Functional Reactive Programming in elm

s1137931

November 7, 2014

1 Abstract

Building reactive graphical user interfaces (GUIs) has proven to be a difficult task. Using a functional reactive programming language that lets build GUIs declaratively is a possible approach to simplifying the task. This report describes how the elm programming language is trying to address this problem, outlines how the programming language works, detailing some of its advantages and limitations, by providing an example of using elm to build a sample website, promoting tourism in Scotland.

2 Introduction

Elm programming language is a functional reactive programming language that provides a declarative way of defining the graphical layout. Moreover, it provides functional reactive programming primitives to describe the interaction between different components of the GUI and build powerful GUIs by writing a relatively small amount of code[1]. Although this approach should be helpful for some advanced programmers, elm might be quite difficult to understand for beginners, due to a limited number of learning resources available on the web.

In my report, I am going to explain the particular problem that elm programming language is trying to address, provide a description of the programming language itself, illustrated by examples of its possible use cases.

3 Context

The process of building a front-end of a website can be divided into two main stages:

- Defining the structure of the web page, usually by writing the HTML markup, and adding styling, traditionally done with CSS.
- By using a language like JavaScript, defining the way that the web page is going to react to the user input, like mouse clicks or key strokes.

Although these two stages might seem to be straightforward and simple, usually they are not. There are a few reasons behind it.

First of all, the initial step of specifying the structure and graphical layout of the page proves to be difficult, because it has to be coded up in an imperative

manner - instead of specifying *what* elements need to appear, the programmer has to specify *how exactly* it must look like. Various tools have been created to address this issue, for example, Adobe Dreamweaver WYSIWYG editor has a drag-and-drop interface for creating the structure of a web page. However, it is an expensive proprietary development tool that, similarly to the other tools available, often produces verbose and poor quality code, which is difficult to maintain.

Secondly, designing interaction between the user and the page in order to make it reactive often results in architecturally complex code[4]. When using an event-driven programming language like JavaScript, the programmer often needs to explicitly traverse and manipulate the DOM tree, attach event listeners, callbacks, provide event handlers and maintain the state of the data structures of the application.

4 Elm programming language

The elm programming language was designed to reduce the complexity of building interactive graphical user interfaces and provides a different approach of implementing both stages of the pipeline[1]. Instead of declaring the graphical layout of the page in an imperative fashion, the language allows specifying it declaratively and provides a way to describe the interaction between the user and the page by using the functional reactive programming (FRP) paradigm.

Functional reactive programming is a high level declarative language for programming reactive systems[2]. Here, the emphasis is put on the concepts of data flows and the propagation of changes through the system. In contrast to imperative programming, where the values need to be explicitly updated, in functional reactive programming each value is connected to the data flow and the values get updated automatically[3].

Although the elm programming language syntax looks very close to Haskell, elm is not a sub-language of Haskell. Although this may sound confusing, since the compiler for elm is written in Haskell, the compiler produces JavaScript, HTML and CSS. Also, elm is not lazy[5].

In order to better understand how FRP works in elm and how elm is converted into code that can be executed by a browser, we need to get familiar with a couple of elm's key concepts.

4.1 Elements

The data type *Element* defines the basic visual block that can be presented on the screen - it is simply a rectangle with some width and height, that can contain values like images, video or text. It is also possible to display various shapes, defined by the datatype *Form* that are not rectangular, for example, a Form can be a circle[1]. Let me illustrate this with an example:

```
main : Element
main = flow down [scotlandFlag, description]


scotlandFlag : Element
scotlandFlag = fittedImage 200 100
  ("https://upload.wikimedia.org/wikipedia/" ++
```

```
   "commons/6/66/Flag_of_Scotland_%28Union_" ++
   "Jack_colours_and_proportion%29.png")

description : Element
description =
  [markdown| Welcome to the official web page of Scotland!
             We are famous for haggis,
             bagpipes and many more other things!|]
```

Here, the element to be displayed is produced by taking elements `scotlandFlag`, `description` and placing one below another using function `flow` with parameter `down`. The value `scotlandFlag` of type *Element* is an image of fixed size and the value `description` of type *Element* is a piece of Markdown.

As we can see in this example, elm makes it easy to declaratively define the graphical layout of a page. Moreover, modifying the code to add a new element or changing the order of the layout is trivial. Although I did not use any styling in my example, elm provides a range of various functions to add style to each element. However, our page is not responsive - resizing the window doesn't scale the flag. In order to enable this, we need to introduce a new concept.

