My adventure with ELM

by Yan Cui
agenda
I’m **not** an expert on Elm.
A functional reactive language for interactive applications

Functional
Features like immutability and type inference help you write code that is short, fast, and maintainable. Elm makes them easy to learn too.

Reactive
Elm is based on the idea of Functional Reactive Programming. Create highly interactive applications without messy callbacks or a tangle of shared state.

Compatible
Elm compiles to HTML, CSS, and JavaScript. It is easy to use HTML and interop with JS, so it is simple to write part of your application in Elm.
Functional Reactive Programming?
Value over Time
Signal

Value

Time

@theburningmonk
private var arrowKeyUp:Bool;
private var arrowKeyDown:Bool;

private var platform1:Platform;
private var platform2:Platform;
private var ball:Ball;
function keyDown(event: KeyboardEvent): Void {
    if (currentGameState == Paused &&
        event.keyCode == 32) {
        setGameState(Playing);
    } else if (event.keyCode == 38) {
        arrowKeyUp = true;
    } else if (event.keyCode == 40) {
        arrowKeyDown = true;
    }
}
function `keyUp`(event:KeyboardEvent):Void {
    if (event.keyCode == 38) {
        `arrowKeyUp` = false;
    }
    else if (event.keyCode == 40) {
        `arrowKeyDown` = false;
    }
}
function **everyFrame**(event:Event):Void {
    if(currentGameState == Playing){
        if (arrowKeyUp) {
            platform1.y -= platformSpeed;
        }
        if (arrowKeyDown) {
            platform1.y += platformSpeed;
        }
        if (platform1.y < 5) platform1.y = 5;
        if (platform1.y > 395) platform1.y = 395;
    }
}

@theburningmonk
function everyFrame(event:Event):Void {
    if (currentGameState == Playing) {
        if (arrowKeyUp) {
            platform1.y -= platformSpeed;
        }
        if (arrowKeyDown) {
            platform1.y += platformSpeed;
        }
        if (platform1.y < 5) platform1.y = 5;
        if (platform1.y > 395) platform1.y = 395;
    }
}

@theburningmonk
source files

@theburningmonk
## mental model

<table>
<thead>
<tr>
<th>input</th>
<th>state</th>
<th>new state</th>
<th>behaviour</th>
</tr>
</thead>
<tbody>
<tr>
<td>↑</td>
<td>{ x; y }</td>
<td>{ x; y-speed }</td>
<td></td>
</tr>
<tr>
<td>↓</td>
<td>{ x; y }</td>
<td>{ x; y+speed }</td>
<td></td>
</tr>
<tr>
<td>timer</td>
<td>{ x; y }</td>
<td>{ x; y }</td>
<td>draw platform</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>
Imperative

x.f()

mutation

Functional

let y = f(x)

transformation

@theburningmonk
“one thing I’m discovering is that transforming data is easier to think about than maintaining state.”

- Dave Thomas
transformations
simplify problem
decomposition
Press SPACE to start
Use ARROW KEYS to move your platform

Move Up

Move Down

@theburningmonk
type alias Platform = {x:Int, y:Int}
defaultPlatform = {x=5, y=0}

delta = Time.fps 20
input = Signal.sampleOn delta Keyboard.arrows

cap x = max 5 <| min x 395

p1 : Signal Platform
p1 = foldp ($x, y$ s -> {s | y = cap <| s.y + 5*y})
defaultPlatform
input
type alias `Platform` = `{x:Int, y:Int}

`defaultPlatform` = `{x=5, y=0}`

delta = Time.fps 20

`input` = Signal.sampleOn delta Keyboard.arrows

cap x = max 5 <l min x 395

p1 : Signal Platform

p1 = foldp (\{x, y\} s -> {s l y = cap <l s.y + 5*y})

defaultPlatform

input

@theburningmonk
type alias Platform = {x: Int, y: Int}
defaultPlatform = {x=5, y=0}

delta = **Time.fps** 20

input = **Signal.sampleOn** delta **Keyboard.arrows**

cap x = max 5 <$l$ min x 395

p1 : Signal Platform
p1 = foldp (\{x, y\} s -> {s | y = cap <$l$ s.y + 5*y})

defaultPlatform input

@theburningmonk
Keyboard.arrows

UP { x=0, y=1 }
DOWN { x=0, y=-1 }
LEFT { x=-1, y=0 }
RIGHT { x=1, y=0 }

@theburningmonk
type alias Platform = {x:Int, y:Int}
defaultPlatform = {x=5, y=0}

delta = Time.fps 20
input = Signal.sampleOn delta Keyboard.arrows

cap x = max 5 <| min x 395

p1 : Signal Platform
p1 = foldp (\{x, y\} s -> \{s | y = cap <| s.y + 5*y\})
defaultPlatform
input

