Linköping University | Department of Computer science
Master thesis | Computer science
Spring term 2016-07-01 | LIU-IDA/LITH-EX-A—16/042--SE

Elmulating JavaScript

Nils Eriksson & Christofer Ärleryd

Tutor, Anders Fröberg
Examinator, Erik Berglund

Linköpings universitet
SE–581 83 Linköping
+46 13 28 10 00 , www.liu.se

Abstract

Functional programming has long been used in academia, but has historically seen little light in industry, where imperative programming languages have been dominating. This is quickly changing in web development, where the functional paradigm is increasingly being adopted.

While programming languages on other platforms than the web are constantly competing, in a sort of survival of the fittest environment, on the web the platform is determined by the browsers which today only support JavaScript. JavaScript which was made in 10 days is not well suited for building large applications. A popular approach to cope with this is to write the application in another language and then compile the code to JavaScript. Today this is possible to do in a number of established languages such as Java, Clojure, Ruby etc. but also a number of special purpose language has been created. These are languages that are made for building front-end web applications. One such language is Elm which embraces the principles of functional programming. In many real life situation Elm might not be possible to use, so in this report we are going to look at how to bring the benefits seen in Elm to JavaScript.

Acknowledgments

We would like to thank Jörgen Svensson for the amazing support thru this journey. You endless knowledge and sharing nature has really gotten us to where we are today. Thank you for the long hours explaining concepts foreign to our understanding, and taking the time to review our work at every step. We wish you all the best and good luck with your endeavours up there in Skelleteå. And may we be the first to congratulate you on your 55 birthday!

Nils & Christofer

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

Today's web applications are getting more and more interactive and browsers that once just showed static text now display rich user experiences that are getting closer and closer to native desktop applications. Web applications today are built around being dynamic and respond to changes in the underlying data. As these applications grow larger, it's getting harder and harder to reason about how these applications work which cause bugs to creep up from everywhere.

The interest in trying to solve these problems has steadily increased with new frameworks and libraries are released literally every week. It's very commonplace today to talk about JavaScript fatigue, where developers don't have the stamina to keep up with all of these new techniques[59]. We want to take a look back at past success and see how one of the earliest programming paradigms, namely functional programming, can help us in the development of JavaScript applications. More specifically, we will compare Elm, a new functional reactive programming language to JavaScript, and determine if it is possible to get the functionality that Elm offers into JavaScript and the reasons for why you would want to do that.

1.2 Aim

Bring the features found in the functional reactive programming language Elm into JavaScript. Find out which frameworks, libraries and tools are needed to achieve this and reflect on how this compares to using Elm.

The point of this is to assess the merits of introducing these concepts to Javascript.

1.3 Research questions

We have one main research question we wish to answer.

- Can we bring the Elm experience to JavaScript?

However, in order to answer this question we first need to answer these two questions.

- What is functional programming with regards to Elm?

- What constitutes the development experience in Elm?

1.4 Delimitations

- We will not be touching on all aspects of functional languages, and will not make an extensive list of what alternatives are available for developing client side web in a functional reactive style.
- We will limit our alternatives to JavaScript client side web development. We think this will allow us to limit our scope while still staying relevant because of the popularity that JavaScript web development has today.
- We will limit the JavaScript comparison to using the framework React. We make this limitation because React fits in our comparison and is very popular. Other reasonable alternatives would have been Angular 2 or Cycle.js.
- We will not cover any questions regarding ethics in this work because we believe it to be irrelevant to the content of this report.

1.5 General terms

Arity

The number of arguments a function takes. A function written as **multiply/2** represents a function with arity 2.

Client

Client references the web browser which shows an application to the user.

Co-effect

A function that depends on some value outside of its inputs, i.e it is dependent on its execution context.[53] Below is a function `addWithMagicNumber( a, b )` which uses a value which is not an input or in its own scope, this means it is using a co-effect.

Listing 1: Example of a co-effect

```
var magicNumber = 6;

function addWithMagicNumber( a, b ){
        return a + b + magicNumber;
}
```

Compilation and transpilation

Both a compiler and a transpiler transforms code from one representation into another. The slight difference in the two is that a transpiler transform code between two representations that are quite close in abstraction level to each other. This is why the process of transforming C into assembly is called compilation and the transformation from CoffeeScript, a slight abstraction for JavaScript, into JavaScript is called transpilation.

Compile-time errors

Compile-time errors are errors that gets found by the compiler before you run your program.

Document Object Model(DOM)

The tree structure of objects that make up a website. A very basic website may consist of a document object at the top, its only child <html> which in turns hold <head> and <body> in which every website consists of.

ECMAScript

A scripting language specification used by JavaScript. ES6 or ES2015 refers to the latest specification of ECMAScript W3C.

Error

An error is not the same thing as a bug. An error is a fundamental problem that causes and exception in you program. A bug can be a logical flaw you have made that leads to unexpected behavior.

Framework

A framework is a package of abstractions that gives you a structured way of developing your app. In contrast to a library which only holds functionality which can be added to your program at will, a framework forces the application to follow a certain structure.

Haskell

A functional programming language. It is highly used in research of programming languages and is thus often mentioned in scientific papers and useful when highlighting concepts of functional programming languages.

JSX

An extension to JavaScript which makes the syntax look similar to XML.[39]

jQuery

A very popular JavaScript library designed to simplify client-side scripting of HTML.[38]

Production

"In Production" means that the system is live towards the end users.

Rendering

Refers to the process of generating visual representation of the website that the users sees and can interact with.

REPL

Read Evaluate Print Loop. It's and interactive environment where the developer can try out things in real time.

Run-time errors

Run-time errors are errors that gets found first when you run your program.

Semantic versioning

Given a version number(ex v1.2.5) **MAJOR.MINOR.PATCH**, increment the:

1. MAJOR version when you make incompatible API changes,
2. MINOR version when you add functionality in a backwards-compatible manner, and
3. PATCH version when you make backwards-compatible bug fixes.

[58]

Stack

The stack is short for tech-stack which are the technologies used to build the application. A couple of examples are the classic LAMP(Linux, Apache, MySQL, PHP) stack or the more modern MEAN(MongoDB, Express, Angular, Node.js) stack.

State

The state of data in an application. When a function of an application makes a lasting change to the data of an application when run, the state becomes more complex because data is now different before and after that function has been run.

TEA

The Elm Architecture see chapter 2.5

World Wide Web Consortium(W3C)

The World Wide Web Consortium(W3C) determines the direction of the ECMAScript specification[66].

2 Theory

2.1 Declarative Programming, *"what"* is that ?

Web applications are becoming more and more advanced for every year that passes, and this leads to the need of managing this increased complexity. This has caused web development technologies to take a more declarative approach to writing code because, among other things, user interfaces become a lot simpler to write when a lot of irrelevant details can be left to the compiler to take care of[11]. Declarative programming is a super-set of functional programming. In its essence, it is a way of constructing programs where the programmer expresses *what* a program should do, rather than explicitly describing the steps involved in *how* it is done.

This is an area that has long been studied and applied in a range of programming paradigms. In 1966, there was a paper *The Next 700 Programming Languages*[40], written by P.J. Landing featuring a discussion about declarative programming. It is clear from the discussion that the concept is something desirable but that attempts at implementing programs in a declarative way have been intolerably inefficient compared to the imperative counterparts. Because of hardware limitations, it was hard to create abstractions out of code without crippling the performance too much. Today, hardware is a lot better which makes it a stronger argument now than ever before to use it in software development.

In this chapter we will present a few different programming paradigms and highlight the benefits of functional programming and the issues with the popular imperative paradigm when developing modern web applications.

2.2 Programming Paradigms

There are a vast amount of problems in programming, and solving them requires the use of the right programming concepts [63]. Because there are so many different problems in programming, thousands of programming languages have been designed to solve specific problems better than others. Illustrating the differences by comparison between languages would take forever, instead it is easier to understand programming by understanding the paradigms that each programming language follow. There are other resources which broadly cover these concepts, we will cover the paradigms we consider most suitable for our end goals.

Imperative Programming

It may be hard to specify which programming language is the most popular, but when referring to the TIOBE index[62] which tries to determine the most popular programming languages, it is clear that imperative programming is a far more popular paradigm compared to functional programming.

The paradigm has its roots in the technological ideas of how a computer is built. Take assembly for example, the most basic language a computer interprets and the language that most compilers target, is imperative. Imperative programming is a necessity for having absolute control over the allocation of memory, which explains why it historically has been the default paradigm of the languages used when faced with limited hardware.[47]

Functional Programming

The functional programming paradigm has its roots in mathematics. In purely functional languages, a function has no side effects(see chapter 2.8 Language features of Elm). If you call a function twice with the same parameters you are guaranteed to get the same result, this property is called referential transparency[34]. This is just how a function in mathematics work and allows for more modular and reusable code because the function is not dependent on the state of the scope it is being used in and can be used anywhere regardless of scope.

Functional programming differs from regular imperative programming in that you describe *"what"* you want done instead of *"how"* the computer should do it[11]. Here is an example of the naive way of summing all the numbers between 1 and a upper limit done with both a functional and an imperative approach.

Listing 2: Functional way to sum from 1 to limit

```
const functionalSum = (limit) => {
  return [...Array(limit+1).keys()]
         .slice(1)
         .reduce((a,b) => a+b, 0);
}
```

Listing 3: Imperative way to sum from 1 to limit

```
const traditionalSum = (limit) => {
  let total = 0;
  for( let i = 1; i<=limit; i++){
    total += i;
  }
  return total;
}
```

Historically, functional programming has mostly been used in academia[52], but has recently gotten a lot of attention in the industry as well. As applications grow in size, they tend to increase in complexity. In order to continue developing efficiently on a large application, it is important to keep the complexity as low as possible. This is where functional programming shines. It allows for better modularity, something that conventional languages place conceptual limits on.[35]

Logic Programming

Logic programming is very different from other programming paradigms. It is based on formal logic and sets up a problem domain containing facts and rules about the problem. Compared to the other paradigms, there are fewer languages to choose from, with the most popular one being Prolog. In Prolog and logic programming in general, problems are solved by unification. Instead of getting a return value from an executed function, Prolog tries to find a solution to the given proposition with the problem domain it has. If there are possible ways to solve the problem, Prolog will return True, and additional information if necessary to explain the solution.

