


WHAT'S SO GREAT ABOUT FUNCTIONAL PROGRAMMING ANYWAY?



To hear some people talk about functional programming, you'd think they'd joined some kind of cult. They prattle on about how it's changed the way they think about code. They'll extol the benefits of *purity*, at length. And proclaim that they are now able to "reason about their code"—as if all other code is irrational and incomprehensible. It's enough to make anyone skeptical.

Still, one has to wonder. There must be a reason these zealots are so excited. In my personal experience, it wasn't the lazy, incompetent programmers who developed an interest functional programming.¹ Instead, the most intelligent coders I knew tended to take it up; the people most

¹ One of the people I showed this to had an interesting reaction. Their response was something like: "Hey! I like functional programming *because* I'm lazy and incompetent. It's about all the things I don't have to think about."

passionate about writing good code. (Though, they did tend towards the boffin end of the spectrum.) And this raises the question: What are they so excited about?

Faced with this question, most educators will start with the basics. They'll take you to the metaphorical baby pool. And they'll try to explain to you what functional programming *is*. They'll talk about "coding with expressions" and side-effects, and purity, and ... they mean well. But telling people what functional programming *is* doesn't explain what functional programming is *good for*.

Lets' be honest. Nobody cares what functional programming *is*, at least, not at first. What we care about is "can we deliver better code, faster?" And our project managers care about those in reverse order. Instead, let's try something different and skip the baby pool. Rather than talking about the definition of functional programming, we'll go straight to the good parts. Let's talk about Algebraic Structures.

§ 2.1. ALGEBRAIC STRUCTURES

Algebraic structures allow us to write expressive code, with more confidence. They're expressive because they convey a wealth of information. They tell us how code can be re-used, optimised, and rearranged. And all these with complete confidence we won't break anything. In some cases, they even enable automatic code generation.

These are bold claims. But by the end of this chapter, we will have demonstrated both:

- Reusable code; and
- Performance optimisation with guaranteed safety.

Furthermore, in later chapters, we'll show how algebraic structures allow our code to convey more information.

WHAT ARE ALGEBRAIC STRUCTURES?

If they're so good, what are algebraic structures? In short, they're what lots of people consider the scary bits of functional programming. They include concepts like 'monoids', 'semigroups', 'functors', and the dreaded 'monad.' They're also super abstract—in the literal sense. Algebraic structures are abstractions of abstractions. In this way, they are a little bit like *design patterns*, such as those described in the 'gang of four' book *Design Patterns: Elements of Reusable Object-Oriented Software*.² But they have some significant differences too.

Once again though, instead of focussing on what they *are*, let's start with what they can do.

A REAL WORLD PROBLEM

If we want to see what functional programming (and algebraic structures) are good for, there's no point solving toy problems. We can do better than adding two numbers together. Instead, let's look at something JavaScript developers deal with often.

Let's imagine we're working on a web application. We have a list of notifications to display to the user. And we have them in an array of plain ol' JavaScript objects (POJOS). But, we need to transform them into a format that the front-end UI code can handle. Suppose the data looks something like this:

```
const notificationData = [
  {
    username: 'sherlock',
    message: 'Watson. Come at once if convenient.',
    date: -1461735479,
    displayName: 'Sherlock Holmes',
```

² Gamma et. al. (1994), *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley.

```

    id: 221,
    read: false,
    sourceId: 'note-to-watson-1895',
    sourceType: 'note',
  },
  {
    username: 'sherlock',
    message: 'If not convenient, come all the same.',
    date: -1461735359,
    displayName: 'Sherlock Holmes',
    id: 221,
    read: false,
    sourceId: 'note-to-watson-1895',
    sourceType: 'note',
  },
  // ... and so on. Imagine we had lots more entries here.
];

```

Now, to convert this data so our templating system can handle it, we need to do the following:

1. Generate a readable date;
2. Sanitise the message to prevent cross-site scripting (xss) attacks;
3. Build a link to the sender's profile page;
4. Build a link to the source of the notification; and
5. Tell the template what icon to display, based on the source type.

