O'REILLY®

Why Elm?

Robust, Functional Programming for the Web Frontend

Matthew Griffith

Why Elm?

Matthew Griffith

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Why Elm?
by Matthew Griffith

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Editors: Dawn Schanafelt and Meg Foley
Production Editor: Shiny Kalapurakkel
Copyeditor: Rachel Head
Proofreader: Amanda Kersey
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

March 2017: First Edition

Revision History for the First Edition
2017-03-09: First Release

978-1-491-98000-2

[LSI]

Table of Contents

CHAPTER 1
Why Elm?

Elm is a functional language that compiles to JavaScript and is specifically for designing the frontend of your web app. One of the main design goals of Elm is to incorporate some of the advances from the last 40 years of programming design (many of which have not made the full transition from academia to common use), and to not require you to learn a bunch of jargon to benefit from those advances. The practical result is that Elm code is fast, hard to break, easily testable, and profoundly maintainable. Above all, Elm is a functional programming language for the *practical* frontend developer.

This report is meant for developers who are largely familiar with JavaScript and other dynamically typed, imperative languages, but who aren't quite as familiar with static types or purely functional programming languages. If you're an established web developer, you might feel a bit overwhelmed by trying to keep track of the recent explosion of frontend technologies—so what makes Elm fundamentally different?

Here's quick overview of some of Elm's best features; we'll cover these topics in more detail later in this report.

Elm eliminates many of the most common pain points of frontend development. That's because many of the common issues that front-end developers have to deal with *just don't exist in Elm*. There is no `null` and no confounding errors such as `undefined is not a function`, but that's just the beginning. Elm really shines when we talk about the *higher-level benefits* it can provide to you.

In Elm it's extremely common to have no runtime exceptions in practice. While you technically *can* have a runtime error in some limited cases, it's actually fairly difficult to accomplish. Instead of runtime exceptions, in Elm you have compile-time errors that check your work. Take a moment to think about this. You can rephrase it as, *Elm ensures that errors will happen at compile time, in front of a developer, instead of runtime, in front of a user*. This is fantastic! Is there any case where you'd *ever* want a user to receive an error instead of a developer?

Elm's compiler is your friend and provides *very specific, human-friendly compile errors* as you develop. These errors generally explain *why* code won't work instead of just *what broke*. They also tend to include hints such as links to recommended design documentation and highly accurate spelling suggestions based on what the compiler knows about your code. Features such as this go a long way toward making the Elm development experience smooth and productive.

Packages in Elm have *enforced semantic versioning*, meaning the version number of an Elm package will tell you whether the package API has been broken or extended, or whether this is simply a patch release fixing an internal detail. Patch changes will *never* break the API. This is enforced automatically on all packages, so you don't need to worry about whether third-party developers are following semantic versioning best practices. The confidence that this builds goes both ways. As a package author, it means you won't break your API by accident.

The reason Elm can provide all these benefits is because of how it handles *types*, though Elm types are likely very different from what you're used to. They don't require a bunch of extra work. In fact, *type annotation is entirely optional* because Elm can infer all the types of your program. This makes Elm types lightweight and easy to work with.

Even though type annotation is optional, Elm types are always there to provide you the benefits we've been talking about. These benefits aren't something you have to *activate* or work hard to get right; they're built into the language. Elm's type system is mandatory (which is how it guarantees no runtime exceptions, superb maintainability of your code, and enforced semantic versioning) but, again, is lightweight and surprisingly conducive to productivity. Even though Elm's type annotations are optional, you may find them incredibly useful as enforced documentation and as a powerful design tool.

Of course, having a fantastic type system doesn't mean you don't have to test your code. What it *does* mean is that you will write fewer tests, and those tests will be more meaningful and easier to write. There are many situations where you don't need to worry about testing in Elm because of the type system. For the areas where we do need testing, Elm code is inherently easy to test because each function can be realistically tested in isolation of all others. This a somewhat subtle concept which we'll cover later, but the gist is that *Elm code is made to be easily testable*.

Keep in mind that *not all typed languages can give you these same benefits*. Java makes you write a lot of boilerplate for your types, but will still throw runtime exceptions. TypeScript can provide incremental benefit to your JavaScript, but only so far as you're willing to put in the work and invoke the right options; even when you do, there is no guarantee that runtime errors, enforced semantic versioning, or developer-friendly error messages won't come up. It's not just types that deliver these benefits, but also Elm's design focus of leveraging types into high-level benefits for the developer.

All these practical properties make Elm code profoundly maintainable. Not just maintainable by a devoloper looking at your code five years from now, but maintainable in a way that immediately and directly affects your developer experience. You're guaranteed that adding code to a large Elm codebase won't break existing code. If there *is* something wrong, the compiler will likely tell you. The strength of Elm's compiler leads to developers having *no fear when refactoring*. Old, unnecessary code can be removed with confidence. Projects can grow as they need to, and development on those projects generally doesn't slow down significantly as they get bigger. *Maintainability* is Elm's killer feature.

Elm's emphasis on maintainability means that even after a few months, *returning to a codebase to add a feature is generally trivial*. It also means you don't have to worry that beginners will silently break existing codebases. Eliminating so many of the common pain points of frontend development allows you to focus on more valuable problems such as design and business logic. (On a more subjective note, for me and many others, Elm's maintainability and developer-friendliness has made frontend programming a blast.)

Another nifty aspect of Elm is that you can *adopt it incrementally*; there's no need to rewrite your entire application in Elm to try it out. Elm compiles to JavaScript and generally renders some HTML, so integrating some Elm code into an existing project is as simple as including a JavaScript file and telling it which HTML node to render to. Existing JavaScript code can easily talk to Elm code through Elm ports.

To get started with Elm, you don't need to know many of the things I'm going to be talking about. I highly recommend starting by writing some Elm code! You can do that by trying out the Elm online editor, or just go ahead and install the Elm platform to start developing for real. I also recommend joining the Elm Slack channel; there are many friendly developers who love helping those who are just getting started. You should also check out the Elm architecture, which is the official guide to Elm written by Evan Czaplicki, Elm's creator.

Elm is designed to make sure you benefit from the strong design of the language without *requiring* you to know the theoretical underpinnings that make it strong. This report is a brief introduction to the language that focuses mostly on the benefits you can expect from Elm and what makes them possible. I'll also compare Elm to other popular technologies to give you some perspective. Let's jump right in.

CHAPTER 2

Reading Elm

It's a little hard to talk too much about Elm without showing some code. This report isn't meant to be a comprehensive guide, but merely to give you a feel for what the language looks like and how it works. So, let's see what some Elm code looks like! (I will warn you that if you're not familiar with this sort of programming, the code presented in this chapter may feel a little weird. I encourage you to squint your eyes and persevere.)

Here's a basic function called `mathPlease` that takes two numbers, `x` and `y`, and does some math with them:

```
mathPlease x y = (x + y) * 5
```

Elm is an expression-oriented language. There's no `return` statement because each expression will naturally result in a value. This may become a bit clearer when we do something familiar like write an `if` expression. Here's a function called `aboveTen` that takes a number, `x`, and returns `True` if it's above 10:

```
aboveTen x =
    if x > 10 then
       True
    else
       False
```

Also note that this function can only result in one *type* of thing. In this case it results in a `Bool`, which can be either `True` or `False`. We couldn't have one branch that returns a `String`, while another branch returns `True`. In order to capture more complex results, we'd have to define a new type, which we'll get to shortly.