@theburningmonk
type alias Platform = {x:Int, y:Int}
defaultPlatform = {x=5, y=0}

delta = Time.fps 20
input = Signal.sampleOn delta Keyboard.arrows

cap x = max 5 <| min x 395

p1 : Signal Platform
p1 = foldp (\{x, y\} s -> {s | y = cap <| s.y + 5*y})
defaultPlatform
    input

@theburningmonk
type alias Platform = {x:Int, y:Int}
defaultPlatform = {x=5, y=0}

delta = Time.fps 20
input = Signal.sampleOn delta Keyboard.arrows

cap x = max 5 <| min x 395

p1 : Signal Platform
p1 = foldp (\{x, y\} s -> {s | y = cap <| s.y + 5*y})
defaultPlatform
input

@theburningmonk
type alias Platform = {x:Int, y:Int}
defaultPlatform = {x=5, y=0}

delta = Time.fps 20
input = Signal.sampleOn delta Keyboard.arrows

cap x = max 5 <| min x 395

p1 : Signal Platform
p1 = foldp (\{x, y\} s -> {s | y = cap <| s.y + 5*y})
defaultPlatform
input
type alias Platform = {x:Int, y:Int}
defaultPlatform = {x=5, y=0}

delta = Time.fps 20
input = Signal.sampleOn delta Keyboard.arrows

cap x = max 5 <| min x 395

p1 : Signal Platform
p1 = foldp (\{x, y\} s -> {s | y = cap <| s.y + 5*y})
    defaultPlatform
    input
Static Graphs

Pursuing Reconfigurable Graphs

Dynamic Graphs

- First-order FRP
- Higher-order FRP
- Asynchronous Data Flow
- Infinite Signals
- Finite Event Streams
- Neither!

see also http://bit.ly/1HPM3ul
see also http://bit.ly/1VqL6SF
```javascript
// GameEnemy

var GameEnemy = new Class{
    Extends: GameObject,

    initialize: function () {
        this.x = 5;
        this.y = 14;
        this.direction = -1;
        this.stompTime = 0;
    },

    tick: function (dt, commands) {
        var isStumped = (this.stompTime > 0);
        if (isStumped) {
            this.stompTime -= dt;
        }

        if (isStumped) {
            this.x += this.direction * dt = 0.0;
        }

        if (this.x < 4.0) {
            this.direction = 1;
        } else if (this.x > 5.0) {
            this.direction = -1;
        }

        stomp: function () {
            this.stompTime = 2.0;
        }
    }
}
```
Idea → See in Action

@theburningmonk
Computer programming

Learn the fundamentals of programming with the popular JavaScript language and ProcessingJS library. Write your own programs and share them, explore programs made by others, and learn from each other’s programs!

Learn Programming

Create Program

Documentation

Intro to JS: Drawing & Animation
In these tutorials, you'll learn how to use the JavaScript language and the ProcessingJS library to create fun drawings and animations. If you've never programmed before, start here to learn how!

Advanced JS: Games & Visualizations
Now that you know how to program in JavaScript and make basic drawings and animations, how could you use that knowledge to make games and visualizations?

