the same during the entirety of our application.

To demonstrate:

```
const tOfM = compose(Task.of, Maybe.of);

liftA2(liftA2(concat), tOfM('Rainy Days and Mondays'), tOfM(' always get me dow
// Task(Maybe(Rainy Days and Mondays always get me down))
```

See, no need to worry about different types getting in the mix.

Time to look at our favorite categorical law: *identity*:

Identity

```
// identity
A.of(id).ap(v) === v;
```

Right, so applying `id` all from within a functor shouldn't alter the value in `v`. For example:

```
const v = Identity.of('Pillow Pets');
Identity.of(id).ap(v) === v;
```

`Identity.of(id)` makes me chuckle at its futility. Anyway, what's interesting is that, as we've already established, `of/ap` is the same as `map` so this law follows directly from functor identity: `map(id) == id`.

The beauty in using these laws is that, like a militant kindergarten gym coach, they force all of our interfaces to play well together.

Homomorphism

```
// homomorphism
A.of(f).ap(A.of(x)) === A.of(f(x));
```

A *homomorphism* is just a structure preserving map. In fact, a functor is just a *homomorphism* between categories as it preserves the original category's structure under the mapping.

We're really just stuffing our normal functions and values into a container and running the computation in there so it should come as no surprise that we will end up with the same result if we apply the whole thing inside the container (left side of the equation) or apply it outside, then place it in there (right side).

A quick example:

```
Either.of(toUpperCase).ap(Either.of('oreos')) === Either.of(toUpperCase('oreos
```

Interchange

The *interchange* law states that it doesn't matter if we choose to lift our function into the left or right side of `ap`.

```
// interchange
v.ap(A.of(x)) === A.of(f => f(x)).ap(v);
```

Here is an example:

```
const v = Task.of(reverse);
const x = 'Sparklehorse';

v.ap(Task.of(x)) === Task.of(f => f(x)).ap(v);
```

Composition

And finally composition which is just a way to check that our standard function composition holds when applying inside of containers.

```
// composition
A.of(compose).ap(u).ap(v).ap(w) === u.ap(v.ap(w));
```

```
const u = IO.of(toUpperCase);
const v = IO.of(concat('& beyond'));
const w = IO.of('blood bath ');

IO.of(compose).ap(u).ap(v).ap(w) === u.ap(v.ap(w));
```

In Summary

A good use case for applicatives is when one has multiple functor arguments. They give us the ability to apply functions to arguments all within the functor world. Though we could already do so with monads, we should prefer applicative functors when we aren't in need of monadic specific functionality.

We're almost finished with container apis. We've learned how to `map`, `chain`, and now `ap` functions. In the next chapter, we'll learn how to work better with multiple functors and disassemble them in a principled way.

Chapter 11: Transformation Again, Naturally

Exercises

Exercise

Write a function that adds two possibly null numbers together using `Maybe` and `ap`.

```
// safeAdd :: Maybe Number -> Maybe Number -> Maybe Number
const safeAdd = undefined;
```

Exercise

Rewrite `safeAdd` from exercise_b to use `liftA2` instead of `ap`.

```
// safeAdd :: Maybe Number -> Maybe Number -> Maybe Number
const safeAdd = undefined;
```

For the next exercise, we consider the following helpers:

```
const localStorage = {
  player1: { id:1, name: 'Albert' },
  player2: { id:2, name: 'Theresa' },
};

// getFromCache :: String -> IO User
const getFromCache = x => new IO(() => localStorage[x]);

// game :: User -> User -> String
const game = curry((p1, p2) => `${p1.name} vs ${p2.name}`);
```

Exercise

Write an IO that gets both player1 and player2 from the cache and starts the game.

```
// startGame :: IO String
const startGame = undefined;
```

Chapter 11: Transform Again, Naturally

We are about to discuss *natural transformations* in the context of practical utility in every day code. It just so happens they are a pillar of category theory and absolutely indispensable when applying mathematics to reason about and refactor our code. As such, I believe it is my duty to inform you about the lamentable injustice you are about to witness undoubtedly due to my limited scope. Let's begin.

Curse This Nest

I'd like to address the issue of nesting. Not the instinctive urge felt by soon to be parents wherein they tidy and rearrange with obsessive compulsion, but the...well actually, come to think of it, that isn't far from the mark as we'll see in the coming chapters... In any case, what I mean by *nesting* is to have two or more different types all huddled together around a value, cradling it like a newborn, as it were.

```
Right(Maybe('b'));


IO(Task(IO(1000)));


[Identity('bee thousand')];
```

Until now, we've managed to evade this common scenario with carefully crafted examples, but in practice, as one codes, types tend to tangle themselves up like earbuds in an exorcism. If we don't meticulously keep our types organized as we go along, our code will read hairier than a beatnik in a cat café.

A Situational Comedy

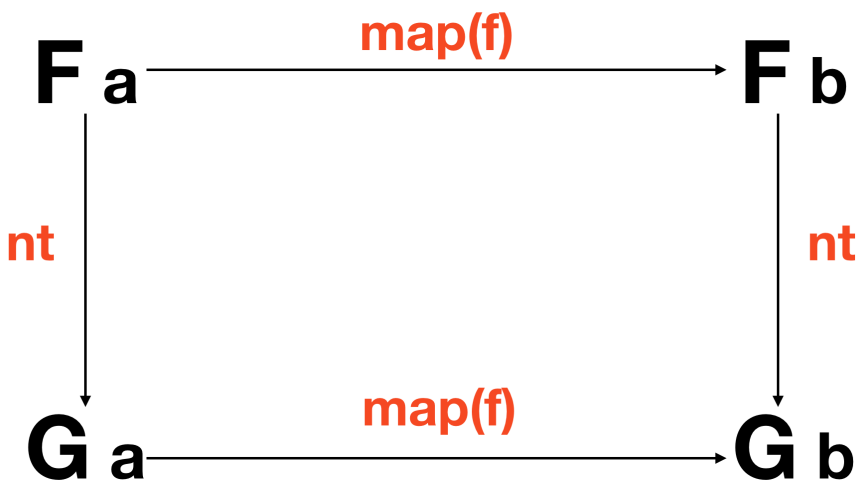
```
// getValue :: Selector -> Task Error (Maybe String)
// postComment :: String -> Task Error Comment
// validate :: String -> Either ValidationError String


// saveComment :: () -> Task Error (Maybe (Either ValidationError (Task Error C
const saveComment = compose(
  map(map(map(postComment))),
  map(map(validate)),
  getValue('#comment'),
);
```

The gang is all here, much to our type signature's dismay. Allow me to briefly explain the code. We start by getting the user input with `getValue('#comment')` which is an action which retrieves text on an element. Now, it might error finding the element or the value string may not exist so it returns `Task Error (Maybe String)`. After that, we must `map` over both the `Task` and the `Maybe` to pass our text to `validate`, which in turn, gives us back `Either` a `ValidationError` or our `String`. Then onto mapping for days to send the `String` in our current `Task Error (Maybe (Either ValidationError String))` into `postComment` which returns our resulting `Task`.

What a frightful mess. A collage of abstract types, amateur type expressionism, polymorphic Pollock, monolithic Mondrian. There are many solutions to this common issue. We can compose the types into one monstrous container, sort and `join` a few, homogenize them, deconstruct them, and so on. In this chapter, we'll focus on homogenizing them via *natural transformations*.