Because Elm is based on expressions, we don't have the list of instructions that is the hallmark of imperative languages. If we need some workspace to make a more complicated function, Elm has a `let...in` construct, which you may find familiar:

```
absolutelyMoreMath x =
    let
        y = 3
        z = 30
    in
        (x + y) * z
```

Function application in Elm looks very different than in JavaScript or Python. Here's how we can call the function `aboveTen` with the argument `5`:

```
aboveTen 5
```

Parentheses are not used for calling functions, but are instead used to group things together. Here's the same function using parentheses to group the first expression:

```
aboveTen (20 / 2)
```

This becomes a bit clearer when we have a function that takes multiple arguments. We don't need commas to delineate our arguments. Calling a function with two arguments just requires them to have whitespace between them. Here's a basic function, `mathPlease`, being called with two arguments:

```
mathPlease 5 7
```

Again, for more involved expressions, we can use parentheses to group our arguments. Here we call that same function, but this time we use parentheses to capture the expression for the first argument:

```
mathPlease (20 * 2) 5
```

Piping Functions

A common pattern in Elm is to create pipelines with the pipe operator, `|>`, which allows you to chain functions together. This feature is borrowed from F# and allows large, multistep data operations to be presented in an intuitive and readable manner. Here's a basic example:

```
import String

formatString myString =
    myString
        |> String.reverse
        |> String.append "Hello "
        |> String.reverse
```

This is equivalent to the following JavaScript:

```
function formatString(myString){
      var reversed = myString.reverse();
      var with_hello = "Hello " + reversed;
      return with_hello.reverse();
}
```

Elm pipelines are analogous to method chaining in JavaScript, where you can call methods from an object in a chain, as long as the function returns a copy of the object. You might have encountered method chaining in jQuery, or in the Underscore library. Here's an example from jQuery:

```
$('#my-div')
  .height(100)
  .css('background', 'blue')
  .fadeIn(200);
```

Enabling method chaining in JavaScript requires some forethought and setup, as every chainable method needs to return a reference to the object instead of some other result. In contrast, pipelines in Elm can be used to chain a series of *any* functions together as long as the result of one function matches what's expected by the next one in line. This means pipelines are used consistently as a pattern of the Elm language instead of only in places where library creators have worked to enable them.

Writing Types

Elm lets you define new types. Coming from JavaScript or Python, that might sound a bit foreign, but writing your own types is a central part of Elm code. Types are how you describe what values something can have. Once the compiler knows what types of values it can expect, it can check your work.

Record Types

In Elm we have records, which are just sets of key/value pairs that feel a lot like JavaScript or Python objects. Using a record type, we can describe exactly what keys a record has and specify the type of value for each field. Here's what it looks like:

```
type alias Person =
  { name : String
  , age : Int
  }
```

Now, whenever we refer to something as a `Person`, we know that it is a record with two fields (`name` and `age`) where the values are a `String` and an `Int`, respectively. Now that we've described a type, we can annotate our functions with a *type signature* that describes the arguments that the function takes and what the function results in. This type signature is entirely optional (you might have noticed that we didn't write one for our previous functions), but such signatures are useful for keeping track of your code!

Here's a function that takes a `Person` and an `Int` representing the number of cats the `Person` has, and returns a `String`. The first line of code is the type signature:

```
greet : Person -> Int -> String
greet person numberOfCats =
    "Hello " ++ person.name ++
    ", you have " ++ toString numberOfCats ++ " cats."
```

This type signature can be read as "`greet` is a function that takes a `Person` and an `Int` and results in a `String`." Arrows separate each argument, with the last type mentioned being the result of the function.

Notice that we don't have to do any defensive checking in this function. We don't have to verify that `person.name` is not `null`, or even verify that `person` has a field called `name`; this is all checked by the compiler. Because we don't need this standard boilerplate, our functions tend to be more concise and meaningful compared to JavaScript.

Even though type signatures are not mandatory, they are a powerful tool. You can think of them as enforced documentation that never gets out of sync with your code. By looking at type signatures, you can know whether two functions can be chained together, because one accepts the output of another, and you can get a better understanding of the overall organization of the code. When writing a library, experienced Elm programmers sometimes start by sketching it out *just as types and type signatures* to get a high-level view of what the library API and organization might look like without having to write any actual code. In a way, by sketching out the types and type signatures first, you're creating a *specification* for your code that will be enforced. This is a powerful design technique that is either impossible or not quite as effective in most of the JavaScript world.

That being said, not only are the type signatures optional, but the compiler can actually write them for you. Compiling a project with the `--warn` flag will cause the compiler to tell you what any missing type signatures should be.

Union Types

Another handy feature in Elm is *union types*. There are many names and variations for this concept—tagged unions, algebraic data types (ADTs), or some flavors of enum—but essentially union types let you declare a new type and specify exactly what values it can have. For example:

```
type HokeyPokey = LeftFootIn | LeftFootOut | ShakeItAllAbout
```

Here we have a type called `HokeyPokey` that *can only have one of three values*: `LeftFootIn`, `LeftFootOut`, or `ShakeItAllAbout`. There's no secret `null`. There are no accidental misspellings either, as the compiler will let you know if something like that pops up. We know exactly what values a `HokeyPokey` can have. This feature of union types means they work *really* well with `case` expressions. These two constructs are common in Elm because they are so clear and powerful.

Here's an example of a function that takes a `HokeyPokey` and uses a `case` expression to cover each situation:

```
dance : HokeyPokey -> String
dance hokey =
    case hokey of
        LeftFootIn ->
            "OK, got it, left foot in."

        LeftFootOut ->
            "Wait, I thought I just put my left foot in?"
        ShakeItAllAbout ->
            "What are we trying to accomplish here?"
```

This approach is much better than the JavaScript `switch` statement because both we and the compiler know exactly what values need to be handled. This means the compiler can enforce exhaustiveness checking, so if you forget a possible value in your `case` expression, the compiler will give you an error indicating that you're not covering all your cases. For example, this code:

```
dance : HokeyPokey -> String
dance hokey =
    case hokey of
        LeftFootIn ->
            "OK, got it, left foot in"

        LeftFootOut ->
            "Wait, I thought I just put my left foot in?"
```

results in the following error:

```
  This 'case' does not have branches for all possibilities.

  37|>  case hokey of
  38|>      LeftFootIn ->
  39|>          "OK, got it, left foot in"
  40|>
  41|>      LeftFootOut ->
  42|>          "Wait, I thought I just put my left foot in?"

You need to account for the following values:

    Main.ShakeItAllAbout

Add a branch to cover this pattern!

If you are seeing this error for the first time, check out
these hints:
<https://github.com/elm-lang/Elm-compiler/\
blob/0.18.0/hints/missing-patterns.md>
The recommendations about wildcard patterns and "Debug.crash"
are important!

  Detected errors in 1 module.
```

Union types don't have to be single values; they can also "contain" other values. We're then able to unpack these values in our `case` statement.