Advanced JS: Natural Simulations
Learn how to use JavaScript, ProcessingJS, and mathematical concepts to simulate nature in your programs. These tutorials
LIGHT TABLE
the next generation code editor

@theburningmonk
let key pressed dt keys
keys
let
ng.watch "mario"

Model -> Model

= 0 && mario.vy == 0 then { mario | vy + dt/4 else 0

Model -> Model

= mario.y > 0 then mario.vy - dt/4 else 0

Model -> Model

=
TodoMVC Benchmark

- Backbone: 951 milliseconds
- Ember: 2191 milliseconds
- Angular: 1713 milliseconds
- React: 1381 milliseconds
- Om: 685 milliseconds
- Mercury: 262 milliseconds
- Elm: 399 milliseconds

Average time in milliseconds over 16 runs (lower is better)

Firefox 30 on MacBook Air with OSX 0.10.9.4

see also http://bit.ly/1wV46XS
Elm Basics
add x y = x + y
add : Int -> Int -> Int
add x y = x + y
add : Int -> Int -> Int
add x y = x + y
calcAngle start end =
  let distH = end.x - start.x
  distV = end.y - start.y
  in atan2 distV distH
calcAngle start end =
    let distH = end.x - start.x
    distV = end.y - start.y
    in atan2 distV distH
calcAngle start end =
  let distH = end.x - start.x
  distV  = end.y - start.y
  in atan2 distV distH
multiply $x \cdot y = x \times y$
triple $= multiply 3$
multiply $x \ y = x \ * \ y$
triple $= \ multiply \ 3$
double list = List.map (\x -> x * 2) list
double list = List.map ((*) 2) list
tuple1 = (2, "three")
tuple2 = (2, "three", [4, 5])
tuple4 = (,) 2 "three"
tuple5 = (,,) 2 "three" [4, 5]
\[ x = \{ \text{age}=42, \text{name}="foo" \} \]
lightweight, labelled data structure
x.age -- 42
x.name -- “foo”
x.age       --  42
x.name      -- "foo"
.age x      --  42
.name x     -- "foo"
-- clone and update

\[ y = \{ x \mid \text{name} = "bar" \} \]
type alias Character =
{ age : Int, name : String }
type alias Named a = { a | name : String }  
type alias Aged a = { a | age : Int }
lady : Named ( Aged {} )
lady = { name="foo", age=42 }
lady : Named ( Aged

lady = { name="foo", age=42 }
lady : Named (Aged {})
lady = { name="foo", age=42 }
lady : Named ( Aged {} )
lady = { name="foo", age=42 }
getName : Named x -> String
getName { name } = name
getName : Named x -> String
getName { name } = name

getName lady -- "foo"
type Tree =
| Leaf
| Branch Tree
aka.
“sums-and-products”
data structures
sums :
choice between variants of a type

```haskell
type Tree =
  Leaf
  | Branch Tree
```
products :
tuple of types

type Tree =
  | Leaf
  | Branch Tree
“...a clean design is one that supports **visual thinking** so people can meet their informational needs with a minimum of conscious effort.”

- Daniel Higginbotham

(www.visualmess.com)
Whilst talking with an ex-colleague, a question came up on how to implement the Stable Marriage problem using a message passing approach. Naturally, I wanted to answer that question with Erlang!

Let's first dissect the problem and decide what processes we need and how they need to interact with one another.

The stable marriage problem is commonly stated as: Given n men and n women, where each person has ranked all members of the opposite sex with a unique number between 1 and n in order of preference, marry the men and women together such that there are no two people of opposite sex who would both rather have each other than their current partners. If there are no such people, all the marriages are “stable”. (It is assumed that the participants are binary gendered and that marriages are not same-sex).

From the problem description, we can see that we need:
* a module for man
* a module for woman
* a module for orchestrating the experiment

In terms of interaction between the different modules, I imagined something along the lines of…

@theburningmonk
public void DoSomething(int x, int y)  
{  
    Foo(y,  
        Bar(x,  
            Zoo(Monkey())));  
}
public void DoSomething(int x, int y)
{
    Foo(y,
        Bar(x,
            Zoo(Monkey())));
}
Whilst talking with an ex-colleague, a question came up on how to implement the Stable Marriage problem using a message passing approach. Naturally, I wanted to answer that question with Erlang!

Let's first dissect the problem and decide what processes we need and how they need to interact with one another.

The stable marriage problem is commonly stated as: Given n men and n women, where each person has ranked all members of the opposite sex with a unique number between 1 and n in order of preference, marry the men and women together such that there are no two people of opposite sex who would both rather have each other than their current partners. If there are no such people, all the marriages are stable (it is assumed that the participants are binary gendered and that marriages are not same-sex).

