The University of Alabama in Huntsville

The Elm Programming Language:

An Analysis of the Objectives and Reasoning behind Elm

Cameron Howell

CS 524 Spring 2020

Dr. Weisskopf

18 March 2020

**Introduction**

When most programmers think of web development, the first languages that come to mind are perhaps JavaScript, HTML, or CSS. While not entirely common, Elm fits right into the list of web development languages; although it is not as popular as other front-end languages, Elm is arguably one of the more efficient and approachable languages of the lot. The importance of Elm in front-end web development is hard to understate, though that is to be discussed in later sections.

*Design and History.* Elm was developed by Harvard alumnus Evan Czaplicki in 2012. Like the beginning of most programming languages, Elm was created to address some void that existing languages do not fill. As described in Harvard’s article, Czaplicki was working with front-end web development using existing languages and came across some annoyances that just kept building up. He eventually decided that these annoyances were too much of a burden, so he resorted to creating his own functional programming language to correct these issues. Czaplicki invented Elm as a senior thesis, and he drew upon many language ideas synthesized by previous computer scientists that just never made the light of day [1].

*Basic Features.* Czaplicki designed Elm as a pure functional programming language, meaning that it is a language which only implements the ability to use the functional paradigms [3]. When compiled, Elm is translated into JavaScript; this is useful because most web browsers only run the JavaScript language. (This is different from the HTML and CSS that serve as the code for webpages. Web browsers only execute JavaScript code embedded into the HTML of a webpage). According to the Elm Language Guide, Elm was designed as a functional programming language because of their inherent ability to be flexible in terms of errors and semantic versioning [2]. Czaplicki designed Elm’s ideology to be divisible into three concepts that comprise a central idea known as “The Elm Architecture.” The three parts – model, view, and update – are crucial to understanding how Elm code works; this is of the utmost importance to software and front-end developers to be able to easily write efficient code. In the same manner that object-oriented programming is central to languages like Java, this concept of “The Elm Architecture” serves as the true definition of what Elm is. A more detailed explanation of this idea will be provided in later sections. One last matter of importance is that Elm is a statically typed language, but almost all types are inferred during the compilation process. It is not necessary to explicitly type variables in Elm, although it is subjectively constructive in practice.

*Domain of Use*. It should be clear that Elm is solely for web development; however, this does not limit its functionality. One of the biggest uses of Elm is for creating web apps or games. While JavaScript is produced by default with the Elm compiler, it can be commanded to output HTML using modules such as elm-html. This is intuitive for several reasons, including the fact that Elm can be compiled into JavaScript and called into another HTML file. Embedding Elm into a webpage only requires a few lines of HTML, so this is especially useful for web developers trying to put their web app or game on their webpage [6]. Since Elm can be manipulated to be compiled into HTML, it is practical to use this language to create a webpage. Given Elm’s ease of use, some programmers may find that creating a webpage with Elm and then compiling it into HTML is much more convenient. This was Czaplicki’s intent in creating this language, though: he sought out to create a language that fixed a lot of what he deemed “historical mistakes” in the realm of modern web development languages [1].

**Characteristics**

Some of Elm’s most intriguing features were just brushed upon in the introduction, but the beauty of the Elm language can be found in breaking it down into its base components. This portion will list some of the features of Elm that are most prevalent and compare them to other languages with similar implementations. Examples of these features are demonstrated in the Elm Language Guide.

*Values*. Every language has some set of primitives. Linguists would define a primitive as “a word, base, or root from which another is historically derived” [5]. This is true for Elm as well, and its primitive is the **value**. Most programming languages have values, and all values are categorized into types. Java’s types, such as integers, floats, doubles, strings, etc., are all labels that describe what values exist in that set. Integers contain numbers, and strings contain alphanumeric characters. In the same manner, Elm’s values include numbers, strings, and Boolean values like true or false. Since values are primitives, they can be used as building blocks for creating larger constructs [2].