For example, here's a union type that can be either `Hello` with a `String` attached to it or the value `DontSayHiToThemTheyreWeird`:

```
type Greeting
    = Hello String
    | DontSayHiToThemTheyreWeird


greet : Greeting -> String
greet action =
    case action of
        Hello name ->
            "Hello " ++ name ++ "!"

        DontSayHiToThemTheyreWeird ->
            "Uhh, nevermind."
```

This is actually how Elm handles the idea of `null`. There is no `null` in Elm as it exists in JavaScript. Instead, it is replaced by the `Maybe` type, which can either be your value, or `Nothing`. For instance:

```
tellMeIfIHaveANumber : Maybe Int -> String
tellMeIfIHaveANumber maybeNumber =
    case maybeNumber of
        Nothing ->
            "Nope, no number"

        Just number ->
            "Yup!  The number is " ++ toString number
```

This is a powerful idea: we can capture the entirety of the *concept* of `null` without having it periodically crash our application because we forgot to check for it for the millionth time. Furthermore, we use this construct intentionally in our model *only* when we need the concept of nullability. This gives us a greater sense of the *shape* of our data model and a better intuition about how our application works.

That does it for the brief intro to Elm's syntax! You should be good to go for reading basic Elm. Let's move on to discussing some of the advantages that you can expect from Elm's type system. In the next chapter, we'll discuss some of the high-level concepts that the Elm language is based on and what practical benefit they bring you.

CHAPTER 3

Why Types?

Now that you know a bit about reading Elm, let's learn more about types. If you're coming from a background in JavaScript or Python, talking directly about types might feel a little foreign, but types in Elm provide developers with specific benefits:

- **No runtime exceptions in practice**.
- **Specific, developer-friendly error messages at compile time.**
- **Easy refactoring.** Elm's types ensure that you won't break anything when you need to make a big change. This means it's easy to keep codebases cruft-free and well designed.
- **Enforced semantic versioning of Elm packages.** There should be no reason that a patch change can break an API. Elm automatically enforces the version number of a package, so you know this is *always* true.
- **Extremely reusable code.** One of the ultimate goals of software is code reuse, but too often it's something that's hard to achieve. Because of Elm's types, Elm functions are *inherently* easy to reuse, much more so than functions in JavaScript, Python, or TypeScript. This, and the fact that Elm's functions are so easy to test, is directly due to Elm's use of *immutable data*, which we'll cover in a little bit.
- **Language-wide optimization of external effects.** You want to send an HTTP request? You're rendering HTML into the DOM? In either situation, Elm will make sure all external actions are

done efficiently, with minimal effort needed from a developer to optimize them.

We'll cover these benefits in more detail in the following sections.

Beautiful Error Messages and Refactoring

Let's start by taking a look at some of Elm's high-quality error messages. These are available because the compiler has enough type information to explain to you what went wrong. We'll start with something basic by trying the classic JavaScript mishap of adding a number and a string. Here's an Elm function called `classicMishap` that tries to do that:

```
classicMishap = 5 + "my string"
```

The Elm compiler will give the following error for this code:

```
The right side of (+) is causing a type mismatch.

3|   5 + "my string"
         ^^^^^^^^^^^
(+) is expecting the right side to be a:

    number

But the right side is:

    String

Hint: To append strings in Elm, you need to use the (++)
operator, not (+).
<http://package.elm-lang.org/packages/elm-lang/core/\
latest/Basics#++>

Hint: With operators like (+) I always check the left side
first. If it seems fine, I assume it is correct and check
the right side. So the problem may be in how the left and
right arguments interact.
```

First, in instances (online), it's nice that we have color, documentation, and hints! Beyond that, though, you might be thinking that this seems pretty basic. There are errors analogous to this in Python and JavaScript, so it might not seem impressive. But there is one big difference between this error and the ones you get in Python and JavaScript: this one was caught by the compiler instead of at runtime. This is important for a few reasons, but the one I want to talk about is that we didn't have to execute our program with just the right data and the right series of operations to get this error message to trigger. In Python and JavaScript, it can be tough to get all errors for every function or method to trigger. I know I've often worried that I didn't cover all my bases in my Python and JavaScript code and that a rarely invoked, dusty corner of my codebase was going to break my app. In Elm, the compiler checks your work no matter where it is. What a relief!

That being said, the preceding error also demonstrates another important aspect of Elm's types: there is no implicit type conversion in Elm. Said another way, Elm doesn't try to guess what you're trying to do.

Elm's type system allows us to go farther than just, "I was expecting this type and got that one." Let's look at another common error, the dreaded typo:

```
charlie = { name = "Charles", age = 27 }

greetCharles = "Hello " ++ charlie.nmae
  "Hello " ++ charlie.nmae
```

This code will produce the following compiler error:

```
-- TYPE MISMATCH-------------------------------------------

`charlie` does not have a field named `nmae`.

5| greetCharles = "Hello " ++ charlie.nmae
                              ^^^^^^^^^^^^
The type of `charlie` is:

    { age : number, name : String }

Which does not contain a field named `nmae`.

Hint: The record fields do not match up. Maybe you made one of
these typos?

    name <-> nmae
```

Elm can make spelling suggestions like this because it knows what fields the record *should* have. Detailed error messages lead to a productive developer experience, allowing you to focus on more valuable pursuits than tracking down subtle typos.

The power of static typing isn't just about small types like `String` and `Int`. One of the more profound benefits of Elm's types is having certainty around your *large* data structures. It's one thing to say, "I know this value is always a `String`" or, "This value will always be an `Int`." Those sorts of statements are useful but may leave you thinking, "Well, that would be nice, but I've been getting by just fine without that sort of checking." The value of static typing is entirely more profound when you're working with a complex record with many fields that contain `Strings`, `Ints`, lists of other records, and every variation of your data; and you can say, "I know exactly what forms my data can take and I know my code handles every single one of them."

This leads us to the topic of refactoring in Elm. If you make a change to a data structure or function, the compiler will tell you what code is affected. This means you avoid situations where a tiny change leads to code breaking in a faraway land. Adding code won't break existing code (as we'll discuss in a moment, this is because Elm is based on immutable data). You can delete code with confidence because you know the compiler will tell you if that code is being relied on somewhere else. This means you can aggressively clean up old code and not let your codebase fall stagnant due to fear of breakage.

From a high-level point of view, the process of making large, systemic changes to Elm code is surprisingly simple. You make the change you want, follow and address the compiler errors, and then, once your project compiles, it's *likely* that you're done. For those cases where you aren't, an easily writable test suite will likely to bring you the rest of the way.

Performance and Reusable Code via Data Immutability

One of the key features related to Elm's type system is data immutability. Immutable data means that once a value has been created, it can't be modified.

Elm is based entirely on immutable data, and this has some profound benefits that aren't obvious at first glance. In fact, your first thought might be, "How could a language based on immutable data ever work? You must be nuts!" It turns out to be super useful though, so let's talk about what data immutability buys you, starting with performance.

Data immutability allows for a number of opportunities for optimization that don't exist otherwise. It's one of the reasons *Elm has such fast HTML rendering*! Because all data is immutable in Elm, we can compare deeply nested data structures via reference instead of having to compare each and every value manually. This can drastically improve performance and is especially relevant in Elm's virtual DOM implementation, which has to perform a comparison to know what, if any, part of the existing DOM needs to be updated.