Listing 4: Recursive function in Prolog

```
even([]).
even([X1, X2|Xs]) :-
    even(Xs).
```

The function even will take a list and tell us whether it contains an even amount of elements. If it does, Prolog will return True, otherwise False. All functions in Prolog have a base case which, when true, returns True. When faced with the following question

Listing 5: Query in Prolog

```
even([6,3,2,8]).
```

Prolog will first check if the base case, `even([])`, is true. If it is not, it will try to run the next predicate, remove the two first elements of itself and call *even* again with the remaining list [2,8]. The is repeated until either the base case is true, i.e. our list is [], or we end up with one element in the list and a solution cannot be found. In our case Prolog will return True.

Object-oriented programming

This paradigm is widely used and can be found in virtually all of software today[8]. It is based on the idea to compose the logic of a program into a objects created from classes. These can easily be passed around and the class code can be reused. Two of the main concepts from Object-oriented programming is inheritance and polymorphism. Inheritance is the reusability of logic from a base class to its subclasses. Polymorphism is the way the program decides which object to use given an expression. Here is an example of these concepts used in a small C++ program:

Listing 6: Example of interface in C++

```
class Animal
  {
  virtual void talk() = 0;
  }
class Cat : Animal
  {
  void talk();
  }
class Dog : Animal
  {
  void talk();
  }
```

Animal is the base class of the subclasses `Cat` and `Dog`. These subclasses inherit the behavior `talk()` and polymorphism lets us create a list of Animals with the different objects Cat and Dog because they both share the same base class. In this example `talk()` is not even implemented in the base class, but we could have the case where a function is defined in `Animal` and can then be used in all of its subclasses. This is a common technique for reusing code in object-oriented programming [36].

Functional Reactive Programming

The paper "Functional Reactive Programming"[49] define the paradigm as *"Functional Reactive Programming (FRP) extends a host programming language with a notion of time flow"*.

Functional Reactive Programming was introduced by Paul Hudak and Conal Elliott. In their paper Functional Reactive Animations[13] they aim to extend the language Haskell with two new types of values, *Behaviors* and *Events*. *Behavior* is analog to a function describing for example velocity as a function of time. This is useful in animations. *Events* are, as the name entails, discrete events over time such as keyboard presses.

In Functional Reactive Programming the behavior between different parts of the program is defined at the time of declaration. This can be viewed in contrast to the usual way of manually updating variables in an imperative manner. Popular techniques for handling such a flow of data in Functional Reactive Programming is through the functions `map`, `fold` and `filter`.

Listing 7: JavaScript representation of changes over time

```
setInterval(function({
  currentTime = (new Date()).getHours();
  updateView(currentTime);
}), 1000);
```

Listing 8: Elm representation of changes over time

```
Signal.map timeView (every second)
```

In the traditional approach the developer must keep track of sending the changes to the view to render them. In the FRP example using Elm the *timeView* gets called every second with the current time. Modeling an application this way creates an abstraction of steps that traditionally would have been explicitly declared. This way the complexity of the application can be kept lower. By describing how everything is connected and how they change over time at time of declaration we keep things simpler.[31]

2.3 What constitutes a functional languages

There is no clear definition what exactly draws the line of a language being functional or not. The programming language C is clearly not functional, but Haskell clearly is, and in between are gray areas. We have decided to use the language Elm as a template for what we think constitutes a functional language. Elm is based on Haskell and is specially designed for front-end web development.

2.4 JavaScript

Background

JavaScript is the language of the web and because of this it is the most ubiquitous programming language in history.[21] JavaScript is a high-level, dynamic, untyped interpreted language which treat functions as first class citizens. When Javascript was first released i was used to create simple interactions on web pages. Today whole sites are written in JavaScript. JavaScript allows for dynamic updating of the site instead of having a server loading a whole new page for every interaction. JavaScript was created in ten days and was originally meant to be based on the functional language Scheme but at Netscape where it was created by Brendan Eich, they decided that the language should appeal more to Java programmers.[3] So the made it look more like Java and threw in some object-oriented constructs and choose the language name JavaScript even though Java and JavaScript are very different [10]. Given that JavaScript was made in only ten days it has a lot of quirks. Just compare the length of the books *JavaScript: The Definitive Guide*(1078 pages) [21] and *JavaScript: The Good Parts*(172 pages). [10]

But even with its quirks, it has managed to became the language of the web. Netscape introduced JavaScript in their browser Netscape Navigator 2 and when Java applets failed, other companies started adopting the language as well. The popularity of the language, because of this, is almost completely independent of its qualities as a programming language.[10] It has simply become the standard because it was, and still is today, the one language that browsers can interpret.

Browser vendors are no longer trying to replace JavaScript with something else, instead the development has moved towards using languages that transpile into JavaScript. It is very popular to use transpilers that can interpret ES6/ES7 syntax which is the coming releases of Javascript. One example of the new features that are in the ES6(ECMAScript 2015) specification, are the keywords `let` and `const`. These replace the `var` keyword which did not have a lexical scope. With the introduction of `let` and `const` this is no longer the case. Another feature that gets standardized in that same specification that has long been needed is a native module system for JavaScript. Until now there has been no standardized way of split code into different files.

Today

JavaScript can be found literally everywhere. It can run on an Arduino [4], it can be used to create web servers with for example Node.js [50], and pretty much anything that's possible to program, can be done with JavaScript. There is even a saying, named Atwood's law, stating that *"any application that can written be JavaScript, will be written in JavaScript"* [14]. JavaScript has taken an interesting turn last couple of years and some believe that you could view JavaScript as the assembly language of the web[5]. Write code in your favorite language and compile down to JavaScript.

Competition

Javascript is the only language that runs in all major browsers [33]. There have been attempts of running different languages but non of them have stuck. In the early days of the web Java had a brief chance to be the default language of the web, but it failed. It was way too slow and heavy to run Java programs in the browser [12].

Recently Google made an attempt for a new language called Dart that was meant to run natively, i.e. be supported just like JavaScript, in the browser but that project changed direction and today it's meant as a transpilation language instead of having its own Virtual Machine like JavaScript does [30]. A good reason why it's not such a good idea to support multiple VM's in browsers are summed up here in a WebKit thread from 2011 *[webkit-dev] WebKit branch to support multiple VMs (e.g., Dart)*

> In this case the feature is exposing additional programming languages to the web, something without any real benefit to anyone other than fans of the current "most awesome" language (not too long ago that might have been Go, a year or so ago this would have been ruby, before than python, i recall i brief surge in haskell popularity not that long ago as well, Lua has been on the verges for a long time, in this case it's Dart – who's to say there won't be a completely different language in vogue in 6 months?), but as a cost it fragments the web and adds a substantial additional maintenance burden – just maintaining the v8 and jsc bindings isn't trivial and they're for the same language.
>
> The issue here isn't "can we make multiple vms live in webkit" it's "can we expose multiple languages to the web", to the former i say obviously as we already do, to the latter I say that we don't want to. [64]

2.5 Elm

Transpiling to JavaScript instead of writing the language directly is a very popular approach and has been so for many years. Two popular transpilation languages out of several hundred are CoffeeScript and TypeScript [42]. These are two fairly simple transpilation languages, Typescript uses a similar syntax and structure to JavaScript, Coffeescript is more like Ruby. Elm is a far different story, it is a completely different language.

Elm is the brainchild of Evan Czaplicki and is currently working full time with the language at the company NoRedInk. Evan got the idea for the language when he did an internship at Google and realized how horrible it was to create user interfaces on the web. He was fascinated by how CSS had been around for twenty years but still you couldn't center a box within a box in a easy way [17].

Czaplicki started working on Elm in his Master Thesis [11] where he continued the research of FRP and made Elm in to a language that avoided two of the biggest problems with previous attempts at FRP[11]. After the thesis the interest for Elm started to grow, and Prezi hired Evan to work on it full-time.

Elm has a syntax reminiscent of Haskell and is different compared to other transpilation languages meant for front-end development in that it compiles to not only JavaScript, but also HTML and CSS. This allows for an entire web-app to be built only using Elm. The language uses immutable data structures and a virtual DOM that it uses to render to the screen.

Something that fell out of the development of Elm was the Elm Architecture that has a very elegant and simple way of thinking about data flow. This architecture has since been copied to a bunch of JavaScript frameworks. Redux created by Dan Abramov is basically a port of the Elm architecture that works well with React.js [54]. Angular 2 is also very influenced by Elm [1]. Even though Elm uses very different concepts compared to traditional web development, Evan Czaplicki constantly tries to focus on making it user friendly. One of his most popular talks is named *Let's be mainstream* [18] which pushes on this point. This focus to be more accessible to new people led to extensive development to make the Elm compiler extremely helpful in its error messages. Below is one such example.

Listing 9: Elm compiler error message

```
The argument to function ‘getFullName‘ is causing a mismatch.

21|             getFullName
22|>            {
23|>                    firstName = "Sam",
24|>                    lastName = "Sample",
25|>
26|>                    hairColor = "Brown",
27|>                    eyeColor = "Brown",
28|>
29|>                    address = "1337␣Elite␣st",
30|>                    phoneNumber = "867-5309",
31|>                    email = "foo@bar.com",
32|>
33|>                    pets = 2
34|>            }

Function ‘getFullName‘ is expecting the argument to be:

        { ..., phoenNumber : ... }

But it is:

        { ..., phoneNumber : ... }

Hint: I compared the record fields
          and found some potential typos.

        phoenNumber <-> phoneNumber
```

[9]

In JavaScript the *"phoneNumber"* would just be set to `undefined` and later on you would get some strange error of

```
undefined is not a function
```

or something similar when trying to access it. This is a huge drain of time and energy in regular JavaScript development. Elm takes safety very serious and lets the developer be confident that if the code compiles, it will work. Another thing that gives confidence to Elm developers is Elm's package manager, which enforces semantic versioning. When a package is updated, Elm check the new version against the old one and determines what the new version should be depending on the changes made to the code. This ensures that a significant change to the package is labeled as such and not as a minor change.

The strictness of the compiler and the syntax of Elm allows for the eco-system to reject code that may cause run-time exceptions and as of version 0.16, run-time exceptions simply do not happen [15]. This is another major selling point for the language because JavaScript itself is plagued with run-time exceptions. Elm compared to most languages that transpile to JavaScript has its own eco-system that is walled of and provides guarantees of no run-time exceptions. Elm should be thought of as a language that compiles to any given platform, but right now it only compiles down to client-side JavaScript. There are experimental libraries running Elm on the server-side.[31]

The Elm Architecture

The Elm Architecture is the preferred way to structure all Elm applications in *"infinitely nested components"*[19]. The pattern is useful for both Elm, JavaScript or any other language for that matter[20]. The architecture is simple, and is nothing like your traditional **MVC**. The Elm architecture is also called model-update-view(MUV) and as the name suggests it consists of three parts.