*Functions*. One such construct would be **functions**. Similar to functions in C-based languages and methods in Java, Elm’s functions accept a value as a parameter, perform some series of operations to change it, and then produce some result based on the value passed to it. A simple example function in Elm would be something like this:



Now, calling this function with a string as a parameter would produce an output like this:

As stated previously, Elm is a functional programming language, meaning that it is perfect for the functional paradigm like the example shown [3]. Elm takes simple commands and functions and produces a direct output instead of causing other issues to juggle like with Java’s methods. Methods require return types and objects to be contained in, but Elm is streamlined and allows simple function execution. All values passed must be primitive, but values can be passed that have had operations performed on them during the function call, like adding two integers or concatenating two strings. As long as the parameter passed is a primitive, Elm’s functions will work [2].

*Records.* A **record** in Elm is comparable to an object in Java and a struct in C++. Like these data structures, records can contain multiple types, and each value has some identifier. Each line in a record is known as a **field**, and, similar to the dot operator in Java, a field’s value can be accessed like so: .name record. If a programmer needed to access the value of the field “salary” in the record “employee,” the function to get this value would simply be .salary employee. Records are useful for creating object-like instances that contain many attributes that can be accessed and updated at will. A field in a record can be updatedby performing an **update** function on a field. If the value for “salary” in the “employee” record needed to be changed to 500, then the function would be { employee | salary = 500} [2].

*If-Else Statements*. Virtually every modern language has some equivalent to the classic **if-else statement**. Elm’s if-else statement is structured like *if <condition> then <expression> else <expression>*. What’s interesting about Elm’s if-else statements is that, unlike other languages, the else statement is required. In a language like Java, the if-statement can stand alone as just a conditional for executing an expression, but for Elm, the else condition is required in order to handle the case in which the if condition proves to be false. When executing Elm in a terminal, an error will be thrown when trying to create an if-statement with no else-statement [2].

*Lists.* Lists are analogous to Lua’s lists or Java’s arrays. A **list** is a data structure that holds values of the same type. Elm will infer the types of the values at compile time to enforce the list’s typing. Attempting to create a list with multiple types will throw a type mismatch error to alert the programmer of the issue. Lists come with a host of built-in operations that can be performed on them, such as getting the length or reversing the order of the values [2].

*Type Inference and Aliasing.* One of Czaplicki’s major design features of Elm is the ability for the compiler to infer types. This has been and will be mentioned often because of its usefulness, and it also affects aspects of the language’s writability and reliability. Elm will never throw a runtime error because all errors in type are discovered through the compilation process. When analyzing the code, the compiler will trace all values as they are passed to functions, manipulated, etc., and *infer* each value’s type. Whenever a value is used in a manner contrary to the inferred type, the compiler will throw an error and show the programmer where in the code the type mismatch occurs. The Elm Language Guide provides the following as an example of the compiler inferring type and finding a type mismatch error:

toFullName person =

person.firstName ++ " " ++ person.lastName

fullName =

toFullName { fistName = "Hermann", lastName = "Hesse" }[11]

Notice that the first name field in the person record is correctly spelled as “firstName”, but the parameter in the function call is misspelled as “fistName.” The compiler flags this as a type mismatch error, and it even has the granularity to suggest to the developer that the error is the result of a simple typo; the error is still considered a type mismatch because, technically, the type passed to the function did not match the inferred type of the field in the record.

It is important to reiterate the point that Elm is statically typed and that types are inferred by the compiler; however, *type aliases* or type annotations can be integrated into a program to assist the compiler in asserting a type during the inference stage. A *type alias* is a shorthand for type annotations, which are single lines of code written before a function definition. The type annotation is essentially “declaring” a type for a value, but the compiler will still infer the type and compare it to the type “declared.” This is useful for extending the specificity of error messages while also providing a method for documentation other than comments. A type alias is a more compact method of annotating, and it is convenient in that annotations require writing out the type each time the value is used whereas aliases provide a one-time type declaration for later references.

The benefit to type inferencing and aliasing/annotating is that programmers don’t have to wait until executing their code to discover something amiss; the compiler will catch mostly all errors (especially type mismatches) before runtime, so there is a significantly minute chance that a runtime error would slip past the type inferencing [2].

