BACKGROUND REPORT: Elm as a Practical Language for Interactive Web Applications

Yang, Jessica jessicayzt@alumni.ubc.ca
van der Kooi, Tim vanderkooi11@gmail.com
Hong, Karen khong@alumni.ubc.ca
Rodgers, Laura laurarodgers@alumni.ubc.ca

ABSTRACT

Elm is a functional language specialized for building robust web applications. This report explores some of the unique features of Elm that makes it well-suited for this particular application. We discuss the implications and impact of these features, and compare and contrast Elm to other more popular languages for web applications. We also investigate features of Elm that are necessary or beneficial to the implementation of an interactive web application in the form of a platform game.

1. OVERVIEW

We will focus on the value of Elm as an alternative programming language for interactive web applications by comparing it to other popular languages and noting their similarities and differences. We will evaluate the unique elements and features of Elm that have particular value to the creation of our project, citing existing applications.

Our project will create a side-scrolling platform game to explore Elm's features that benefit and hamper designing an interactive web application. In particular, this project will provide first-hand insight into the advantages and disadvantages of Elm's features, such as the enforced modularity through the Elm Architecture, the virtual DOM, no runtime exceptions, and the impact of Elm's static typing and type inference system in producing reliable, easy-to-debug web applications. By creating a non-trivial platform game in Elm, our project will uncover how Elm's unique features and structure influence development strategies in web application development.

All in all, this report will examine the features that make Elm unique and advantageous as compared to its peers, while keeping the context of the project in its investigative scope.

2. INTRODUCTION

Elm is a functional language based on the Haskell paradigm but designed for straightforward use in front-end web applications. It compiles to JavaScript and runs in the browser. [1]

3. HIGHLIGHTED FEATURES

In this section, we will highlight several features of the Elm language that sets it apart from similar languages, and discuss them.

3.1. Syntactical Differences

Elm's syntax has been designed to be clean, readable, and intuitive. It has several notable differences compared to other languages such as JavaScript, Python, or Java, one of which is function definition and application, which is denoted with a space [1]. For example, in

```
isPositive n = n > 0
isPositive 3
```

the function application of the function `isPositive` is denoted with a space after the function name. This light syntax avoids the common use of parentheses and commas, which cause clutter and limit readability in large applications. Like Python, indentations and

spaces are part of Elm's syntax, which means that it is not a freeform language.

3.2. "Building Block" Features

Like one of its parents, Haskell, the most common data structures in Elm are homogenous lists [1]. Elm also features tuples and records, which are similar to objects in JavaScript or Python. However, unlike JavaScript, you cannot ask for a field that does not exist, and no field will ever be `undefined` or `null` without a known expectation of this possibility. This leads to more safety and reliability.

3.3. Static Typing and Type Inference System

Here, we will outline what in our opinion is Elm's most major benefit, and what sets it apart from other languages for web applications. Elm is statically typed, which means that type errors are found at compile time, contrary to JavaScript, which discovers type errors at runtime due to being a dynamically typed language [1]. Elm's types are also inferred by the Elm compiler, which means that the input and output types of functions are inferred by deduction. Due to this, type annotations or function contracts can be omitted. If one adds type annotations, the compiler will still perform type inference and check its results against the provided type annotations [1]. This ensures that there will be no incorrect type annotations littering the application.

3.4. Tagged Union Types

Elm provides tagged union types that are designed to deal with difficult-to-represent data (ie. data that contains values that could take on several different fixed types) [1]. Creating a union type generates constructors for the types in the union type, which tag the type that is in use at runtime, hence the name of "tagged union types." For example, in

```
type Character = Jack | Zombie
```

We have three cases defined along with their constructors. Tagged unions are typically paired with `case` statements [4], such as

```
type Character = Jack | Zombie
speed : Character -> Float
speed character =
      case character of
            Jack ->
                  4.0
            Zombie ->
                  2.5
```

Union types are easily extensible, as new cases will be checked by the compiler. In the above example, if we were to add a `Ghost` type into the union type without changing the `case`, the code would not compile due to the lack of a `Ghost` type in the `case`. Another benefit of union types is that they encourage the development of individual subcases, leading to small, reusable parts. The reusability of these parts is due to their independence. Each "subproblem" or type in a tagged union type can be considered independently from the others.