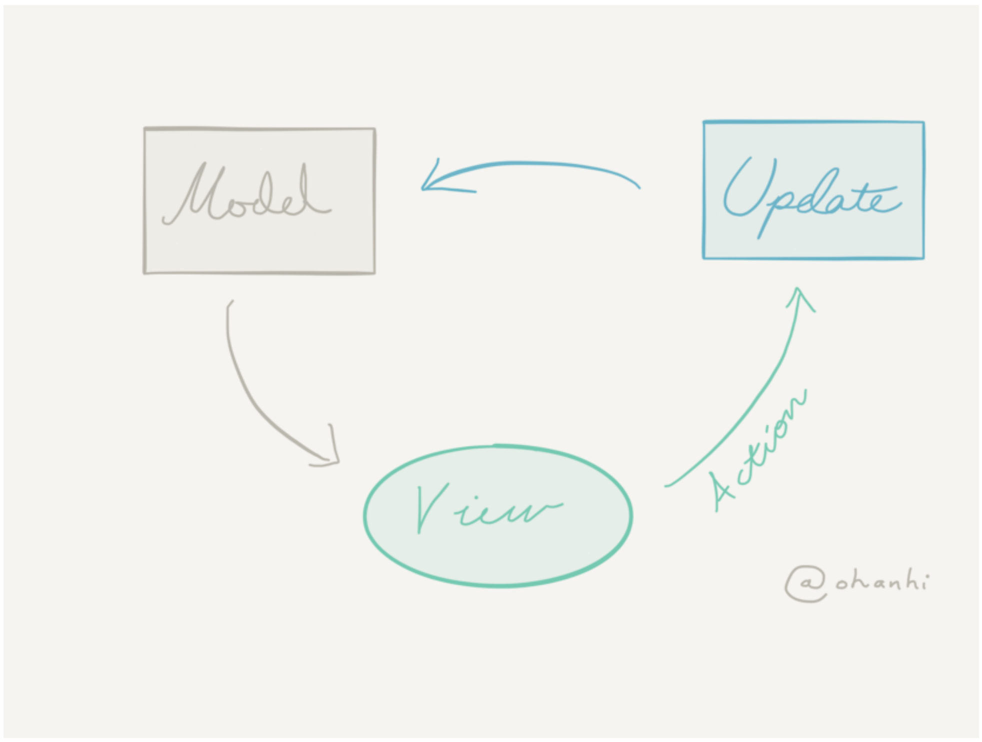

Figure 2.1: The Elm Architecture[32]

StartApp

StartApp is the official implementation of The Elm Architecture. It packages up the pattern in a easy to use micro framework. StartApp consists has two versions, one simple that does not use Signals and one that does.

Signals

Values that change over time are called Signals. Signals can represent any mutable value. One simple example is the position of the mouse, every time the mouse moves the value of `Mouse.position` changes. When a change is triggered the update-view cycle is run again. To not be inefficient, only things affected by the signal change is recalculated. The collection of all the signals are called the signal graph.

Model

The model is an immutable data structure that describes the data of the application, in Elm there is only one model for the whole application. The top level model contains references to the children models but not the details. Here is an example of that.

Listing 10: Model the Elm Architecture

```
module App.Model (..) where

import ChildComponent.Model
import ChildComponentTwo.Model


type alias Model =
  { childComponent : ChildComponent.Model.Model
  , childComponentTwo : ChildComponentTwo.Model.Model
  }
```

`ChildComponent.Model` can also have it's own children in the same fashion. The benefits of this approach is that the we get one source of truth in our app which prevents the state from becoming inconsistent, as is possible in Object-Oriented languages and most Flux implementations. This also makes it easier to figure out how the application got into a particular state and gives us the ability to utilize Time Travel.

Time travel

Given one source of truth it is possible to create a Signal Graph that represents all changes that can occur in our Elm program. If we keep track of the history of all our models when we run the program, we can go back and forth in the history, or "time" if you will. This technique is cleverly called time traveling[37]. When running this time traveling debugger, we can change the behavior of our application and see the changes in this recorded history change depending on how they would replay given these new changes to the application.

Update

The update function is the one taking care of updating our model. Every change to the model goes through the update function. Below is a simple update function in Elm.

Listing 11: Model the Elm Architecture

```
type Action = Increment | Decrement

update : Action -> Model -> Model
update action model =
  case action of
    Increment ->
      model + 1

    Decrement ->
      model - 1
```

Elm has support for union types, which is what you see as `Action`. This says that there is a type `Action` that can either be `Increment` or `Decrement`. Our `update` function takes an `Action` and a `Model` and returns a whole new `Model` rather than mutating the old one.

```
update : Action -> Model -> Model
```

In this example the model is just an `Int` and we return an incremented or decremented version of the model.

View

The `view` is what the end user sees and interacts with. Reactive programming is about reacting to events and that is what Elm does. The user presses the `+` sign in the application that sends an action to the `update` function that returns a new model. The `view` then renders view again with the new model.

Listing 12: Example of a View in Elm

```
view : Signal.Address Action -> Model -> Html
view address model =
  div []
    [ button [ onClick address Decrement ] [ text "-" ]
    , div [ countStyle ] [ text (toString model) ]
    , button [ onClick address Increment ] [ text "+" ]
    ]
```

Virtual DOM

The virtual DOM is a virtual representation of the actual DOM of the website. This virtual DOM is part of the application model and the environment figures out how to make the minimal amount of changes to the actual DOM in order to represent the current state of the application model. This let's the programmer focus on *what* the page should look like and not *how* to make it look like that. In traditional web development the developer had to explicitly change the DOM. Removing this responsibility from the developer creates abstractions that make DOM manipulation easier and more effective.

2.6 React

As can be read on the official website of the framework it is "A JavaScript Library For Building User Interfaces"[2]. React is being developed by Facebook and was released in March of 2013. It has since then gained a lot of popularity and is today one of the most popular frameworks used in industry. This is illustrated by the vast amount of companies using the framework in production, Netflix, Paypal, Imgur and Airbnb to name a few[41].

React uses techniques such as a virtual DOM, one-way data flow and declarative style of writing in JavaScript [65].

More than just the V in MVC

The bare bones of React can be seen as the view in the classic web design pattern model-view-controller. To use React as more than just a view layer, an application architectures for handling the rest of the front-end logic needs to be added. One popular alternative that Facebook themselves has come up with is Flux.

Flux utilizes a unidirectional data flow(covered later in this section) and has three different components: a dispatcher, stores, and views.

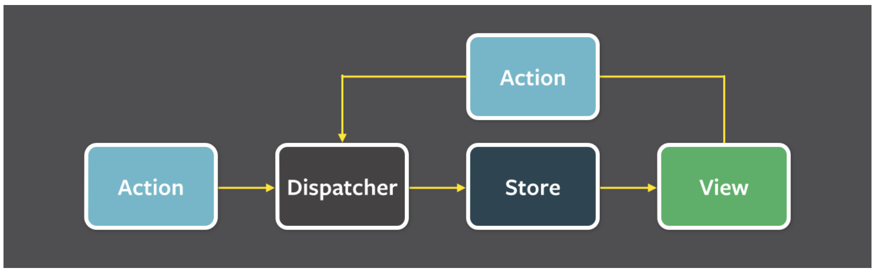

Figure 2.2: Flux architecture[25]

The flow of data in the Flux architecture consists of *Actions* which are sent to a *Dispatcher*. From there *Stores* are updated with new data, which then triggers the *View Controllers* which re-renders with that new data.

Components

In React, applications are structured with components. Each component have their own model which keeps track of the state. If the state changes React re-renders the component. Below is a simple timer component taken from the React website [61].

Listing 13: React Timer component [61]

```
var React = require('react');

var Timer = React.createClass({
  getInitialState: function() {
    return {secondsElapsed: 0};
  },
  tick: function() {
    this.setState({
    secondsElapsed: this.state.secondsElapsed + 1
    });
  },
  componentDidMount: function() {
    this.interval = setInterval(this.tick, 1000);
  },
  componentWillUnmount: function() {
    clearInterval(this.interval);
  },
  render: function() {
    return (
      <div>Seconds Elapsed: {this.state.secondsElapsed}</div>
    );
  }
});
module.exports = Timer;
```

The model is `this.state`, it has one property `secondsElapsed` which gets updated every second by an interval, initialized when the component was created. The view logic resides in the `render` function and that's it.

This component can be easily reused by any other component that first requires it and and then makes use of it in it's own render function by writing `<Timer />`.

A React component is almost completely declarative. By looking at the example above we can see a completely declarative `render` function.

This is no different from normal HTML, that too is declarative. However React does not present traditional JavaScript solutions for changing the DOM by using `document.getElementById("\#divName")`, but instead declare *how* the component should change and behave on new events by declaring functions such as `getInitialState, tick, componentDidMount, componentWillUnmount`.

Virtual DOM

React popularized the idea of a virtual DOM [7], which is a JavaScript object which represents the DOM. It makes a diff with that object on state changes and goes on to apply the smallest amount of actual DOM manipulations to have the view sync with the Virtual DOM.

Unidirectional data flow

React combined with Flux makes data flow explicit by only allowing direct data flow to be passed from an parent component to its child components. If one of these components want to change data in the owner, an action needs to be sent to the dispatcher which in turns makes the change to the store which holds the component's data. [24]

Because of this restriction on data flow, it is easier to follow where changes to data come from. Nested child components cannot directly change the state of any component higher up the tree, it has to instead communicate that through actions which in turn perform these changes inside of that component.

2.7 Language features of Elm

These are language features that exist in Elm. The first five are features that are very prevalent to functional programming and are taken from the paper "Why functional programming matters", written by John Hughes [35]. The last four points are more unique to Elm.

- Immutable data structures
- Pure functions
- Higher-order functions
- Currying
- Static type checking

These are not general features of functional programming but important to Elm

- No concept of Null
- Reactivity
- The Elm Architecture
- Declarative UI

Immutable data structures

Mutating values over time is very commonplace in imperative programming. Given a variable that is set to be the array

```
var myArray = [ 1, 2, 4, 5 ]
```

and you then want to append **6** to this array, this would change the original array to now be

```
[ 1, 2, 4, 5, 6 ]
```

By using immutable data structures, the value that `myArray` points to can not be mutated like this. Instead, a new array would be returned on every change to `myArray` and `myArray` would stay the same. This restriction is popular in functional programming languages.

According to Eric Meijer, a programming language cannot become more powerful by removing features [16], but it can be easier to reason about for humans. The trouble with mutability is that unexpected things can happen to your data.