To get started, we write a function for each:³

³ Please don't write your own xss sanitizing function. Use a battle-tested library, or let your view library (like React) take care of it. This is but an example for educational purposes.

```

const getSet = (getKey, setKey, transform) => (obj) =>
({
  ...obj,
  [setKey]: transform(obj[getKey]),
});
const addReadableDate = getSet(
  'date',
  'readableDate',
  t => new Date(t * 1000).toGMTString()
);
const sanitizeMessage = getSet(
  'message',
  'message',
  msg => msg.replace(/</g, '&lt;');
);
const buildLinkToSender = getSet(
  'username',
  'sender',
  u => `https://example.com/users/${u}`
);

const buildLinkToSource = (notification) => ({
  ...notification,
  source: `https://example.com/${
    notification.sourceType
  }/${notification.sourceId}`
});
const iconPrefix = 'https://example.com/assets/icons/';
const iconSuffix = '-small.svg';
const addIcon = getSet(

```

```
'sourceType',
'icon',
sourceType => `${urlPrefix}${sourceType}${iconSuffix}`
);
```

One way to wire all these together is to run them one-by-one, and store the results in named variables. For example:

```
const withDates = notificationData.map(addReadableDate);
const sanitized = withDates.map(sanitizeMessage);
const withSenders = sanitized.map(buildLinkToSender);
const withSources = withSenders.map(buildLinkToSource);
const dataForTemplate = withSources.map(addIcon);
```

Those interstitial variables don't add any new information, though. We can see what's going on from the name of the function we're mapping. Another way to wire it up would be to use some boring old JavaScript array method chaining. And as we do that, the code starts to look a little bit 'functional.'

```
const dataForTemplate = notificationData
  .map(addReadableDate)
  .map(sanitizeMessage)
  .map(buildLinkToSender)
  .map(buildLinkToSource)
  .map(addIcon);
```

Now, while this is truly 'functional' code, it's not overly special. Weren't we supposed to be talking about the wondrous benefits of algebraic structures?

Bear with me. We're going to rewrite this code using a couple of helper functions. The first is not complicated. We'll write a `map()` function that,

well, calls `.map()`.⁴

```
const map = f => functor => functor.map(f);
```

Next, we write a `pipe()` function that lets us ‘pipe’ a value through a series of functions. It’s a variation on function composition.⁵

```
const pipe = (x0, ...funcs) => funcs.reduce(  
  (x, f) => f(x),  
  x0  
);
```

The `pipe` function uses the spread operator to turn all but the first argument into an array. Then it passes that first argument to the first function. And the result of that to the next function. And so on.

Now we can rewrite our transform code like so:

```
const dataForTemplate = pipe(  
  notificationData,  
  map(addReadableDate),  
  map(sanitizeMessage),  
  map(buildLinkToSender),  
  map(buildLinkToSource),  
  map(addIcon)  
);
```

The first thing to notice here, is that it looks a lot like the previous version using chained methods. But aside from that, it’s still rather banal

⁴ If you’re not used to seeing arrow functions that return arrow functions like that, check out the appendix on higher-order functions. We’ll also explain this a bit more in Chapter 3 when we discuss currying.

⁵ If you’re not familiar with function composition, you can read more in the blog post *JavaScript function composition: What’s the big deal?* <https://jrsinclair.com/articles/2022/javascript-function-composition-whats-the-big-deal/>

code. We can map over an array, so what? And worse still, it's inefficient.⁶
Hang in there. It's about to get more interesting.

§ 2.2. MAYBE

For the sake of argument, let's change the scenario a little. Instead of a list of notifications, suppose we've received the most recent one. But, we don't have complete confidence in our server. On occasion, something goes wrong, and it sends us an HTML page instead of JavaScript object notation (JSON) data. And we end up with `undefined` rather than a notification.