All Natural

A *Natural Transformation* is a "morphism between functors", that is, a function which operates on the containers themselves. Typewise, it is a function `(Functor f, Functor g) => f a -> g a`. What makes it special is that we cannot, for any reason, peek at the contents of our functor. Think of it as an exchange of highly classified information - the two parties oblivious to what's in the sealed manila envelope stamped "top secret". This is a structural operation. A functorial costume change. Formally, a *natural transformation* is any function for which the following holds:



or in code:

```
// nt :: (Functor f, Functor g) => f a -> g a
compose(map(f), nt) === compose(nt, map(f));
```

Both the diagram and the code say the same thing: We can run our natural transformation then `map` or `map` then run our natural transformation and get the same result. Incidentally, that follows from a free theorem though natural transformations (and functors) are not limited to functions on types.

Principled Type Conversions

As programmers we are familiar with type conversions. We transform types like `Strings` into `Booleans` and `Integers` into `Floats` (though JavaScript only has `Numbers`). The difference here is simply that we're working with algebraic containers and we have some theory at our disposal.

Let's look at some of these as examples:

```
// idToMaybe :: Identity a -> Maybe a
const idToMaybe = x => Maybe.of(x.$value);

// idToIO :: Identity a -> IO a
const idToIO = x => IO.of(x.$value);

// eitherToTask :: Either a b -> Task a b
const eitherToTask = either(Task.rejected, Task.of);

// ioToTask :: IO a -> Task () a
const ioToTask = x => new Task((reject, resolve) => resolve(x.unsafePerform()))

// maybeToTask :: Maybe a -> Task () a
const maybeToTask = x => (x.isNothing ? Task.rejected() : Task.of(x.$value));

// arrayToMaybe :: [a] -> Maybe a
const arrayToMaybe = x => Maybe.of(x[0]);
```

See the idea? We're just changing one functor to another. We are permitted to lose information along the way so long as the value we'll `map` doesn't get lost in the shape shift shuffle. That is the whole point: `map` must carry on, according to our definition, even after the transformation.

One way to look at it is that we are transforming our effects. In that light, we can view `ioToTask` as converting synchronous to asynchronous or `arrayToMaybe` from nondeterminism to possible failure. Note that we cannot convert asynchronous to synchronous in JavaScript so we cannot write `taskToIO` - that would be a supernatural transformation.

Feature Envy

Suppose we'd like to use some features from another type like `sortBy` on a `List`. *Natural transformations* provide a nice way to convert to the target type knowing our `map` will be sound.

```
// arrayToList :: [a] -> List a
const arrayToList = List.of;

const doListyThings = compose(sortBy(h), filter(g), arrayToList, map(f));
const doListyThings_ = compose(sortBy(h), filter(g), map(f), arrayToList); // 
```

A wiggle of our nose, three taps of our wand, drop in `arrayToList`, and voilà! Our `[a]` is a `List a` and we can `sortBy` if we please.

Also, it becomes easier to optimize / fuse operations by moving `map(f)` to the left of *natural transformation* as shown in `doListyThings_`.

Isomorphic JavaScript

When we can completely go back and forth without losing any information, that is considered an *isomorphism*. That's just a fancy word for "holds the same data". We say that two types are *isomorphic* if we can provide the "to" and "from" *natural transformations* as proof:

```
// promiseToTask :: Promise a b -> Task a b
const promiseToTask = x => new Task((reject, resolve) => x.then(resolve).catch

// taskToPromise :: Task a b -> Promise a b
const taskToPromise = x => new Promise((resolve, reject) => x.fork(reject, reso

const x = Promise.resolve('ring');
taskToPromise(promiseToTask(x)) === x;

const y = Task.of('rabbit');
promiseToTask(taskToPromise(y)) === y;
```

Q.E.D. `Promise` and `Task` are *isomorphic*. We can also write a `listToArray` to complement our `arrayToList` and show that they are too. As a counter example, `arrayToMaybe` is not an *isomorphism* since it loses information:

```
// maybeToArray :: Maybe a -> [a]
const maybeToArray = x => (x.isNothing ? [] : [x.$value]);

// arrayToMaybe :: [a] -> Maybe a
const arrayToMaybe = x => Maybe.of(x[0]);

const x = ['elvis costello', 'the attractions'];

// not isomorphic
maybeToArray(arrayToMaybe(x)); // ['elvis costello']

// but is a natural transformation
compose(arrayToMaybe, map(replace('elvis', 'lou')))(x); // Just('lou costello')
// ==
compose(map(replace('elvis', 'lou')), arrayToMaybe)(x); // Just('lou costello')
```

They are indeed *natural transformations*, however, since `map` on either side yields the same result. I mention *isomorphisms* here, mid-chapter while we're on the subject, but don't let that fool you, they are an enormously powerful and pervasive concept. Anyways, let's move on.

A Broader Definition

These structural functions aren't limited to type conversions by any means.

Here are a few different ones:

```
reverse :: [a] -> [a]

join :: (Monad m) => m (m a) -> m a

head :: [a] -> a

of :: a -> f a
```

The natural transformation laws hold for these functions too. One thing that might trip you up is that `head :: [a] -> a` can be viewed as `head :: [a] -> Identity a`. We are free to insert `Identity` wherever we please whilst proving laws since we can, in turn, prove that `a` is isomorphic to `Identity a` (see, I told you *isomorphisms* were pervasive).

One Nesting Solution

Back to our comedic type signature. We can sprinkle in some *natural transformations* throughout the calling code to coerce each varying type so they are uniform and, therefore, `join` able.

```
// getValue :: Selector -> Task Error (Maybe String)
// postComment :: String -> Task Error Comment
// validate :: String -> Either ValidationError String

// saveComment :: () -> Task Error Comment
const saveComment = compose(
  chain(postComment),
  chain(eitherToTask),
  map(validate),
  chain(maybeToTask),
  getValue('#comment'),
);
```

So what do we have here? We've simply added `chain(maybeToTask)` and `chain(eitherToTask)`. Both have the same effect; they naturally transform the functor our `Task` is holding into another `Task` then `join` the two. Like pigeon spikes on a window ledge, we avoid nesting right at the source. As they say in the city of light, "Mieux vaut prévenir que guérir" - an ounce of prevention is worth a pound of cure.

In Summary

Natural transformations are functions on our functors themselves. They are an extremely important concept in category theory and will start to appear everywhere once more abstractions are adopted, but for now, we've scoped them to a few concrete applications. As we saw, we can achieve different effects by converting types with the guarantee that our composition will hold. They can also help us with nested types, although they have the general effect of homogenizing our functors to the lowest common denominator, which in practice, is the functor with the most volatile effects (`Task` in most cases).

This continual and tedious sorting of types is the price we pay for having materialized them - summoned them from the ether. Of course, implicit effects are much more insidious and so here we are fighting the good fight. We'll need a few more tools in our tackle before we can reel in the larger type amalgamations. Next up, we'll look at reordering our types with *Traversable*.