3.5. The Elm Virtual DOM

As one of many languages used for web development, Elm must also address the sluggishness of DOM manipulations and their impact on performance. Updating nodes in the DOM is a lot of work. The browser must consider the children of updated nodes, the recomputation of physical dimensions, browser extensions, and more. In order for Elm to yield competitive performance, it needs to allow users to make changes to views quickly and simply. Elm's Virtual DOM is a construct largely responsible for the speed of the language. By providing an abstraction on top of the document object model, the work of updating the DOM is automatically minimized without the need for additional optimizations.

When making changes to the view, the virtual DOM uses a process called diffing to determine

the updates that need to be made. This process involves creating a new virtual DOM tree and comparing it with the old one to obtain a record of changes. “Batching the instructions” allows for multiple edits to occur within an update. Instead of rendering each separate change, multiple changes to the DOM are made before re-rendering. Since the virtual DOM has been designed for operations such as diffing, this all happens substantially faster than when comparing to the real DOM. And because these strategies are taken care of by Elm's runtime, users of the language are free to concentrate on developing modular code without focusing on these low level details.
But the virtual DOM is by no means a new concept. In fact, React first popularized the idea in 2013 [5]. Yet, when comparing the performance of JavaScript frameworks such as React, Ember, and Angular, Elm comes out on top [6]. So how is it that Elm’s Virtual DOM is able to outperform these longer established implementations?

In a benchmark test where React, Angular, and Elm were stripped of optimizations, Evan Czaplicki, the creator of Elm, showed that the actual diffing of a simple TodoMVC app is significantly faster in Elm [6]. He credits this performance to the considerations taken towards using fast data structures and avoiding slow operations and memory allocation. But where Elm really shines is when its performance optimization capabilities are fully taken advantage of. Its ease of use and the difficulty of introducing bugs makes it extremely simple to render views quickly in Elm.

3.6. Performance Optimization Strategies

The two main virtual DOM optimization strategies are skipping and aligning work. Both concepts aim to avoid unnecessary computations in the diffing process. What is meant by “skipping work,” is to avoid comparing parts of the view that we know has not changed. In Elm, this idea can be applied through the use of the Html.Lazy module. Because Elm’s functions are pure, by simply comparing the inputs of the functions, we can definitively know if the output will change. Therefore, if we compose a view with Html.Lazy, the diffing process is able to use the inputs to determine equality while delaying the building of the virtual DOM. Additionally, since Elm’s data structures are immutable, it can check any models inputted to the view referentially instead of structurally.
The second concept, aligning work, aims to avoid pointless diffing when adding or removing child nodes. In Elm, this is accomplished by using the Html.Keyed module which pairs a unique identifier to every child node. This identifier allows the diffing algorithm to optimize any changes to the child nodes.

Both of these optimization techniques are common to languages that use virtual DOMs. However, when applied in Elm, developers need only add optimizations to the view in a way that is nearly impossible to introduce bugs. This is not the case for other web frameworks such as React. In order to get the same performance benefits of using Html.Lazy in React, we need to write our own update logic and may need to start making architectural changes. Its equivalent, shouldComponentUpdate, lets React know if it can skip re-rendering a component. To take advantage of this functionality, the programmer is responsible for writing the logic to determine whether a module should be re-rendered. This poses a risk of introducing bugs into your code. In addition, it may be necessary to create smaller and smaller components the more you want to optimize. These are problems that Elm developers need not worry about and as a result Elm code tends to be highly optimized and fast.

3.7. Functional Language Features

As Elm is a functional language, it supports many functional language features based on Haskell's semantics, such as stateless functions and higher-order functions. The usage of anonymous functions are alike to Haskell both syntactically and semantically,

```
\n -> n * 2
```

being a nameless function that doubles its argument. As with Haskell, every function is by default curried,

```
mult x y = x / y
```

having the type of `Float -> Float -> Float`. This allows for clean usage of higher-order functions, even with chaining, through Elm's `<|` and `|>` pipe operators.

With pure functions come immutability of data structures, which allows for optimization through the reusability of shared memory whenever possible, which is feasible due to the lack of mutable state [2].

3.8. The Elm Architecture Pattern

The Elm architecture pattern is a simple pattern that breaks the application down into 3 parts, the "model," or state, a way to "update" the state, and a way to "view" the state [1].

