Background Report: Functional Reactive Programming with Elm

I. Alcuaz (14633127), C. Balladares (53443123), E. Feng (32827131),
J. Parkes (37890126), and S. Wang (57023146)

Department of Computer Sc.,
University of British Columbia,
2329 West Mall, Vancouver, BC,
Canada

(Dated: November 16, 2015)

Functional reactive programming (FRP) is a programming paradigm which commonly sees usage in highly reactive systems, such as graphical user interfaces (GUIs), robotics, and music synthesis. FRP combines the principles of declarative/functional programming, of which are predictability and immutable states, with those of reactive programming - flexibility around data flow. GUIs, which serve to simplify our interactions with a computer, are inherently dynamic; this is just due to the fact that the underlying data is constantly changing due to user input. FRP is a unique approach which contrasts to more popular techniques, such as coding in traditional JavaScript and JQuery languages, by explicitly modelling time in the context of data flow (e.g. a mouse that moves across a browser). We opted to implement an HTML 2D-platformer game for our project as this would be great example of states which are constantly in flux, and as such, we will be able to concretely highlight the advantages/disadvantages of using an FRP based language such as Elm as opposed to the other, more popular languages which currently dominate the web development industry. In this report we will specifically highlight Elm's place in the FRP landscape as well as underline the motivation behind our project choice.

CONTENTS

I. INTRODUCTION

Of the examples we listed in the abstract concerning FRP use, we will focus this report and our project on the first one on the list, namely, GUIs. GUIs constantly transmute as a response to a constantly changing user input. For example, a mouse that moves across the screen will need to be handled by the program by constantly recording its new position, and then display feedback to the user. Traditional non FRP based languages such as JavaScript would need to specify exactly how to handle these events by defining handlers accordingly, but with FRP, the focus shifts on what should happen (i.e. how would the GUI change in response) as opposed to how, the latter of which is left to the compiler (Czaplicki, 2012). Elm is one such example of a programming language which integrates the principles of FRP to try to simplify the GUI implementation process.

II. BACKGROUND INFORMATION

A. Functional reactive programming (FRP)

FRP is a programming paradigm which combines the principles of functional/declarative design with reactive programming. The purpose is such that the implementation of systems which demand high reactivity are simplified. This is achieved due to the predictable nature of functional programming; states are immutable and functions are expected to return the same basic type of result for each evaluation. Reactive programming on the other hand is centred around the concept of asynchronous data flow, which can be represented in terms of variables, data structures, user inputs, etc. Consider the equation below:

$$\begin{aligned} a &= 1 \\ b &= 1 \\ sum &= a + b \\ a &= 11 \end{aligned} \tag{1}$$

The key idea behind reactive programming is such that the dependency on "sum" on Eq. (1) above is tied to the values of a and b for all consecutive time, making it reactive, so to speak. In traditional imperative languages, the variable "sum" in Eq. (1) would equate to 2, and not 12; from a reactive standpoint, the result should be 12, since variables are tied to some input data stream for all time. Therefore, when functional and reactive programming are combined, the characteristics which define both are meshed together - hence the term FRP. As such, a stream of events (see Fig. 1) are considered immutable, and functions which act on such a stream do not modify the data, but instead make an entirely new stream to manipulate (Staltz, 2015). This is in direct contrast to how data is handled from traditional imperative approaches, where it is allowed to change.

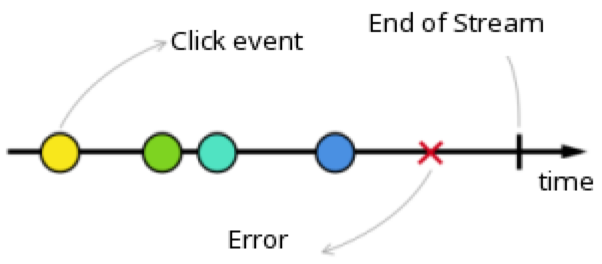


FIG. 1 In FRP, events are immutable (represented here as circles) and are treated as a function of time. (Staltz, 2015)

B. Discrete versus continuous semantics

There are two flavours of FRP: one which views data flow in terms of discrete events, and another as continuously propagating change throughout the system. Elm and other event driven FRP implementations require the former; namely, that updates due to an external input are modelled in terms of discrete events (n.d., 2015b). Values which can vary over time are termed "signals", and from a discrete semantics standpoint, events (such as user input) would be combined into signals to give it an initial value, but of which can change in a discrete manner.

C. Existing FRP frameworks

There are several examples of FRP based languages, one of which is already mentioned - Elm - which compiles to HTML, CSS, and JavaScript for the sole purpose of making web based GUIs. Another is Flapjax, and like Elm, it is an event-driven FRP implementation, except it compiles only to JavaScript. This is not an exhaustive list and there are many other FRP based languages on the web, but they all share the goal of simplifying the implementation process for fundamentally reactive systems.