NOTE If you're unfamiliar with the DOM, it's just a browser's internal representation of your HTML. We're simply talking about Elm making changes to what's displayed on the page.

Data immutability is implemented in Elm through the use of a persistent data structure. When updates are made to your data model, the new data is attached at a specific place in the data structure, marking it as the most recent version of the data but also preserving previous states. This persistent model is highly efficient because even though consecutive versions of your model are being stored, common data is being *shared* between these versions. Thus, when a piece of your model doesn't change, no additional operations are performed on it.

Maintaining an efficient history of your data allows for some powerful debugging opportunities. Elm's debugger allows you to navigate and replay the history of states your app has been through; we'll cover that in a later section.

Beyond performance, immutability serves as a powerful means for maintaining a separation of concerns in your code. Again, this probably isn't immediately obvious, but it means that Elm can guarantee that functions don't interfere with each other's internal details, which is the key to why Elm code is so reusable and easy to write tests for.

Specifically, by basing a language on immutable data, we're saying that a function *does not have permission* to change the internal details of another function. To put this another way, in Elm we have the guarantee that *a function will always return the same result if you* *give it the same arguments*. When you have a global guarantee that all functions *only* operate on the arguments they are given, it means that you can test each function in isolation.

This building-block characteristic of Elm functions will also help you know when you *can* reuse a function. You no longer have to worry about the entire state of your program when trying to use a function; you only have to pay attention to the arguments going in and the data coming out. This drastically simplifies thinking about your code and also leads to a natural way of organizing code. Because we know functions only operate on their arguments, we can organize functions by grouping the ones that operate on the same *type* of data.

In order to have the same-arguments/same-result property of functions that makes Elm code so reusable and easy to test, we can't allow just *any* function to "talk to the outside" by sending an HTTP request, modifying the DOM, or opening a web socket. If we did, then a function could be basing its result on something other than the arguments!

Of course, "talking to the outside" is ultimately what we care about in software. Code that performs these kinds of external tasks is generally referred to as code that has *side effects*. Instead of allowing *every* function to have side effects, each effect is managed in exactly one place in Elm, called an *effect manager*. Elm isn't limiting your functionality by doing this; it's enforcing a separation of concerns at the language level. This creates opportunities for language-wide optimization that the entire Elm ecosystem can benefit from.

Immutability in JavaScript

There are a number of libraries (such as Immutable.js and `seamless-immutable`) that can be used to provide immutable data structures for JavaScript. They're definitely worth checking out and can provide some strong gains in performance and modularity. This is another example of how, if you're sufficiently knowledgeable and willing to put in the additional work, you can bring some of the benefits of Elm to your JavaScript projects, though you'll have to remain vigilant that all new code is written with these techniques. You'll also have no guarantee that the packages you're using are similarly knowledgeable and vigilant.

In Elm you benefit from data immutability without any additional work or advanced knowledge. This means beginners often write performant, modular code without needing to know that these benefits are coming from the idea of immutability. In fact, you can be sure that all Elm code is going to receive these benefits, whether it's a third-party package or code written by your top developer, because these features are integrated directly into the language.

Language-wide Optimization of External Effects

In Elm we do things like perform an HTTP request or open a web socket by describing what we want to happen and letting the Elm runtime actually do the dirty work. What is happening is that we're separating *describing what we want done* from actually *performing that action*. Elm's types are how Elm enforces that separation. By keeping these two things separate we open up opportunities for consistent optimization, and we have a strong assurance that the effect we want is executed correctly every time. This is the essence of Elm's managed effects.

This isn't that foreign of an idea. A virtual DOM (like the one used in Elm or even the one implemented by React) is similar. The essence of a virtual DOM is that we describe what the HTML should be at any given point and the virtual DOM is in charge of efficiently batching updates to the page being displayed. We're putting all the code for giving updates to the DOM in one place and saying, "If you want to update the DOM, talk to the code that's managing that." This solves the problem of separate parts of a large codebase each trying to update the DOM independently and giving heart attacks to developers in the process. This approach not only made for performance gains because DOM updates could be batched, but also laid the foundation for handling larger codebases.

Elm uses this idea of separating *describing* what we want done and *executing* it as the standard pattern for handling all effects, generally to great benefit. Because each effect has a single codebase that's in charge of managing that effect (generally referred to as an *effect manager*—who could have guessed?), we can optimize that one piece of code and have the entire Elm ecosystem enjoy the benefits.

What do these optimizations look like? The specifics can vary, but generally involve batching work. For example, if multiple GraphQL queries are made to a single resource, Elm's GraphQL effect manager can batch them into one query. The web sockets effect manager takes care of keeping a web socket open. If a connection goes down in the middle of sending data, the data is queued and will be sent as soon as the connection comes back online. Since this behavior is handled automatically by the effect manager, Elm code dealing with web sockets tends to focus specifically on what data to send and what to do with the data that's received. That tends to be the important part, right?

Should we care if an HTTP request is synchronous or asynchronous? In Elm, we don't need to; we just describe what we want (such as a `GET` request to a URL that should return JSON with a certain structure), and the `Http` package takes care of executing it for us. This is great, though you might be wondering what happens when an effect fails, such as when an HTTP response is something other than a 2XX code. Wouldn't this generate a runtime error?

Operations that can fail (which are generally only associated with effect manager code) will pass back any error messages to you as data so that you can gracefully handle the error in the context of your app. In fact, for every effect that can fail, you're required by the language to account for what the app should do if that effect fails, even if it's to do nothing. Coming from a language where anything can fail in any number of confounding ways, this may sound like a lot of work, but because Elm has isolated the tasks that can fail into one place, the amount of code we need to write to ensure that every possible outcome is covered is minimal.

To be a little more concrete, in Elm there's no special syntax to handle an error. Specifically when dealing with an effect that is failable, we get back a `Result` type, which can be either the value `Ok` with the data that we requested or the value `Err` with a string describing what went wrong. We can use a `case` expression to account for each situation:

```
handleResponse response =
    case response of
        Ok yayData ->
            -- Now we handle our new data.

        Err errorString ->
```

```
            -- We had an error instead.
            -- Better handle it gracefully.
            -- Pirouettes?
```

Where Are JavaScript Promises?

If you're working with JavaScript, it's likely you've encountered JavaScript promises as a way to structure multistep asynchronous operations.

In Elm, we already have a separation between describing what we want done and execution. So it probably isn't that surprising that describing a *series* of things we want done, executing them, and bailing out early if one of them fails isn't that difficult. In fact, this is the job of the `Task` module.

The advantage of Elm tasks over JavaScript promises is based on Elm's type checking. Long chains of asynchronous operations that depend on one another can be tough to code and test, so it's a relief to have Elm's assurances that every type of data is being handled correctly, that we've covered every case that can fail, and that our (possibly long-running) series of operations isn't going to create a delayed runtime error. We know that all our external errors are accounted for gracefully.