Chapter 12: Traversing the Stone

Exercises

Exercise

Write a natural transformation that converts `Either b a` to `Maybe a`

```
// eitherToMaybe :: Either b a -> Maybe a
const eitherToMaybe = undefined;
```

```
// eitherToTask :: Either a b -> Task a b
const eitherToTask = either(Task.rejected, Task.of);
```

Exercise

Using `eitherToTask`, simplify `findNameById` to remove the nested `Either`.

```
// findNameById :: Number -> Task Error (Either Error User)
const findNameById = compose(map(map(prop('name'))), findUserById);
```

As a reminder, the following functions are available in the exercise's context:

```
split :: String -> String -> [String]
intercalate :: String -> [String] -> String
```

Exercise

Write the isomorphisms between String and [Char].

```
// strToList :: String -> [Char]
const strToList = undefined;

// listToStr :: [Char] -> String
const listToStr = undefined;
```

Chapter 12: Traversing the Stone

So far, in our cirque du conteneur, you've seen us tame the ferocious functor, bending it to our will to perform any operation that strikes our fancy. You've been dazzled by the juggling of many dangerous effects at once using function application to collect the results. Sat there in amazement as containers vanished in thin air by joining them together. At the side effect sideshow, we've seen them composed into one. And most recently, we've ventured beyond what's natural and transformed one type into another before your very eyes.

And now for our next trick, we'll look at traversals. We'll watch types soar over one another as if they were trapeze artists holding our value intact. We'll reorder effects like the trolleys in a tilt-a-whirl. When our containers get intertwined like the limbs of a contortionist, we can use this interface to straighten things out. We'll witness different effects with different orderings. Fetch me my pantaloons and slide whistle, let's get started.

Types n' Types

Let's get weird:

```
// readFile :: FileName -> Task Error String

// firstWords :: String -> String
const firstWords = compose(intercalate(' '), take(3), split(' '));

// tldr :: FileName -> Task Error String
const tldr = compose(map(firstWords), readFile);

map(tldr, ['file1', 'file2']);
// [Task('hail the monarchy'), Task('smash the patriarchy')]
```

Here we read a bunch of files and end up with a useless array of tasks. How might we fork each one of these? It would be most agreeable if we could switch the types around to have `Task Error [String]` instead of `[Task Error String]`. That way, we'd have one future value holding all the results, which is much more amenable to our async needs than several future values arriving at their leisure.

Here's one last example of a sticky situation:

```
// getAttribute :: String -> Node -> Maybe String
// $ :: Selector -> IO Node

// getControlNode :: Selector -> IO (Maybe (IO Node))
const getControlNode = compose(map(map($)), map(getAttribute('aria-controls')),
```

Look at those `IO`s longing to be together. It'd be just lovely to `join` them, let them dance cheek to cheek, but alas a `Maybe` stands between them like a chaperone at prom. Our best move here would be to shift their positions next to one another, that way each type can be together at last and our signature can be simplified to `IO (Maybe Node)`.

Type Feng Shui

The *Traversable* interface consists of two glorious functions: `sequence` and `traverse`.

Let's rearrange our types using `sequence`:

```
sequence(List.of, Maybe.of(['the facts'])); // [Just('the facts')]
sequence(Task.of, new Map({ a: Task.of(1), b: Task.of(2) })); // Task(Map({ a:
sequence(IO.of, Either.of(IO.of('buckle my shoe'))); // IO(Right('buckle my sho
sequence(Either.of, [Either.of('wing')]); // Right(['wing'])
sequence(Task.of, left('wing')); // Task(Left('wing'))
```

See what has happened here? Our nested type gets turned inside out like a pair of leather trousers on a humid summer night. The inner functor is shifted to the outside and vice versa. It should be known that `sequence` is bit particular about its arguments. It looks like this:

```
// sequence :: (Traversable t, Applicative f) => (a -> f a) -> t (f a) -> f (t
const sequence = curry((of, x) => x.sequence(of));
```

Let's start with the second argument. It must be a *Traversable* holding an *Applicative*, which sounds quite restrictive, but just so happens to be the case more often than not. It is the `t (f a)` which gets turned into a `f (t a)`. Isn't that expressive? It's clear as day the two types do-si-do around each other. That first argument there is merely a crutch and only necessary in an untyped language. It is a type constructor (our *of*) provided so that we can invert map-reluctant types like `Left` - more on that in a minute.

Using `sequence`, we can shift types around with the precision of a sidewalk thimblerigger. But how does it work? Let's look at how a type, say `Either`, would implement it:

```
class Right extends Either {
  // ...
  sequence(of) {
    return this.$value.map(Either.of);
  }
}
```

Ah yes, if our `$value` is a functor (it must be an applicative, in fact), we can simply `map` our constructor to leap frog the type.

You may have noticed that we've ignored the `of` entirely. It is passed in for the occasion where mapping is futile, as is the case with `Left`:

```
class Left extends Either {
  // ...
  sequence(of) {
    return of(this);
  }
}
```

We'd like the types to always end up in the same arrangement, therefore it is necessary for types like `Left` who don't actually hold our inner applicative to get a little help in doing so. The *Applicative* interface requires that we first have a *Pointed Functor* so we'll always have a `of` to pass in. In a language with a type system, the outer type can be inferred from the signature and does not need to be explicitly given.

Effect Assortment

Different orders have different outcomes where our containers are concerned. If I have `[Maybe a]`, that's a collection of possible values whereas if I have a `Maybe [a]`, that's a possible collection of values. The former indicates we'll be forgiving and keep "the good ones", while the latter means it's an "all or nothing" type of situation. Likewise, `Either Error (Task Error a)` could represent a client side validation and `Task Error (Either Error a)` could be a server side one. Types can be swapped to give us different effects.

```
// fromPredicate :: (a -> Bool) -> a -> Either e a

// partition :: (a -> Bool) -> [a] -> [Either e a]
const partition = f => map(fromPredicate(f));

// validate :: (a -> Bool) -> [a] -> Either e [a]
const validate = f => traverse(Either.of, fromPredicate(f));
```

Here we have two different functions based on if we `map` or `traverse`. The first, `partition` will give us an array of `Left`s and `Right`s according to the predicate function. This is useful to keep precious data around for future use rather than filtering it out with the bathwater. `validate` instead will give us the first item that fails the predicate in `Left`, or all the items in `Right` if everything is hunky dory. By choosing a different type order, we get different behavior.

Let's look at the `traverse` function of `List`, to see how the `validate` method is made.

```
traverse(of, fn) {
    return this.$value.reduce(
      (f, a) => fn(a).map(b => bs => bs.concat(b)).ap(f),
      of(new List([])),
    );
  }
```

This just runs a `reduce` on the list. The reduce function is `(f, a) => fn(a).map(b => bs => bs.concat(b)).ap(f)`, which looks a bit scary, so let's step through it.

1. `reduce(..., ...)`

   Remember the signature of `reduce :: [a] -> (f -> a -> f) -> f -> f`. The first argument is actually provided by the dot-notation on `$value`, so it's a list of things. Then we need a function from a `f` (the accumulator) and a `a` (the iteree) to return us a new accumulator.

2. `of(new List([]))`

   The seed value is `of(new List([]))`, which in our case is `Right([]) :: Either e [a]`. Notice that `Either e [a]` will also be our final resulting type!

3. `fn :: Applicative f => a -> f a`

   If we apply it to our example above, `fn` is actually `fromPredicate(f) :: a -> Either e a`.