Take the following code as an example mutation in JavaScript.

Listing 14: JavaScript mutation

```
function printMyArray(array){
  console.log(array);
  array.push('Mutated');
}

var myArray = [1,2,3,4];
printMyArray(myArray);
console.log(myArray);
```

Listing 15: JavaScript mutation output

```
[1, 2, 3, 4]
[1, 2, 3, 4, "Mutated"]
```

The variable `myArray` appear to unintentionally have been changed by `printMyArray`. This would not have been possible if JavaScript had enforced immutable data structures, in which case `myArray` would not have been mutated and print as expected. It is possible to use `concat` instead of `push` in Javascript. This would return a new array with the value concatenated but leave the original unchanged.

Pure functions

Mathematical functions take inputs, defines some calculations inside that function, and returns an output. Such a function is referentially transparent because the function can at any time be replaced by the value it returns without changing the behavior of the program [34]. A function that changes the surroundings or depends on some external data is an impure function *side-effects* [53].

The functional programming paradigm stems from the idea of pure functions and the use of them is one of the paradigm's main concepts and claimed benefits. Impure functions come in different forms and can thus be defined differently. One type of impure function is one that changes the surrounding application state. This type of function is said to have *side-effects*. However a function that is dependent on external API's is said to have *co-effects* [53].

Using an impure function can often be flexible and in some cases necessary, but it should be used with caution because it increases the complexity of the application code which increases the risk of bugs and makes the code harder to test. One such example may be the use of a global variable, which may be changed in unexpected and almost untraceable places, which causes the developer to spend time following a foggy trail through the code. It is however impossible to only have pure functions in an interactive environment. Every state change such as printing to the screen needs an impure function. So the best approach is to handle impure functions with care, which is something that has been researched to great extent in the functional programming language Haskell. Haskell has explicit features to handle cases of impure functions, by the usage of a so called *state monad*.

Higher-order functions

In functional languages functions can be treated like any other values and can be passed as arguments to other functions. This allows for higher order functions, which are functions that composes other functions[27].

This in combination with pure functions increases the modularity in a program [35]. Imagine a function which prints the salary of the employees in the company. Now if this function needs to be used with different sorting methods, this sorting function could be given as a parameter. Without higher-order functions, this would not have been as flexible. They give a lot of powerful ways to abstract away logic and compose programs from smaller decoupled functions.

Currying

This is a recurring supported feature in functional programming languages which allows for functions to be partially applied. Let us take a look at the following C code.

Listing 16: Definition for multiply/3 in C

```c
int multiply(x, y, z) {
        return x * y * z;
}
```

This function requires to be called with three and only three arguments, because C does not support currying. The same function written in Haskell look similarly.

Listing 17: Definition for multiply/3 in Haskell

```haskell
multiply x y z = x * y * z
```

This function allows for being called with zero, one, two or three arguments because Haskell supports *currying*. If the function is called with three arguments, it will work just like the function written in C. If however it is called with fewer arguments, a partially applied function would be returned.

Listing 18: Multiply50/1 partially applied from multiply/3

```haskell
multiply50 = multiply 5 10
```

The new function `multiply50` is `multiply` partially applied with the arguments 5 and 10 and expects one more argument to return a value. An identical function could have been written without currying as well with the exact same behavior.

Listing 19: Definition for multiply50/1

```
multiply50 x = 5 * 10 * x
```

Lazy evaluation

Instead of evaluating something right away the program wait until something is really needed before it evaluates the expression. This gives the possibility for, among other things infinite arrays from 1 to infinity. The numbers will appear only when you actually need them in some other calculation otherwise they are not put in memory. This can avoid unnecessary memory usage and in some cases give significant speed increases[7].

There are of course downsides to this as well. If too many calculations are put in memory for future use, memory may run out. Also if to many calculations need to be evaluated at once that can create a performance bottleneck. One example where lazy evaluation is bad is when dealing with time. Imaging a clock updating every second, creating an update to be made to the model of the clock every second, but without any part of the program actually using the clock. When the clock is later used, there will be a huge stack of useless updates executed at once which is bad for performance.

Static type checking

Data types are classifications for what a variable holds, for example an integer holds whole numbers. The types used by programming languages vary, and some choose to explicitly declare what type a variable should have, and others let this be inferred by either the compiler or the run-time.

Languages that use static type checking has explicit data types set for variables. This can improve readability of the code and is crucial for certain compile-time checking. Consider the following code snippet:

Listing 20: C++ example

```
int value = 5;
string color = "blue";
int newValue = value * color;
```

This would clearly not pass the compile step because a *string* cannot be multiplied with an *integer*. Languages that use dynamic type checking do not explicitly declare the types of their variables and the same snippet written in a dynamically typed language such as JavaScript would look like this:

Listing 21: JavaScript example

```
var value = 5;
var color = "blue";
var newValue = value * color;
```

These lines of code would pass perfectly fine in JavaScript, `newValue` resulting in `NaN`(Not a number), which is obviously a mistake by the programmer. It is however not caught by the compiler, and may cause unexpected errors right away or sometime in the future.

The choice between the two is not always this black and white, as Erik Meijer and Peter Drayton argues in their paper "Static Typing Where Possible, Dynamic Typing When Needed"[44]. They prefer languages make use of, as the title clearly states, to allow both types to be used where needed or preferred. This is exactly the type of type checking Elm has. There are certain problems that really benefit from dynamic typing, such as dynamic loading, where loading of a library can be done in run-time without prior knowledge and then released when used. This allows for efficient memory management because the machine running the script does not need to initially allocate memory for the library.[44]

Declarative UI

The code that make up the UI is declarative when we specifically state how it should look, rather than specifying the steps it takes to make it look a certain way. This is how the UI logic is written in both Elm and React.

No concept of null

null or *nil* in some languages is the absence of a value. You might think you have a *String* but in fact you have a *null*. Should this then be check for? Or was the value checked by the person who gave it to you? Maybe everything will work or maybe you whole program will crash.[46]

The concept of *null* vary in many language and it has caused a lot of trouble since it was invented in 1965. Here is what its inventor has to say about it.

> I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.[51]

In Elm there is a *type* named *Maybe*. This makes it explicit that what is returned from a function might not have a value. For example

Listing 22: Maybe elm-repl

```
> import List
> myList = [1,2,3,4]
[1,2,3,4] : List number
> List.head myList
Just 1 : Maybe.Maybe number
> emptyList = []
[] : List a
> List.head emptyList
Nothing : Maybe.Maybe a
```

As seen in the above example `head` returns a *Maybe* this is because of what is shown with `emptyList`, we can not be sure that there is a first value in the *List*. So when using `head` we need to take care of what to do in the case that the List is empty. This explicit way of taking cake of potential problems is what makes Elm able to not have any run-time exceptions. Here is a small program showing the use of this.

Listing 23: Minimal elm program with Maybe

```
import Html exposing (Html, text, div, input)
import Html.App exposing (beginnerProgram)
import Html.Events exposing (onInput)
import String

main =
  beginnerProgram {
      model = initalModel
    , view = view
    , update = update }

-- Model
type alias Model = String
initalModel = ""


-- UPDATE

type Msg = NewContent String

update : Msg -> Model -> Model
update msg model =
  case msg of
  NewContent content -> content


-- VIEW

view content =
  div []
    [ input [  onInput NewContent ] []
    , div [ ] [ text (findFirstChar content) ]
    ]


findFirstChar : String -> String
findFirstChar input =
  case List.head ( String.toList input) of
  Just a -> toString a ++ "␣is␣the␣first␣character"
  Nothing -> "Empty␣String"
```

The example might not show best practices since you most likely want `findFirstChar` to return a *Maybe String* and then let the developer using the function decide what should happen on `Nothing`. This could be done with the `Maybe.withDefault` function that lets us set a default value in case `Nothing` is returned. But the example shows a way of dealing with *Maybe* that illustrates the point that you have to be explicit.

Reactivity

A reactive way of handling data changes is to declaratively define how to react to new changes. This is in contrast to polling for changes and handle the new data as we find it. The idea is to directly get notified when a change occur and handle it as quickly as possible. The programming paradigm Functional Reactive Programming stems from this concept and seeks to hide the mechanism that controls time-flow under an abstraction layer[49]. This is covered in chapter 2.2.

2.8 Benefits of Elm features

By following the concepts used in Elm, a lot of benefits are gained to the code that is written. These benefits are not unique to Elm, but to any language that uses the features and techniques listed in the previous chapter on "Language features of Elm".

- Declarative code
- No flow of control
- Increased modularity
- Reduced state

Declarative code

Code that is written in a way that states *how* something should work without explicitly describing the steps involved in accomplishing it. Declarative code is generally shorter than code written imperatively because a lot of intricate detail is hidden by the compiler. In chapter 2.2 we mentioned the two paradigms functional programming and logic programming. These are both proponents of declarative programming while imperative programming is not.

No flow of control

The flow of control is the described way in which a program may execute. In a language that supports mutability, the result of a conditional branch may completely change the resulting control flow of a program. Imagine a variable being mutated differently depending on the outcome of a conditional branch, and that in turn changes the control flow in another conditional branch farther down the line. This quickly results in a lot of possible paths in which a program can execute, making the program more complex and error-prone. In functional programming, because there is no mutability, a conditional branch cannot change the outcome of future conditional branches, thus lowering the complexity of the application.

Increased modularity

Modularity is the concept that functionality can be broken down and reused throughout an application. According to the paper Why Functional Programming Matters [35] functional programming increases modularity of the program by using first-order functions and lazy evaluation. There are many techniques and concepts to improve or reduce modularity of a function. One way to achieve this is to make functions adhere to the single responsibility principle, which states that a function should only have one purpose. When a function breaks this, it becomes more specific and may contain both desirable and undesirable code for different parts of the application [43].

Reduced state

Every time a function causes a side-effect, state becomes larger and more complex. This makes code less traceable mathematically according to the paper "Functional Programming Matters" [35]. Pure functions cause no side effects and are thus generally preferred when possible.

This may not be a problem in a small application, but generally as an application grows this becomes a larger issue. Elm achieves a small state from using pure functions and immutable data structures.

3 Method

This thesis work will be conducted using the principles of action research. Action research is, when applied to technology, "the study of how technology is applied in the real world and the practical consequences of technology-enabled action"[48]. This correlates to our work because we seek to implement different features and functionality to a programming language, namely JavaScript, and conclude whether this is a possibility. More precisely, this thesis work will be conducted using a waterfall method with the following stages. First we will conduct ten weeks of literary study on the area of functional programming, followed by seven weeks of implementation of two application, one that will give us the insight needed of what developing in Elm is like and what concepts are prevalent and connected to the relevant literature. After the Elm application is done we will build the same application in JavaScript and use as many tools as necessary accomplish this.