4.2 Signals

There are various kinds of input that our program can get from the world - the position of the mouse, the value of the key that has been pressed by the user, etc.. These values are called signals in FRP and in elm, they are modelled by the data type *Signal*. The value of each signal changes with respect to the user actions, so we can use signals to create GUI that reacts to the input. Signals can be transformed and combined into new signals[1]. Using signals we can make our previous example rescale when the window size changes:

```
import Window

main : Signal Element
main = lift layout Window.width


layout : Int -> Element
layout x = flow down [scotlandFlag x, description]


scotlandFlag : Int -> Element
scotlandFlag x = fittedImage x (x // 2)
  ("https://upload.wikimedia.org/wikipedia/" ++
   "commons/6/66/Flag_of_Scotland_%28Union_" ++
   "Jack_colours_and_proportion%29.png")

description : Element
description =
  [markdown| Welcome to the official web page of Scotland!
             We are famous for haggis,
             bagpipes and many more other things!|]
```

Here, we are using the signal `Window.width`, which provides the width of the browser window. In order to pass the signal value into a function, we must use a function `lift`. It is passed to the function `layout` by *lifting* the signal that uses the value to scale the `scotlandFlag` element to fit the width of the window.

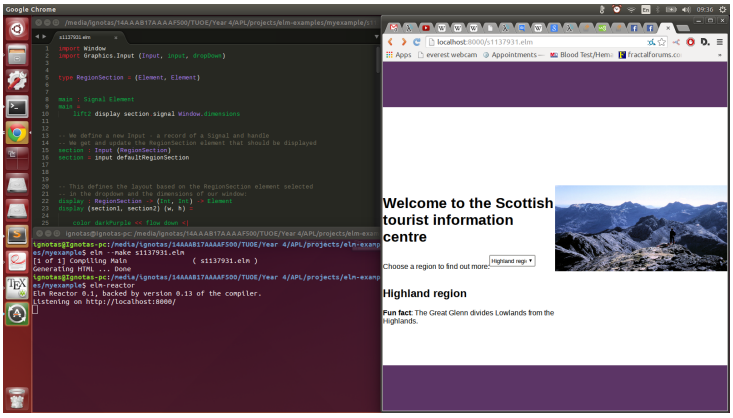
4.3 Signal graphs and concurrent execution

After defining the basic building blocks of the language, we can finally talk about the architecture of the program and explain how the input values propagate through the GUI. Each GUI can be represented as a directed acyclic graph, where each node represents an element of the layout and each vertex represents a signal being passed from one element to another. The creator of elm uses the term *signal graph* to denote the directed graph that represents the system[6].

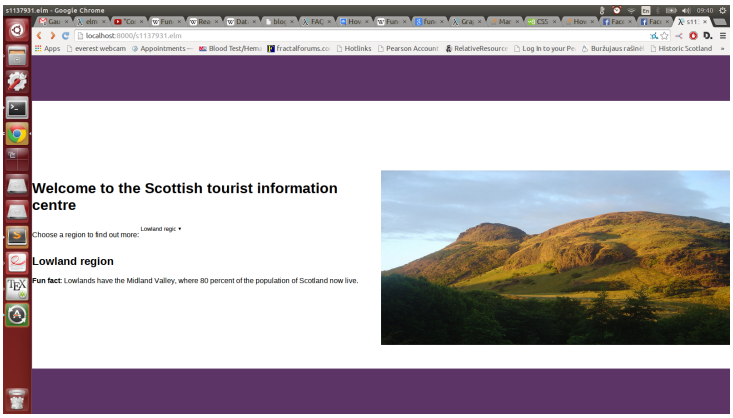
Each time a new event occurs, the signal representing the event is passed into the signal graph. It is then processed at the node and the resulting signal is sent to all of the descendent nodes. In elm, no two events can occur at the same time, which means that there is a strict ordering of the times of each event. Moreover, elm must ensure that each event finishes being processed in the same order it came in (first-in-first-out). One of the advantages of elm is that it is possible to be processing several events at the same time. Since one node can be receiving input from several other nodes, the node might have to wait for a signal from one of the nodes, before it can start running the computation. This means that nodes have to synchronize with each other. This is achieved by running each node on a separate thread and making threads synchronize by exchanging messages[1].

5 Example

To illustrate the concepts outlined before and show that elm can be used for designing reactive GUIs in real world, I constructed an example of a minimalistic Scottish tourist information centre. Initially, none of the regions in Scotland are selected in the drop down, so the default information is shown and the Scottish flag is displayed. However, after picking the "Highlands region" option in the drop down, the description and the image changes. This is what it looks like:



As you can see, the page is responsive - it reacts when it's dimensions are changed and rescales. This is how the web page looks in full scale:



In my sample program, I defined a type called `RegionSection` that simply contains the text element and the image element. I use `RegionSection` type to encapsulate the text and image values that are displayed on the page when a region is selected. The `options` value holds the mapping between the name of each region that can be selected in the dropdown and the `RegionSection` containing the data for the selected region.

In order to be able to obtain the current value selected in the drop down menu and update it when the value changes, I used `section`. This value is a `Input` record that is defined as `signal : Signal a, handle : Handle a` [7].

Finally, I defined the `display` function that takes the value of the section that should be shown and the dimensions of the browser window. It colors the background as darkPurple and inserts the main element, containing textual description and an image, into the middle of the page.