   > fn(a) :: Either e a

4. `.map(b => bs => bs.concat(b))`

   When `Right`, `Either.map` passes the right value to the function and returns a new `Right` with the result. In this case the function has one parameter (`b`), and returns another function (`bs => bs.concat(b)`, where `b` is in scope due to the closure). When `Left`, the left value is returned.

   > fn(a).map(b => bs => bs.concat(b)) :: Either e ([a] -> [a])

5. . `ap(f)`

   Remember that `f` is an Applicative here, so we can apply the function `bs => bs.concat(b)` to whatever value `bs :: [a]` is in `f`. Fortunately for us, `f` comes from our initial seed and has the following type: `f :: Either e [a]` which is by the way, preserved when we apply `bs => bs.concat(b)`. When `f` is `Right`, this calls `bs => bs.concat(b)`, which returns a `Right` with the item added to the list. When `Left`, the left value (from the previous step or previous iteration respectively) is returned.

   > fn(a).map(b => bs => bs.concat(b)).ap(f) :: Either e [a]

This apparently miraculous transformation is achieved with just 6 measly lines of code in `List.traverse`, and is accomplished with `of`, `map` and `ap`, so will work for any Applicative Functor. This is a great example of how those abstraction

can help to write highly generic code with only a few assumptions (that can, incidentally, be declared and checked at the type level!).

Waltz of the Types

Time to revisit and clean our initial examples.

```
// readFile :: FileName -> Task Error String

// firstWords :: String -> String
const firstWords = compose(intercalate(' '), take(3), split(' '));

// tldr :: FileName -> Task Error String
const tldr = compose(map(firstWords), readFile);

traverse(Task.of, tldr, ['file1', 'file2']);
// Task(['hail the monarchy', 'smash the patriarchy']);
```

Using `traverse` instead of `map`, we've successfully herded those unruly `Task`s into a nice coordinated array of results. This is like `Promise.all()`, if you're familiar, except it isn't just a one-off, custom function, no, this works for any *traversable* type. These mathematical apis tend to capture most things we'd like to do in an interoperable, reusable way, rather than each library reinventing these functions for a single type.

Let's clean up the last example for closure (no, not that kind):

```
// getAttribute :: String -> Node -> Maybe String
// $ :: Selector -> IO Node

// getControlNode :: Selector -> IO (Maybe Node)
const getControlNode = compose(chain(traverse(IO.of, $)), map(getAttribute('ar
```

Instead of `map(map($))` we have `chain(traverse(IO.of, $))` which inverts our types as it maps then flattens the two `IO`s via `chain`.

No Law and Order

Well now, before you get all judgemental and bang the backspace button like a gavel to retreat from the chapter, take a moment to recognize that these laws are useful code guarantees. 'Tis my conjecture that the goal of most program architecture is an attempt to place useful restrictions on our code to narrow the possibilities, to guide us into the answers as designers and readers.

An interface without laws is merely indirection. Like any other mathematical structure, we must expose properties for our own sanity. This has a similar effect as encapsulation since it protects the data, enabling us to swap out the interface with another law abiding citizen.

Come along now, we've got some laws to suss out.

Identity

```
const identity1 = compose(sequence(Identity.of), map(Identity.of));
const identity2 = Identity.of;

// test it out with Right
identity1(Either.of('stuff'));
// Identity(Right('stuff'))

identity2(Either.of('stuff'));
// Identity(Right('stuff'))
```

This should be straightforward. If we place an `Identity` in our functor, then turn it inside out with `sequence` that's the same as just placing it on the outside to begin with. We chose `Right` as our guinea pig as it is easy to try the law and inspect. An arbitrary functor there is normal, however, the use of a concrete functor here, namely `Identity` in the law itself might raise some eyebrows. Remember a category is defined by morphisms between its objects that have associative composition and identity. When dealing with the category of functors, natural transformations are the morphisms and `Identity` is, well identity. The `Identity` functor is as fundamental in demonstrating laws as our `compose` function. In fact, we should give up the ghost and follow suit with our Compose type:

Composition

```
const comp1 = compose(sequence(Compose.of), map(Compose.of));
const comp2 = (Fof, Gof) => compose(Compose.of, map(sequence(Gof)), sequence(Fo

// Test it out with some types we have lying around
comp1(Identity(Right([true])));
// Compose(Right([Identity(true)]))

comp2(Either.of, Array)(Identity(Right([true])));
// Compose(Right([Identity(true)]))
```

This law preserves composition as one would expect: if we swap compositions of functors, we shouldn't see any surprises since the composition is a functor itself. We arbitrarily chose `true`, `Right`, `Identity`, and `Array` to test it out. Libraries

like quickcheck or jsverify can help us test the law by fuzz testing the inputs.

As a natural consequence of the above law, we get the ability to fuse traversals, which is nice from a performance standpoint.

Naturality

```
const natLaw1 = (of, nt) => compose(nt, sequence(of));
const natLaw2 = (of, nt) => compose(sequence(of), map(nt));

// test with a random natural transformation and our friendly Identity/Right fu

// maybeToEither :: Maybe a -> Either () a
const maybeToEither = x => (x.$value ? new Right(x.$value) : new Left());

natLaw1(Maybe.of, maybeToEither)(Identity.of(Maybe.of('barlow one')));
// Right(Identity('barlow one'))

natLaw2(Either.of, maybeToEither)(Identity.of(Maybe.of('barlow one')));
// Right(Identity('barlow one'))
```

This is similar to our identity law. If we first swing the types around then run a natural transformation on the outside, that should equal mapping a natural transformation, then flipping the types.

A natural consequence of this law is:

```
traverse(A.of, A.of) === A.of;
```

Which, again, is nice from a performance standpoint.

In Summary

Traversable is a powerful interface that gives us the ability to rearrange our types with the ease of a telekinetic interior decorator. We can achieve different effects with different orders as well as iron out those nasty type wrinkles that keep us from `join`ing them down. Next, we'll take a bit of a detour to see one of the most powerful interfaces of functional programming and perhaps even algebra itself: Monoids bring it all together

Exercises

Considering the following elements:

```
// httpGet :: Route -> Task Error JSON

// routes :: Map Route Route
const routes = new Map({ '/': '/', '/about': '/about' });
```

Exercise

Use the traversable interface to change the type signature of `getJsons` to Map Route Route → Task Error (Map Route JSON)

```
// getJsons :: Map Route Route -> Map Route (Task Error JSON)
const getJsons = map(httpGet);
```

We now define the following validation function:

```
// validate :: Player -> Either String Player
const validate = player => (player.name ? Either.of(player) : left('must have r
```

Exercise

Using traversable, and the `validate` function, update `startGame` (and its signature) to only start the game if all players are valid

```
// startGame :: [Player] -> [Either Error String]
const startGame = compose(map(map(always('game started!'))), map(validate)
```

Finally, we consider some file-system helpers:

```
// readfile :: String -> String -> Task Error String
// readdir :: String -> Task Error [String]
```

Exercise

Use traversable to rearrange and flatten the nested Tasks & Maybe

```
// readFirst :: String -> Task Error (Maybe (Task Error String))
const readFirst = compose(map(map(readfile('utf-8'))), map(safeHead), read
```

Chapter 13: Monoids bring it all together

Wild combination

In this chapter, we will examine *monoids* by way of *semigroup*. *Monoids* are the bubblegum in the hair of mathematical abstraction. They capture an idea that spans multiple disciplines, figuratively and literally bringing them all together. They are the ominous force that connects all that calculates. The oxygen in our code base, the ground on which it runs, quantum entanglement encoded.