From the problem description, we can see that we need:
* a module for man
* a module for woman
* a module for orchestrating the experiment

In terms of interaction between the different modules, I imagined something along the lines of…

```
public void DoSomething(int x, int y)
{
    Foo(y, Bar(x, Zoo(Monkey())));
}
```

@theburningmonk
“...a clean design is one that supports visual thinking so people can meet their informational needs with a minimum of conscious effort.”
drawCircle x y radius =
circle radius |> filled (rgb 150 150 150) |> alpha 0.5 |> move (x, y)
drawCircle x y radius =
circle radius
|> filled (rgb 150 170 150)
|> alpha 0.5
|> move (x, y)
drawCircle x y radius =

circle radius
|> filled (rgb 150 170 150)
|> alpha 0.5
|> move (x, y)
drawCircle x y radius =

circle radius

| > filled (rgb 150 170 150)

filled : Color -> Shape -> Form

| > alpha 0.5

| > move (x, y)
drawCircle x y radius =

  circle radius
  |> filled (rgb 150 170 150)
  |> alpha 0.5
  |> move (x, y)
drawCircle : Int -> Int -> Float -> Form
drawCircle x y =
circle
  >>= filled (rgb 150 170 150)
  >>= alpha 0.5
  >>= move (x, y)
drawCircle x y =
circle
  >> filled (rgb 150 170 150)
  >> alpha 0.5
  >> move (x, y)
circle : Float -> Shape
drawCircle x y =
  (Float -> Shape)
  >> filled (rgb 150 170 150)
  >> alpha 0.5
  >> move (x, y)
drawCircle x y =
  (Float -> Shape)
  >> filled (rgb 150 170 150)
  >> alpha 0.5
  >> move (x, y)

@theburningmonk
drawCircle x y =

(Float -> Shape)

>> (Shape -> Form)

>> alpha 0.5

>> move (x, y)
drawCircle x y =
(FLOAT -> SHAPE)
>> (SHAPE -> FORM)
>> alpha 0.5
>> move (x, y)
drawCircle x y =
  (Float -> Shape)
  >>= (Shape -> Form)
  >>= alpha 0.5
  >>= move (x, y)
drawCircle x y =
  (Float -> Shape)
  >> (Shape -> Form)
  >> alpha 0.5
  >> move (x, y)
drawCircle x y =
  (Float -> Form)
  >> alpha 0.5
  >> move (x, y)
drawCircle x y =
  (Float -> Form)
  >> (Form -> Form)
  >> move (x, y)
drawCircle x y =
  (Float -> Form)
>> (Form -> Form)
>> move (x, y)
drawCircle x y =
(FLOAT -> FORM)
>> move (x, y)
drawCircle x y =
  (Float -> Form)
>> (Form -> Form)
drawCircle x y =
(FLOAT -> FORM) >> (FORM -> FORM)
drawCircle x y = (Float -> Form)
drawCircle : Int -> Int -> (Float -> Form)
greet name =
  case name of
    "Yan"  -> "hi, theburningmonk"
    _     -> "hi, “ ++ name
greet name =
  case name of
    "Yan"  -> "hi, theburningmonk"
    _      -> "hi, " ++ name
Mouse.position
Mouse.clicks
Mouse.isDown
...

@theburningmonk
Window.dimension
Window.width
Window.height

@theburningmonk
Time.every
Time.fps
Time.timestamp
Time.delay
...

@theburningmonk
Mouse.position : Signal (Int, Int)
Mouse.position : Signal (Int, Int)
Mouse.position : Signal (Int, Int)

(10, 23) → (3, 16) → (8, 10) → (12, 5) → (18, 3)
Keyboard.presses : Signal Int
Keyboard.

presses : Signal Int

H — E — L — L — O — space →
Keyboard.presses : Signal Int

<table>
<thead>
<tr>
<th>H</th>
<th>E</th>
<th>L</th>
<th>L</th>
<th>O</th>
<th>space</th>
</tr>
</thead>
<tbody>
<tr>
<td>72</td>
<td>69</td>
<td>76</td>
<td>76</td>
<td>79</td>
<td>32</td>
</tr>
</tbody>
</table>