Now, one way to handle this would be to litter our code with if-statements. First, we catch the error, and return `undefined` if the response doesn't parse.

```
const parseJSON = (dataFromServer) => {
  try {
    const parsed = JSON.parse(dataFromServer);
    return parsed;
  } catch (_) {
    return undefined;
  }
};
```

Then we add if-statements to each of our utility functions.

```
const addReadableDate = (notification) => {
  if (notification !== undefined) {
    return getSet(
      'date',
```

⁶ We'll come back to this in Chapter 4.


```

        'readableDate',
        t => new Date(t * 1000).toGMTString()
    )(notification);
} else {
    return undefined;
}
}

const sanitizeMessage = (notification) => {
    if (notification !== undefined) {
        return getSet(
            'message',
            'message',
            msg => msg.replace(/</g, '&lt;')
        )(notification)
    } else {
        return undefined;
    }
};

const buildLinkToSender = (notification) => {
    if (notification !== undefined) {
        return getSet(
            'username',
            'sender',
            u => `https://example.com/users/${u}`
        );
    } else {
        return undefined;
    }
};

const buildLinkToSource = (notification) => {

```

```

if (notification !== undefined) {
  return ({
    ...notification,
    source: `https://example.com/${
      notification.sourceType
    }/${notification.sourceId}`
  });
} else {
  return undefined;
}
};

const iconPrefix = 'https://example.com/assets/icons/';
const iconSuffix = '-small.svg';
const addIcon = (notification) => {
  if (notification !== undefined) {
    getSet(
      'sourceType',
      'icon',
      sourceType =>
        `${urlPrefix}${sourceType}${iconSuffix}`
    );
  } else {
    return undefined;
  }
};
};

```

After all that, our main `pipe()` call still looks the same.

```

const dataForTemplate = pipe(
  notificationData,
  map(addReadableDate),

```

```
    map(sanitizeMessage),
    map(buildLinkToSender),
    map(buildLinkToSource),
    map(addIcon)
  );
```

But, as you can see, it makes our individual functions verbose and repetitive. Surely there must be an alternative? And indeed, there is. We'll write a pair of functions like so:

```
const Just = (val) => ({
  map: f => Just(f(val)),
});

const Nothing = () => {
  const nothing = { map: () => nothing };
  return nothing;
};
```

Both `Just` and `Nothing` return an object with a `.map()` method. When used together, we call this pair a `Maybe`. And we use it like so:

```
const parseJSON = (data) => {
  try {
    return Just(JSON.parse(data));
  } catch () {
    return Nothing();
  }
}

const notificationData = parseJSON(dataFromServer);
```

With that in place, let's look at our mapping code. In this new sce-

nario, we're no longer working with arrays. Instead, we have a single value that may be `Nothing`. Or, it may be `Just` a notification. But, as a reminder, here's the code we had for arrays again:

```
const dataForTemplate = pipe(
  notificationData,
  map(addReadableDate),
  map(sanitizeMessage),
  map(buildLinkToSender),
  map(buildLinkToSource),
  map(addIcon)
);
```

What do we need to make this work with `Maybe` a single value? Almost nothing. All we need is a way to get our value out of the `Just` wrapper at the end. To do that, we'll add another method to `Just` and `Nothing`.

```
const Just = (val) => ({
  map: f => Just(f(val)),
  reduce: (f, x0) => f(x0, val),
});

const Nothing = () => {
  const nothing = {
    map: () => nothing,
    reduce: (_, x0) => x0,
  };
  return nothing;
};
```

Notice how we've added `reduce()` to both `Just` and `Nothing`. That allows us write a stand-alone `reduce()` function, much like we did for `map()`:

```
const reduce = (f, x0) => foldable =>
  foldable.reduce(f, x0);
```

If we want to get our value out of a `Just`, we can call `reduce()` like so:

```
reduce( (_, val) => val, fallbackValue);
```

If `reduce()` encounters `Nothing`, it will return the fallback value. Otherwise, it will ignore fallback value and returns the data.