Monoids are about combination. But what is combination? It can mean so many things from accumulation to concatenation to multiplication to choice, composition, ordering, even evaluation! We'll see many examples here, but we'll only tip-toe on the foothills of monoid mountain. The instances are plentiful and applications vast. The aim of this chapter is to provide a good intuition so you can make some *monoids* of your own.

Abstracting addition

Addition has some interesting qualities I'd like to discuss. Let's have a look at it through our abstraction goggles.

For starters, it's a binary operation, that is, an operation which takes two values and returns a value, all within the same set.

```
// a binary operation
1 + 1 = 2
```

See? Two values in the domain, one value in the codomain, all the same set - numbers, as it were. Some might say numbers are "closed under addition", meaning the type won't ever change no matter which ones get tossed into the mix. That means we can chain the operation since the result is always another number:

```
// we can run this on any amount of numbers
1 + 7 + 5 + 4 + ...
```

In addition to that (what a calculated pun...), we have associativity which buys us the ability to group operations however we please. Incidentally, an associative, binary operation is a recipe for parallel computation because we can chunk and distribute work.

```
// associativity
(1 + 2) + 3 = 6
1 + (2 + 3) = 6
```

Now, don't go confusing this with commutativity which allows us to rearrange the order. While that holds for addition, we're not particularly interested in that property at the moment - too specific for our abstraction needs.

Come to think of it, what properties should be in our abstract superclass anyways? What traits are specific to addition and what ones can be generalized? Are there other abstractions amidst this hierarchy or is it all one chunk? It's this kind of thinking that our mathematical forefathers applied when conceiving the interfaces in abstract algebra.

As it happens, those old school abstractionists landed on the concept of a *group* when abstracting addition. A *group* has all the bells and whistles including the concept of negative numbers. Here, we're only interested in that associative binary operator so we'll choose the less specific interface *Semigroup*. A *Semigroup* is a type with a `concat` method which acts as our associative binary operator.

Let's implement it for addition and call it `Sum`:

```
const Sum = x => ({
  x,
  concat: other => Sum(x + other.x)
})
```

Note we `concat` with some other `Sum` and always return a `Sum`.

I've used an object factory here instead of our typical prototype ceremony, primarily because `Sum` is not *pointed* and we don't want to have to type `new`. Anyways, here it is in action:

```
Sum(1).concat(Sum(3)) // Sum(4)
Sum(4).concat(Sum(37)) // Sum(41)
```

Just like that, we can program to an interface, not an implementation. Since this interface comes from group theory it has centuries of literature backing it up. Free docs!

Now, as mentioned, `Sum` is not *pointed*, nor a *functor*. As an exercise, go back and check the laws to see why. Okay, I'll just tell you: it can only hold a number, so `map` does not make sense here as we cannot transform the underlying value to another type. That would be a very limited `map` indeed!

So why is this useful? Well, as with any interface, we can swap out our instance to achieve different results:

```
const Product = x => ({ x, concat: other => Product(x * other.x) })

const Min = x => ({ x, concat: other => Min(x < other.x ? x : other.x) })

const Max = x => ({ x, concat: other => Max(x > other.x ? x : other.x) })
```

This isn't limited to numbers, though. Let's see some other types:

```
const Any = x => ({ x, concat: other => Any(x || other.x) })
const All = x => ({ x, concat: other => All(x && other.x) })

Any(false).concat(Any(true)) // Any(true)
Any(false).concat(Any(false)) // Any(false)

All(false).concat(All(true)) // All(false)
All(true).concat(All(true)) // All(true)

[1,2].concat([3,4]) // [1,2,3,4]

"miracle grow".concat("n") // miracle grown"

Map({day: 'night'}).concat(Map({white: 'nikes'})) // Map({day: 'night', white:
```

If you stare at these long enough the pattern will pop out at you like a magic eye poster. It's everywhere. We're merging data structures, combining logic, building strings...it seems one can bludgeon almost any task into this combination based interface.

I've used `Map` a few times now. Pardon me if you two weren't properly introduced. `Map` simply wraps `Object` so we can embellish it with some extra methods without altering the fabric of the universe.

All my favourite functors are semigroups.

The types we've seen so far which implement the functor interface all implement semigroup one as well. Let's look at `Identity` (the artist previously known as Container):

```
Identity.prototype.concat = function(other) {
  return new Identity(this.__value.concat(other.__value))
}

Identity.of(Sum(4)).concat(Identity.of(Sum(1))) // Identity(Sum(5))
Identity.of(4).concat(Identity.of(1)) // TypeError: this.__value.concat is not
```

It is a *semigroup* if and only if its `__value` is a *semigroup*. Like a butterfingered hang glider, it is one whilst it holds one.

Other types have similar behavior:

```
// combine with error handling
Right(Sum(2)).concat(Right(Sum(3))) // Right(Sum(5))
Right(Sum(2)).concat(Left('some error')) // Left('some error')



// combine async
Task.of([1,2]).concat(Task.of([3,4])) // Task([1,2,3,4])
```

This gets particularly useful when we stack these semigroups into a cascading combination:

```
// formValues :: Selector -> IO (Map String String)
// validate :: Map String String -> Either Error (Map String String)

formValues('#signup').map(validate).concat(formValues('#terms').map(validate))
formValues('#signup').map(validate).concat(formValues('#terms').map(validate))

serverA.get('/friends').concat(serverB.get('/friends')) // Task([friend1, frie

// loadSetting :: String -> Task Error (Maybe (Map String Boolean))
loadSetting('email').concat(loadSetting('general')) // Task(Maybe(Map({backgrou
```

In the top example, we've combined an `IO` holding an `Either` holding a `Map` to validate and merge form values. Next, we've hit a couple of different servers and combined their results in an async way using `Task` and `Array`. Lastly, we've stacked `Task`, `Maybe`, and `Map` to load, parse, and merge multiple settings.

These can be `chain`ed or `ap`'d, but *semigroups* capture what we'd like to do much more concisely.

This extends beyond functors. In fact, it turns out that anything made up entirely of semigroups, is itself, a semigroup: if we can concat the kit, then we can concat the caboodle.

```
const Analytics = (clicks, path, idleTime) => ({
  clicks,
  path,
  idleTime,
  concat: other =>
    Analytics(clicks.concat(other.clicks), path.concat(other.path), idleTime.co
})

Analytics(Sum(2), ['/home', '/about'], Right(Max(2000))).concat(Analytics(Sum(1
// Analytics(Sum(3), ['/home', '/about', '/contact'], Right(Max(2000)))
```

See, everything knows how to combine itself nicely. Turns out, we could do the same thing for free just by using the `Map` type:

```
Map({clicks: Sum(2), path: ['/home', '/about'], idleTime: Right(Max(2000))}).co
// Map({clicks: Sum(3), path: ['/home', '/about', '/contact'], idleTime: Right(
```

We can stack and combine as many of these as we'd like. It's simply a matter of adding another tree to the forest, or another flame to the forest fire depending on your codebase.