D. FRP versus other paradigms

Altogether, FRP makes for a very robust and unique approach to the domain associated with intrinsically reactive systems. Due to the nature of FRP, many of the hurdles in this problem area are simplified, such as the debugging process; variables are immutable, and the states of a program are modelled as a function of time, allowing FRP languages such as Elm to boast unusual but useful features such as a "time travelling debugger" (Czaplicki, 2012). However, FRP based systems lack the flexibility which imperative based languages such as Java and Ruby offer. With imperative languages one has complete control over how the program behaves, and the programmer can implement details at a much lower level than what functional programming offers. This is just due to the nature of FRP based languages; abstraction is emphasized so that the developer can focus on coding the behaviour of the program, instead of worrying about the underlying details. Also consider the fact that there exists much more libraries/solutions to common implementation problems in web development in imperative languages, of which are highly optimized. In short, anything which an FRP language can do, these imperative languages can do much faster and with more control, which is one of the key reasons they dominate the HTML landscape. FRP based languages instead offer reliability and predictability, which leads to easier maintenance and an arguably smoother development process, but at the cost of speed and versatility.

III. THE ELM PROGRAMMING LANGUAGE

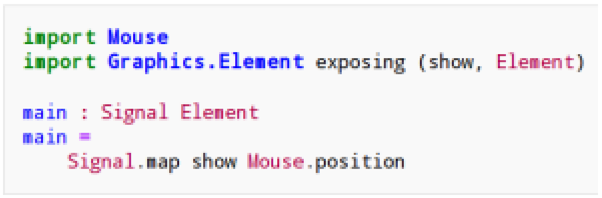
A. What is Elm?

Elm is a relatively new programming language that is based around FRP, and was built in 2012 with the sole purpose of making HTML rendered GUIs. Elm attempts to streamline the HTML GUI implementation process by addressing the shortcomings of traditional imperative programming techniques. An example of such problem involves the complexity with debugging a program in Ruby, where functions are allowed to transform one another and variables are mutable. With a functional language like Elm, the debugging process is simplified since variables are immutable and function outputs are well predicted. Another example are segmentation faults in a C program; since the program is allowed to change states, the problem could've arisen from any of the other previous states leading up to the runtime error. Since

Elm is specialized towards browser applications, its compiler renders code as HTML, JavaScript, and CSS, which provides an abstraction for the developer, as the tedious syntax associated with these languages are streamlined.

B. Typing and abstractions

Elm is a type-safe, inferred, and statically typed language. As such, it is up to the developer to specify a primitive type (e.g. strings, integers) or make their own data type (the syntax involved is extremely similar to PLAI's define-type/type-case), or let the compiler handle it for them. The main abstraction that Elm features are "Signals", which are essentially variables, only that they are allowed to change over time (unlike a variable with a primitive type). For example in Fig. 2, the "main" variable is automatically updated based on the position of the mouse, and "Signal.map" calls the map function in the Signal library to map the current position of the mouse to the "main" variable. The mouse position is retrieved through the function call "Mouse.position" and the "show" command then applies the mapping to convert it to a Document Object Model (DOM) element for the user to see.



```
import Mouse
import Graphics.Element exposing (show, Element)

main : Signal Element
main =
    Signal.map show Mouse.position
```

FIG. 2 The Signal library provides developers with the ability to handle data flow, while avoiding having to resort to mutable memory. (n.d., 2015a)

C. Implementation

The Elm compiler was written using the Haskell programming language. This choice was due to the fact that Haskell "provides many useful libraries that make [it] a good choice. Many parsing libraries exist, making it relatively pleasant to create a lexer and parser" (Czaplicki, 2012). The compiler then generates HTML, CSS, and JavaScript, which handles events that change the state of the program.

D. Benefits

In addition to the advantages that FRP possesses, of which Elm and other FRP languages share, Elm is also able to boast many benefits over other FRP-based languages. One such advantage is easy setup times: Elm was built with the priority of making the implementation process for HTML based apps easier by minimizing the difficulties involved with creating apps from scratch. At its first release, it utilized an easy to use in-browser editor, and as of late features a StartApp package that allows developers to almost completely eliminate setup time by providing an additional layer of abstraction, similar to what the industry standards Ruby on Rails and Django provide, except from a functional programming perspective. Another advantage would be very fast render times: Elm has a steadily growing collection of third-party libraries due to its rising popularity, many of which improve rendering speed. Elm also features a virtual document object model which allows developers to use HTML, CSS, and JavaScript directly. As such, implementations in Elm are able to emphasize code abstraction, modularization, and reusability (Czaplicki, 2014), all of which aid in the surprisingly difficult but supposedly simple tasks involved with web development, by providing a high level abstraction for low level API's. Developers also need not worry about browser incompatibilities provided the browser being used supports the latest web standards. Lastly, Elm's compiler catches mostly everything at compile time, and as such, there are for the most part no runtime exceptions (except for beginner cases, such as mismatching the head and tail of a list). What this means is that runtime exceptions must be invoked explicitly through Elm's crash debug function.