Effect managers in Elm not only allow us to keep the guarantees that make our code modular and easily testable, but also to interact with the real world and make cool stuff. Even beyond this, they allow for consistent optimization across the language and for handling the boilerplate details necessary for an effect so that the code in our app is more focused on the meaningful parts.

Elm Types Versus TypeScript

TypeScript is Microsoft's typed superset of JavaScript that compiles to standard Javascript. It has types, so how is this different from Elm?

TypeScript is based on the idea of adding incremental value to your JavaScript by adding the option of type checking. This means you can get a guarantee that a function can only be called with the right arguments. This is definitely an improvement over standard JavaScript, but we still miss out on some of the strongest benefits that we get in Elm.

Elm's errors are much cleaner and more informative than TypeScript's. Here's an example taken from the TypeScript website:

```
function greeter(person: string) {
    return "Hello, " + person;
}

var user = [0, 1, 2];

document.body.innerHTML = greeter(user);
```

which results in the following compiler error:

```
greeter.ts(7,26): Supplied parameters do not match any signature
of call target
```

An analogous error message in Elm would be:

```
-- TYPE MISMATCH ------------------------------ src/Code.Elm

The argument to function `greeter` is causing a mismatch.

86|     greeter [ 0, 1, 2 ]
                ^^^^^^^^^^^
Function `greeter` is expecting the argument to be:

    String

But it is:

    List number

Detected errors in 1 module.
```

The differences between these error messages may seem unimportant, but take a look at how much more *specific* the Elm error message is. This specificity is important when we progress beyond tiny example functions. If the argument required by a function is something like a complex object, TypeScript will still just report that something is wrong. Elm, by contrast, will tell you specifically *what* is wrong and sometimes give you hints about how to fix it.

In addition to the differences in error messages, we can't have the same level of confidence in TypeScript's types as we can in Elm's. TypeScript's `any` type lets you opt out of type checking. This might appear to be a good idea because it seems like less work, but this is a shortcut that lets in all sorts of subtle errors that Elm will always protect you against. Even beyond the `any` type in TypeScript, common JavaScript pitfalls also still exist: two prime examples are the fact that TypeScript allows any value to also be `null` and that TypeScript doesn't provide an error for `case` statements that don't cover all situations. Both of these things can be caught by the TypeScript compiler if you enable the right options, which is fine on a certain level—except now we've lost the "no runtime errors" guarantee of Elm and replaced it with, "If you have some compiler options turned on and do all the necessary work, some code will be better, but who knows about that library you're using. It was probably created by good developers, right?"

In Elm we enjoy a strong baseline of confidence in other people's code. Protecting so many of the error-prone areas with types and the compiler means that *all* Elm code enjoys effectively no runtime exceptions, enforced semantic versioning practices, and ease of maintenance—not just code written by developers who have the right knowledge, and enough time and awareness to *always* do things correctly. (The best developers I know don't even trust themselves to always do things correctly!)

Another difference is that TypeScript, like JavaScript, is based on mutable data, while Elm is based completely on immutable data. Because of this, all the benefits of data immutability that we just discussed, including performance optimization, strong modularity, and enforced separation of concerns, are much harder to attain in TypeScript. While you can implement immutable techniques in TypeScript, it's not a language that's really meant to work well with immutability. The property of same-arguments/same-result can never be assured in TypeScript, making tests harder to write and reuse harder to achieve.

TypeScript can improve your JavaScript. You'll still have a number of JavaScript "gotchas" to deal with, but you will have some tools to lessen their impact, even if they are behind a compiler option. Elm's mentality, however, is that best practices should be turned on by default or even be mandatory. In short, while TypeScript can bring incremental benefits to your JavaScript, Elm doesn't have a large number of the issues in the first place. Elm goes beyond using a type system to check that some of your arguments are correct, and offers high-level benefits like no runtime errors in practice, enforced semantic versioning of packages, and fantastic error messages.

Now it's time to take a tour of the Elm architecture.

CHAPTER 4

The Elm Architecture

Elm programs have a standard architecture which consists of a data model, a `view` function that renders that model into HTML, and an `update` function that handles all updates to the model. You might find this familiar, as it's a variation of the Model-View-Controller pattern.

The first step in writing an Elm app is to register these components with the Elm runtime. A basic Elm application looks like this:

```
main =
    Html.beginnerProgram
          { view = view
          , model = 0
          , update = update
          }

type Msg
    = Increment
    | Decrement

update : Msg -> Model -> Model
update msg model =
    case msg of
        Increment ->
            model + 1

        Decrement ->
            model - 1

view : Model -> Html Msg
view model = div [ onClick Increment ] [ text (toString model) ]
```

Notice that it doesn't matter in which order everything is declared in our code (in our `main` definition, we reference our `view` function and our `update` function even though they are defined later in the file). This is because our file isn't executed from top to bottom, but is instead a collection of types and functions. Execution is handled separately by the Elm runtime using the functions that we provide it.

All the data used in an Elm application is described in the data model. Commonly this is captured as an Elm record, but any type can be used as the model. In this example, our model is just a `number`. We didn't need to tell the compiler this because it inferred that fact when we used addition and specified the initial value as `0`.

Our `update` function takes a `Msg` and our `Model` and results in an updated `Model`. `Msg` is a normal union type like we discussed in Chapter 2. These `Msgs` are handed to the `update` function when the specified event fires. This means our `update` function generally takes the form of a comprehensive `case` expression. In human terms, this means the `update` function serves as a table of contents for everything that can be done to our model. The `update` function is also the *only* place where changes to the model can be made. This drastically simplifies navigating the code and tracking down where something happens.

Elm's `view` function makes use of a virtual DOM. Similar to in React, in Elm you describe your view as functions, which the Elm runtime will render as HTML. When your model changes, the virtual DOM performs a diff operation to see what needs to be updated, and then makes the necessary changes to the DOM as efficiently as possible.

Upon closer inspection, we can see that our view can also send `Msgs` to our `update` function. In our view we have `onClick Increment`, which should feel intuitive. When we put this in our code, the Elm runtime takes care of calling our `update` function with the value `Increment` when a click event occurs.

Standardizing the language on a strong architecture means that beginners and experts alike have a solid starting point for well-organized code. It also simplifies navigating other developers' Elm code, because you know to look for a model, a view, and an `update` function.

Interop with JavaScript

The main method that JavaScript can use to communicate with Elm is via ports. Data coming in from JavaScript land first needs to be translated into Elm types. For most common types this can be done automatically. For something more nuanced like Elm's union types, where we don't have a direct analog in JSON, you'll have to write a small bit of code called a `JSON decoder`, which creates the type you need from the data that's provided. There are also tools that can automatically generate Elm decoder code directly from JSON . They can't generate code for every situation, but they can get you most of the way there and point you to what still needs attention.

This process is fairly straightforward (and beyond of the scope of this report) and is how most communication with Elm works. When outside data comes in, it's checked at the gate. If the data isn't well formed, then Elm doesn't throw a runtime error (that would be silly), but instead captures an error message describing what went wrong and provides it to you in a manner where you can have your application gracefully deal with it.

Adopting Elm Incrementally

There's no need to adopt Elm for the entirety of your project in one go. You can adopt Elm incrementally by giving it control of a single HTML node of your app. That way you know that this one piece of HTML is completely managed by Elm, and the rest of your page can be run by other technologies as you see fit. This is obviously useful when dealing with a large codebase that's already in production.