Given that the scope of the application will be quite small, we will most likely have to do some side experiments to try out all features that are available in Elm. This will be done the last three weeks.

The applications will not be part of the thesis but are instead meant as a means of experimenting to get a feel for what Elm is and what JavaScript constructs and tools can be used to emulate it.

For the interested reader, here are the links to the applications (73 GitHub stars combined)

1. `https://github.com/carleryd/elm-hipster-stack`
2. `https://github.com/graphql-elixir/phoenix-hipster-stack`

4 Result

4.1 Introduction

In the Theory chapter we covered the functional and imperative paradigm, and also JavaScript which allows for imperative programming, and Elm which enforces functional programming.

In this chapter we will answer our research questions. This will mainly be done by showing how the different features of Elm can be brought into JavaScript, but also what the experience of working with the two languages in two different code bases was like.

4.2 Implementing features of Elm in JavaScript

To achieve the benefits found in Elm(chapter 2.9), we had to find frameworks, tools and libraries to use with JavaScript to enable the usage of these concepts. This is the list of the concepts we will focus to implement, as can be found in the theory chapter:

- Immutable data structures
- Pure functions
- Higher-order functions
- Currying
- Static type checking
- No concept of Null
- Reactivity
- The Elm Architecture
- Declarative UI

[60]

Some features may already exist but are not enforced by the language, some may be extended with tooling, and some we may be unable to achieve at all.

Immutable data structures

Since JavaScript does not support immutablity at the language level we need to use a library. There are a couple of libraries that does this, Mori.js and Immutable.js are the two most popular. Immutable.js is created by Lee Byron who works at Facebook. The library gives you access to the following data structures *List*, *Stack*, *Map*, *OrderedMap*, *Set*, *OrderedSet* and *Record*.

Using immutable data structures may seem like something that would hurt performance. But Immutable.js is smart when it creates new data structures. It reuses most of the previous data structures by setting pointers to it from the new one. We can do this since we know that the first data structure will never change.

Listing 24: Example using Immutable.js

```
var Immutable = require('immutable');
var map1 = Immutable.Map({a:1, b:2, c:3});
var map2 = map1.set('b', 50);
map1.get('b'); // 2
map2.get('b'); // 50
```

Pure functions

Pure functions can, just as in any other language, be implemented in JavaScript. Below is a simple example of a pure function in JavaScript.

Listing 25: JavaScript pure function

```
function circleArea(radius) {
        var PI = 3.14;
        return radius * radius * PI;
}
```

Higher-order functions

Since JavaScript has functions as first-class objects, higher-order functions are supported at the language level and require no additional library for them to be used.

Here is an example of using higher-order functions in JavaScript.

Listing 26: JavaScript higher-order function

```
function numberFilter(arr, constraint){
  var filteredArray = [];
  for(var i = 0; i < arr.length; i++){
      if (constraint(arr[i])){
        filteredArray.push(arr[i]);
      }
  }
  return filteredArray;
}


function biggerThanTwo(value){
  return value > 2;
}

function even(value){
  return value%2===0;
}

function prime(value){
  for(var i = 2 ; i < value; i ++){
    if( value%i === 0 ){
      return false;
    }
  }
  return value > 1;
}

function numbersFromZeroTo(size){
  var numberArray = []
  for(i = 0; i<=size; i++){
    numberArray.push(i);
  }
  return numberArray;
}

console.log(numberFilter(numbersFromZeroTo(10), biggerThanTwo)));
console.log(numberFilter(numbersFromZeroTo(250), even)));
console.log(numberFilter(numbersFromZeroTo(1000), prime)));
```

As is clearly shown, higher-order functions are fully supported at the language level in JavaScript.

Currying

Elm comes with currying as a language feature but in JavaScript we have to write something like [28]

Listing 27: Naive JavaScript currying [29]

```
function sum(x) {
  return function(y) {
    return x + y;
  };
}
var addTen = sum(10);
addTen(20); // 30
addTen(55); // 65
```

This is a bit inflexible though since if we extend it we would have to do this.

Listing 28: Naive JavaScript currying extended example [29]

```
function sumFour(w) {
  return function (x) {
    return function (y) {
      return function (z) {
        return w + x + y + z;
      }
    }
  }
}

sumFour(1)(2)(10)(20); // 33
var addTen = sumFour(3)(3)(4);
addTen(20); // 30
```

But in JavaScript we have the ability for higher order functions so we can make a curried version of a function that has the correct arity and can be called like a regular JavaScript function with comma separated values.

Listing 29: JavaScript currying [29]

```
function curry(fx) {
  var arity = fx.length;

  return function f1() {
    var args = Array.prototype.slice.call(arguments, 0);
    if (args.length >= arity) {
      return fx.apply(null, args);
    }
    else {
      return function f2() {
        var args2 = Array.prototype.slice.call(arguments, 0);
        return f1.apply(null, args.concat(args2));
      }
    }
  };
}
```

This function uses some pretty advanced JavaScript concepts but also shows the flexibility of JavaScript. Now we can create curried functions of arbitrary length.

Listing 30: Creating a JavaScript curried function[29]

```
var sumFour = curry(function(w, x, y, z) {
  return w + x + y + z;
});
```

And the following calls will work.

Listing 31: Calles to the curried function

```
var
  f1 = sumFour(10),         // awaiting three arguments
  f2 = sumFour(1)(2, 3),    // awaiting one argument
  f3 = sumFour(1, 2, 7),    // awaiting one argument
  x  = sumFour(1, 2, 3, 4); // x is equal to 1+2+3+4=10
  x2 = sumFour(1)(2)(3)(4); // fully applied; x2 equals 10
```

Luckily there are already libraries that does this and the one that we recommend to use is called Ramda.js[**https://github.com/ramda/ramda**]. Ramda.js is similar in scope to the libraries lodash and underscore, but uses a functional approach.

Static type checking

One of the things we really loved with Elm was the static type checking. Paired with the awesome compiler, it really improved the development experience. JavaScript don't have any types but the demand for it has made several alternatives to achieve this available. The two most popular ones are Flow.js(Facebook) and TypeScript(Microsoft). Flow works well in conjunction with React so that's the one we chose.

Flow is a static type checker that allows for introducing types to JavaScript code. This aids with finding bugs and errors that can be inferred in compile-time. The types are never seen by the run-time, they are stripped from the code when compiled with Flow. Examples are taken from flowtype.org [23] In the below example we call a function with `null` and then try to get the length of `null`. Since `null` does not have a `length` property, Flow finds this error by following the flow of data trought the program and checking for inconsistencies.

Listing 32: Flow null example

```
// @flow

function length(x) {
  return x.length;
}

var total = length('Hello') + length(null);
```

Here we see the error message we get, when we run Flow on the above example.

Listing 33: Flow null example output

```
7: var total = length("Hello") + length(null);
                                 ^^^^^^^^^^^^ function call
4:   return x.length;
              ^^^^^^ property `length`.
             Property cannot be accessed on possibly null value
4:   return x.length;
            ^ null
```

The error messages as surprisingly good, and it points us towards where the error happens and why. I does not suggest a solution but still this is very useful and several times while developing using flow it found bugs we had missed.
Below we correct the example so that is does not throw any more errors.

Listing 34: Flow null example corrected

```
function length(x) {
  if (x) {
    return x.length;
  } else {
    return 0;
  }
}

var total = length('Hello') + length(null);
//Gives no errors.
```

One really nice feature with Flow is that you can add it incrementally to your project. Every file that you want to be checked by Flow you type

```
// @flow
```

With flow we can add types to our JavaScript project, below is and example of that. The variable `x` is of type `string` and `y` is of type `number`. The function `foo` should return a `string`, which is obviously incorrect when we are in fact supposed to return a `number`.

Listing 35: Flow with types

```
// @flow

function foo(x: string, y: number): string {
  return x.length * y;
}

foo('Hello', 42);
```

Here we can see that Flow complains that `number` is incompatible with `string`.

Listing 36: Flow with types output

```
type_annotations.js:4
  4:    return x.length * y;
               ^^^^^^^^^^^^ number. This type is incompatible with
  3: function foo(x: string, y: number): string {
                                         ^^^^^^ string
```

To fix the error we change the return type to `number`.

Listing 37: Flow with types corrected

```
// @flow

// Changing the return type to number fixes the error
function foo(x: string, y: number): number {
  return x.length * y;
}

foo('Hello', 42);
```

No concept of Null

JavaScript like many other languages has *null*, and this can lead to some very problematic bugs where a null *value* has been sent around the whole application before it is accessed by, for example, `myValue.length` and find out that somewhere along the way something broke. To deal with the chance of variables being *null* or *undefined* we usually write a lot of defensive checking to guard against such errors.

Listing 38: Defensive programming JavaScript [57]

```
var person = {
    "name":"Homer␣Simpson",
    "address": {
        "street":"123␣Fake␣St.",
        "city":"Springfield"
    }
};

if (person != null && person["address"] != null) {
    var state = person["address"]["state"];
    if (state != null) {
        console.log(state);
    }
    else {
        console.log("State␣unknown");
    }
}
```

To be explicit and take care of situations where we are not sure that what we want returned will be returned we can introduce the *Maybe* data type into JavaScript.

Listing 39: Maybe implemented in JavaScript [57]

```
Maybe = value => {
  const Nothing = {
    bind(fn) {
      return this;
    },
    isNothing() {
      return true;
    },
    val() {
      throw new Error("cannot␣call␣val()␣nothing");
    },
    maybe(def, fn) {
      return def;
    }
  };

  const Just = value => ({
    bind(fn) {
      return Maybe(fn.call(this, value));
    },

    isNothing() {
      return false;
    },

    val() {
      return value;
    },

    maybe(def, fn) {
      return fn.call(this, value);
    }
  });

  if (typeof value === 'undefined' || value === null)
    return Nothing;

  return Just(value);
};
```

This JavaScript implementation of Maybe has the same *type signature* as in Elm `type Maybe a = Just a | Nothing`

We can now write the defensive programming example using Maybe.

Listing 40: JavaScript defensive with Maybe

```
const state =
        Maybe(person)
        .bind(p => p["address"])
        .bind(a => a["state"])
        .maybe("State␣unknown", s => s);
console.log(state);
```

In the example we try to bind `p => p["address"]` which works just fine because a `person` has that field. When we then try to bind `a => a["state"]` the `Maybe` construct will return a `Nothing` because we try to access a key that does not exist. When we lastly run `.maybe` on our now `Nothing` value, we get the return value `"State unknown"`. Had we not bound the `state` value we would have instead received a `Just(value)`, resulting in the actual address when calling `.maybe`.

