2015

# Functional Reactive Programming For Games

Peter Adewunmi Salu
*University of Mississippi*

# FUNCTIONAL REACTIVE PROGRAMMING FOR GAMES

A Thesis
presented in partial fulfillment of requirements
for the degree of Master of Science
in the Department of Computer and Information Science
The University of Mississippi

by

Peter Salu

December 2015

ABSTRACT

We investigate the effectiveness of functional reactive programming for games. To accomplish this, we clone aa, an existing game, in Elm, a purely functional programming language. We find that functional reactive programming offers an excellent alternative to event driven programming in purely functional languages. Elm still needs more work if it aims to compete with JavaScript libraries. Games, which typically need several inputs at the same time, benefit from the first class status of Signals, which allow them to be combined.

ACKNOWLEDGEMENTS

Thanks to github user eugenioclrc for cloning aa in JavaScript and giving me insight into how he did it. I would also like to thank Michael Macias for helping me with the math in aa, pointing me in the direction of the eugenioclrc aa clone, and reviewing my paper.

TABLE OF CONTENTS

LIST OF FIGURES

# CHAPTER 1

# INTRODUCTION

The goal of this thesis is to study the use of functional reactive programming for video games. We achieve this by implementing a game in a purely functional language and making note of the problems and advantages of this style.

## 1.1 Previous Work

One prominent paper that already shows how functional reactive programming can be used for games is the Yampa arcade [4]. This paper shows how to implement Space Invaders using Yampa which is a Functional Reactive Library that uses Haskell as a host language. This paper serves as a template for the thesis.

# CHAPTER 2

# OVERVIEW OF ELM

For the reader to properly understand the programs as described in the thesis, they must first understand Elm a purely functional, functional reactive programming language the games are written in. It compiles to JavaScript and runs in any capable web browser.

## 2.1 History Of Elm

Elm was first released in April of 2012. It was part of the thesis of Evan Czaplicki at Harvard [5]. The syntax of Elm resembles Haskell [2] with ML [3] style semantics. The big difference from Haskell is that Elm is not lazy. Laziness in languages means that expressions are not evaluated until their result is needed. This can allow for the creation of infinite data structures whose later values are not known until they are needed.

## 2.2 Basic Syntax

Using the REPL (Read Evaluate Print Loop) interactive interpreter interface, we show various features of Elm. The inputs to the REPL prefix with ”>”. After the statement, the REPL displays the resulting value and its type.

The basics of math work exactly like they do in other languages.

```
> x = 1 + 1
2 : number
> 22 / 7
3.142857142857143 : Float
> 32 % 3
2 : Int
```

The first line also shows how to assign values to variables.

As Elm is a functional language, function calls are very lightweight.

```
> sin 32
0.5514266812416906 : Float
> max 3 5
5 : comparable
```

The function name comes first, with the arguments separated only by whitespace.

Declaring functions works by giving the name of the function, followed by the parameters, then "=" before the body of the function.

```
> lessThan5 x = x < 5
<function> : comparable -> Bool
> lessThan5 12
False : Bool
> lessThan5 1
True : Bool
```

The first example works because of type inference, which enables Elm to determine the type of functions and other objects by how they are used without requiring explicit type declarations.

For documentation reasons, it is typical to declare the type of a function before the body of the function.

```
displayBackground : Int -> Int -> Form
displayBackground width height =
filled white (rect (toFloat width) (toFloat height))
```

The name of the function is first, and ":" means has a type. The parameters of this function are Ints, two shown, separated by "− >". The return value Form is last and is also separated by "− >". Form is a type internal to Elm that is used to represent shapes.

Functions can be partially applied meaning we can create new functions by applying existing functions to some of their arguments.

```
> is10 = (==) 10
<function> : number -> Bool
> is10 3
False : Bool
> is10 10
True : Bool
```

We have to wrap the function == in parentheses because it is a function whose name consists entirely of symbols. 10 is the first argument in the new function.