So the pipeline would look like so:

```
const dataForTemplate = pipe(
  notificationData,
  map(addReadableDate),
  map(sanitizeMessage),
  map(buildLinkToSender),
  map(buildLinkToSource),
  map(addIcon),
  reduce( (_, val) => val, fallbackValue),
);
```

Now, you may be wondering, why all this rigmarole with `.reduce()`? Why not add a method that provides the fallback value straight away? For example:

```
const Just = (val) => ({
  map: f => Just(f(val)),
  fallbackTo: ( _) => val,
});

const Nothing = () => {
```

```

const nothing = {
  map: () => nothing,
  fallbackTo: (x0) => x0,
};
return nothing;
};

```

Once again, because we've added `.fallbackTo()` to both, we can write another utility function. This will work regardless of whether we get `Just` or `Nothing`. It will do what we expect either way.

```

const fallbackTo = (x0) => (m) => m.fallbackTo(x0);

```

This utility function, `fallbackTo()` is concise and effective. Why bother with `reduce()`?

It's a good question. At first glance, it appears to be the kind of needlessly complicated code that makes functional programmers so annoying. Always adding in layers of abstraction that make code harder to read, and confusing for juniors. Right?

There's a good reason for using `reduce()` instead of `fallbackTo()`, though. Because `reduce()` can work with other data structures besides `Just` and `Nothing`. It's portable code. The truth is, for this code, we can replace `Just` and `Nothing` with something else. What would happen if we rewrote the parsing code like this:

```

const parseJSON = strData => {
  try { return [JSON.parse(strData)]; }
  catch () { return []; }
};

const notificationData = parseJSON(dataFromServer);

```

Instead of using `Just` and `Nothing`, we're now returning plain ol' JavaScript arrays. If we look at our pipeline again:

```
const dataForTemplate = pipe(
  notificationData,
  map(addReadableDate),
  map(sanitizeMessage),
  map(buildLinkToSender),
  map(buildLinkToSource),
  map(addIcon),
  reduce((_, val) => val, fallbackValue),
);
```

We don't change a single line. But it still produces the same result.

§ 2.3. RESULT

Let's stick with this scenario a moment longer. In our JSON parsing code, we ignore the error in the `catch` clause. But, what if that error has useful information inside? We may want to log the error somewhere so we can debug issues.

Let's go back to our old `Just/Nothing` code. We'll switch out `Nothing` for a slightly different function, `Err`. And while we're at it, we'll also rename `Just` to `OK`.

```
const OK = (val) => ({
  map: (f) => OK(f(val)),
  reduce: (f, x0) => f(x0, val),
});

const Err = (e) => ({
```

```

const err = {
  map: (_) => err,
  reduce: (_, x0) => x0,
};
return err;
});

```

We'll call this new pair of functions, `Result`.⁷ With that in place, we can change our `parseJSON()` code so that it uses `Result`.

```

const parseJSON = strData => {
  try { return OK(JSON.parse(strData)); }
  catch (e) { return Err(e); }
}

const notificationData = parseJSON(dataFromServer);

```

Now, instead of ignoring the error, we capture it in an `Err` object. If we go back to the pipeline, we don't have to change anything. Since `Err` has compatible `.map()` and `.reduce()` methods, it still works.

```

const dataForTemplate = pipe(
  notificationData,
  map(addReadableDate),
  map(sanitizeMessage),
  map(buildLinkToSender),
  map(buildLinkToSource),
  map(addIcon),
  reduce(_, val) => val, fallbackValue),
);

```

⁷ You'll find functional libraries often offer a similar structure called 'Either'.

Of course, we're still ignoring the error when we get to that final `reduce()`. To fix that, we need to make a firm decision about what we want to do with that error. Do we want to log it to the console, introducing a side-effect? Do we want to send it over the network to a logging platform? Or do we want to extract something from it and display it to the user?

