

L. MORSE, S. WEIRICH: FUNCTIONAL REACTIVE PROGRAMMING USING ELM

# FUNCTIONAL REACTIVE PROGRAMMING USING ELM

Undergraduate Intern

**LEONDRA MORSE\***

*University of Maryland Eastern Shore*

*Department of Mathematics and Computer Science*

*Undergraduate Student: Junior*

*Bachelor Degree (Expected: Spring 2016) in Computer Science*

*E-mail: [lnmorse@umes.edu](mailto:lnmorse@umes.edu)*

Research Mentor

**STEPHANIE WEIRICH**

*University of Pennsylvania*

*Associate Professor of Computer and Information Science*

*School of Engineering and Applied Science*

*Ph. D (2002) Computer Science: Cornell University*

*E-mail: [sweirich@cis.upenn.edu](mailto:sweirich@cis.upenn.edu)*

**Abstract:** Functional Programming is a programming paradigm. A style of building elements and structure of computer programs. This will treat computation as the evaluation of mathematical functions and will avoid changing-state and mutable data. Functional Reactive Programming is a programming paradigm for asynchronous data-flow using the building blocks of functional programming such as map, filter, reduce, etc. In the field of computer science, we are especially interested in forming uncomplicated and simple approaches in order to simplify a series of actions or steps taken to achieve a finish product. These procedures provide a collection of processes in the development and construction of algorithms. This will enable clarity, understanding, simplicity, and elegance to the knowledge in comprehending codes of various languages from different authors and editors. Functional style is different to current promoted languages such as Java, C++, C, and Visual Basic. Software engineers are always looking for the next best step to a concise, abstracted manner of coding. The goal in this research project is to learn, understand, and comprehend functional reactive programming by using Elm language to design and develop a program.

**Keywords:** Functional Reactive Programming, Asynchronous data-flow, Simplified approaches, Functional style, Abstract concise code

# Table of Contents

<b>Table of Contents</b> .....	<b>ii</b>
<b>1. Introduction</b> .....	<b>2</b>
1.1 Purpose .....	2
<b>2. Overall Description</b> .....	<b>2</b>
2.1 Functional Reactive Programming .....	2
2.2 Elm Programming Language .....	2
<b>3. Elm Programming Final Project</b> .....	<b>3</b>
3.1 Project Purpose .....	3
3.2 Product Scope .....	3
3.3 Intended Audience and Reading Suggestions .....	3
3.4 Product Functions .....	4
3.5 Elm Requirements .....	4
3.5.1 Model Configurations .....	4
3.5.2 Model .....	5
3.5.3 Input .....	6
3.5.4 View Configurations .....	6-7
3.5.5 Updates .....	8-9
3.5.6 Display .....	10
<b>4. Implementing Elm</b> .....	<b>10</b>
4.1 Accessing Elm .....	10
4.2 Executing Elm Program .....	12
<b>5. Conclusion</b> .....	<b>12</b>
<b>6. Other Requirements</b> .....	<b>13</b>
<b>Appendix A: References</b> .....	<b>13</b>

## 1. INTRODUCTION

### 1.1 Purpose

This summer will incorporate the learning of a new language called “*Elm*”.



Figure 1: “*Elm*” Product Logo

Czaplicki, Evan. "Elm." Elm. N.p., 2011 -15. Web. 08 June 2015.

Elm is a language based on the idea of Functional Reactive Programming. This is believed to be made easy to create interactive applications than most languages. The code can compile to HTML, CSS, and JavaScript. Elm is great for 2D and 3D games, diagrams, widgets, and websites. Learning the language of Elm helps create an easier way to create different Html JavaScript assignments. By analyzing the different elements of the Elm language will justify information, and the experience of learning functional reactive programming enabling full understand of Elm language style.

## 2. OVERALL DESCRIPTION

### 2.1 Functional Reactive Programming

Functional Reactive Programming is a concise approach to create a functional reactive programming system because the system requires interaction with a variety of different signals from the keyboard, and/or outside sources [5]. These systems consist of many different subjects such as videos games, animations, graphical user interfaces, modeling, and robotics.