Here is the code for my example:

```
import Window
import Graphics.Input (Input, input, dropDown)


type RegionSection = (Element, Element)


main : Signal Element
main =
    lift2 display section.signal Window.dimensions


-- We define a new Input - a record of a Signal and handle
-- We get and update the RegionSection element that should be displayed
section : Input (RegionSection)
section = input defaultRegionSection



-- This defines the layout based on the RegionSection element selected
-- in the dropdown and the dimensions of our window:
display : RegionSection -> (Int, Int) -> Element
display (section1, section2) (w, h) =

    color darkPurple << flow down <|
        [ spacer w 100
        , container w (h-200) midTop

        (color white <| container w (h-100) middle <| flow right
            [
            flow down
                [
                width (w // 2) title,
                flow right [chooseOptionText , dropDown section.handle options],
                width (w // 2) section1
                ],
            size (w // 2) (w // 4) section2
```

```
           ]),
        spacer w 100
        ]


-- These are the options available in the dropdown:
options : [(String, RegionSection)]
options = [("", defaultRegionSection),
           ("Island region", islandRegionSection),
           ("Highland region", highlandRegionSection),
           ("Lowland region", lowlandRegionSection)]


chooseOptionText : Element
chooseOptionText = [markdown|
Choose a region to find out more:
|]



title : Element
title = [markdown|
Welcome to the Scottish tourist information centre
==================================================
|]

-- These are the section descriptions for each region in Scotland:

-- Default region - entire Scotland

-- Image from: https://en.wikipedia.org/wiki/Flag_of_Scotland#mediaviewer/File:Flag_of_Scotland_(Union_Jack_colours_and_proportion).png

defaultRegionSection : RegionSection
defaultRegionSection = (defaultDescription, defaultImage)

defaultDescription = [markdown|
Regions of Scotland
-------------------
Select a region from the dropdown menu to find out more about them!
|]

defaultImage : Element
defaultImage = image 100 100 "flag.png"


-- Island region

-- Image from: https://en.wikipedia.org/wiki/Outer_Hebrides#mediaviewer/File:View_south_from_Heaval.jpg

islandRegionSection : RegionSection
islandRegionSection = (islandDescription, islandImage)

islandDescription : Element
islandDescription = [markdown|
Island region
---------------------------------
**Fun fact**: This region is famous for the number of islands it has
|]

islandImage : Element
islandImage = image 100 100 "islands.jpg"

-- Highland region

-- Image from: https://en.wikipedia.org/wiki/Scottish_Highlands#mediaviewer/File:Main_ridge_of_the_cuillin_in_skye_arp.jpg

highlandRegionSection : RegionSection
highlandRegionSection = (highlandDescription, highlandImage)

highlandDescription = [markdown|
Highland region
---------------------------------
**Fun fact**: The Great Glenn divides Lowlands from the Highlands.
|]

highlandImage : Element
highlandImage = image 100 100 "highlands.jpg"

-- Lowland region

-- Image from: https://en.wikipedia.org/wiki/Central_Lowlands#mediaviewer/File:Edinburgh_Arthur_Seat_dsc06165.jpg

lowlandRegionSection : RegionSection
lowlandRegionSection = (lowlandDescription, lowlandImage)


lowlandDescription = [markdown|
Lowland region
---------------------------------
**Fun fact**: Lowlands have the Midland Valley, where 80 percent of the population of Scotland now live.
|]

lowlandImage : Element
lowlandImage = image 100 100 "lowlands.jpg"
```

6 Conclusion

In conclusion, I believe that the elm programming language provides a useful alternative to the traditional imperative of describing GUIs. However, it would be naive to think that just because a reactive GUIs can be specified in a declarative way, they are easy to understand and write. The elm programming language is only 2 years old, so, naturally, there are very few examples and tutorials for learning elm. Based on my experience, elm has a steep learning curve and although a small number of experienced programmers might benefit by using it, it is not an easy tool to use for those with no prior experience in functional programming.

7 Bibliography

1. Evan Czaplicki, 2012. Elm: Concurrent FRP for Functional GUIs. Harvard, MA, USA.

   Retrieved October 10, 2014, from http://elm-lang.org/papers/concurrent-frp.pdf.

2. Taha, Walid, P. Hudak, and Z. Wan. "Event-driven FRP." Proceedings of the 4th international symposium on practical aspects of declarative languages (PADL). 2002.

3. Bainomugisha, Engineer, et al. "A survey on reactive programming." ACM Computing Surveys (CSUR) 45.4 (2013): 52.

4. Järvi, Jaakko, et al. "Property models: from incidental algorithms to reusable components." Proceedings of the 7th international conference on Generative programming and component engineering. ACM, 2008.

5. Evan Czaplicki. Elm language FAQ. Retrieved November 7, 2014, from http://elm-lang.org/learn/FAQ.elm.

6. "Controlling Time and Space: understanding the many formulations of FRP" by Evan Czaplicki.

   Retrieved November 7, 2014, from https://www.youtube.com/watch?v=Agu6jipKfYw

7. Evan Czaplicki. Elm language documentation. Retrieved November 7, 2014, from http://library.elm-lang.org/catalog/elm-lang-Elm/0.13/JavaScript-Experimental.