For now, let's assume we're OK with a small side effect, and we'll log it to the console. We add a `.peekErr()` method to both `OK` and `Err` like so:

```
const OK = (val) => ({
  map: (f) => OK(f(val)),
  reduce: (f, x0) => f(x0, val),
  peekErr: () => OK(val),
});

const Err = (e) => {
  const err = {
    map: (_) => err,
    reduce: (_, x0) => x0,
    peekErr: (f) => { f(x); return err; }
  }
  return err;
};
```

The version we add to `OK` does nothing, because there's no error to peek at. But having it there allows us to write a utility function that works with both `OK` and `Err`.

```
const peekErr = (f) => (result) => result.peekErr(f);
```

Then we can add `peekErr()` to our pipeline:

```

const dataForTemplate = pipe(
  notificationData,
  map(addReadableDate),
  map(sanitizeMessage),
  map(buildLinkToSender),
  map(buildLinkToSource),
  map(addIcon),
  peekErr(console.warn),
  reduce( (_, val) => val, fallbackValue),
);

```

If there happens to be an error, we log it and move on. If we needed more complex error handling, we might use other structures.

Of course, adding `peekErr()` breaks compatibility with Arrays and the Maybe structure. And that's fine. Arrays and Maybe don't have this extra error data to deal with.

§ 2.4. TASK

Now, this is all well and good, but we've been ignoring something important. All along we've been saying that this data comes from a server. But retrieving data from a server implies that there's some sort of network call involved. And in JavaScript, that most often means asynchronous code.

For example, suppose we have some code that fetches our notification data using standard JavaScript promises:

```

const notificationDataPromise = fetch(urlForData)
  .then(response => response.json());

```

Let's see if we can build a structure that works for asynchronous code

too. To do this, we're going to create a structure with a constructor function much like a Promise. It expects a function that takes two arguments:

1. One to call on successful resolution; and
2. Another to call if something goes wrong.

We can call it like so:

```
const notificationData = Task((resolve, reject) => {
  fetch(urlForData)
    .then(response => response.json())
    .then(resolve)
    .catch(reject);
});
```

In this example, we call `fetch` and pass it the URL for our notification. Then we call `.json()` on the response to parse the data. And with that done, we `resolve()` if the call was successful, or `reject()` if it wasn't. It looks a little awkward compared to the Promise-only `fetch()` code. But that's so we can wire up the `resolve` and `reject`. We'll add a helper for wiring up asynchronous functions like `fetch()` in a moment.

The implementation for our `Task` structure is not too complex:

```
const Task = (run) => {
  map: (f) => Task((resolve, reject) => {
    run(
      (x) => (resolve(f(x))),
      reject
    );
  }),
  peekErr: (f) => Task((resolve, reject) => {
    run(
      resolve,
```

```

      (err) => { f(err); reject(err); }
    )
  }),
  run: (onResolve, onReject) => run(
    onResolve,
    onReject
  );
}

```

We have `.map()` and `.peekErr()`, as we did for `Result`. But a `.reduce()` method doesn't make sense for asynchronous code. Once you go asynchronous, you can never go back. We've also added a `.run()` method to kick off our `Task`.

To make working with Promises a little easier, we can add a static helper to `Task`. And another helper for fetching JSON data:

```

Task.fromAsync = (asyncFunc) => (...args) =>
  Task((resolve, reject) => {
    asyncFunc(...args).then(resolve).catch(reject);
  });

const taskFetchJSON = Task.fromAsync(
  (url) => fetch(url).then(data => data.json())
);

```

With those helpers, we can define `notificationData` like so:

```

const notificationData = taskFetchJSON(urlForData);

```

To work with `Task`, we need to change our pipeline a little. But it's a small change:

```

const dataForTemplate = pipe(
  notificationData,
  map(addReadableDate),
  map(sanitizeMessage),
  map(buildLinkToSender),
  map(buildLinkToSource),
  map(addIcon),
  peekErr(console.warn),
);

```

Most of it still works, except for the `reduce()` function. But we still want some way to introduce a fallback value if the network request or parsing fails. To make that happen, we'll add a method called `.scan()`. It will be like `.reduce()`, but we give it a different name to acknowledge that the result will still be 'inside' a `Task`.

```

const Task = (run) => {
  map: (f) => Task((resolve, reject) => {
    run(
      (x) => (resolve(f(x))),
      reject
    );
  }),
  peekErr: (f) => Task((resolve, reject) => {
    run(
      resolve,
      (err) => { f(err); reject(err); }
    )
  }),
  run: (onResolve, onReject) => run(
    onResolve,