Here's some basic HTML that shows how to embed Elm code that has been compiled to *my-elm.js* and attach it to an HTML node:

```
<body>
  <div id="elm"></div>
</body>
<script src="my-elm.js"></script>
<script>
    var node = document.getElementById('elm');
    var app = Elm.Main.embed(node);
</script>
```

Some people have even done work that shows how to write React components in Elm, so more sophisticated incremental integration of Elm is possible.

Elm Versus React

React is a JavaScript library made for the view part of the frontend, which means taking your data model and rendering it as HTML.

Both React and Elm utilize the idea of a virtual DOM, which decouples declaring what HTML you want and actually rendering the changes as HTML. This is done so that all changes to the DOM can be done in one batch and rendered efficiently by the browser. This decoupling is great for another reason: it allows for a much more declarative style of programming. Instead of creating a tangle of functions that can each modify the DOM independently, we can describe what the HTML should look like for a given model and let a function calculate what changes need to be made.

Many of the differences between React and Elm can be traced back to the languages themselves. In Elm, we still have the guarantee of no runtime exceptions in practice. We're also able to catch virtually all trivial—and a large number of meaningful—errors at compile time without needing to run our program in a certain way with a certain state to see if it errors. React does have some type-checking abilities, but this is all done at runtime and requires the code with the bug to be executed before React logs an error. It's also not nearly as comprehensive as Elm's checking.

React is designed to support using immutable data to store state, though it can't make you use it in your entire project. Because of this, there are a number of subtle bugs that can occur in your React code as a result of hidden mutable state. An example of this (as given by the React documentation) is that a component may not render if you mutate certain props. This is behavior that would be hard to mimic in Elm. If there was an issue like this in Elm code, it'd likely show up as a compiler error. The React documentation recommends focusing on maintaining immutable state to avoid these issues. This is in contrast to Elm having data immutability built in, which means you can enjoy the performance benefits without having to worry about subtle pitfalls of switching between mutable and immutable data.

React incorporates some of the same ideas that power Elm, but can only fully implement a portion of them due to the limitations of JavaScript. It's forced to implement the rest as recommendations and incremental (meaning not language-wide) tools. You can get *some* of the reliability of Elm with React, but only if you know all the subtle traps to avoid and have the time to ensure that you've avoided them.

Many best practices in Elm are simply *built into the language*. In React (and most projects written in JavaScript), best practices and warning signs are communicated through the documentation. The question is, would you prefer be continuously scouring the documentation for these hidden surprises, warnings, and performance tips, or do you want the lion's share of this work to be taken care of by the compiler?

Elm Versus Vue.js

There's a trend toward libraries being more stripped down and closer to the technologies that they're trying to use (HTML, CSS, and Vanilla JS). This is the general approach of Vue.js, which makes the framework feel incredibly familiar. The idea is that Vue.js allows you to write *simple* JavaScript with the hope that when something does break, there won't be mountains of abstract indirection to throw you off course.

Vue.js doesn't change anything fundamental about JavaScript. It can't provide you with the high-level benefits that are built into Elm. If you're worried about code breaking and being hard to debug, maybe the answer isn't to write *simpler JavaScript* but to use a language that has effectively eliminated runtime errors in the first place and provides amazing, specific compile-time errors that tell you exactly *why something might break.*

It's tempting to think that Vue.js will be easier to learn than Elm. While Elm may seem foreign to established frontend developers, much of the learning curve comes from habits that are ingrained from having done things differently for many years and having to reevaluate our coding intuitions. Let's be real—given a week, I'd be surprised if a competent web developer couldn't become productive in Vue.js *or* Elm. The learning curve for each is generally low, and both have excellent documentation.

Once you get past the initial learning curve, however, Vue.js and Elm aren't equal. In my opinion, beginners who code in Elm end up with maintainable, performant, runtime error–free code from the start because these things are built into the language. The same can't be said for Vue.js code.

Now that we've looked at the standard Elm architecture, let's take a look at the tools that are available to us when we program in Elm.

CHAPTER 5

Elm Tooling

The Elm platform consists of several executables that all help with Elm development. `elm-make` is the compiler, which can compile Elm either to a JavaScript file or to an HTML file with that JavaScript embedded. `elm-repl`, Elm's REPL, is very useful, especially when you're learning Elm.

This chapter provides a rundown of the other major Elm tools.

elm-package

Elm has its own package manager, `elm-package`, which is backed by GitHub.

Elm is able to calculate and enforce semantic versioning for all packages published through `elm-package`, which is fairly extraordinary. A version number that uses semantic versioning takes the form three numbers separated by periods. Each number indicates what *sort* of change has occurred compared to previous versions. Here's what the three numbers mean, from left to right:

Major
: A piece of the existing API has been changed or removed.

Minor
: Something has been added, but nothing existing has changed.

Patch
: Something has been changed which does not affect the API, such as documentation or an internal implementation detail.

It’s worth restating this: *Elm enforces semantic versioning for all Elm packages.* If you see an Elm package move from 2.0.4 to 2.0.5, for example, you are guaranteed that the API has not changed.

This is only possible because of static typing and is a direct result of being able to know the type signatures for every function that is exposed in the API of your package.

This also means that you can ask `elm-package` what your version number will be, and it’ll tell you both what your new version number is and specifically *what changes* are driving this new version number. Here’s an example that was created by deleting a function called `repeat` from a package and then running `elm-package diff`:

```
Comparing mdgriffith/elm-example 1.0.0 to local changes...
This is a MAJOR change.

------ Changes to module String - MAJOR ------

    Removed:
        reverse : String -> String
```

It tells us immediately that a function called `reverse` which takes a `String` and results in a `String` has been removed. How cool is that?

This adds confidence in Elm packages both on the user side and on the package creator/maintainer side. From the user side, it makes the process of updating your dependencies straightforward and understandable. From the package creator/maintainer side, knowing when you’ve broken your API is an enormous benefit. It means you’ll never *accidentally* break your API. This is another example of the way in which the guarantees that Elm can provide about your code not only improve the quality of the software you’re writing, but also raise the level of confidence you can have in other developers’ code and packages.

Beyond enforced semantic versioning, packages published through `elm-package` are also *required* to have documentation comments on all functions and types that are exposed as part of the API. HTML documentation is then automatically generated from the source code and published on the Elm packages website.

The Elm Debugger

Recreating bugs in a browser can be challenging. This was recently addressed in Elm's 0.18 release, which focused on creating a debugger that could provide insight into what is going on in your application and as well as including tools that let you replay and examine bugs.

In an Elm program, the model is updated every time the `update` function receives a `Msg`. When we turn on the debugger with the debug flag (`--debug`), we get an interactive menu that shows us each data update as it's performed. We can then visually navigate what's going on and *rewind* to a specific state and inspect our code.

This `Msg` history is also exportable/importable, so it can be replayed. This is a powerful tool for cross-browser testing.

There are also advantages to shipping the Elm debugger as a standard language tool. It means that everyone can standardize on a centralized toolset, and it provides clear guidance to beginners on how they can get better insight into their Elm programs.