This is very similar to how Elm would achieve the same thing with the `Maybe.withDefault` function.

Listing 41: Maybe in Elm

```
Maybe.withDefault "State␣unknown" (getState person)
```

But in Elm we would need a slightly different object representation that at least has the attribute we call on, otherwise the program would not compile.

Instead of implementing the Maybe implementation by ourselves, there are obviously libraries which do this. The most popular one is *folktale/data.maybe* [26].

Flow also has functionality for dealing with this issue where we try to access data that does not exist. If we have defined a *Person type* in Flow and it does not include the `state` key, it will give us an error at compile-time. If we have run-time problems where we get back JSON that might not have that field, this kind of solution is very useful.

This is a great pattern to follow but like with all things javascript it is not enforced and the developer must be be aware of situations where null and undefined might popup. But being aware of the issue and thinking in this terms is really helpful in developing JavaScript applications we found.

Reactivity

JavaScript does not support reactivity at the language level. But there are a lot of libraries that do offer this functionality. These are three such libraries written for JavaScript that we find interesting.

- Flapjax[22]
- RxJS[56]
- Bacon.js[6]

Flapjax is created by a couple of students from Brown University as their thesis. It's designed as a language that compiles to JavaScript but can also be used as a library[45]. It's the least popular of the three alternatives but uses the concept of signals to implement FRP which is interesting in relation to Elm.

Bacon.js and RxJS are pretty similar, however RxJS is more performant and twice as popular on Github. RxJS is being maintained by Netflix and is the basis of the recently popular

Cycle.js framework that is a competitor to React.js that is fully reactive.

Since JavaScript does not support FRP out of the box the implementation will be a bit more complicated that it otherwise would be. The following are a couple of examples using RxJS 4.

Listing 42: RxJs Example [**RxJs**]

```
/* Get stock data somehow */
const source = getAsyncStockData();

const subscription = source
  .filter(quote => quote.price > 30)
  .map(quote => quote.price)
  .subscribe(
    price => console.log(‘Prices higher than $30: ${price}‘),
    err => console.log(‘Something went wrong: ${err.message}‘);
  );

/* When we’re done */
subscription.dispose();
```

In Elm the equivalent would be.

Listing 43: RxJs example in Elm [31]

```
moreThan30 = List.filter (\quote -> quote.price > 30)
logItem v = log "Prices␣higher␣than␣$30:␣$"
Signal.map (moreThan30 >> List.map logItem) getAsyncStockData
```

In RxJS the developer must manually subscribe and dispose the subscription. In Elm, `Signal` is not coupled with the filtering, instead we just decide that we want to use those functions with our asyncData. Signals are introduced when we need them and are decoupled from the implementation details. In Elm, `Signal` is in the core library and as such it’s not a foreign concept as in JavaScript.

The function *Signal.map* that we used above has the following type signature

```
Signal.map : (a -> b) -> Signal a -> Signal b
```

It takes some transformation function `(a->b)` and a `Signal a` and return a `Signal b`. In React there is no support for using streams but it can be added with the help of any of the above mentioned libraries.

The Elm Architecture

There are a several different implementations of the *Flux* architecture. One of the most popular has taken a lot of inspiration from Elm. It is called Redux.

Both TEA and Redux takes an *action* and a *state* and gives back a new *state*.

```
(state, action) => state
```

Redux assumes that you never mutate your data just like in Elm.[55] This makes it pair very well with a library like *Immutable.js*. This is obviously not a requirement for not mutating data, it is also possible to consistently use techniques like `concat(newData)` instead of `push(newData)` but it's a good idea to enforce immutability with a library. In Redux you can mutate data, and in some edge cases you might want to, this differs from TEA where it's impossible to do this. Elm achieves its reactivity with the use of signals. Redux can achieve this too when paired with RxJS. Here is an example. To make Redux work even more like Elm using TEA, it can emulate the use of signals with the library RxJS. Here is an example.

Listing 44: Redux with RxJS [55]

```
function toObservable(store) {
  return {
    subscribe({ onNext }) {
      let dispose = store.subscribe(() => {
        onNext(store.getState())
      });
      onNext(store.getState())
      return { dispose }
    }
  }
}
```

Redux like TEA has three fundamental principles.

Single source of truth

That is one model for the whole application. Redux calls this the *Store*, Elm calls it the *Model*.

Listing 45: Single source of truth in Redux [55]

```
console.log(store.getState())

/* Prints
{
  visibilityFilter: 'SHOW_ALL',
  todos: [
    {
      text: 'Consider using Redux',
      completed: true,
    },
    {
      text: 'Keep all state in a single tree',
      completed: false
    }
  ]
}
*/
```

No direct mutation of state

The only way to mutate the state is to emit an *Action*. Action in Redux is the same as in Elm, only it's a JavaScript object instead of a special *Type*.

Listing 46: Dispatch action in Redux [55]

```
store.dispatch({
  type: 'COMPLETE_TODO',
  index: 1
})

store.dispatch({
  type: 'SET_VISIBILITY_FILTER',
  filter: 'SHOW_COMPLETED'
})
```

Changes to state are made by pure function

In Elm we have the *update* function, in Redux it's called the *Reducer*. In the Reducer does not mutate the state, but instead returns a totally new state object with the changes applied. This let's Redux, just like Elm, have a time traveling debugger.

Listing 47: Reducers in Redux

```
function visibilityFilter(state = 'SHOW_ALL', action) {
  switch (action.type) {
    case 'SET_VISIBILITY_FILTER':
      return action.filter
    default:
      return state
  }
}

function todos(state = [], action) {
  switch (action.type) {
    case 'ADD_TODO':
      return [
        ...state,
        {
          text: action.text,
          completed: false
        }
      ]
    case 'COMPLETE_TODO':
      return state.map((todo, index) => {
        if (index === action.index) {
          return Object.assign({}, todo, {
            completed: true
          })
        }
        return todo
      })
    default:
      return state
  }
}

import { combineReducers, createStore } from 'redux'
let reducer = combineReducers({ visibilityFilter, todos })
let store = createStore(reducer)
```

Here is a simple example combining the three concepts.

Listing 48: Complete Redux example

```
import { createStore } from 'redux'

function counter(state = 0, action) {
  switch (action.type) {
  case 'INCREMENT':
    return state + 1
  case 'DECREMENT':
    return state - 1
  default:
    return state
  }
}

store.subscribe(() =>
  console.log(store.getState())
)

store.dispatch({ type: 'INCREMENT' })
// 1
store.dispatch({ type: 'INCREMENT' })
// 2
store.dispatch({ type: 'DECREMENT' })
// 1
```

Redux is a great alternative when developing with React and makes it easy to follow the data flow in the application. One downside to Redux compared to Elm, except that it is not enforced, is that Redux has a lot more boilerplate and verbose syntax.

Declarative UI

As has been mentioned many times before, Elm takes a declarative approach to UI development. This is not something you can do with vanilla JavaScript as we will show below. But declarative UI is starting to become a popular approach for JavaScript frameworks and we are going to have a look at React.js and compare it to Elm. It's the most popular declarative framework at this point in time.

In React as well as in Elm, there is a declarative UI that lets you state *"what"* you want the page to look, and then *"how"* that is accomplished gets taken care of behind the scene. Here is an example view in Elm.

Listing 49: Elm simple program

```
import Html exposing (..)

main : Html
main =
  view 5


view : Int -> Html
view count =
  div []
    [ text count ]
```

These functions are all pure, if view is given the same input twice, it is guaranteed to produce the same output both times. They have no side effects on the language level, there is process that actually draws the message to the screen that is impure but that is not a part of the program. A benefit of having all pure functions is that you can perform caching of data, given the same input we can skip recalculating the result but instead just return the cached value. This cannot be done with full accuracy if we have side effects because the side effects may have caused some function to behave differently after subsequent runs.

The same function in React using ES6 syntax looks like this.

Listing 50: React Main component with separated View

```
class Main extends React.Component {
        render(){
            return (
                <View count={5} />
            )
        }
}

class View extends React.Component {
        render () {
        return (<div>{this.props.count}</div>);
  }
}

React.render(<Main />, document.getElementById('container'));
```

The React version can be considered pure, even though it has a co-effect due to the use of `this`. The `View.render` function takes in an object called props as argument that contains data passed into the component. `View.render` is a pure function that returns an immutable JavaScript object that later gets converted into HTML by React's renderer.

A program with the same behavior written in vanilla JavaScript would look like this.

Listing 51: Vanilla JavaScript replica of a React Main component

```
const view = (number) => {
  const div = document.createElement("div");
  div.innerHTML = number;
  return div;
}

const main = () => { return view(5); }

document.body.appendChild(main());
```

The pure JavaScript view is pretty similar to React, but now we don't have any framework taking care of how to render our component. We also have to explicitly add it to the DOM.

One major difference from the vanilla JavaScript version is that it requires the global object `document`. This is always available to any JavaScript app, but in React it's abstracted out, resulting in the the developer not having to access variables outside of the scope.

Interactivity

A web page without interactivity is not very exciting in this day and age. Such a web page is called static and does by definition not respond to any user input. Today all web pages have some kind of interaction, you may fill in a form, click a button, or just be able to navigate around with the help of tabs.

There are a myriad of different solutions to handle such interactions in JavaScript. Early solutions involved callbacks. The JavaScript framework Angular.js introduced a concept of two-way data binding. React.js and Angular 2 both use a one-way data flow with abstractions to make it more declarative.

One point which has differed a lot and still is under debate is how to store state. In jQuery the state is stored in the DOM. In Angular it's stored in each controller. React has several different solutions for this. The most popular one, Redux, tries to store as much as possible in a global state. But some state is still stored in each component. If we add some interactivity to our previous examples they would look like this.

Listing 52: Interactive React View

```
class Main extends React.Component {
    render(){
        return (
            <View />
        )
    }
}

class View extends React.Component {
        constructor(props) {
      super(props);
      this.state = { count: 0}
    }

    handleClick(){
      console.log("I␣am␣a␣side␣effect!")
      this.setState({count: this.state.count + 1});
    }

    render () {
      return (
        <div>
          <div>{this.state.count}</div>
          <button onClick={this.handleClick.bind(this)}>+
          </button>
        </div>
        );
    }
}

React.render(<Main />, document.getElementById('container'));
```

React uses an object-oriented approach where a component has an internal state that it mutates. Even though the state is encapsulated we have side effects with both `handleClick` and `render`.