### 2.2 Elm Programming Language

Elm is a Functional Reactive Programming Language. Therefore, this programming language is widely used for programs that consist of interactions from input and output sources. The features of Elm contains an abundance of valuable aspects such as no runtime exceptions (Elm’s compiler will locate and find the errors in your program before they impact your user) [1], Blazing fast rendering (Elm can convert elm based programs into JavaScript using elm-html library) [1], Libraries with guarantees (Elm language libraries are automatically enforced for all community libraries) [1], Clean syntax (No semicolons. No mandatory parentheses for calling functions. Everything is an expression. This allows more concise coding with the use of pattern matching, automatic currying, and destructive assignments.) [1], Smooth JavaScript interop (Elm consists of a JavaScript library that can communicate with elm code directly without sacrificing guarantees.) [1], and Time-traveling debugger (Elm allows the changing of code as it progresses without page refresh) [1].

The values in Elm are immutable, meaning the values cannot be modified after they are created, such that the values in Java, C++, and C are allowed to be modified after being defined. Values have advantages, they can be given a type annotation that describes the exact shape of the value [4]. These types include custom types called ADTs, this allows one to create new types, primitive types (strings and integers), and basic data structures (lists, tuples, and extensible records) [4].

There are 6 basic sections that are important to Elm. These sections of Elm consists of the model configurations, view configurations, inputs, model, updates, and the display of the program [2]. Each section is crucial to the functioning of elm programs. The model configurations are for the programs necessary data-flow, such as the size and speeds of objects within the program. On the other hand, the view configurations creates a foundation for color, and text, this will be displayed within the program. The inputs will come from the user, and or keyboard. The model (map) will create a format of where objects should or should not be placed in one's program before displayed on one's screen. Updates will provide renewal of interactions between objects, and inputs. This section is very important to the existence of interactions in one's program. Finally, one will have to display the program on the screen through the display (show) section of the program.

### **3. ELM PROGRAMMING FINAL PROJECT**

#### **3.1 Product Purpose**

The purpose of this product is to create a program that will include everything that one will learn about functional reactive programming. In this case, one will create a program that consist of interactions between several objects within a graphical interface. This program will be replicated from a game called Brick Buster, but will be created with the use of Elm language.

#### **3.2 Product Scope**

A web-based applet that allows the player to move the paddle from left to right using the input from the keyboard in order to rack up points by hitting all the bricks that are in the field to complete the game. This game is a technical way to understand how elm programming works in the access of interacting objects.

#### **3.3 Intended Audience and Reading Suggestions**

This program is intended for persons of the DREU Computing Research Association, and the Computer Science Department. The sequence for reading this documents begins with the overview sections and proceeds through the sections that are most pertinent to each reader type. Please refer to the table of contents to find area of interest.

### 3.4 Product Functions

This product will consist of many functions between objects and collisions between them objects within the program. The paddle will have to interact with the ball, and the ball will have to interact with each brick within the display of the program providing points for each brick that is hit.

### 3.5 Elm Requirements

#### 3.5.1 Model Configurations

The game width and height is set to 600px to 400px. The half width and half height splits the whole game as a whole in half from 300px to 200px. As one will see that there boundaries, that will be half of the games width, and game height. Also, there are four record types created here known as Brick, Ball, Player, and Game type. These particular types define key elements that are needed to define each type. Each element are defined as a data type “Float” known as the storage of floating-point values, that is, values that have potential decimal places [7], and “Int” known as the storage of integral values, that is, whole numbers [7].

The Brick type consists of the x and y coordinates, the height and width of the brick, and the number of rows and the number of columns of bricks. The Ball type consists of x and y coordinate, and the “vy” and “vx” velocities for the speed of the ball. Such as the player, and game type. All models consist of the same elements that are necessary for that particular type.

```
(gameWidth,gameHeight) =
(600,400)
(halfWidth,halfHeight) =
(300,200)

paddleYPos= -100
ballRadius = 15
paddleWidths = 50
paddleHeight = 10

{--This type defines the game
state changes. Will the game be
played, paused, or Setup--}

type State = Play | Pause |
Setup

{--The types are created, and
Float, Int are defined for each
element within the record --}

type alias Brick= --Brick type
{ posX: Float
, posY: Float
, brkH: Float
, brkW: Float
, brkR: Float
, brkC: Float
}

type alias Ball = --Ball type
{ x : Float
, y : Float
, vx : Float
, vy : Float }

type alias Player ==--Player type
{ x : Float
, y : Float
, vx : Float
, vy : Float }
```

```
, score : Int
, width: Float
, height: Float }
```

```
type alias Game = --Game type
{ state : State
, ball : Ball
, player1 : Player
, brick : List Brick
,score : Int }
```

```
type alias Input =
{ space : Bool
, enter: Bool
, dir1 : Int
, delta : Time }
```