Compared to the Redux Debugger

Redux is a common library for managing a data model in JavaScript apps. Like Elm, it has a debugger in its developer tools, which allows you to navigate and rewind to a specific point in the history of updates in your model. There are also some additional packages that allow you to export/import your update history from Redux.

The main difference between the Redux developer tools and the Elm debugger is that Elm is able to verify that an exported update history *can* be replayed with the code that is running. Elm accomplishes this using the same mechanism that's used to enforce semantic versioning. Specifically, we know that if the types that are required by an update history have changed, then the history won't be replayable. Because Redux doesn't have access to the type information that Elm has, it can't verify that a set of replay data is compatible with the running code.

This is an important but subtle point. If you aren't sure that a replay can be used with your current code, then you'll likely end up debugging your debugging tool, or debugging bizarre side effects of incompatible code, instead of actually solving the bug you captured.

elm-reactor

`elm-reactor` is a small local server that automatically compiles and serves any *.elm* files in the local directory. This makes development pretty quick and is a good way to get started developing! Because this feature is used purely for development, it has the Elm debug mode enabled by default.

elm-format

`elm-format`, as you might imagine, is an automatic code formatter for Elm, similar to `gofmt` for the Go language. At the time of writing `elm-format` isn't part of the official Elm platform, though it's likely to be included at some point in the future. You can download it from the `elm-format` GitHub page.

`elm-format` will automatically reformat your Elm code into Elm's best practices for code style. Most developers set it to run whenever an Elm file is saved. Using `elm-format` means not having to worry about minor style issues, so you can focus just on coding. This cuts out quite a bit of busy work!

Having code that's always in a consistent style helps development significantly, especially when you have to navigate someone else's code. This effect is compounded the more people use this tool, and fortunately it is used extensively throughout the Elm ecosystem.

It may not be immediately obvious, but having a strong code formatter also improves your development workflow by giving you quick, informal feedback on your syntax. `elm-format` will only reformat valid Elm code, so if you save a file and it *doesn't* correct your slightly off indentation, you can infer something is wrong with your syntax. Many times this nudge is all you need to catch a missing parenthesis or a misplaced letter.

You might be a little wary of using a code formatter if you're not used to working with one, especially if you've worked with a *bad* code formatter in the past. Nobody wants to fight with a tool to get their code to look right. However, this rarely happens with `elm-format`. A lot of thought went into the style decisions that drive `elm-format`, and it's frequently mentioned as one of the most useful tools in the Elm ecosystem.

Similarly, you may be a little surprised, and maybe initially frustrated, to learn that you can't configure the tool, for example, by setting your own indentation level. This is by design and key to what `elm-format` is trying to achieve. If `elm-format` was configurable, then the idea of a globally consistent, best-practices style would be lost.

elm-test

`elm-test` is another tool that isn't part of the official Elm platform but has become a de facto community standard.

Just because Elm is based on static types and has a wonderful compiler doesn't mean we don't have to write tests for our code. It just means that our tests don't have to cover the cases that the compiler covers for us. We know that all functions will be called with the correct type, and that there are no hidden `null`s or functions that are being called incorrectly. We just need to test for values that aren't caught by the type checker.

We have some things going for us in Elm that help with testing. We know that a function that is given the same arguments *will always give the same result*. This means we don't have to set up a huge environment to run a test; we can test any function in isolation and know that it will work the same no matter where it occurs in our code.

What does a test suite in Elm look like? Here's an example showing a simple test for the `String.reverse` function:

```
testReverse =
    test "reverses a known string"
        (\() ->
            "ABCDEFG"
                |> String.reverse
                |> Expect.equal "GFEDCBA"
        )
```

The only thing in this example that we haven't seen before is the `\ ->` syntax, which is how you define an anonymous function in Elm. The main thing here is that `test` is a function that takes a string and a function that returns a test result. We mentioned the pipe operator (`|>`) before, and here we can see it in action. It chains together a series of functions that can be read from top to bottom.

elm-test supports fuzz testing, which is a powerful technique. Essentially, we describe the types of values that our function takes and let elm-test generate a large number of random arguments to this function. By doing this, we increase our testing coverage enormously. This is useful for catching nonobvious corner cases hiding in your code. Here's a simple fuzz tester that tests String.reverse, this time with randomly generated input:

```
stringFuzzTest =
    fuzz string "Test with 100 randomly generated strings!"
        (\randomlyGeneratedString ->
            randomlyGeneratedString
                |> String.reverse
                |> String.reverse
                |> Expect.equal randomlyGeneratedString
        )
```

Tests in Elm are generally easy to write, not only because of the excellent elm-test library, but also because of the guarantees the language provides around functions. Even though we just mentioned it, it bears repeating: the same-arguments/same-result property of Elm makes testing easy and robust, especially in large codebases.

Now that we've covered some of the tools at your disposal when working with Elm, let's look at a few specific Elm packages that are commonly used.

CHAPTER 6

A Tour of the Ecosystem

Elm is still a young language. It doesn't yet offer the vast number of packages that are in the JavaScript NPM ecosystem, though the packages that do exist are generally high quality and have all the same guarantees that the Elm language offers. While the number of available packages is growing fast, it's unlikely that there is an out-of-the-box solution for *absolutely* everything. That being said, you can always interop with existing JavaScript libraries or even write an Elm package yourself.

The primary hubs of the Elm community are the elm-discuss Google Group, the Elm subreddit, and the elm-lang Slack channel, which is generally full of people who are happy to assist newcomers and answer questions. The Elm community has been incredibly positive and helpful in my journey to writing better frontend code; it's one of the reasons I keep coming back to Elm (in addition to what the language can do technically).

Let's take a look at a couple of Elm packages and discuss how they approach their areas of frontend development compared to solutions outside of the Elm ecosystem.

elm-css

CSS presents a number of challenges, including debugging subtle style errors and the unenviable task of maintaining CSS for large projects. Fortunately, we have a tool in Elm that makes handling CSS much less error prone: `elm-css`.

`elm-css` is a CSS preprocessor akin to Less or Sass. The main advantage of describing CSS in Elm is that we bring type checking to CSS, which makes it nearly impossible to write invalid CSS. In practice this means receiving well-written Elm error messages at compile time for our CSS instead of having to track down typos and silently invalid properties (were you aware that the RGB color channels can't be floats, and will fail to render if they are?). Properties can't be written incorrectly because the compiler won't allow it. You can't provide an incorrect value.

Of course, we can't catch absolutely everything with our type system. For values that can't be type-checked (such as validating that a hexcode for color is valid), `elm-css` will log a build-time validation error.