In Elm the same program would look like this.

Listing 53: Simple Elm counter

```
import Html exposing (div, button, text)
import Html.Events exposing (onClick)
import Signal exposing (Mailbox,mailbox,foldp,map)

actions: Mailbox Action
actions =
  mailbox NoOp

model: Signal Int
model =
  foldp update 0 actions.signal

main =
  map (view actions.address) model


view address model =
  div []
    [ div [] [ text (toString model) ]
    , button [ onClick address Increment ] [ text "+" ]
    ]


type Action = Increment | NoOp


update action model =
  case action of
    Increment -> model + 1
    NoOp -> model
```

Elm uses something called `mailboxes`. They utilize an address and a signal. The `address` is where we send an `Action` to in our case. The other part of the `mailbox` is the `signal`, which, as we see here our model uses `actions.signal`. Every time the mailbox receives an action, it sends out a `signal` that in our case the model listens to.

The function `foldp` *"folds from the past"*. By accumulating the past models, we get data persistence in the application. Otherwise we would not be able to add one to the state of the model. By subscribing to the `mailbox`, `foldp` receives an action triggered on the `mailbox`. This action is then sent to the update function which calculates a new model. In turn, main recalculates because the model signal has changed, updating the view and the user sees the correct change occurring.

Compared to React, changes to the model can *only* occur at one place, in the update function. This gives the developer a straight forward way to follow the flow in the application. There is no need to second guess what may have caused the change.

We can abstract away a lot of the wiring that make up every Elm application by using `StartApp`.

Listing 54: Simple Elm counter using StartApp

```elm
import Html exposing (div, button, text)
import Html.Events exposing (onClick)
import StartApp.Simple as StartApp

main =
  StartApp.start { model = 0, view = view, update = update }


view address model =
  div []
    [ div [] [ text (toString model) ]
    , button [ onClick address Increment ] [ text "+" ]
    ]


type Action = Increment


update action model =
  case action of
    Increment -> model + 1
```

4.3 Summary

It was possible to achieve all the features from functional programming in Elm, as explained in chapter 2.7 and implemented in chapter 3.2.

Below are those features listed once more, in a comprehensive list describing how each feature was implemented in JavaScript. It is also mentioned if the feature was already present in the language, and if it was not, whether it could be fully utilized in it.

- Immutable data structures
- Pure functions
- Higher-order functions
- Currying
- Static type checking

These are not general features of functional programming but important to Elm

- No concept of Null
- Reactivity
- The Elm Architecture
- Declarative UI

Immutable data structures

Immutable data structures are not part of the JavaScript language at this day in time. But can be emulated with libraries. In the report we described how Immutable.js was able to emulate this feature.

Pure functions

It is possible to write pure functions in JavaScript, but it is not mandatory and the language does not direct you towards that kind of a pattern. There is also no requirement to be explicit when causing side-effects as in Elm.

Higher-order functions

Functions are first class citizens in JavaScript so there is no issue creating higher order functions.

Currying

Currying can natively be achieved in JavaScript but there are libraries which make the usage easier. A good library to achieve this is Ramda.js

Static type checking

Flow is a static type checker for Javascript. Is adds types to Javascript and supports incremental addoptation. This is not a native javascript solution and flow annotations are striped from the code before it's possible to run it in the browser.

No concept of Null

The concept of null will always be in JavaScript, but it is possible to avoid it with consistent usage of a *Maybe* data type. This can be manually created quite easily or used from a library. We chose to use *folktale/data.maybe*.

Reactivity

Reactivity is not nativly a part of JavaScript as it is in Elm but is achievable with the use of librarys such as *Flapjax*, *RxJS* or *Bacon.js*. In the report we showed a bit of *RxJS*

The Elm Architecture

Structuring an application using The Elm Architecture was possible in JavaScript with the usage of Redux.

Declarative UI

By using the framework React we could write declarative UI in JavaScript.

4.4 List of JS libraries needed to emulate Elm

- Immutable data structures -> Immutable.js
- Pure functions -> Native
- Higher-order functions -> Native

- Currying -> Ramda.js
- Static type checking -> Flow (Not JavaScript)
- No concept of Null -> folktale/data.maybe
- Reactivity -> Rx.js
- The Elm Architecture -> Redux
- Declarative UI -> React.js

5 Discussion

5.1 Result

We were able to implement all the features we set out to implement in JavaScript that we found beneficial from Elm. This does however not mean that the implementation came anywhere close to mimicking the experience of working with Elm. It is still a completely different eco-system which required a lot of libraries to check off all the features.

If a comparison was made between the amount of documentation pages in all those libraries combined, and the Elm documentation, the latter would surely be a shorter read. This would not be because Elm has scarce documentation, but because there is just so much more to cover when introducing all those libraries on top of an already existing language. With Elm you only need Elm. The cost of learning all these tools can make the cost higher than the gain. As with Immutable.js, Facebook themselves recommend not using the library with React for small projects because of this.

We certainly would prefer writing Elm instead of using all of these libraries, and it is hard to say whether it would be worth all the effort and maintenance to use them. Then mental load of keeping all this librarys close at hand and use them in the right way is very hard. In Elm you get all that for free, in Javascript you are threading in a minefield using all this librarys and if you start doing stuff like "I just want it to work so i will do this quick fix here" you are probably in for a rough time. There is certainly times where this librarys will not work well together. With this said, learning Elm and takeing concepts from it turns out to be immensely useful for Javascript development. You don't need them all, all the time use the once that gives the most bang for buck and don't over do it. One example of this would be to try to curry everything when you don't even use the function in a partially applied way anywhere. One of the pros with JavaScript is that it is easy to get started with because most people with some programming experience are likely to be more comfortable with such a flexible and traditional language compared to Elm.

All benefits with Elm were not covered in this report because they did not relate to our aim of implementing features of functional programming. Some of those benefits are that Elm has a unified style for how to write elm applications and a program called *elm-format* that auto styles the code in this way. This together with that almost all projects follow the Elm architecture makes it really easy to follow along in other peoples code. Another feature that elm has strait out of the box is the *elm-reactor* that is a webserver that serves your elm projects and lets you quickly test out your ideas. Elm also comes with a *repl* which is very useful. Even thou we talked about the compiler in the theory and how *Flow* fills some of its shoes in Javascript. The compiler is one strong selling point for Elm that we diden't really compare in detail. The effectiveness as a developer and the general work experience is far greater if you are presented with a helpful error message rather than the experience of solving errors in JavaScript, where the developer has to go through the following steps:

1. Refresh the browser
2. Manually use the feature
3. Realize nothing happens
4. Open up console and see an error message, which is usually not very helpful, such as "Undefined is not a function" at a line in your code.
5. Go to that line, make a change that hopefully resolves the issue and repeat

In comparison to the *elm-compiler* which has a great understanding of the structure of the program and can make an intelligent assessment over what is wrong. Not only can it make a more accurate guess than the JavaScript run-time, it can also do this *before* actually running the code. This results in *no run-time exceptions* which drastically improves development experience. We must say that we are thoroughly impressed by the power of *Flow* and will use it in all our future Javascript projects. It's no where near as powerful as the Elm-compiler but still offers some impressive error detection Sure its one extra thing to setup and get into your Javascript pipeline but with the incremental adoptation it supports and the power that comes with it is certainly a good choice.

In chapter 2.8 we listed the benefits of features in Elm. We believe these benefits can partly be gained by the methods we used, but are very elusive when the language does not enforce using the features.

5.2 Approach

The approach used to try to implement features of one language into another is perhaps not fair when comparing languages. We have not highlighted the benefits of JavaScript compared to Elm which surely exist, instead we have highly criticized the shortcomings of the language and the disadvantages of the techniques it uses. Because of this, we believe our report succeeds in conveying the benefits of using the functional reactive programming language Elm when developing frontend applications, but should not be seen as a comparison between two programming languages. That said, we believe we managed to shed light on the problems with extending a language with features to make it greater. This pushed on the point that language design is very important and a great language cannot be created by simply assembling a list of nice features without a grand design of how they should be structured in the language.

The sources used were most often from scientific papers in the area. When other online sources were used, they were done so as resources to code examples or different programming libraries.

6 Conclusion

6.1 Feedback on Aim and Research questions

We managed to fulfill the aim of this report by assessing which frameworks, libraries and tools were needed to implement a chosen set of beneficial features found in Elm. This chosen set of beneficial features came to answer our research question regarding what functional programming is with regards to Elm.

By researching how these different frameworks, libraries and tools work, and the theoretical benefits of the principles they implement, we were able to argue the benefit of introducing them to JavaScript. We came to a conclusion regarding the merits of doing this, although it could have been made clearer with an in-depth comparison of an application developed once in each direction, however we found this to be outside the scope of this thesis.

It was hard to pinpoint what constitutes the Elm experience. We argued that there were certain benefits to Elm that were impossible to replicate in JavaScript, such as the elm-compiler and elm-format, and thus concluded that the development experience of Elm could not be replicated in JavaScript.

6.2 Future work

It would be interesting to create a rigorous application with the libraries and tools we chose to use, and compare that to an Elm implementation in terms of performance, lines of code and development time.

6.3 What we learned

When we started the work we had preconceived notions that the JavaScript implementation wouldn't be as good Elm itself. The work on this thesis has not changed that notion, however there were a few things we learned on the way.

Before this thesis we had very limited experience with functional programming, we believed it was useful in some areas but either too difficult to use or perhaps not applicable in the applications we were writing in school and at work. But what we found was that, even though your manager won't let you write Haskell, a lot of the concepts from functional programming can be used on all projects. We learned the importance of limiting the state of the application, and how this can be achieved with ideas such as immutable data structures and pure functions. We believe these concepts can, and should be, applied to any programming language as often as possible to limit complexity of the application. One more valuable lesson we learned was the difference between *easy/simple* and *hard/complex*. That easy solutions is something that is close at hand and something you have experience doing, it does not have to be a simple solution. And many times as developers we get stuck in our old ways and take the easy way out just to later down the line be punished by the added complexity it brought.

Bibliography

[1] *175 JSJ Elm with Evan Czaplicki and Richard Feldman*. https://devchat.tv/js-jabber/175-jsj-elm-with-evan-czaplicki-and-richard-feldman. (Accessed on 06/07/2016).

[2] *A JavaScript library for building user interfaces | React*. https://facebook.github.io/react/. (accessed on 05/05/2016).

[3] *A Short History of JavaScript - Web Education Community Group*. https://www.w3.org/community/webed/wiki/A_Short_History_of_JavaScript. (Accessed on 06/20/2016).