### 3.5.2 Model

The model section consists of the positioning and the sizes of each of the element types located in the model configurations. Located below in the Elm code the Player, Game, Ball, and Brick types are defined. Each element within each type are defined, and all elements within that element type have definite values. These elements are stored in what we call data functions.

```
{--This creates a function
named player that takes a float
and gives it to the player. In
which you define each of the
elements within the type Player
recorded --}
```

```
player : Float -> Player
player x1 =
{ x=0, y=x1, vx=0, vy=0,
score=0, width=40, height=8 }
```

```
{-- This creates a function
named defaultGame that takes
the type record Game and
defines each element within the
Game record--}
```

```
defaultGame : Game
defaultGame =
{ state = Setup
```

```
, ball = Ball 0 (paddleYPos +
ballRadius) 0 0
, player1 = Player 0
paddleYPos 0 0 0
paddleWidths paddleHeight
, brick = [ brick1, brick2,
brick3, brick4 ]
, score =0
}
```

```
{--This creates a function
named intball that uses the
record Ball and fills in each
element --}
```

```
intball: Ball
intball = Ball 0 (paddleYPos +
ballRadius) 0 0
```

```
{-- This creates a function
named brick1, brick2, brick3,
and brick4 that defines the size
of the brick and the position --}
```

```
brick1: Brick
brick1 =
  { posX = 50
  , posY = 50
  , brkH = 15
  , brkW = 30
  , brkR = 5
  , brkC = 5 }
```

```
brick2: Brick
brick2 =
  { posX = -50
  , posY = 50
  , brkH = 15
  , brkW = 30
  , brkR = 5
  , brkC = 5 }
```

```
brick3: Brick
brick3 =
  { posX = -100
  , posY = 50
  , brkH = 15
  , brkW = 30
  , brkR = 5
  , brkC = 5 }
```

```
brick4: Brick
brick4 =
  { posX = 100
  , posY = 50
  , brkH = 15
  , brkW = 30
  , brkR = 5
  , brkC = 5 }
```

### 3.5.3 Input

The input section is the signal that is coming or will be read by the keyboard. The following code shows what will be used to help move and interact with objects. These elements consist of `signal.map4 Input [3]` (Input takes 4 arguments, therefore `map4` will need to be used), the `Keyboard.space [3]` (shortcut command for computer keyboard spacebar), the `Keyboard.enter [3]` (shortcut command for computer keyboard enter bar), and the `Signal.map.x Keyboard.wasd [3]` (creates a signal for the keyboard to use the “w,a,s,d” for one element mapping) for the program to interact with the program.

```
input: Signal Input
input =
  Signal.sampleOn delta <|
    Signal.map4 Input
      Keyboard.space
      Keyboard.enter
      (Signal.map .x Keyboard.wasd)
    delta
```

### 3.5.4 View configuration

The view configurations create a foundation for color, and text that one will like to have displayed within the program. This part of the program is very easily constructed.

```
{-- Creates a function named
view that accepts 2 integers and
passes it through the Game
type to return an element. This
is also where the game, paddle,
ball, and score is created to be
print to the screen with color --
}
```

```
view : (Int,Int) -> Game ->
Element
view (w,h)
{state,ball,player1,brick} =
  let
    scores =
      txt (Text.height 50)
      (toString player1.score)
  in
    container w h middle <|
      collage gameWidth
      gameHeight
      [ rect gameWidth
      gameHeight
        |> filled pongGreen
        , oval 15 15
        |> make ball
        , group (List.map
      viewBrick brick)
        , rect player1.width
      player1.height
        |> make player1
        , toForm scores
        |> move (0,
      gameHeight/2 - 40)
        , toForm (if state == Play
      then spacer 1 1 else txt identity
      msg)
        |> move (0, 40 -
      gameHeight/2)
      ]
```

```
{-- Creates a function called
viewBrick and colors the bricks
with a specific color --}
```

```
viewBrick: Brick -> Form
viewBrick brick =
  filled Color.white (rect
brick.brkW brick.brkH)
  |> move (brick.posX,
brick.posY)
```

```
{-- Colors the background of the
game --}
```

```
pongGreen =
  rgb 0 0 0
```

```
{--Colors the text within the
program --}
```

```
textGreen =
  rgb 160 200 160
```

```
txt f string =
  Text.fromString string
  |> Text.color textGreen
  |> Text.monospace
  |> f
  |> leftAligned
```

```
{-- Creates the message that will
be displayed on the screen --}
```

```
msg = "SPACE to start, AD to
move, Enter to Setup"
```

```
make obj shape =
  shape
  |> filled white
  |> move (obj.x,obj.y)
```