*Error Handling.* Czaplicki realized that not all errors will be related to type mismatching, and some errors may still make it into runtime; however, his counter to this was Elm’s intuitive **error** **handling**. According to the Elm Language Guide, all errors that occur during runtime are never shown as errors to the user, i.e., the errors do not halt runtime. All errors are typed as a form of data, so any programming mistake that slips past the compiler will show through the data output rather than the compiler’s feedback terminal. The programmer can create custom data types so that, if an error should occur during runtime, the custom data type would be used instead, and the error can be discovered by observing the output data instead of the program exiting. This enforces Czaplicki’s “guarantee” that runtime errors will never occur with Elm [2]. This concept is simple and convenient, as shown with this example provided in the Elm Language Guide:

type MaybeAge

= Age Int

| InvalidInput

toAge : String -> MaybeAge

toAge userInput =

...

The function toAge takes a String as an input and outputs the custom type MaybeAge (side note: MaybeAge is considered a *Union* type because it is a *union* of different types. Union types will be described in the *Writability* section because of their relevance to that topic). Notice that the type definition for MaybeAge has two possible outputs: Age Int (which would output Age and then the integer input by the user) **or** InvalidInput. So, if the user input 21 for the age, the value printed to the screen would read “Age 21,” but if the user input something like “twenty-one” (which, even though it is of type String, is still erroneous because it cannot be converted into an Int), then the output would read “InvalidInput.” This is what is meant when Elm’s error handling outputs errors as data: the developer should create custom data types in anticipation of any erroneous values input by a user so that the program will still compile. The error shows only in the output, but the program still runs as directed. It is essentially like Java’s exceptions where the developer can define exception handlers to allow the program to execute while pushing error messages to the output.

*Purpose of Elm.* While these features roughly define Elm as a programming language, it is still necessary to delve further into the smallest units that make Elm work. The general purpose for Elm was given in the introduction, but what specifically was Czaplicki attempting to correct with his senior thesis project? The answer lies in Czaplicki’s thesis: Elm’s two major features solve the issue that he found with graphical user interfaces (GUIs). In Czaplicki’s abstract, he states that Elm would “simplify the complicated task of creating responsive and usable graphical user interfaces” [8]. When interviewing for Harvard’s Alumni Profile article series, Czaplicki remarked that he constantly ran into problems such as “trying to center an image in a box or reus[ing] visual elements on multiple web pages” [1]. Another problem he found was that JavaScript had a “notoriously small standard library,”[8] but Elm would remedy all these development woes of the web language standards of the time.

Elm’s status as a functional language plays a major part in Czaplicki’s quest to solve his perceived issues. While Elm is a pure functional programming language, Czaplicki refers to it as a “concurrent FRP language” in his thesis. FRP is a type of GUI development approach known as Functional Reactive Programming [8]. He aimed for Elm to be a language that streamlined graphical development so that creating complex visual structures could be simplified and consistent. Perhaps the best example that Czaplicki gives of this simplification in action is his demonstration of the mouse tracking program. He shows that a simple GUI can be created using just a single line of code to track the position of a mouse cursor over the GUI’s area:

Czaplicki’s Elm accomplished a task in a single line that may have required a much greater length of code in another language [8, F.1].

*General Syntax*. It should be noted that Elm is, by design, an event-driven language, and with all event-driven languages, there are units known as signals. A signal is another name for any type of mutable – or time-varying – value in a language. Consequently, the syntax for Elm’s core language reflects the concepts of events. Czaplicki’s syntax in his thesis paper is much more involved and higher-level than the syntax described in CS 524, but this is what he describes as thecore syntax of the Elm language:

e ::= () | n | λx. e | e1e2 | x | let x =e1 in e | i | liftn e e1 ... en | foldp e1 e2 e3 | async e

n ∈ Z, x ∈ Var, i ∈ Input [8, p.14].

This is complex notation, but this syntax can be broken down and expressed in terms of CS 524-level detail. Each unit separated by vertical bars represents a “reactive primitive” for Elm. Every primitive represents a building block unit that can be combined with other units to create a much more complex structure. For instance, the “n” primitive is described as being in the set of integers, so this means that one of the reactive primitives in Elm is that of integer values, or numbers. The primitive “x” is in the set of all possible variables, meaning that variable names are a primitive; in like manner, the primitive “i” is in the set of all possible identifiers. These identifiers are representative of input signals for events. The primitives “liftn” and “foldp” allow Elm to transform or combine signals to or from events. Some primitives are for the compiler’s use only, such as the “async” primitive. This indicates to the compiler that two or more manipulations of signals can be concurrent so that events do not have to take place at the exact same time as other events elsewhere [8].