[4] *Arduino - Home*. https://www.arduino.cc/. (Accessed on 05/09/2016).

[5] *asm.js*. http://asmjs.org/spec/latest/. (Accessed on 05/09/2016).

[6] *Bacon.js - Functional Reactive Programming library for JavaScript*. https://baconjs.github.io/. (Accessed on 05/13/2016).

[7] *blazing-fast-html*. http://elm-lang.org/blog/blazing-fast-html. (Accessed on 06/07/2016).

[8] Timothy Budd. *An Introduction to Object-Oriented Programming (3rd Edition)*. Pearson, 2001. ISBN: 0201760312.

[9] *Compilers as Assistants*. http://elm-lang.org/blog/compilers-as-assistants. (Accessed on 04/04/2016).

[10] Douglas Crockford. *JavaScript: The Good Parts*. O'Reilly Media, 2008. ISBN: 0596517742. URL: http://www.amazon.com/JavaScript-Good-Parts-Douglas-Crockford/dp/0596517742?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=0596517742.

[11] Evan Czaplicki. "Elm: Concurrent FRP for Functional GUIs". In: *Senior thesis, Harvard University* (2012).

[12] *Douglas Crockford: Java was a colossal failure...JavaScript is succeeding because it works. - JAXenter*. https://jaxenter.com/douglas-crockford-java-was-a-colossal-failure-javascript-is-succeeding-because-it-works-105395.html. (Accessed on 04/04/2016).

[13] Conal Elliott and Paul Hudak. "Functional Reactive Animation". In: *International Conference on Functional Programming*. 1997. URL: http://conal.net/papers/icfp97/.

[14] Eric Elliott. *Programming JavaScript Applications: Robust Web Architecture with Node, HTML5, and Modern JS Libraries*. O'Reilly Media, 2014. ISBN: 1491950293. URL: http://www.amazon.com/Programming-JavaScript-Applications-Architecture-Libraries/dp/1491950293?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=1491950293.

[15] *Elm*. http://elm-lang.org/. (Accessed on 04/04/2016).

[16] *Erik Meijer: Functional Programming - YouTube*. https://www.youtube.com/watch?v=z0N1aZ6SnBk. (Accessed on 04/04/2016).

[17] *Erlang Factory SF 2016 Keynote Phoenix and Elm – Making the Web Functional - YouTube*. https://www.youtube.com/watch?v=XJ9ckqCMiKk&feature=youtu.be&t=29m5s. (Accessed on 04/04/2016).

[18] *Evan Czaplicki - Let's be mainstream! User focused design in Elm - Curry On - YouTube*. https://www.youtube.com/watch?v=oYk8CKH7OhE. (Accessed on 06/07/2016).

[19] *evancz/elm-architecture-tutorial at dc1f52b1560fb1719242920d97cb009d49d8fcd2*. https://github.com/evancz/elm-architecture-tutorial/tree/dc1f52b1560fb1719242920d97cb009d49d8fcd2. (Accessed on 05/03/2016).

[20] *evancz/elm-architecture-tutorial: How to create modular Elm code that scales nicely with your app*. https://github.com/evancz/elm-architecture-tutorial. (Accessed on 05/12/2016).

[21] David Flanagan. *JavaScript: The Definitive Guide: Activate Your Web Pages (Definitive Guides)*. O'Reilly Media, 2011. ISBN: 0596805527. URL: http://www.amazon.com/JavaScript-Definitive-Guide-Activate-Guides/dp/0596805527?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=0596805527.

[22] *Flapjax*. http://www.flapjax-lang.org/. (Accessed on 05/13/2016).

[23] *Flow | Five simple examples*. http://flowtype.org/docs/five-simple-examples.html. (Accessed on 05/11/2016).

[24] *Flux | Application Architecture for Building User Interfaces*. https://facebook.github.io/flux/docs/overview.html. (Accessed on 05/11/2016).

[25] *Flux architecture*. https://facebook.github.io/flux/docs/overview.html.

[26] *folktale/data.maybe: A structure for values that may not be available or computations that may fail*. https://github.com/folktale/data.maybe. (Accessed on 05/13/2016).

[27] *Functional programming - HaskellWiki*. https://wiki.haskell.org/Functional_programming. (Accessed on 04/04/2016).

[28] *Gettin' Freaky Functional w/Curried JavaScript*. http://blog.carbonfive.com/2015/01/14/gettin-freaky-functional-wcurried-JavaScript/. (Accessed on 04/04/2016).

[29] *Gettin' Freaky Functional w/Curried JavaScript*. http://blog.carbonfive.com/2015/01/14/gettin-freaky-functional-wcurried-javascript/. (Accessed on 05/11/2016).

[30] *Google Dart Will Not Be The Next Default Programming Language Of The Web - ARC - ARC*. http://arc.applause.com/2015/03/27/google-dart-virtual-machine-chrome/. (Accessed on 04/04/2016).

[31] Noah. Hall. "Elm - a typesafe language for the web". In: ().

[32] *How Elm made our work better — Futurice*. http://futurice.com/blog/elm-in-the-real-world. (Accessed on 05/09/2016).

[33] *How to enable JavaScript in your browser and why*. http://enable-javascript.com/. (Accessed on 06/07/2016).

[34] J. Hughes. “Why Functional Programming Matters”. In: *Comput. J.* 32.2 (Apr. 1989), pp. 98–107. ISSN: 0010-4620. DOI: 10.1093/comjnl/32.2.98. URL: http://dx.doi.org/10.1093/comjnl/32.2.98.

[35] J. Hughes. “Why Functional Programming Matters”. In: *Computer Journal* 32.2 (1989), pp. 98–107.

[36] Ashley Hull and Hasmet Genceli. “A SYNOPSIS OF SOFTWARE TECHNOLOGIES USED IN TODAY’S ENGINEERING SOFTWARE”. In: ().

[37] *interactive-programming*. http://elm-lang.org/blog/interactive-programming. (Accessed on 05/12/2016).

[38] *jQuery*. https://jquery.com/. (Accessed on 05/11/2016).

[39] *JSX | XML-like syntax extension to ECMAScript*. https://facebook.github.io/jsx/. (Accessed on 05/17/2016).

[40] P. J. Landin. “The Next 700 Programming Languages”. In: *Commun. ACM* 9.3 (Mar. 1966), pp. 157–166. ISSN: 0001-0782. DOI: 10.1145/365230.365257. URL: http://doi.acm.org/10.1145/365230.365257.

[41] *Libscore top companies using React*. http://libscore.com. (Accessed on 05/05/2016).

[42] *List of languages that compile to JS · jashkenas/coffeescript Wiki*. https://github.com/jashkenas/coffeescript/wiki/list-of-languages-that-compile-to-js. (Accessed on 04/04/2016).

[43] Robert C. Martin. *Agile Software Development, Principles, Patterns, and Practices*. Pearson, 2002. ISBN: 0135974445. URL: http://www.amazon.com/Software-Development-Principles-Patterns-Practices/dp/0135974445?SubscriptionId=0JYN1NVW651KCA56C102&tag=techkie-20&linkCode=xm2&camp=2025&creative=165953&creativeASIN=0135974445.

[44] Erik Meijer and Peter Drayton. “Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages”. In: Citeseer. 2004.

[45] Leo A Meyerovich et al. “Flapjax: a programming language for Ajax applications”. In: *ACM SIGPLAN Notices*. Vol. 44. 10. ACM. 2009, pp. 1–20.

[46] *Model The Problem*. http://elm-lang.org:1234/guide/model-the-problem. (Accessed on 05/13/2016).

[47] Sebastian Nanz and Carlo A. Furia. “A Comparative Study of Programming Languages in Rosetta Code”. In: *CoRR* abs/1409.0252 (2014). URL: http://arxiv.org/abs/1409.0252.

[48] Lene Nielsen. “Personas In "The Encyclopedia of Human-Computer Interaction, 2nd Ed."." In: ed. by Mads Soegaard and Rikke Friis (eds.). Dam. 2013. URL: http://www.interaction-design.org/encyclopedia/personas.html.

[49] Henrik Nilsson, Antony Courtney, and John Peterson. “Functional reactive programming, continued”. In: *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*. ACM. 2002, pp. 51–64.

[50] *Node.js*. https://nodejs.org/en/. (Accessed on 05/09/2016).

[51] *Null References: The Billion Dollar Mistake*. http://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare. (Accessed on 05/13/2016).

[52] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Inc, 2008. ISBN: 0981531601.

[53] Tomas Petricek, Dominic Orchard, and Alan Mycroft. “Coeffects: A calculus of context-dependent computation”. In: *Proceedings of International Conference on Functional Programming*. ICFP 2014. Gothenburg, Sweden.

[54] *Prior Art | Redux*. http://redux.js.org/docs/introduction/PriorArt.html. (Accessed on 06/07/2016).

[55] *Prior Art | Redux*. http://redux.js.org/docs/introduction/PriorArt.html. (Accessed on 05/16/2016).

[56] *Reactive-Extensions/RxJS: The Reactive Extensions for JavaScript*. https://github.com/Reactive-Extensions/RxJS. (Accessed on 05/13/2016).

[57] *Sean Voisen » A Gentle Intro to Monads … Maybe?* http://sean.voisen.org/blog/2013/10/intro-monads-maybe/. (Accessed on 05/13/2016).

[58] *Semantic Versioning 2.0.0*. http://semver.org/. (Accessed on 06/07/2016).

[59] *The Deep Roots of Javascript Fatigue*. https://segment.com/blog/the-deep-roots-of-js-fatigue/. (Accessed on 06/06/2016).

[60] *The JavaScript Problem - HaskellWiki*. https://wiki.haskell.org/The_JavaScript_Problem. (Accessed on 04/04/2016).

[61] *Timer component*. https://facebook.github.io/react/. (Accessed on 05/05/2016).

[62] *TIOBE Index | Tiobe - The Software Quality Company*. http://www.tiobe.com/tiobe_index. (Accessed on 04/04/2016).

[63] Peter Van Roy et al. “Programming paradigms for dummies: What every programmer should know”. In: *New computational paradigms for computer music* 104 (2009).

[64] *[webkit-dev] WebKit branch to support multiple VMs (e.g., Dart)*. https://lists.webkit.org/pipermail/webkit-dev/2011-December/018806.html. (Accessed on 06/07/2016).

[65] *Why React? | React*. https://facebook.github.io/react/docs/why-react.html. (Accessed on 06/07/2016).

[66] *World Wide Web Consortium (W3C)*. https://www.w3.org/. (Accessed on 05/12/2016).