The default, intuitive behavior is to combine what a type is holding, however, there are cases where we ignore what's inside and combine the containers themselves. Consider a type like `Stream`:

```
const submitStream = Stream.fromEvent('click', $('#submit'))
const enterStream = filter(x => x.key === 'Enter', Stream.fromEvent('keydown',

submitStream.concat(enterStream).map(submitForm) // Stream()
```

We can combine event streams by capturing events from both as one new stream. Alternatively, we could have combined them by insisting they hold a semigroup. In fact, there are many possible instances for each type. Consider `Task`, we can combine them by choosing the earlier or later of the two. We can always chose the first `Right` instead of short circuiting on `Left` which has the effect of ignoring errors. There is an interface called *Alternative* which implements some of these, well, alternative instances, typically focused on choice rather than cascading combination. It is worth looking into if you are in need of such functionality.

Monoids for nothing

We were abstracting addition, but like the Babylonians, we lacked the concept of zero (there were zero mentions of it).

Zero acts as *identity* meaning any element added to `0`, will return back that very same element. Abstraction-wise, it's helpful to think of `0` as a kind of neutral or *empty* element. It's important that it act the same way on the left and right side of our binary operation:

```
// identity
1 + 0 = 1
0 + 1 = 1
```

Let's call this concept `empty` and create a new interface with it. Like so many startups, we'll choose a heinously uninformative, yet conveniently googleable name: *Monoid*. The recipe for *Monoid* is to take any *semigroup* and add a special *identity* element. We'll implement that with an `empty` function on the type itself:

```
Array.empty = () => []
String.empty = () => ""
Sum.empty = () => Sum(0)
Product.empty = () => Product(1)
Min.empty = () => Min(Infinity)
Max.empty = () => Max(-Infinity)
All.empty = () => All(true)
Any.empty = () => Any(false)
```

When might an empty, identity value prove useful? That's like asking why zero is useful. Like not asking anything at all...

When we have nothing else, who can we count on? Zero. How many bugs do we want? Zero. It's our tolerance for unsafe code. A fresh start. The ultimate price tag. It can annihilate everything in its path or save us in a pinch. A golden life saver and a pit of despair.

Codewise, they correspond to sensible defaults:

```
const settings = (prefix="", overrides=[], total=0) => ...

const settings = (prefix=String.empty(), overrides=Array.empty(), total=Sum.emp
```

Or to return a useful value when we have nothing else:

```
sum([]) // 0
```

They are also the perfect initial value for an accumulator...

Folding down the house

It just so happens that `concat` and `empty` fit perfectly in the first two slots of `reduce`. We can actually `reduce` an array of *semigroup*'s down by ignoring the *empty* value, but as you can see, that leads to a precarious situation:

```
// concat :: Semigroup s => s -> s -> s
const concat = x => y => x.concat(y)

[Sum(1), Sum(2)].reduce(concat) // Sum(3)

[].reduce(concat) // TypeError: Reduce of empty array with no initial value
```

Boom goes the dynamite. Like a twisted ankle in a marathon, we have ourselves a runtime exception. JavaScript is more than happy to let us strap pistols to our sneakers before running - it is a conservative sort of language, I suppose, but it stops us dead in our tracks when the array is barren. What could it return anyhow? `NaN`, `false`, `-1`? If we were to continue on in our program, we'd like a result of the right type. It could return a `Maybe` to indicate the possibility of failure, but we can do one better.

Let's use our curried `reduce` function and make a safe version where the `empty` value is not optional. It shall henceforth be known as `fold`:

```
// fold :: Monoid m => m -> [m] -> m
const fold = reduce(concat)
```

The initial `m` is our `empty` value - our neutral, starting point, then we take an array of `m`'s and crush them down to one beautiful diamond like value.

```
fold(Sum.empty(), [Sum(1), Sum(2)]) // Sum(3)
fold(Sum.empty(), []) // Sum(0)

fold(Any.empty(), [Any(false), Any(true)]) // Any(true)
fold(Any.empty(), []) // Any(false)



fold(Either.of(Max.empty()), [Right(Max(3)), Right(Max(21)), Right(Max(11))]) /
fold(Either.of(Max.empty()), [Right(Max(3)), Left('error retrieving value'), R

fold(IO.of([]), ['.link', 'a'].map($)) // IO([<a>, <button class="link"/>, <a>
```

We've provided a manual `empty` value for those last two since we can't define one on the type itself. That's totally fine. Typed languages can figure that out by themselves, but we have to pass it in here.

Not quite a monoid

There are some *semigroups* that cannot become *monoids*, that is provide an initial value. Look at `First`:

```
const First = x => ({ x, concat: other => First(x) })


Map({id: First(123), isPaid: Any(true), points: Sum(13)}).concat(Map({id: First
// Map({id: First(123), isPaid: Any(true), points: Sum(14)})
```

We'll merge a couple of accounts and keep the `First` id. There is no way to define an `empty` value for it. Doesn't mean it's not useful.

Grand unifying theory

Group theory or Category theory?

The notion of a binary operation is everywhere in abstract algebra. It is, in fact, the primary operation for a *category*. We cannot, however, model our operation in category theory without an *identity*. This is the reason we start with a semi-group from group theory, then jump to a monoid in category theory once we have *empty*.

Monoids form a single object category where the morphism is `concat`, `empty` is the identity, and composition is guaranteed.

Composition as a monoid

Functions of type `a -> a`, where the domain is in the same set as the codomain, are called *endomorphisms*. We can make a *monoid* called `Endo` which captures this idea:

```
const Endo = run => ({
  run,
  concat: other =>
    Endo(compose(run, other.run))
})

Endo.empty = () => Endo(identity)



// in action

// thingDownFlipAndReverse :: Endo [String] -> [String]
const thingDownFlipAndReverse = fold(Endo(() => []), [Endo(reverse), Endo(sort)

thingDownFlipAndReverse.run(['let me work it', 'is it worth it?'])
// ['thing down', 'let me work it', 'is it worth it?']
```

Since they are all the same type, we can `concat` via `compose` and the types always line up.

Monad as a monoid

You may have noticed that `join` is an operation which takes two (nested) monads and squashes them down to one in an associative fashion. It is also a natural transformation or "functor function". As previously stated, we can make a category of functors as objects with natural transformations as morphisms. Now, if we specialize it to *Endofunctors*, that is functors of the same type, then `join` provides us with a monoid in the category of Endofunctors also known as a Monad. To show the exact formulation in code takes a little finagling which I encourage you to google, but that's the general idea.

Applicative as a monoid

Even applicative functors have a monoidal formulation known in the category theory as a *lax monoidal functor*. We can implement the interface as a monoid and recover `ap` from it:

```
// concat :: f a -> f b -> f [a, b]
// empty :: () -> f ()

// ap :: Functor f => f (a -> b) -> f a -> f b
const ap = compose(map(([f, x]) => f(x)), concat)
```

In summary

So you see, everything is connected, or can be. This profound realization makes *Monoids* a powerful modelling tool for broad swaths of app architecture to the tiniest pieces of datum. I encourage you to think of *monoids* whenever direct accumulation or combination is part of your application, then once you've got that down, start to stretch the definition to more applications (you'd be surprised how much one can model with a *monoid*).

Exercises

Appendix A: Essential Functions Support

In this appendix, you'll find some basic JavaScript implementations of various functions described in the book. Keep in mind that these implementations may not be the fastest or the most efficient implementation out there; they *solely serve an educational purpose*.