The greatest thing about this syntax is that these primitives aren’t just for abstract notation: the programmer can directly use these primitives in Elm code to manipulate the flow of compilation at the signal level, i.e., the primitive “lift” exists as a functioning unit in the Elm language. Refer to the mouse cursor code and notice that the word “lift” exists as a command. The entirety of the Elm syntax exists in the notation shown above and studying and understanding the complex notation involved will give the programmer a greater grasp on how to abstractly derive functional programs in Elm [8].

*Type Features.* The descriptions of the type features in relation to the syntax is analyzed in-depth on page 17 of the thesis, but the main focus here is on the actual rules for typing in Elm. Czaplicki again uses a different notation, but this time, his descriptions are much clearer:

τ ::= unit | number | τ → τ′

σ ::= τ signal | τ → σ | σ → σ′

η ::= τ | σ

It should be noted that τ represents primitive type, σ represents signal type, and η represents either. The primitive type, as the name implies, is a typing for Elm’s primitives. The signal type is a typing for Elm’s signals, and Czaplicki notes that signal types should not be used for signals, despite the name. The reasoning, he says, is because allowing signals to be of the signal type would open up the possibility for a program to alter the runtime system while the program is executing [8].

To simplify all this notation and complex jargon, primitive types are traditional types such as integers, strings, etc., whereas signal types denote the type of dataflow occurring with synchronous events. Having said this, it is also important to note that Elm is a strongly typed – or statically typed – language that just so happens to also implement type inference. All types decided by type inference by the compiler are one of the two types described above. Using the type definitions, Czaplicki then derives the rules behind inferring or deciding a primitive or signal type, respectively, in Figure 2 on page 17 of his thesis [8]. The rules derived there are much more complex lambda calculus, and as such, it is not necessary to decipher because of the scope of this analysis paper.

*Concurrency in Elm.* Since Elm uses a method of FRP known as Concurrent FRP, Elm is considered *concurrent* at runtime; consequently, Elm can perform two operations of major concern in modern computer science: parallelism and pipelining. Parallelism ensures that Elm can have multiple processes running at the same time to increase efficiency and latency, and pipelining allows for simultaneous updates for a single computation. Both concepts improve Elm’s efficiency and throughput, both of which are important for running complex programs that would normally require a lot of time and resources. With the advent of parallel computing using multiple processors, cores, and/or threads simultaneously, Elm’s concurrency ensures that programs can be optimized to fit this hardware paradigm for quicker runtimes and more equal distribution of resources.

*Semantic Versioning.* Developing with Elm is made even simpler because of a concept known as Enforced Semantic Versioning. When compiling code, the type system will check for any changes in package API versions. This is beneficial for a few reasons: first, programmers are constantly informed of changes in API versions so that they can maintain consistency with the current standards; second, package developers are required to follow the semantic versioning rules to ensure that all packages are up to standard; and third, deprecated API versions are less likely to be used because of Elm’s clarity in prompting the programmer to evaluate the versioning discrepancies [9].

**Evaluation**

Now that an extensive description of Elm has been proposed, Elm as a programming language must be analyzed in terms of how these features affect readability, writability, and reliability.

*Readability.* Czaplicki intended for Elm to be simple, and because of his effort to do so, the language is high-level and easy to understand if someone unfamiliar with the language were to attempt to understand some of the code. Also, as shown with the mouse pointer example, a normally complex operation can be performed in a single line, so Elm is simple in the fact that its functionality is extensive in comparison to the relatively simple syntax used to produce those functions. With the consolidated syntax in mind along with the language’s constructs listed previously, i.e., lists, records, etc., it is a fair assumption to declare that Elm possesses a high level of orthogonality in proportion to its simplicity. Elm can do a lot with a little, to say the least, and this is indicative of the overall simplicity and orthogonality that Czaplicki was aiming for in his implementation of Elm.

Counting each subtype of data type in Elm, there are six total data types native to the language: Int, Float, number, String, Char, and Bool. Elm also supports custom data types, but this doesn’t mitigate Elm’s overall readability. Elm is strongly typed, and its type checking mechanism is one of its defining features; subsequently, it is a reasonable assessment to say that Elm’s data types do not impact its readability.