```

```

    onReject
  );
  scan: (f, x0) => Task((resolve, reject) => run(
    x => resolve(f(x0, x)),
    e => resolve(x0),
  )),
}

```

Notice that `.scan()` doesn't call `reject()`. This is because it's analogous to `.reduce()`. It lets us fall back to a default value if we have an error.

And, as usual, we'll create a matching utility function:

```

const scan = (f, x0) => (scannable) =>
  scannable.scan(f, x0);

```

With that in place, we can adjust our pipeline like so:

```

const taskForTemplateData = pipe(
  notificationData,
  map(addReadableDate),
  map(sanitizeMessage),
  map(buildLinkToSender),
  map(buildLinkToSource),
  map(addIcon),
  peekErr(console.warn),
  scan((_, val) => val, fallback)
);

```

And to run it, we do something like this:

```
taskForTemplateData.run(  
  renderNotifications,  
  handleError  
);
```

WHY NOT USE PROMISES?

Someone might be wondering, JavaScript already has a built-in data structure for asynchronous code. Why not use Promises? Why bother with this Task business? What's the point, if it's going to confuse everyone?

There's at least three reasons. The first is that Promises don't have a `.run()` method. This means they kick off as soon as you create them. Using `Task` gives us precise control over when everything starts.

Now, we don't *need* `Task` to get this control. If we want to, we can delay our Promises by putting them inside a function. Then, the Promise won't 'kick off' until we call the function. Along the way though, we've as-good-as reinvented `Task`. But with a different syntax and less flexibility.

The second reason for preferring `Task` is it has abilities Promises don't. The main one is being able to nest Tasks. We can run a `Task`, and get back another `Task`. We can then wait and decide when to run that next `Task`. This isn't possible with Promises.⁸ Promises smooch `.map()` and `.flatMap()` together into a single `.then()` method. And as a consequence, we lose flexibility (again).

The final reason for preferring `Task` is that it's consistent with other algebraic structures. If we keep using these structures often enough, they become familiar. And in turn, it becomes easier to make inferences about what the code is doing. Or (more importantly) *not* doing. We'll discuss this further in a moment.

⁸ At least, it's impossible with Promises, unless you return a function that returns a Promise. But, as discussed, a function that returns a Promise is another way of constructing a `Task`.

In summary, Task gives us more power, flexibility, and consistency. This isn't to say that there's no tradeoffs using tasks. With the `async . . . await` keywords, JavaScript supports Promises 'out of the box'. We may not want to give up that convenience to use Tasks. And that's okay.

§ 2.5. SO YOU USED POLYMORPHISM. BIG DEAL.

We started this chapter asking the question “What’s so great about functional programming?” But all we’ve done so far is talk about a handful of objects that share some method names. That’s plain old polymorphism. Object Oriented Programming (oop) gurus have been banging on about polymorphism for decades. We can’t claim that functional programming is awesome because it uses polymorphism.

Or can we?

It’s not polymorphism itself that makes algebraic structures (and functional programming) so awesome. But polymorphism makes algebraic structures possible in JavaScript. In our notifications example, we defined some methods with matching names and signatures. For example, `.map()` and `.reduce()`. Then we wrote utility functions that work with methods matching those signatures, for example, `map()` and `reduce()`. Polymorphism makes those utility functions work.

Those method definitions (and utility functions) aren’t arbitrary. They’re not design patterns that someone made up by observing common architectural patterns. No, algebraic structures come from mathematics; from fields like set theory and category theory. This means that, as well as specific method signatures, these structures come with *laws*.

At first, this doesn’t sound wonderful. We associate mathematics with confusion and boredom. And we associate laws with restriction. Laws get in our way. They stop us doing what we want. They’re inconvenient. But if you take a moment to read these laws, they may surprise you. Because they’re *boring*. Incredibly boring.