@theburningmonk
map : (a -> b) -> Signal a -> Signal b
Signal of num of pixels in window
map (\(w, h\) -> \(w\times h\)) Window.dimensions
map \((w, h) \rightarrow w \times h\) Window.dimensions

Signal (Int, Int)

map (\(w, h\) \rightarrow w*h) Window.dimensions

(Int, Int) \rightarrow Int

@theburningmonk
map (\( (w, h) \rightarrow w \times h \)) Window.dimensions

\[
signal \ int
\]

\[
(\text{int, int}) \rightarrow \text{int}
\]
map (\(w, h\))
map2 : (a -> b -> c)
   -> Signal a
   -> Signal b
   -> Signal c
a -> b -> (a, b)

map2 (,) Window.width Window.height

Signal Int  

Signal Int Int
\( \text{Int} \rightarrow \text{Int} \rightarrow (\text{Int}, \text{Int}) \)

\text{map2} \ (,) \ \text{Window.width} \ \text{Window.height}

\text{Signal} \ \text{Int} \quad \text{Signal} \ \text{Int}

@theburningmonk
map3 : (a -> b -> c -> d)
    -> Signal a
    -> Signal b
    -> Signal c
    -> Signal d
map3 (,,) signalA signalB signalC
map4 : ....
map5 : ....
map6 : ....
map7 : ....
map8 : ....

@theburningmonk
foldp : (a -> b -> b) -> b
    -> Signal a
    -> Signal b
@theburningmonk
foldp : (a -> b -> b) -> b
      -> Signal a
      -> Signal b
Mapping function

foldp : (a -> b -> b) -> b

Input signal

- Signal a
- Signal b
foldp : (a -> b -> b) -> b

Initial value: b

Input signal: Signal a -> Signal b

Mapping function
foldp (\_ n -> n + 1) 0 Mouse.clicks
foldp (\_ n -> n + 1) 0 Mouse.clicks
foldp (\_ n -> n + 1) 0 Mouse.clicks
foldp (\_ n -> n + 1) 0 Mouse.clicks
UP \{ x=0, y=1 \}
DOWN \{ x=0, y=-1 \}
LEFT \{ x=-1, y=0 \}
RIGHT \{ x=1, y=0 \}
merge : Signal a -> Signal a -> Signal a
mergeMany : List (Signal a) -> Signal a
...
Js Interop, WebGL, HTML layout, dependency management, etc.
Demo
Press SPACE to start.
direction
direction
YUM YUM YUM!
direction
+1 segment
import Graphics.Element
import Keyboard
import Signal
import Signal (..)
import Text
import Time
import Window

type alias UserInput = {}

userInput : Signal UserInput
userInput = Signal.constant {}

type alias Input =
  { timeDelta : Float
  , userInput : UserInput
  }

type alias GameState = {}

defaultGame : GameState
defaultGame = {} 

stepGame : Input \rightarrow GameState \rightarrow GameState
stepGame {timeDelta, userInput} gameState = gameState

display : (Int,Int) \rightarrow GameState \rightarrow Element
display (w,h) gameState = Text.asText gameState

delta : Signal Float
delta = Time.fps 30

input : Signal Input
input = Signal.sampleOn delta (Input<<delta~userInput)

gameState : Signal GameState
gameState = Signal.foldp stepGame defaultGame input

main : Signal Element
main = display<>Window.dimensions~gameState
import Graphics.Element
import Keyboard
import Signal
import Signal (..)
import Text
import Time
import Window

type alias UserInput = {}  

userInput : Signal UserInput
userInput = Signal.constant {}

type alias Input = 
    { timeDelta : Float
    , userInput : UserInput
    }

type alias GameState = {}  

defaultGame : GameState
defaultGame = {}  

stepGame : Input -> GameState -> GameState
stepGame {timeDelta, userInput} gameState = gameState

display : (Int,Int) -> GameState -> Element
display (w,h) gameState = Text.asText gameState

delta : Signal Float
delta = Time.fps 30

input : Signal Input
input = Signal.sampleOn delta (Input<<delta~userInput)

gameState : Signal GameState
gameState = Signal.foldp stepGame defaultGame input

main : Signal Element
main = display<<Window.dimensions~gameState
import Graphics.Element
import Keyboard
import Signal
import Signal(..)
import Text
import Time
import Window