Some special functions are used to reduce the number of parentheses in code, thereby making it more readable [6].

```
scale 2 (move (10,10) (filled blue (ngon 5 30)))


ngon 5 30
  |> filled blue
  |> move (10,10)
  |> scale 2
```

The function | > (forward function application) is like a pipe in Unix, taking the output of one function, and using it as the input to the function on the right. There is another function < | (backward function application) which does the same thing, but it passes its results in the opposite direction. Mathematically, this is function composition as indicated by the first line of code above.

## 2.3 Tuples and Records

Tuples are a way to group objects together.

```
> (12, "Something")
```

```
(12,"Something") : ( number, String )
```

This way, we can easily return multiple values from a function or group arguments that are related together.

Records are a more robust way to group objects together. They give the component names that can be used to access the value.

```
> person = {name = "Peter", age = 25}
{ age = 25, name = "Peter" } :
{ age : number, name : String }


> person.name
"Peter" : String
```

This creates a new record whose member types can be inferred.

We can create a new record by updating the existing fields of a record.

```
> {person | age <- 12}
{ age = 12, name = "Peter" } :
{ name : String, age : number }
```

We can also create new records by adding new fields to existing records.

```
> {person | school = ``Ole Miss''}
{ age = 25, name = "Peter", school = ``Ole Miss'' }
: { age : number, name : String, school : String }
```

We can create parameterized record types by passing type parameters during record creation [9].

```
type alias Named a =
{ a | name : String }
```

```
lady : Named { age:Int }
lady =
{ name = ``Lois Lane''
, age = 31
}
```

In this case age was added during the declaration of the type of lady. This allows us to reuse records by defining common fields, adding more if we need them.

## 2.4 Unions

Unions allow us to define a type that is one of several other types.

```
type Direction = Left | Right
```

This creates a type which can only have the values Left or Right.

# CHAPTER 3

# FUNCTIONAL REACTIVE PROGRAMMING

Functional reactive programming is a way of obtaining values that can vary over time [1] [11].

It has emerged recently as an alternative to event driven programming. A typical event driven program can be difficult to structure because in reaction to an event, global state typically needs to be modified [1]. Another problem is needing the values provided by several events like the current mouse position when the mouse is clicked.

A typical event driven program has at least two parts to use an event. The first is the callback, which is a function that will be called when an instance of the event has occurred. The second is the event object, which will be passed in to the callback and have the relevant information e.g., the mouse position.

A signal is a variable whose value can change over time [8]. Signals are the equivalent construct in a functional reactive language to events in other languages. They also have two parts to use them. The first is the signal function, which will be called each time a new value of the signal arrives. The second is the signal itself, which pushes values to the signal function. The big difference from events is that signals are first class values. This means they are used as arguments to, declared in, and returned from functions. In other words, they are valid anywhere other values, like ints and booleans, are. Values from signals combine in a way events cannot.

## 3.1 Functional Reactive Programming in Elm

An example of displaying the mouse position in Elm is simple.

```
show <~ Mouse.position
```

Show is a function that takes an object and displays it on the screen. Mouse.position is a Signal which gives a tuple containing the x and y coordinates of the mouse position, e.g., (2, 226). $<\sim$ (pronounced map) is a function that applies the signal on the right to the function on the left. We can also consider map as transforming an ordinary function into a signal function. Each time the mouse position changes, the show function is called, and the tuple on the screen changes.

Multiple signals may need to be passed into a signal function. This can be accomplished by using one of the mapN functions, which takes in N signals as parameter and calls the signal function with the current value of any of the signal changes. The other way is to use the $\sim$ function, which allows multiple signals to be passed into a function.

```
Input <~ Keyboard.space ~ Keyboard.enter
```

The advantage signals have over events is their ability to be used as ordinary variables.

```
show <~ sampleOn Mouse.clicks Mouse.position
```

The sampleOn function returns the current value of the second parameter when the value in the first changes. The ability for signals to be combined is a powerful one allowing for ease of expression without global variables. This ability is similar to functional languages, where functions are able to be used as first class values.

# CHAPTER 4

# IMPLEMENTATION OF AA

To judge how useful functional reactive programming is for games, we implement aa [14]. To avoid getting caught in tricky game design work, we use an existing game.

aa is a game where the objective is to shoot a number of darts into a circular board. Once a dart hits the board, it begins to rotate. The player completes levels by shooting the required number of darts, and loses by having two darts collide. Two darts collide when the ball at the end of each dart overlap. The game becomes more challenging by having more darts on the board initially, and moving faster.