### 3.5.5 Updates

The update section of the program is a crucial part of the program. This section of the program keeps track of the programs progress. This is an essential portion to the functioning of the program and the interactions within the objects created within the program itself. It keeps track of the states that the programs may be confined to, and the collisions between the objects. In the code below there are several updates from the collisions between the ball and paddle, the ball and the brick, and also the game states.

```

update : Input -> Game -> Game
update {space, enter, dir1,
delta} ({state,ball,player1,
brick, score} as game) =
  let

    maybeBrick=
      updateHit ball brick

    points =
      case maybeBrick of

        Just b -> 1

        Nothing -> 0

    dropped =
      ball.y < -halfHeight

    newBrick =
      case maybeBrick of

        Just b -> removeBrick b
brick

        Nothing -> brick

    newState =
      if

        | enter ->
          Setup

        | space ->
          Play

        | dropped ->
          Pause

        | otherwise ->
          state

    newPlayer =
      updatePlayer delta dir1
points player1

    newBall =
      if | newState == Pause ->
        ball

        | space ->
          { ball |
            vx <- 90,
            vy <- 70
          }

        | enter ->
          { intball|
            x <- newPlayer.x
          }

        | newState == Setup ->
          { ball |
            x <- newPlayer.x
          }

        |otherwise ->

          updateBall delta ball
player1 maybeBrick
in
      { game |
        state <- newState,

```

```

    ball <- newBall,
    player1 <- newPlayer,
    brick <- newBrick
  }

updateBall : Time -> Ball ->
Player -> Maybe Brick -> Ball
updateBall t ({x,y,vx,vy} as
ball) p1 b1 =
  if not (ball.y |> near 0
halfHeight) then
    intball
  else
    let

    topHit =
      case b1 of

        Just b -> ball.y > b.posY
          Nothing -> False
        bottomHit =
          case b1 of
            Just b -> ball.y < b.posY
              Nothing -> False
            in
              physicsUpdate t
            { ball |
              vy <- stepV vy (ball
`within` p1 || topHit)(y >
halfHeight-7 || bottomHit),
              vx <- stepV vx (x < 7-
halfWidth) (x > halfWidth-7)
            }

    updateHit: Ball -> List Brick ->
Maybe Brick
    updateHit ball brick =
      case brick of
        [] -> Nothing

        head::tail ->
          if hitBrick ball head

          then Just head

          else updateHit ball tail

    removeBrick: Brick -> List
Brick -> List Brick
    removeBrick b1 b2 =
      case b2 of
        [] -> []

        head::tail ->
          let o= removeBrick b1 tail
          in
            if b1 == head
            then o
            else head :: o

    updatePlayer : Time -> Int -> Int
-> Player -> Player
    updatePlayer t dir points player
=
    let
      player1 =
        physicsUpdate t { player |
vx <- toFloat dir * 200 }
      in
        { player1 |
          x <- clamp (22-halfWidth)
(halfWidth-22) player1.x, --
make sures within players range
          score <- player.score +
points
        }

    physicsUpdate t ({x,y,vx,vy} as obj)
=
    { obj |
      x <- x + vx * t,
      y <- y + vy * t
    }

    near k c n =
      n >= k-c && n <= k+c

```

```
hitBrick: Ball-> Brick -> Bool
hitBrick ball brick =
  near ball.x brick.brkW brick.posX && near
  ball.y brick.brkH brick.posY --}
```

```
within ball paddle =
  near paddle.x paddle.width ball.x && near
  paddle.y paddle.height ball.y
```

```
stepV v lowerCollision upperCollision =
  if | lowerCollision ->
    abs v
  | upperCollision ->
    -(abs v)
  | otherwise ->
    v
```

### 3.5.6 Display

This section displays the game.

```
main =
  Signal.map2          view
  Window.dimensions gameState
  gameState : Signal Game

gameState =
  Signal.foldp          update
  defaultGame input

delta =
  Signal.map inSeconds (fps 35)
```

## 4. IMPLEMENTING ELM

### 4.1 Accessing Elm

In order to access Elm on the system one must install the Elm-Package [1]. After installing the Elm-Package one will have to install Git (allows access to install evancz files to your system) [6]. Then one will proceed to activate the Elm-Reactor.