Here's an example of what a stylesheet in `elm-css` looks like. You have the option of rendering this as an actual stylesheet or rendering a specific style inline in your `view` function:

```
module MyCss exposing (..)

import Css exposing (..)
import Css.Elements exposing (body, li)
import Css.Namespace exposing (namespace)


type CssClasses
    = NavBar


type CssIds
    = Page


css =
    (stylesheet << namespace "my-css")
    [ body
        [ overflowX auto
```

```
            , minWidth (px 1280)
            ]
        , class Page
            [ backgroundColor (rgb 200 128 64)
            , color (hex "CCFFFF")
            , width (pct 100)
            , height (pct 100)
            , boxSizing borderBox
            , padding (px 8)
            , margin zero
            ]
        , id NavBar
            [ margin zero
            , padding zero
            , children
                [ li
                    [ (display inlineBlock) |> important
                    , color primaryAccentColor
                    ]
                ]
            ]
        ]


primaryAccentColor =
    hex "ccffaa"
```

`elm-css` represents CSS classes and IDs as union types instead of strings, which means the compiler will let you know if you misspell one and even make suggestions about what you might have meant. If you try to use a class or ID that hasn't been written yet, the compiler will also complain. This feature of `elm-css` is a powerful tool for stylesheet maintenance because it means that if you *delete a class* and its style definition from the stylesheet, the compiler will give you an error if that class is still in use.

Taking inspiration from CSS modules, `elm-css` has *namespacing*, which allows you to scope your styles as you see fit. Similarly, nested media queries akin to those in Sass and Less are also available, as is support for mixins.

All in all, `elm-css` is about bringing the robustness that Elm enjoys to modern CSS. Because `elm-css` covers the most common, trivial mistakes and protects against some of the subtler aspects of CSS with compile-time and build-time errors, we can focus on more important issues such as design and user experience.

elm-style-animation

Depending on the design goals of your app, clean animation can be a crucial component or necessary polish to keep things feeling modern. It's easy enough to use CSS animations and transitions in Elm, but they have several limitations that make them unsuitable for more complex interactions. First, they can't be interrupted smoothly. You also can't attach a callback to be called when an animation hits a certain keyframe or finishes. And you have no way of utilizing springs to model your movement. (In case you're new to animation, springs can be used instead of easings to make it much easier to create natural-looking animations.)

`elm-style-animation` is an Elm library that lets us take animation in Elm beyond what we can accomplish with CSS animations. It does this by handling the animation math manually and rendering it as inline CSS. Here's what a reasonably complex animation looks like in Elm:

```
myAnimation =
  Animation.interrupt
      [ to [ opacity 1
           , left (px 200)
           ]
      , send DoneFadingIn
      , loop
            [ to [ rotate (degrees 360) ]
            -- Reset to 0 degrees (happens instantaneously).
            , set [ rotate (degrees 0) ]
            ]
      ]
```

This example starts with `Animation.interrupt`, which means that this animation will interrupt any ongoing animation if necessary. This is done smoothly, maintaining momentum values behind the scenes so that each property changes direction and velocity in a nice physics-based movement. Then the code sends a `DoneFadingIn` message to the main `update` function once the first step has been completed. The animation then begins to rotate forever, or at least until another animation interrupts it.

Here's what the same animation would look like in Velocity.js:

```
// We start with an element selected from the DOM.
$element
    .velocity({ opacity: [1, "spring"]
              , left: ["200px", "spring"]
              },{
                complete: function(elements)
                { console.log(elements); }
              })
    .velocity({ rotateX: "+=360deg" },
              { loop:true, easing:"linear" });
```

If you squint you can see the similarities between these two code examples. There's an obvious sequence of events. Both examples have the capability to notify other code; the Elm code sends a message while the JavaScript code calls a callback function. Implementation-wise, they're both synced to the browser animation frame, ensuring as close to 60 frames per second as possible. So, what are the differences?

Velocity.js was originally written as a performant, drop-in replacement for jQuery's `.animate()`, and it benefits from jQuery's ability to get something working quickly. However, both Velocity.js and jQuery predate the recent explosion of frontend frameworks, and it's not entirely clear how Velocity.js would integrate with any given JavaScript framework, especially one with a virtual DOM. In some cases there are specific modules that bring Velocity.js to a framework, like `velocity-react`, which encapsulates the Velocity API as a React component. In fact, most JavaScript frameworks have their own custom way of describing and implementing animations. There are many options, each with its own intricacies and pitfalls.

In contrast, our Elm animation can be used in *any* Elm application without any special modifications. That will likely continue to be true as Elm evolves, and this reusability is directly because of Elm's types.

This is how software *should* be. Every animation library for the browser that's not based on CSS animations does two things: allows you to describe an animation and renders that animation as inline CSS properties. How many truly different ways of doing this do we need, and why should we have to continuously adapt this concept to new frameworks? In an ideal world there would be one library that could be used anywhere. Elm's types enable strongly reusable code, which makes that much more of a possibility than in JavaScript.

Of course, an Elm animation also has all the guarantees of the Elm language. The compiler won't let us write an invalid animation. If we run into an issue, the compiler will gently point us in the right direction. This trickles down to each component of the animation: it's impossible to write a length unit (such as pixels) when an angle unit is required, or to provide a bare number when a number with a unit is needed. The type system guarantees that our `DoneFadingIn` message is a message that our `update` function knows about and has covered.

Hopefully learning about these two packages gives you some sense of the kinds of goodies the Elm community has to offer. The number of Elm packages is growing by the day, covering many areas of development. Who knows, maybe you'll get inspired and contribute one yourself!

CHAPTER 7

So, Why Elm?

You may think I sound like a broken record, but here are the explicit benefits Elm can provide you:

- No runtime exceptions in practice
- Beautiful compile-time errors
- Enforced semantic versioning for Elm packages
- A refactoring experience that makes you feel invincible because it's so easy and robust
- Language-wide optimization of external effects, which results in strong performance and best practices built into the language
- A litany of well-designed devoloper tools

Elm provides these benefits as a part of the language and the ecosystem. Most of them are built into every Elm project, so you don't need to turn them on or track down some specific technique in the documentation. You can get some, but not all, of these benefits in a JavaScript project if you know the right concepts and are willing to work hard. But Elm takes care of these things for you. Because these features are built into the language, *the entire Elm ecosystem* benefits from them. This means you can have high confidence in third-party Elm code.

The learning curve for Elm is short even though parts of the language may feel foreign. Beginners and experienced web developers alike should have no trouble getting started. This is especially the case if you engage with the community on the Elm Slack channel or the Elm subreddit.

If you want to give Elm a test drive, you can adopt it incrementally alongside existing technologies. You don't need to make a large commitment to try it out in a real-life scenario. Give it a go to see what the experience is like. Most companies that adopt Elm try it with a low-stakes example first.

The ecosystem and community are growing. The first Elm conference happened in 2016 in St. Louis, with a second, separate one coming up in Paris in June 2017. High-quality Elm packages are being published with surprising frequency, with more projects on the horizon.

Elm is being used by companies to solve real problems. As of this writing, NoRedInk has 95k lines of Elm code in production and still hasn't encountered a single runtime exception. That's a bit mind-boggling. Other companies, such as Pivotal Tracker, Futurice, and Gizra, all tell similar stories of no runtime exceptions, a simple learning curve, and improved developer productivity.

Elm could be your super power. By freeing you from having to deal with many of the frontend issues that waste time and money, Elm lets you focus on the important and inherently more valuable problems of user experience, design, and business logic. Elm made me fall in love with frontend development again after many frustrating experiences. I highly recommend giving it a try.

About the Author

Matthew Griffith is a developer-in-residence at Cornell Tech with a passion for clean code that doesn't break. He's worked in cheminformatics and web development.