In order to find functions that are more production-ready, have a peek at ramda, lodash, or folktale.

Note that some functions also refer to algebraic structures defined in the Appendix B

always

```
// always :: a -> b -> a
const always = curry((a, b) => a);
```

compose

```
// compose :: ((y -> z), (x -> y),  ..., (a -> b)) -> a -> z
const compose = (...fns) => (...args) => fns.reduceRight((res, fn) => [fn.call(
```

curry

```
// curry :: ((a, b, ...) -> c) -> a -> b -> ... -> c
function curry(fn) {
  const arity = fn.length;

  return function $curry(...args) {
    if (args.length < arity) {
      return $curry.bind(null, ...args);
    }

    return fn.call(null, ...args);
  };
}
```

either

```
// either :: (a -> c) -> (b -> c) -> Either a b -> c
const either = curry((f, g, e) => {
  if (e.isLeft) {
    return f(e.$value);
  }

  return g(e.$value);
});
```

identity

```
// identity :: x -> x
const identity = x => x;
```

inspect

```
// inspect :: a -> String
const inspect = (x) => {
  if (x && typeof x.inspect === 'function') {
    return x.inspect();
  }

  function inspectFn(f) {
    return f.name ? f.name : f.toString();
  }

  function inspectTerm(t) {
    switch (typeof t) {
      case 'string':
        return `'${t}'`;
      case 'object': {
        const ts = Object.keys(t).map(k => [k, inspect(t[k])]);
        return `{${ts.map(kv => kv.join(': ')).join(', ')}}`;
      }
      default:
        return String(t);
    }
  }

  function inspectArgs(args) {
    return Array.isArray(args) ? `[${args.map(inspect).join(', ')}]` : inspect
  }

  return (typeof x === 'function') ? inspectFn(x) : inspectArgs(x);
};
```

left

```
// left :: a -> Either a b
const left = a => new Left(a);
```

liftA2

```
// liftA2 :: (Applicative f) => (a1 -> a2 -> b) -> f a1 -> f a2 -> f b
const liftA2 = curry((fn, a1, a2) => a1.map(fn).ap(a2));
```

liftA3

```
// liftA3 :: (Applicative f) => (a1 -> a2 -> a3 -> b) -> f a1 -> f a2 -> f a3 -
const liftA3 = curry((fn, a1, a2, a3) => a1.map(fn).ap(a2).ap(a3));
```

maybe

```
// maybe :: b -> (a -> b) -> Maybe a -> b
const maybe = curry((v, f, m) => {
  if (m.isNothing) {
    return v;
  }

  return f(m.$value);
});
```

nothing

```
// nothing :: Maybe a
const nothing = Maybe.of(null);
```

reject

```
// reject :: a -> Task a b
const reject = a => Task.rejected(a);
```

Appendix B: Algebraic Structures Support

In this appendix, you'll find some basic JavaScript implementations of various algebraic structures described in the book. Keep in mind that these implementations may not be the fastest or the most efficient implementation out there; they *solely serve an educational purpose*.

In order to find structures that are more production-ready, have a peek at folktale or fantasy-land.

Note that some methods also refer to functions defined in the Appendix A

Compose

```
const createCompose = curry((F, G) => class Compose {
  constructor(x) {
    this.$value = x;
  }

  [util.inspect.custom]() {
    return `Compose(${inspect(this.$value)})`;
  }

  // ----- Pointed (Compose F G)
  static of(x) {
    return new Compose(F(G(x)));
  }

  // ----- Functor (Compose F G)
  map(fn) {
    return new Compose(this.$value.map(x => x.map(fn)));
  }

  // ----- Applicative (Compose F G)
  ap(f) {
    return f.map(this.$value);
  }
});
```

Either

```
class Either {
  constructor(x) {
    this.$value = x;
  }

  // ----- Pointed (Either a)
  static of(x) {
    return new Right(x);
  }
}
```

Left

```
class Left extends Either {
  get isLeft() {
    return true;
  }

  get isRight() {
    return false;
  }

  static of(x) {
    throw new Error('`of` called on class Left (value) instead of Either (type)
  }

  [util.inspect.custom]() {
    return `Left(${inspect(this.$value)})`;
  }

  // ----- Functor (Either a)
  map() {
    return this;
  }

  // ----- Applicative (Either a)
  ap() {
    return this;
  }

  // ----- Monad (Either a)
  chain() {
    return this;
  }

  join() {
    return this;
  }

  // ----- Traversable (Either a)
  sequence(of) {
    return of(this);
  }

  traverse(of, fn) {
    return of(this);
```

```
    }
}
```

Right

```
class Right extends Either {
  get isLeft() {
    return false;
  }

  get isRight() {
    return true;
  }

  static of(x) {
    throw new Error('`of` called on class Right (value) instead of Either (type
  }

  [util.inspect.custom]() {
    return `Right(${inspect(this.$value)})`;
  }

  // ----- Functor (Either a)
  map(fn) {
    return Either.of(fn(this.$value));
  }

  // ----- Applicative (Either a)
  ap(f) {
    return f.map(this.$value);
  }

  // ----- Monad (Either a)
  chain(fn) {
    return fn(this.$value);
  }

  join() {
    return this.$value;
  }

  // ----- Traversable (Either a)
  sequence(of) {
    return this.traverse(of, identity);
  }

  traverse(of, fn) {
    return fn(this.$value).map(Either.of);
```

```
    }
}
```

Identity

```
class Identity {
  constructor(x) {
    this.$value = x;
  }

  [util.inspect.custom]() {
    return `Identity(${inspect(this.$value)})`;
  }

  // ----- Pointed Identity
  static of(x) {
    return new Identity(x);
  }

  // ----- Functor Identity
  map(fn) {
    return Identity.of(fn(this.$value));
  }

  // ----- Applicative Identity
  ap(f) {
    return f.map(this.$value);
  }

  // ----- Monad Identity
  chain(fn) {
    return this.map(fn).join();
  }

  join() {
    return this.$value;
  }

  // ----- Traversable Identity
  sequence(of) {
    return this.traverse(of, identity);
  }

  traverse(of, fn) {
    return fn(this.$value).map(Identity.of);
  }
}
```

IO

```
class IO {
  constructor(fn) {
    this.unsafePerformIO = fn;
  }

  [util.inspect.custom]() {
    return 'IO(?)';
  }

  // ----- Pointed IO
  static of(x) {
    return new IO(() => x);
  }

  // ----- Functor IO
  map(fn) {
    return new IO(compose(fn, this.unsafePerformIO));
  }

  // ----- Applicative IO
  ap(f) {
    return this.chain(fn => f.map(fn));
  }

  // ----- Monad IO
  chain(fn) {
    return this.map(fn).join();
  }

  join() {
    return new IO(() => this.unsafePerformIO().unsafePerformIO());
  }
}
```

List

```
class List {
  constructor(xs) {
    this.$value = xs;
  }

  [util.inspect.custom]() {
    return `List(${inspect(this.$value)})`;
  }

  concat(x) {
    return new List(this.$value.concat(x));
  }

  // ----- Pointed List
  static of(x) {
    return new List([x]);
  }

  // ----- Functor List
  map(fn) {
    return new List(this.$value.map(fn));
  }

  // ----- Traversable List
  sequence(of) {
    return this.traverse(of, identity);
  }

  traverse(of, fn) {
    return this.$value.reduce(
      (f, a) => fn(a).map(b => bs => bs.concat(b)).ap(f),
      of(new List([])),
    );
  }
}
```