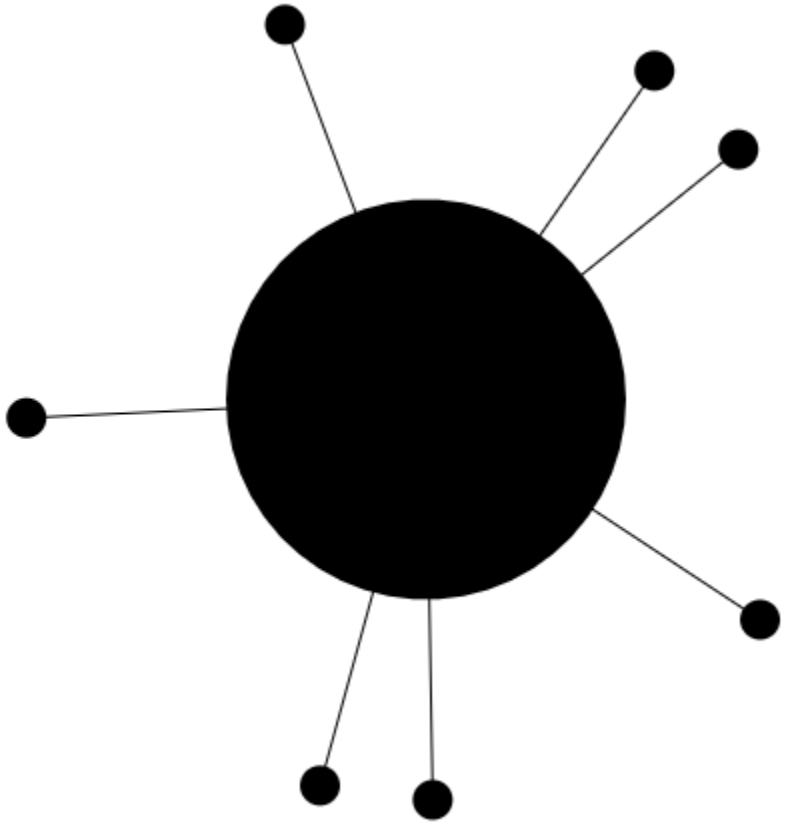
The code template followed for aa is set by Elm's Pong tutorial [7]. The tutorial divides the structure of data and functions into four parts: inputs, model, updates, and view.

The inputs for aa are the space bar, which fires darts, and delta time which is the amount of time that passes between updates. The delta is set at 60 times a second, and at each time delta, the state of the space bar is checked.

```
--Inputs to the game.
type alias Input =
{ space : Bool
, enter : Bool
, delta : Time
}


delta : Signal Time
delta = inSeconds <~ (fps 60)
```

Figure 4.1. aa clone with 7 darts on the board



```
input : Signal Input
input =
Signal.sampleOn delta
<| Input <~ Keyboard.space ~ Keyboard.enter ~ delta
```