Elm has 14 keywords, and while having more reserved words can sometimes help readability (as in Java, which has 53 reserved words), Elm’s reserved words are concise, and the words reflect exactly what their purpose is [13]. Although the keywords are simple and easy to understand, there are not enough to really enhance the overall readability, because, in the case of loops and if-statements, the lack of keywords to denote the end of these blocks of code means that there must be a heavy reliance on indentation to show where code blocks begin and end.

When analyzing Elm’s readability, the word “simple” is heavily used, but not in vain: Czaplicki’s objective in developing his new language was founded on the principle of cutting out the clutter and functional issues that he found in his daily use of other languages. Elm is designed to have compact, well-formatted code that communicates its purpose and function. Consider a function that greets you in Java and in Elm, respectively:

//Java:

public static void hello(String name){

System.out.println(“Hello “ + name);

}

--Elm:

hello name=

"hello “ ++ name

Excluding the main driver functions and calls, Elm’s implementation is concise, compact, and convenient compared to Java (or, to sum Elm up in a word: simple!). With all these aspects of readability considered, Elm has a fairly high level of readability.

*Writability*. To preface, Czaplicki’s entire goal was to create a language that consolidated a lot of the verboseness of existing languages to allow for complexity with simple commands. It would then be expected that Elm has a high index of writability. Considering the explanation for the use of the word “simple” in the readability analysis, Elm’s simplicity clearly helps with writability. Consider the Java versus Elm example provided in the previous subsection: Elm performs the same task in far fewer lexemes. While neither take longer than a few seconds to type out, scale this up a few orders of magnitude, and suddenly that difference in writing time is much more valuable. Though this one example may not prove the point alone, in general, Elm programs are measurably more compact when compared to those in other languages.

Introduced in the *Type Inference and Aliasing* subsection, Union types are critical to Elm’s expressivity. After conducting an online search of the general consensus on Elm’s level of abstraction and expressivity, the conclusion was that public opinion varies, but all sides agree that Elm’s Union types contribute to the language’s expressivity. Union types are the same as the custom types that are effective in handling errors. They are essentially logical or statements, meaning that if the value assigned is not of the first type, then it is of the second, or third, and so on. Union types provide a large range of custom types, meaning that developers can perform abstraction and create their own data types whether they be primitive or complex. The very definition of “expressive” in terms of computer science means that a language is capable of defining new data types that are not “hard-coded” into the language: by definition, Elm does just that, meaning it is expressive.

*Reliability*. Section 1.3.1 of the textbook defines reliability as a collection of all the previous metrics plus type checking, exception handling, and restricted aliasing. Elm has type inference with a broad yet powerful ability to detect type mismatch errors at compile time while also having the ability to produce custom error messages in the code output to prevent runtime errors. The “restricted aliasing” metric should not be confused with Elm’s type alias feature, which means that types can be declared, inferred, and compared rather than just being inferred by the compiler.

With all previous metrics considered, combined with the final three in deciding reliability, Elm generally performs well over readability, writability, and reliability. Elm is not perfect, but Czaplicki’s goals have been realized in the production of Elm in making a simple, compact, easy-to-use web development language.

*Cutsem Analysis*. The final analysis of Elm is regarding Cutsem’s paper “Why Programming Languages?” and the four reasons listed therein. According to Cutsem, Elm’s existence is explained by at least one of the four reasons he listed in his paper. So, which of these does Elm satisfy?

Cutsem states that functional languages are at the forefront of thought shapers. Elm is a pure functional programming language, meaning that it is of a different ideology than imperative languages. Something that is quite interesting about Elm being a functional programming language is that it compiles into JavaScript, which is not typically considered a pure functional programming language. By creating a language that compiles into another language of a different type, Czaplicki surely sought to change the consensus on web development. Czaplicki created Elm to change the way modern languages handled web development and design, so Elm satisfies Cutsem’s criteria for being a though shaper.