**Step 1:** Elm-Reactor Activation

- a. Open Command Prompt
- b. Type: elm-reactor.exe

Example: "C:\Users\ (System Name)> elm-reactor.exe"

- c. If the file "elm-reactor.exe" is located it should create a platform where elm files can be access and executed.

**Note:** If "elm-reactor.exe" is **not found** on your system after elm-package is installed one must open their computer registry found on their system and create a direct path to the file for execution. (Shown in steps a.1- a.3)

**Step a.1:** Access Registry Editor

- a. Open Command Prompt
- b. Type: regedit

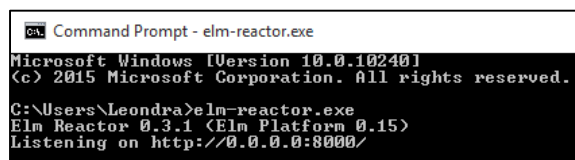
Example: "C:\Users\ (System Name)> regedit"

**Note:** may get a pop up about access permission to run the regedit as known as "User Account Control", accept the pop up.

- c. Registry will Open with the title "Registry Editor"

**Step a.2:** Creating a Path for Elm

- a. Now find the folder known as "HKEY\_CURRENT\_USER" extend the folder
- b. Under the "HKEY\_CURRENT\_USER" folder go to the "Environment" folder
- c. Right Click -> New -> Expandable String Value, Name that file "PATH"
- d. close Registry Editor

**Step a.3:** Repeat Step 1.

```
Command Prompt - elm-reactor.exe
Microsoft Windows [Version 10.0.10240]
(c) 2015 Microsoft Corporation. All rights reserved.

C:\Users\Leondra>elm-reactor.exe
Elm Reactor 0.3.1 (Elm Platform 0.15)
Listening on http://0.0.0.0:8000/
```

**Figure 3**

**\*\*MUST** keep "elm-reactor.exe" active in order for elm platform to work! In other words leave open the command prompt while in use.

## 4.2 Executing Elm Program

Executing Elm program is easy after executing the elm-reactor. One will have to access the platform through a browser application on the Local Host [2]. Shown in the following step.

### Step 2: Accessing Elm Platform

- a. Open up your Internet browser
  - i. Firefox
  - ii. Google Chrome
  - iii. Internet Explorer
- b. Type in address bar: `http://localhost:8000`
  - i. Elm Platform opens in browser
  - ii. Directories should then be visible on browser
  - iii. Locate Elm document
- c. Running Elm document
  - i. Click located document
  - ii. Elm document will open in browser executed if code is correct without errors( located in Figure 4)



Figure 4

## 5. CONCLUSION

This document presents a technical description of the project and results that were obtained during the summer of 2015. The design and implantation of a final Elm program using all fundamental, and advanced styles of functional reactive programming is described to broaden the understanding of the programs functionality. The style of functional reactive programming used in this program shows what has been learned and understood throughout the summer. Perhaps functional reactive programming is more complicated than JavaScript, C++, and C, but the coding is more concise and easier to understand when looking at others code. Functional reactive programming is believed to become the new programming style used across the world because of its easy functionality among the interaction between varieties of different domains.

## 6. OTHER REQUIREMENTS

### Appendix A: References

- [1] Czaplicki, E. (2015, April 20). Elm. Retrieved August 5, 2015, from <http://elm-lang.org/>
- [2] Czaplicki, E. (2015, April 20). Learn by Example. Retrieved August 7, 2015, from <http://elm-lang.org/examples>
- [3] Czaplicki, E. (2015, June 25). Elm-lang / core. Retrieved August 7, 2015, from <http://package.elm-lang.org/packages/elm-lang/core/2.1.0/>
- [4] Elm (programming language). (2015, July 31). Retrieved August 4, 2015, from [https://en.wikipedia.org/wiki/Elm\\_\(programming\\_language\)](https://en.wikipedia.org/wiki/Elm_(programming_language))
- [5] Reactive programming. (2015, May 15). Retrieved August 5, 2015, from [https://en.wikipedia.org/wiki/Reactive\\_programming](https://en.wikipedia.org/wiki/Reactive_programming)
- [6] Torvalds, L. (2005, April 7). Git. Retrieved August 5, 2015, from <http://git-scm.com/>
- [7] What is the difference between the float and integer data type when the size is the same? (2013, October 10). Retrieved August 7, 2015. from <http://stackoverflow.com/questions/4806944/what-is-the-difference-between-the-float-and-Integer-data-type-when-the-size-is>