The model for the game holds relevant data for all the game's objects. This includes the coordinates of objects, their velocities, the number of darts to be fired, whether the game is being played or paused, and the current level. Some of the fields of records are there for the view, which is where objects get displayed on screen, such as the radius of the board and darts.

```
type Direction = Left | Right
```

```
type alias Object a =
{ a |
x : Float
, y : Float
, vx : Float
, vy : Float
, angle : Float
, angularVelocity : Float
, direction : Direction
}

type alias Board =
Object { radius : Float
, numberOfDarts : Int
, collisionY : Float
}

type alias Dart =
Object { radius : Float
, isFired : Bool
, collidedWithBoard : Bool
, collidedWithOtherDart : Bool
, collidedSpeed : Float
}

type alias Darts = List Dart
```

```
type alias Player =
Object { darts : Darts
, isShooting : Bool
, indexOfDartToBeFired : Int
}

type State = LoadLevel | Play | Pause

type alias Game =
{ state : State
, board : Board
, player : Player
, spaceCount : Int
, level : Int
}

defaultBoard : Board
defaultBoard =
{ x = 0
, y = 130
, vx = 0
, vy = 0
, angle = 0
, angularVelocity = 5
, direction = Left
, radius = 100
, numberOfDarts = 10
```

```
, collisionY = -85
}

defaultDart : Dart
defaultDart =
{ x = 0
, y = -300
, vx = 0
, vy = 0
, angle = (3 * pi) / 2 --270 degrees in radians.
, angularVelocity = 0
, direction = Right
, radius = 10
, isFired = False
, collidedWithBoard = False
, collidedWithOtherDart = False
, collidedSpeed = 0
}

defaultPlayer : Player
defaultPlayer =
{ x = 0
, y = 0
, vx = 0
, vy = 0
, angle = 0
, angularVelocity = 0
```

```
, direction = defaultBoard.direction
, darts = []
, isShooting = False
, indexOfDartToBeFired = -1
}

defaultGame : Game
defaultGame =
{ state = LoadLevel
, player = defaultPlayer
, board = defaultBoard
, spaceCount = 0
, level = 0
}
```

Updating the game involves moving all objects in reaction to the inputs. The board keeps track of the number of darts in the level. For the player, updating involves moving the darts toward the board when space is pressed, making them orbit when they collide with the board and checking to see if they collide with each other.

```
--Update the game.
stepObject : Time -> Object a -> Object a
stepObject delta ({x
, y
, vx
, vy
, angle
, angularVelocity
```

```
, direction
} as object) =
{ object |
x <- x + vx * delta
, y <- y + vy * delta
}

collidedWithBoard : Dart -> Board -> Bool
collidedWithBoard dart board =
dart.y >= board.collisionY

unsafeGet : Int -> Array a -> a
unsafeGet index array =
case Array.get index array of
Just item ->
item
Nothing ->
Debug.crash "Unknown Index"

stepPlayer : Time -> Board -> Bool -> Player -> Player
stepPlayer delta board space player =
let indexOfDartToBeFired' =
if space || player.isShooting then
let indexOfDartToBeFired =
if space then
player.indexOfDartToBeFired + 1
else
```

```
player.indexOfDartToBeFired
in
indexOfDartToBeFired
else
player.indexOfDartToBeFired

dartsArray =
Array.fromList player.darts
|> Array.map (\dart -> stepDart delta dart board)
dartsArray' =
if (space || player.isShooting)
&& indexOfDartToBeFired' >
 defaultPlayer.indexOfDartToBeFired
&& indexOfDartToBeFired' <
 Array.length dartsArray then
  let dartToBeFired =
   unsafeGet indexOfDartToBeFired' dartsArray
  dartsArrayWithFired =
  Array.set
  indexOfDartToBeFired'
  {dartToBeFired | isFired <- True}
  dartsArray
  in
    collidedWithOtherDarts
     dartToBeFired
     dartsArrayWithFired
else
```

```
dartsArray
darts' = Array.toList dartsArray'

isShooting' = space || anyInFlight darts' board
in
{ player |
darts <- darts'
, isShooting <- isShooting'
, indexOfDartToBeFired <- indexOfDartToBeFired'
}

dartsInFlight : Darts -> Board -> Darts
dartsInFlight darts board =
let check dart =
  not dart.collidedWithBoard && dart.y > defaultDart.y
in
List.filter check darts

anyInFlight : Darts -> Board -> Bool
anyInFlight darts board =
dartsInFlight darts board
|> List.length
|> (/=) 0

collidedWithOtherDarts : Dart -> Array Dart ->
 Array Dart
collidedWithOtherDarts dart darts =
```

```
let collided aDart =
let dartDistance = distance
 aDart.x
 aDart.y
 dart.x
 dart.y
in
if aDart.collidedWithBoard
&& aDart /= dart
&& dartDistance <= dart.radius * 2 then
{aDart | collidedWithOtherDart <- True}
else
aDart
in
Array.map collided darts

distance : Float -> Float -> Float -> Float -> Float
distance x1 y1 x2 y2 =
sqrt <| ((x2 - x1) ^ 2) + ((y2 - y1) ^ 2)

stepDart : Time -> Dart -> Board -> Dart
stepDart delta dart board =
let vy' = if dart.isFired &&
not dart.collidedWithBoard then 600 else 0
collidedWithBoard' =
if not dart.collidedWithBoard then
collidedWithBoard dart board
```

```
else
dart.collidedWithBoard

angle' = dart.angle +
  if collidedWithBoard' then dart.collidedSpeed else 0

(x', y') =
if dart.collidedWithBoard then
(board.x + 2 * board.radius * cos angle'
,board.y + 2 * board.radius * sin angle'
)
else
(dart.x, dart.y)
dart' = stepObject
delta
{dart |
x <- x'
, y <- y'
, vy <- vy'
, angle <- angle'
, collidedWithBoard <- collidedWithBoard'
}
in
dart'

initialBoardDarts : Int -> Darts
initialBoardDarts n =
```

```
--360 degrees divided by n.
let delta = (2 * pi) / toFloat n
nDeltas = List.repeat n delta
angles = List.scanl (+) 0 nDeltas
updateAngle dart angle =
{dart | angle <- angle
, collidedWithBoard <- True
}
defaultDarts = List.repeat n defaultDart
in
List.map2 updateAngle defaultDarts angles

stepBoard : Time -> Board -> Board
stepBoard delta board =
stepObject delta board

loadLevel : Game -> Game
loadLevel game =
let level = unsafeGet game.level Level.levels
initialNumberOfDarts = level.initialNumberOfDarts
dartsToWin = level.dartsToWin
speed = level.speed

indexOfDartToBeFired' = initialNumberOfDarts - 1
darts' = initialBoardDarts initialNumberOfDarts
++ List.repeat
dartsToWin
```

```
{defaultDart | collidedSpeed <- speed}

player' =
{defaultPlayer |
darts <- darts'
, indexOfDartToBeFired <- indexOfDartToBeFired'
}
in
{game |
player <- player'
}

stepGame : Input -> Game -> Game
stepGame input game =
let
{space, enter, delta} = input

game' =
  if game.state == LoadLevel then
    loadLevel game
  else
    game
{state, board, player, spaceCount} = game'

state' =
case state of
LoadLevel -> Play
```

```
    _ -> state

  --TODO: Move this into a function.
  (spacePressed, spaceCount') =
  if space then
  if spaceCount == 0 then
  (space, spaceCount + 1)
  else
  (False, spaceCount + 1)
  else
  (space, 0)

  board' = stepBoard delta board
  player' = stepPlayer delta board' spacePressed player
  in
  {game |
  state <- state'
  , player <- player'
  , board <- board'
  , spaceCount <- spaceCount'
  }

  gameState : Signal Game
  gameState = Signal.foldp stepGame defaultGame input
```