E. Shortcomings

Elm shares the same disadvantages as FRP and all other FRP-based languages, which stems from the lack of flexibility that imperative based languages offer. This is especially important for the developer who wants to fine tune the little things in their program. Elm focuses on ease of use and relies on external libraries, which effectively provides an additional layer of abstraction for the developer, since the focus is on what the program should do and not how to do it. Furthermore since the Elm compiler relies heavily on JavaScript implementations, tail-call recursion, which is so prevalent in the functional programming world, is non optimized since JavaScript currently provides no support for it. This JavaScript reliance comes at another cost: support for concurrent programs are extremely limited, and this is again due to the fact that JavaScript provides limited support for the functionality. Current implementations of JavaScript provide a library to implement workers in a program, however these workers are not able to pass functions around, only defined primitive types such as strings and arrays. Elm also has no support for common functional programming-esque functions that you would typically see in Scheme for example, such as map, foldl, filter, etc. These are just some of the current disadvantages associated with Elm, but its rising popularity may contribute to more robust solutions to the outlined problems.

IV. PROJECT TOPIC: AN HTML 2D PLATFORMER WITH ELM

We decided to implement an HTML 2D platformer game to showcase the advantages and disadvantages of FRP based systems in the context of Elm. Since the topic choice itself runs into many of the challenges associated with GUI behaviour, such as an unpredictable and reactive domain, we feel it is a great example to compare and contrast the benefits/disadvantages of FRP based GUI design versus imperative based implementations.

A. Motivation for project topic

This project topic allows us to experience first hand the numerous advantages as well as disadvantages of a functional programming based approach in the context of FRP for an HTML application. Having taken CPSC 310 before, we all have some experience already with implementing web based applications, except from an imperative programming standpoint. As such, this would be a valuable opportunity to be able to see things at a different perspective, and allow us to appreciate both ends of the spectrum. This project also allows us to learn a practical and interesting language, namely Elm, as well as sharpen our functional programming skills. Functional programming is especially interesting as it forces you to think in a different way, such as by forcing you to apply recursion instead of iteration, and treating variables and functions as immutable. As such, the exposure we receive from functional programming is quite valuable, especially since only a limited amount of courses provide the opportunity to work with such languages. We are also provided with the opportunity to practice our research and writing skills through these end-of-term assignments and reports, which of course is extremely valuable, considering that some of us are planning to progress further into graduate studies.

B. Potential 100% project

The minimal, low risk milestone at the 80% level our project hopes to achieve is the implementation of a very basic version of the platformer game discussed at length in the beginning sections of this report. This implementation would highlight the key benefits concerning FRP, its shortcomings, as well as its context in the place of other HTML based systems and how our implementation differs from such approaches. However, the game itself would be extremely simple, namely moving from point A to point B with some obstacles to introduce a layer of complexity, with minimal additional effects. In other words, it wouldn't be a complete (pretty) application, but the focus instead would be on highlighting FRP and Elm as discussed earlier. A full 100% project would then go beyond this scope, and as such we would introduce more dynamic and complex variables into the game, as well as a significant amount of artwork, which of course is not a focus of the project, but would overall result in a more polished and presentable application. This would introduce new issues such as concurrency into the system, but would overall result in a much more complete application than at the 80% level, which is ultimately what we hope to achieve.

V. CONCLUSIONS

FRP is a unique approach which is primarily used in the realm of GUI implementation, robotics, and music synthesis. It is a particularly interesting paradigm in that it combines functional programming with the concepts associated with reactive programming, of which is centred around how to handle constant data flow through a system. Together, the two combine to simplify the process involved with the implementation of such reactive systems. Although it has its benefits (predictability and ease of use) it also has its share of flaws, the main one being not having as much freedom with the coding process as an imperative language would. Elm is one of the more popular FRP languages which is specifically geared towards building HTML rendered GUIs. We decided to use Elm to implement a 2D platformer for our project to explore the advantages/disadvantages of FRP design, and such a project is a good example of a very reactive system. We hope to build a well polished and complete implementation, but the main focus would again be on highlighting Elm and FRP, and comparing it to other more popular approaches.

REFERENCES

Czaplicki, Evan (2012), *Elm: Concurrent FRP for Functional GUIs*.

Czaplicki, Evan (2014), "Blazing fast html:virtual dom in elm." `http://elm-lang.org/blog/blazing-fast-html`, accessed: 2015-15-11.

n.d., (2015a), "Elm (programming language)," `https://en.wikipedia.org/wiki/Elm_(programming_language)`, accessed: 2015-15-11.

n.d., (2015b), "Functional reactive programming," `https://en.wikipedia.org/wiki/Functional_reactive_programming`, accessed: 2015-15-11.

Staltz, Andre (2015), "The introduction to reactive programming you've been missing," `https://gist.github.com/staltz/868e7e9bc2a7b8c1f754`, accessed: 2015-15-11.