The model, update, and view are entirely decoupled, allowing for more modularity and extensibility. This architecture pattern allows you to build an application without focusing on the architecture of the code, and add to it without spending lots of time refactoring the structure as it will remain well-architected. With this pattern, it encourages the writing of code that is easy to reuse, as it is difficult to introduce duplicated code.

Due to all Elm programs following the same architecture [1], having a thorough understanding of the architecture leads to understanding the architecture of all Elm programs.

3.9. Declarative Programming Paradigm

In the broadest sense, languages such as Elm which use a declarative programming paradigm are distinguished by referential transparency: if a given reference is immutable and thus interchangeable at compile time, static analysis and compiler optimizations are much more easily implemented. This stands in contrast to imperative languages, for which execution is highly dependent on mutable state.

Declarative programming lends itself to abstracted structure and straightforward refactoring, as it foregrounds what a program or component should accomplish and not the lockstep details of how it should be done.

No matter how cleanly they are presented or how clearly they are commented, the minutiae of control flow tend to introduce opacity when people read code. Replacing these details with higher-order functions, for example, can make the meaning of a passage easier to grasp.

Greater readability also leads to greater extensibility. When a program is a book one can open at any page and still expect to make some sense of, it's much easier to insert (or excerpt) a chapter.

3.10. Error Types

Elm's architects advertise a guarantee of "no runtime exceptions," which is majorly rooted in Elm's handling of nullable values. Contrary to JavaScript, there are no null or undefined values. In Elm, errors are treated as data that can be manipulated. There are three main data structures for error handling, `Maybe`, `Result`, and `Task`.

3.10.1. Maybe Type

The Elm `Maybe` type is the heart of why there is no null dereferences in Elm. It allows for cleaner handling of nullable values by expressing nullability as data [1]. This could be compared with Optional in Java, which is "a container object which may or may not contain a non-null value." In the same sense, the Elm `Maybe` type indicates that a value may be there, or may not be there. It is a tagged union type, defined as

```
type Maybe a = Nothing | Just a
```

The usage of a `Maybe` type for handling of null values is simple, as the union type creates two constructors for each type, `Nothing` and `Just`. The data can then be differentiated with a `case` expression, allowing for both types to be accounted for [1]. Due to the prevalence of nullable values in applications, the Elm `Maybe` type is quite valuable when used correctly.

3.10.2. Result Type

With Elm's `Result` type, the possibility of failure is always handled. This could be compared to exception handling in other languages, such as Java's try/catch/finally blocks. Unfortunately, there is always the risk of unhandled exceptions leading to an application crash. Elm avoids this with the `Result` type, which makes sure that failure is *always* handled.

In this example use of `Result`

```
String.toInt : String -> Result String
Int
```

we can see the similarities to Java's String.parseInt which throws a NumberFormatException that must be remembered to be dealt with by the programmer. However, in Elm, as the `Result` type is returned [1], defined as

```
type Result error value
  = Err error
  | Ok value
```

the `Result` type must be accessed with `case`, where the compiler will force all cases (both `Err` and `Ok`), to be handled appropriately.

3.10.3. Task Type

The `Task` type is similar to the `Result` type and allows for smooth error handling for errors from asynchronous operations, such as HTTP requests [2]. `Task` types can be mapped together, much like JavaScript `Promises`, and chained together. The `Task` type also allows use of a callback for recovery if a `Task` fails [2], allowing for clean error handling in asynchronous situations.

4. IMPLICATIONS OF FEATURES

In this section, we will elaborate on the implications of the aforementioned features we have chosen to highlight, and the combined impact these features have on the Elm language.

4.1. Separation of Data and Logic

There lies a very discrete difference between Elm records and JavaScript or Python objects, or objects in any object-oriented language, which is that an Elm record cannot be self-referential. This means that there exists no `this` or `self` keyword to refer to the record [1]. This self-referential nature of objects in object-oriented languages is what Elm avoids to allow for a separation of data (the objects, or the records, in Elm's case) and logic.

4.2. Types to Model Invariants

With Elm's type system, designers of an application are forced to model their domain more carefully [3]. With Elm being statically typed, types can be used to model invariants in the application, and violations of these invariants can be found at compile time. This

ensures that these invariants are maintained throughout the application.

4.3. Safe Refactoring and Extensibility

With Elm's pure functions, refactoring is safe as functions can be moved around as individual components. Safe refactoring also comes from Elm's compiler, which checks all branches in conditional expressions such as `case` or `match`. If something is not accounted for, the application will fail to compile. This has led to Elm's architects coining a process called "compiler driven design," inspired by the Elm compiler's friendly, helpful messages, and the fact that an Elm application that compiles is likely to not exhibit runtime errors in practice [1].