The view simply takes data presented to it from the model, and displays it on the screen. The board is a circle with the number of darts left. The darts for the player draw separately, with the ball for the darts drawn from the model, and the line drawn from the center of the darts to the

center of the board.

```
-- View for the game.
displayBackground : Int -> Int -> Form
displayBackground width height =
filled white (rect (toFloat width) (toFloat height))

displayObject : Float -> Float -> Form -> Form
displayObject x y form =
move (x, y) form

drawBoard : Board -> Form
drawBoard board =
group
[
(filled black <| circle board.radius)
, (text
<| Text.height 40
<| Text.color white
<| Text.fromString
<| toString board.numberOfDarts
)
]

--Draw the board grouping darts that have collided
--with the board to the board.
displayBoard : Board -> Form
displayBoard board =
```

```
displayObject board.x board.y <| drawBoard board

dartColor : Color.Color
dartColor = black

drawDart : Dart -> Form
drawDart dart =
let drawDartColor =
  if dart.collidedWithOtherDart then red else dartColor
in
circle dart.radius
|> filled drawDartColor

drawLine : Dart -> Form
drawLine dart =
segment (defaultBoard.x, defaultBoard.y)
        (dart.x, dart.y)
|> traced (solid dartColor)

displayDart : Dart -> Form
displayDart dart =
  displayObject dart.x dart.y (drawDart dart)

display : (Int, Int) -> Game -> Element
display (width, height) {board, player} =
let dartForms = List.map displayDart player.darts
```

```
--Lines for the darts drawn separately so they
--won't move when the darts are relocated.
lineForms =
List.filter .collidedWithBoard player.darts
|> List.map drawLine
in
container width height middle
<| collage width height
<| displayBackground width height
:: lineForms
++ dartForms
++ [ displayBoard board ]


main : Signal Element
main = display <~ Window.dimensions ~ gameState
```