type alias UserInput = {};

userInput : Signal UserInput
userInput = Signal.constant {}

type alias Input = {
    timeDelta : Float,
    userInput : UserInput
}

type alias GameState = {};

defaultGame : GameState
defaultGame = {}

stepGame : Input -> GameState -> GameState
stepGame {timeDelta, userInput} gameState = gameState

display : (Int,Int) -> GameState -> Element
display (w,h) gameState = Text.asText gameState

delta : Signal Float
delta = Time.fps 30

input : Signal Input
input = Signal.sampleOn delta (Input<-delta->userInput)

gameState : Signal GameState
gameState = Signal.foldp stepGame defaultGame input

main : Signal Element
main = display<-Window.dimensions->gameState
import Graphics.Element 
import Keyboard
import Signal
import Signal (..)
import Text
import Time
import Window

type alias UserInput = {} 

userInput : Signal UserInput
userInput = Signal.constant {} 

type alias Input = 
{ timeDelta : Float 
, userInput : UserInput  
}

type alias GameState = {} 

defaultGame : GameState
defaultGame = {}

stepGame : Input -> GameState -> GameState
stepGame {timeDelta, userInput} gameState = gameState

display : (Int,Int) -> GameState -> Element
display (w,h) gameState = Text.asText gameState

delta : Signal Float
delta = Time.fps 30

input : Signal Input
input = Signal.sampleOn delta (Input<<delta~userInput)

gameState : Signal GameState
gameState = Signal.foldp stepGame defaultGame input

main : Signal Element
main = display~Window.dimensions~gameState
import Graphics.Element 
import Keyboard 
import Signal 
import Signal (...) 
import Text 
import Time 
import Window 

type alias UserInput = {} 

userInput : Signal UserInput  
userInput = Signal.constant {} 

type alias Input =  
  { timeDelta : Float,  
    userInput : UserInput  
  } 

type alias GameState = {} 

defaultGame : GameState  
defaultGame = {} 

stepGame : Input \rightarrow GameState \rightarrow GameState  
stepGame {timeDelta, userInput} gameState = gameState  

display : (Int,Int) \rightarrow GameState \rightarrow Element  
display (w,h) gameState = Text.asText gameState  

delta : Signal Float  
delta = Time.fps 30  

input : Signal Input  
input = Signal.sampleOn delta (Input<<delta~userInput)  

gameState : Signal GameState  
gameState = Signal.foldp stepGame defaultGame input  

main : Signal Element  
main = display<<Window.dimensions~gameState
import Graphics.Element ..
import Keyboard
import Signal
import Signal ..
import Text
import Time
import Window

type alias UserInput = {}

userInput : Signal UserInput
userInput = Signal.constant {}

type alias Input =
  { timeDelta : Float
  , userInput : UserInput
  }

type alias GameState = {}

defaultGame : GameState
defaultGame = {}

stepGame : Input \rightarrow GameState \rightarrow GameState
stepGame {timeDelta, userInput} gameState = gameState

display : (Int,Int) \rightarrow GameState \rightarrow Element
display (w,h) gameState = Text.asText gameState

delta : Signal Float
delta = Time.fps 30

input : Signal Input
input = Signal.sampleOn delta (Input<>delta<>userInput)

gameState : Signal GameState
gameState = Signal.foldp stepGame defaultGame input

main : Signal Element
main = display<>Window.dimensions<>gameState
Demo

@theburningmonk
Snake
github.com/theburningmonk/elm-snake

Missile Command
github.com/theburningmonk/elm-missile-command

@theburningmonk
Guide the ball safely to its goal (green).
Architecture in Elm: bit.ly/1l0neiE

Kris Jenkins - Elm is Coming: bit.ly/1X1ynXb

NoRedInk - Introducing Elm to a JS web app: bit.ly/1QR3Ruk

Richard Feldman - Elm in production: bit.ly/1lxy4m8

Even Czaplicki - Controlling time and space: bit.ly/1HPM3ul

Evan Czaplicki - User focused design in Elm: bit.ly/216AM2l

@theburningmonk
@theburningmonk

JUST EAT is hiring :-)  
http://tech.just-eat.com/jobs

@theburningmonk