The features that lead to safe refactoring also lead to increased extensibility in applications, as new code will be verified against the compiler for type errors and flows through the application that are unaccounted for.

4.4. Impact of Error Types and Type System

As errors are treated as types in Elm (via the `Maybe`, `Result`, and `Task` types), coupled with the Elm type system, this becomes very powerful. Through type inference, the possibility of failure through these error types are detected at compile time [3], and therefore, if error handling is not implemented, compilation is not possible.

5. FEATURES REQUIRED FOR EFFECTIVE GAME DESIGN

In this section, we will discuss the elements of a language that are necessary for game design, and how Elm provides these elements.

5.1. Managing State in Game Design

Managing state is an important design consideration in any video game. Therefore, it might seem counter-intuitive to program a platform video game in Elm, a functional programming language that lacks mutable state. How will this language provide advantages in crafting a video game?

Elm decouples the complex dependencies of state seen in interactive games and uses an immutable approach to programming. Instead of updating global variables or changing fields inside data structures or objects, Elm produces new objects to update the changing state of the game. The Elm Architecture makes it easy to implement this functional approach for games, which modularizes the architecture into model, update, view, and subscriptions. In this context, a game of Pong can be decomposed and handled in each of these four sections:

- The subscription section will listen for keyboard events from the user, dictating where the player's paddle will move
- The update section will produce the next frame of the game from the user input by producing new objects of the model
- The view section will represent these changes visually based on the new model objects
- The state of model will remain constant; no mutability will occur inside of the objects from one frame to the next

The Elm Architecture lends itself nicely to game design, since the architecture's constraints enforce this type of modular and stateless development [7], which makes it easier to program and extend the game. Instead of worrying about whether refactoring code will break something elsewhere, the modularity of the Elm Architecture allows programmers to makes changes without affecting the other aspects of the game.

5.2. User Interface

User input is necessary for any interactive game. In Elm, input is managed through subscriptions, which listen for external input from keyboard events and mouse movements. The Elm Architecture pattern allows this, with a cycle of presenting an initial model, the user issuing an input message through the view, updating the model based on those messages, and presenting the updated model back to the users [8]. This architecture allows for easy extension of user input, since assuming the developers have included `KeyUp` and `KeyDown` in the union type for `Msg`, new keys can be added by key codes and translated into a union type named `Key`. For example:

```
fromCode : Int -> Key
fromCode keyCode =
 case keyCode of
      32 ->
            Space
      37 ->
            ArrowLeft
      _ ->
            Unknown
```

This code could be extended to take on more key codes and added to `Key`'s union type without touching any other parts of the code.

6. CONCLUSION

Our report has outlined the unique and standout features of Elm that make it suitable for web application development, within the context of creating a side-scrolling platform game. By judging its structure, performance, error handling, and suitability, this report has demonstrated the value of Elm in comparison to similar languages. Most importantly, it has shown how specific language features and structure can have an impact on the development of a web application.

REFERENCES

[1] Czaplicki, Evan. "Introduction." An Introduction to Elm, guide.elm-lang.org/.

[2] Czaplicki, Evan. Elm, Evan Czaplicki, 2012, elm-lang.org/.

[3] Rolo, Pedro. "Elm = Javascript Reinvented." https://www.imaginarycloud.com/blog/elm-javascript-reinvented-1-overview/

[4] Waselnuk, Adam. "Understanding the Elm Type System." Adam Waselnuk - Front End Web Developer, Adam Waselnuk, 27 May 2016, www.adamwaselnuk.com/elm/2016/05/27/understanding-the-elm-type-system.html.

[5] Czaplicki, Evan. "Blazing Fast HTML: Virtual DOM in Elm." http://elm-lang.org/blog/blazing-fast-html

[6] Czaplicki, Evan. "Blazing Fast HTML: Round Two." http://elm-lang.org/blog/blazing-fast-html-round-two

[7] Sampellegrin, Marco. "In depth overview of Elm and Purescript. Lessons learned porting a game from Purescript to Elm" https://alpacaaa.net/blog/post/elm-purescript-in-depth-overview/

[8] Poudel, Pawan. Beginning Elm. elmprogramming.com/.a