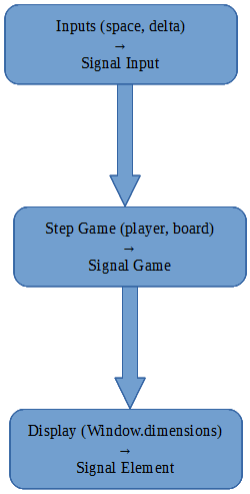
The inputs combine into a new signal, and trigger the generation of a signal of Game objects. A new instance of the game object is created by updating the model based on input. Finally, a signal of Element, Elm's way of displaying graphics on the screen, is created by reading the Signal of Game and displaying the data in the model.

```
input : Signal Input
input =
Signal.sampleOn delta
<| Input <~ Keyboard.space ~ Keyboard.enter ~ delta


gameState : Signal Game
gameState = Signal.foldp stepGame defaultGame input
```

```
main : Signal Element
main = display <~ Window.dimensions ~ gameState
```

Figure 4.2. The flow of data for aa.




## 4.1 Advantages of Functional Reactive Programming

Signals, combined with the structure of the game, reduced the overall complexity of the code. Since the inputs were composed together into a single record, there was no need for multiple event handlers setting the values of global variables.

In JavaScript and other imperative languages, mutation forms the basis for how data from multiple events are gathered. This is because events typically cannot return values, so saving the relevant variable involves setting a global variable from inside the event handler. Also making sure the values generated from events are updated in all the functions that need them can be tricky [1].

In purely functional languages like Elm, variables cannot be modified. This means that there is no way to set a global variable from inside an event handler. By passing in relevant values to signal functions, and allowing signals to be composed with each other, Signals provide a way

for all the functions that need a value to use the latest value.

## 4.2 Problems with the implementation

Most of the problems with the implementation are at the language level. Elm is a new language, and not all of the library features work as expected, or aren't as robust as they should be. For aa, only one dart should be fired regardless of how long space is pressed. Signal.dropRepeats is a function that only lets the new value out if it is not a duplicate of the last value. For an input record consisting of only the space parameter, this works as expected, but because delta, the time since the last update, is constantly changing, the entire record is not considered a duplicate. There was no way to modify the way to compare records, and there was no dropRepeatsWith that would allow filtering values with a custom function, although this has been discussed in [10]. Alternatively, Signal.foldp, which allows past values of signals to be inspected, also did not work because it could only look at the previous value. This meant that the value in the space signal would rapidly flip from true to false.

```
type alias Input =
{ space : Bool
, space2 : Bool
, delta : Float
}


delta : Signal Float
delta = Time.inSeconds <~ (Time.fps 60)


input : Signal Input
input =
let space' = dropRepeats <| sampleOn delta space
space2' = dropRepeats space
```

```
in Input <~ space' ~ space2' ~ delta
```

The solution ended up being a counter in the game record that counted the number of times it had seen the space pressed and would reset when it saw a false value.

## 4.3 Non-Signal Problems

There were a few problems implementing the game that had nothing to do with functional reactive programming. These were aside from the usual crashes and odd behavior that you can expect to see from a new language.

The Pong tutorial made it seem like it is possible to implement a game by planning everything from the start. In reality, the process was a lot more iterative. There are always new considerations and alterations to each of the four sections of code.