When discussing simplifier languages, Cutsem uses the phrase “less is more.” Elm is enrolled in this school of thought, too, as so much can be said with Elm in few words. The level of abstraction available in so few lines of code is tremendous, and certainly convenient for writing more complex programs. The word “simple” has perhaps been used almost 30 times in this paper alone to describe Elm, and that is done to stress ad nauseum the idea that Elm was designed for simplicity. It goes without saying, then, that Elm fits Cutsem’s description of a “simplifier.”

Not much can be said concerning law enforcement and syntactic abstraction to justify conclusively labelling Elm as such, though Elm is not excluded from these categories. Elm’s level of abstraction is on par with languages like Python (which is used as an example of a language with syntactic abstraction), and Elm reuses some traditional primitives and constructs found in other languages, but in terms of truly shaking up these realms of thought, Elm doesn’t quite make it. This isn’t a bad thing, however, as Elm need only satisfy at least one of Cutsem’s criteria to legitimize its existence [10].

**Conclusion**

With the objective analysis complete, I have my own subjective commentary on Elm as a language. Until this project was assigned, I had yet to work with – much less, even hear of – Elm. After doing this extensive amount of research and analysis of it in accordance with the standards discussed in class and in Tom Cutsem’s paper, I have to say that I’d love to start working with Elm more extensively. It has such a high level of expressivity, and the fact that it compiles into JavaScript and can be so readily embedded into a webpage is intriguing. I have worked with HTML and JavaScript in the past, and after studying all these features of Elm, it seems insane that Elm isn’t a standard language for front-end web development. I understand that Elm just has yet to have the traction necessary for pushing it into the mainstream, but with its ease of use, it is astonishing that it is still just an emerging language.

I find it particularly interesting that Elm is strongly typed but also implements type inferencing. This seems oxymoronic, but its brilliance reflects the intuition of its creator in creating a compiler-based typing system that guarantees absolutely no runtime errors. The levels of customization in typing is intriguing, also, with the implementation of Union types, type annotations, and type aliases. Czaplicki clearly emphasized the role of the compiler’s type checking in creating Elm, and this directly prevents runtime errors in Elm because the program will not compile if any type mismatches are detected through either type inferencing or comparing the inferred type to the annotated/aliased type. This idea seems so elementary, yet Czaplicki revolutionized type checking with Elm’s extensive type system.

With all the free time that I have now with the quarantine in place, Elm would be a fun language to study considering I have not ever worked with it until now. Analyzing Elm has also helped me understand the granular details and idiosyncrasies of other languages I am more familiar with, such as C++ or Java. Aside from that, Elm is quite a beautifully formed language, and I will take due diligence in learning how to program with it.

References

[1] Alumni profile: Evan Czaplicki, A.B. ’12, https://www.seas.harvard.edu/news/2015/10/alumni-profile-evan-

czaplicki-ab-12, Feb. 23, 2020

[2] An Introduction to Elm, https://guide.elm-lang.org/, accessed Feb. 2, 2020.

[3] Functional Programming Paradigm, https://www.geeksforgeeks.org/functional-programming-paradigm/,

accessed Feb. 2, 2020.

[4] Getting Started with the Elm Programming Language, https://www.toptal.com/front-end/getting-started-elm-

language, accessed Feb. 2, 2020.

[5] Lexico, https://www.lexico.com/en/definition/primitive, accessed Feb. 28, 2020.

[6] What is Elm? Essentials of the Elm Language, https://pragmaticstudio.com/tutorials/what-is-elm, accessed Feb.

25, 2020.

[7] Why Elm?, https://elmprogramming.com/why-elm.html, accessed Feb. 23, 2020.

[8] Elm: Concurrent FRP for Functional GUIs, https://elm-lang.org/assets/papers/concurrent-frp.pdf, accessed Mar.

2, 2020

[9] Elm: Features , https://elm-lang.org/, accessed March 7, 2020.

[10] Why Programming Languages?. http://soft.vub.ac.be/~tvcutsem/whypls.html, accessed March 14, 2020.

[11] Types, https://guide.elm-lang.org/types/, accessed March 30, 2020.

[12] Making Elm code more compact and more readable, https://medium.com/elm-shorts/making-elm-code-more-

compact-and-more-readable-935067f81829, accessed March 30, 2020.

[13] Elm Syntax, https://elm-lang.org/docs/syntax, accessed April 2, 2020.