Now, you may be thinking “I’m not sure why you thought that would be surprising. That’s the least surprising thing ever.” But these laws are a particular kind of boring. They’re boring in the sense that they state the obvious. The kind of thing where you wonder why anyone bothered to write it down. We read them and tend to think “Of course it works like that. In what scenario would it ever be different?” And that, there, is the beauty of algebraic structures.

To illustrate, let’s look back at our notifications example. We’ve made use of at least two algebraic structures. One of them, we call Functor. All that means is that in `Maybe`, `Result`, and `Task`, we wrote a `.map()` method. And the way we’ve written those `.map()` methods, each one follows some laws. We also used another algebraic structure called Foldable. We call a data structure Foldable if it has a `.reduce()` method, and that method obeys some laws.

One of the laws for Functor says that the following two pieces of code must always produce the same result. No matter what. Assuming we have two pure functions, `f`, and `g`, our first piece of code is:

```
const resultA = a.map(f).map(g);
```

And the second piece of code is:

```
const resultB = a.map(x => g(f(x)));
```

These two pieces of code must produce the same result when given the same input. That is, `resultA ≡ resultB`. We call this the composition rule. And we can apply it to our pipeline code. Because `x => g(f(x))` is the same as writing `x => pipe(x, f, g)`. That is, our `pipe()` function is a form of composition. Thus, if we go all the way back to the array-based version of our pipeline, we have:

```
const dataForTemplate = pipe(
  notificationData,
  map(addReadableDate),
  map(sanitizeMessage),
  map(buildLinkToSender),
  map(buildLinkToSource),
  map(addIcon),
);
```

We can rewrite it as:

```
const dataForTemplate = map(x => pipe(x,
  addReadableDate,
  sanitizeMessage,
  buildLinkToSender,
  buildLinkToSource,
  addIcon,
)) (notificationData);
```

Because of the composition law, we know these two pieces of code are equivalent. It doesn't matter if we're working with a Maybe, Result, Task, or an Array. These two pieces of code will always produce the same result.

Now, it's possible, that doesn't look like a big deal to you. And you may even think the second version is uglier, and overly complex. But for arrays, that second version will be more efficient. The first version will produce at least five intermediate arrays as it passes data through the pipe. The second version does it all in one pass. We get a performance improvement that's guaranteed to produce the same result as the code we started with. Well, guaranteed, so long as we're using pure functions.

It's all about confidence. Those laws tell me that if I use an algebraic structure, it will behave as I expect. And I have a mathematical guarantee that it will continue to do so. 100%. All the time.

As promised, we've demonstrated code that we can re-use. Our utility functions like `map()`, `reduce()` and `pipe()` work with a bunch of different structures. Structures like `Array`, `Maybe`, `Either`, and `Task`. And we showed how the laws of algebraic structures helped us rearrange the code with complete safety. And showed how that rearrangement provided a performance improvement. Again, with complete confidence.

This, in turn, gets to the heart of what's great about functional programming. It's not mainly about algebraic structures. They're but one set of tools in a gigantic tool chest. Functional programming is all about having confidence in our code. It's about knowing that our code is doing what we expect, and *nothing but* what we expect.

Once we understand this, the eccentricities of functional programming start to make a little more sense. This is why, for example, functional programmers are so careful about side-effects. The more we work with pure functions, the more certainty we gain. It also explains the love affair some programmers have with fancy type systems like the one in Haskell.⁹ They're addicted to the drug of certainty.

This knowledge—that functional programming is about confidence in your code—is like having a secret key. It explains why functional programmers get all worked up about ostensibly trivial matters. It's not that they enjoy pedantry. (Well, okay, *some* of them appear to enjoy pedantry a lot). Most of the time, they're fighting to preserve confidence. And they're willing to do whatever it takes. Even if it involves delving into the dark arts of mathematics.

⁹ To be clear, not all functional programmers end up besotted with fancy type systems.