Another problem involves getting the darts to orbit the board. Initially, darts had a pin part a line that extended from the ball in the model. This caused problems when trying to have darts orbit around the board as they would have to rotate. The first solution tried involved grouping the darts when they collide with the board, then rotating the entire object. This did not work as the changes would exist only in the view, and the dart coordinates are used in updates for calculating dart on dart collisions. Rotating the dart by itself using an angle parameter also does not work as this causes the dart to spin about its own axis. The correct solution is to add an angle parameter using $\cos(angle)$ for the x coordinate and $\sin(angle)$ for the y coordinate [12]. The pins of the darts were also removed from the model, and exist only in the view. They rotate properly since we continually redraw them from the center of the ball of the dart to the center of the board.

# CHAPTER 5

# CONCLUSION AND FUTURE WORK

Functional reactive programming provides a cleaner way than event driven programming of modeling input and state updates. This is because of its ability to treat signals the same way as any other variable. This ability is similar to the power functions have in a functional programming language, where functions have first class status.

In the short term, Elm needs to have a more stable language with a lot more features. This includes a more sophisticated type system like Haskell's type classes [13]. This will allow for a greater degree of abstraction and code reuse. A more traditional step debugger would also be helpful even though the time-traveling debugger was useful.

In the long run, Elm might have trouble attracting a large audience because most programmers are unfamiliar with pure functional programming. There are also several libraries like flapjax [15] that can serve programmers that want reactive programming, but don't want to leave JavaScript. It might do well with Haskell and ML users that want a similar language in their browser.

In the future, a cleaner way to ensure no duplicate values for a single input in a record of inputs needs to be found. The given functions for filtering inputs do not work as expected. Looking into the JavaScript Elm outputs and the Haskell Elm is written in might give a clearer explanation why.

More games with a variety of inputs will need to be written. The difference in Signals used between already established games (Pong, aa) will likely only be in the inputs, as the structure for games seems to be robust.

BIBLIOGRAPHY

# BIBLIOGRAPHY


[1] Engineer Bainomugisha, Andoni Lombide Carreton, Tom Van Cutsem, Stijn Mostinckx, and Wolfgang De Meuter. A survey on reactive programming. *ACM Computing Surveys*, 2012.

[2] Haskell Community. Haskell language. `https://www.haskell.org/`, 2015.

[3] Ocaml Community. Ocaml - ocaml. `http://ocaml.org/community/`, 2015.

[4] Antony Courtney, Henrik Nilsson, and John Peterson. The yampa arcade. In *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, pages 7–18. ACM, 2003.

[5] Evan Czaplicki. Elm: Concurrent frp for functional guis. *Senior thesis, Harvard University*, 2012.

[6] Evan Czaplicki. Basics. `http://package.elm-lang.org/packages/elm-lang/core/2.1.0/Basics#|>`, 2015.

[7] Evan Czaplicki. Making pong an intro to games in elm. `http://elm-lang.org/blog/making-pong`, 2015.

[8] Evan Czaplicki. Reactivity. `http://elm-lang.org/guide/reactivity`, 2015.

[9] Evan Czaplicki. Records. `http://elm-lang.org/docs/records`, 2015.

[10] Evan Czaplicki. Why no droprepeats in elm-0.15. `https://github.com/elm-lang/core/issues/200`, 2015.

[11] Conal Elliott and Paul Hudak. Functional reactive animation. In *ACM SIGPLAN Notices*, volume 32, pages 263–273. ACM, 1997.

[12] Gamegur-us. Gamegur-us/aa. `https://github.com/Gamegur-us/aa`, 2015.

[13] Miran Lipovaca. *Learn you a haskell for great good!: a beginner's guide*. no starch press, 2011.

[14] General Adaptive Apps Pty Ltd. aa. `https://play.google.com/store/apps/details?id=com.aa.generaladaptiveapps&hl=en`, 2015.

[15] Leo A Meyerovich, Arjun Guha, Jacob Baskin, Gregory H Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: a programming language for ajax applications. In *ACM SIGPLAN Notices*, volume 44, pages 1–20. ACM, 2009.

# VITA

Bachelor of Science in Computer Science The University of Mississippi (July 2007 - May 2011)

Software Engineer at Notify Corporation (Now Globo Mobile Plc.) (October 2011 - January 2014)

Graduate Assistant at the Department of Outreach (January 2014 - August 2014)

Graduate Assistant at Mississippi Center for Supercomputing Research (September 2014 - December 2015)