Map

```
class Map {
  constructor(x) {
    this.$value = x;
  }

  [util.inspect.custom]() {
    return `Map(${inspect(this.$value)})`;
  }

  insert(k, v) {
    const singleton = {};
    singleton[k] = v;
    return Map.of(Object.assign({}, this.$value, singleton));
  }

  reduceWithKeys(fn, zero) {
    return Object.keys(this.$value)
      .reduce((acc, k) => fn(acc, this.$value[k], k), zero);
  }

  // ----- Functor (Map a)
  map(fn) {
    return this.reduceWithKeys(
      (m, v, k) => m.insert(k, fn(v)),
      new Map({}),
    );
  }

  // ----- Traversable (Map a)
  sequence(of) {
    return this.traverse(of, identity);
  }

  traverse(of, fn) {
    return this.reduceWithKeys(
      (f, a, k) => fn(a).map(b => m => m.insert(k, b)).ap(f),
      of(new Map({})),
    );
  }
}
```

Maybe

> Note that `Maybe` could also be defined in a similar fashion as we did for `Either` with two child classes `Just` and `Nothing`. This is simply a different flavor.

```
class Maybe {
  get isNothing() {
    return this.$value === null || this.$value === undefined;
  }

  get isJust() {
    return !this.isNothing;
  }

  constructor(x) {
    this.$value = x;
  }

  [util.inspect.custom]() {
    return this.isNothing ? 'Nothing' : `Just(${inspect(this.$value)})`;
  }

  // ----- Pointed Maybe
  static of(x) {
    return new Maybe(x);
  }

  // ----- Functor Maybe
  map(fn) {
    return this.isNothing ? this : Maybe.of(fn(this.$value));
  }

  // ----- Applicative Maybe
  ap(f) {
    return this.isNothing ? this : f.map(this.$value);
  }

  // ----- Monad Maybe
  chain(fn) {
    return this.map(fn).join();
  }

  join() {
    return this.isNothing ? this : this.$value;
  }

  // ----- Traversable Maybe
  sequence(of) {
    return this.traverse(of, identity);
  }

  traverse(of, fn) {
```

```
    return this.isNothing ? of(this) : fn(this.$value).map(Maybe.of);
  }
}
```

Task

```
class Task {
  constructor(fork) {
    this.fork = fork;
  }

  [util.inspect.custom]() {
    return 'Task(?)';
  }

  static rejected(x) {
    return new Task((reject, _) => reject(x));
  }

  // ----- Pointed (Task a)
  static of(x) {
    return new Task((_, resolve) => resolve(x));
  }

  // ----- Functor (Task a)
  map(fn) {
    return new Task((reject, resolve) => this.fork(reject, compose(resolve, fn
  }

  // ----- Applicative (Task a)
  ap(f) {
    return this.chain(fn => f.map(fn));
  }

  // ----- Monad (Task a)
  chain(fn) {
    return new Task((reject, resolve) => this.fork(reject, x => fn(x).fork(reje
  }

  join() {
    return this.chain(identity);
  }
}
```

Appendix C: Pointfree Utilities

In this appendix, you'll find pointfree versions of rather classic JavaScript functions described in the book. All of the following functions are seemingly available in exercises, as part of the global context. Keep in mind that these implementations may not be the fastest or the most efficient implementation out there; they *solely serve an educational purpose*.

In order to find functions that are more production-ready, have a peek at ramda, lodash, or folktale.

Note that functions refer to the `curry` & `compose` functions defined in Appendix A

add

```
// add :: Number -> Number -> Number
const add = curry((a, b) => a + b);
```

append

```
// append :: String -> String -> String
const append = flip(concat);
```

chain

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

concat

```
// concat :: String -> String -> String
const concat = curry((a, b) => a.concat(b));
```

eq

```
// eq :: Eq a => a -> a -> Boolean
const eq = curry((a, b) => a === b);
```

filter

```
// filter :: (a -> Boolean) -> [a] -> [a]
const filter = curry((fn, xs) => xs.filter(fn));
```

flip

```
// flip :: (a -> b -> c) -> b -> a -> c
const flip = curry((fn, a, b) => fn(b, a));
```

forEach

```
// forEach :: (a -> ()) -> [a] -> ()
const forEach = curry((fn, xs) => xs.forEach(fn));
```

head

```
// head :: [a] -> a
const head = xs => xs[0];
```

intercalate

```
// intercalate :: String -> [String] -> String
const intercalate = curry((str, xs) => xs.join(str));
```

join

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

last

```
// last :: [a] -> a
const last = xs => xs[xs.length - 1];
```

map

```
// map :: Functor f => (a -> b) -> f a -> f b
const map = curry((fn, f) => f.map(fn));
```

match

```
// match :: RegExp -> String -> Boolean
const match = curry((re, str) => re.test(str));
```

prop

```
// prop :: String -> Object -> a
const prop = curry((p, obj) => obj[p]);
```

reduce

```
// reduce :: (b -> a -> b) -> b -> [a] -> b
const reduce = curry((fn, zero, xs) => xs.reduce(fn, zero));
```

replace

```
// replace :: RegExp -> String -> String -> String
const replace = curry((re, rpl, str) => str.replace(re, rpl));
```

reverse

```
// reverse :: [a] -> [a]
const reverse = x => (Array.isArray(x) ? x.reverse() : x.split('').reverse().jo
```

safeHead

```
// safeHead :: [a] -> Maybe a
const safeHead = compose(Maybe.of, head);
```

safeLast

```
// safeLast :: [a] -> Maybe a
const safeLast = compose(Maybe.of, last);
```

safeProp

```
// safeProp :: String -> Object -> Maybe a
const safeProp = curry((p, obj) => compose(Maybe.of, prop(p))(obj));
```

sequence

```
// sequence :: (Applicative f, Traversable t) => (a -> f a) -> t (f a) -> f (t
const sequence = curry((of, f) => f.sequence(of));
```

sortBy

```
// sortBy :: Ord b => (a -> b) -> [a] -> [a]
const sortBy = curry((fn, xs) => xs.sort((a, b) => {
  if (fn(a) === fn(b)) {
    return 0;
  }

  return fn(a) > fn(b) ? 1 : -1;
}));
```

split

```
// split :: String -> String -> [String]
const split = curry((sep, str) => str.split(sep));
```

take

```
// take :: Number -> [a] -> [a]
const take = curry((n, xs) => xs.slice(0, n));
```

toLowerCase

```
// toLowerCase :: String -> String
const toLowerCase = s => s.toLowerCase();
```

toString

```
// toString :: a -> String
const toString = String;
```

toUpperCase

```
// toUpperCase :: String -> String
const toUpperCase = s => s.toUpperCase();
```

traverse

```
// traverse :: (Applicative f, Traversable t) => (a -> f a) -> (a -> f b) -> t
const traverse = curry((of, fn, f) => f.traverse(of, fn));
```

unsafePerformIO

```
// unsafePerformIO :: IO a -> a
const unsafePerformIO = io => io.unsafePerformIO();
```