



Functional Reactive Programming

for Natural User Interface

“I have no special talents. I am only passionately curious.”

- Albert Einstein



Agenda

What is Functional Reactive Programming – FRP vs RP

FRP foundations and motivations

FRP implemented in F# with Code Samples – Original Paper

FRP implemented in F# with Code Samples – Modern Paper

Natural User Interface with Leap Motion in Action

Functional Reactive Programming

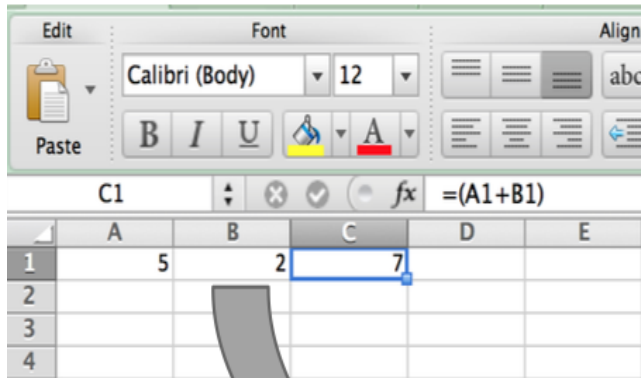
Functional Reactive Programming

- declarative
- functions as values
- side-effects free
- referential transparency
- immutable
- composition

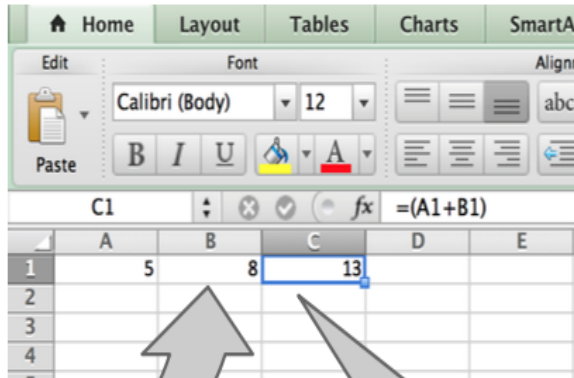
Functional Reactive Programming

```
sample.TweetReceived
|> Event.filter(fun tweet -> tweet.lang == "en")
|> Event.choose(fun tweet -> tweet.Text)
|> Event.map(fun text ->
    if Set.contains text positive then 1
    elif Set.contains text negative then -1
    else 0)
|> Event.scan (+)
|> Event.add(fun n -> printfn "Mood=%d" n)
```

SpreadSheet == Mother of All Reactive Programming



	A	B	C	D	E
1	5	2	7		
2					
3					
4					



	A	B	C	D	E
1	5	8	13		
2					
3					
4					

Result cell automatically updated whenever any input cell changes

Functional Reactive Programming

**Composable Dynamic evolving
values over time**





Functional Reactive ANimation

Functional Reactive Animation

Conal Elliott
Microsoft Research
Graphics Group
conal@microsoft.com

Paul Hudak
Yale University
Dept. of Computer Science
paul.hudak@yale.edu

Abstract

Fran (Functional Reactive Animation) is a collection of data types and functions for composing richly interactive, multi-media animations. The key ideas in *Fran* are its notions of *behaviors* and *events*. Behaviors are time-varying, reactive values, while events are sets of arbitrarily complex conditions, carrying possibly rich information. Most traditional values can be treated as behaviors, and when images are thus treated, they become animations. Although these no-

- capturing and handling sequences of motion input events, even though motion input is conceptually continuous;
- time slicing to update each time-varying animation parameter, even though these parameters conceptually vary in parallel; and

By allowing programmers to express the “what” of an interactive animation, one can hope to then automate the “how” of its presentation. With this point of view, it should

Functional Reactive Programming adoption



- ▣ *Graphical User Interfaces (GUI)*
- ▣ *Digital Music*
- ▣ *Robotics*
- ▣ *Graphical Animation*
- ▣ *Sound Synthesis*
- ▣ *Virtual Reality Environments*
- ▣ *Games*

FRP becomes Main-Stream



FRP has evolved in a number of directions and into different concrete implementations



What is Functional Reactive Programming



“FRP is about handling time-varying values like they were regular values.”

- Haskell Wiki

Functional Reactive Programming is:

- Temporally continuous (*Natural & Composable*)
- Denotative (*Elegant & Rigorous*)

Denotational Semantics



Denotational Semantics map each part of a program to a mathematical object (denotation), which represents the meaning of the program in question.

Consider the definition of a factorial function

`fact n = product [1..n]`

```
int fact(int n) {  
    int i;  
    int result = 1;  
    for (i = 2; i <= n; ++i)  
        result *= i;  
    return result;  
}
```

Denotational Semantics = Simple Design



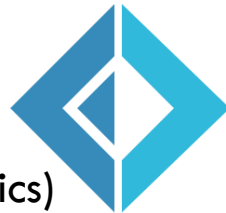
Denotational Semantics map each part of a program to a mathematical object (denotation), which represents the meaning of the program in question.

Denotational Semantics properties

- *leads to simple design*
- *emphasizes declarative programming style (What vs How)*
- *uses math to prove a property of a program*
- *proves that compositionality holds for all building blocks*

Foundation of FRP – Time

(precise & simple semantics)



```
type Time = { Time : float }
```

A hand is holding a large, ornate clock face with Roman numerals. Inside this clock face is a smaller, similar clock face, which in turn contains an even smaller one, creating a recursive effect. The text "Continuous Time" is overlaid in red on the top half of the image.

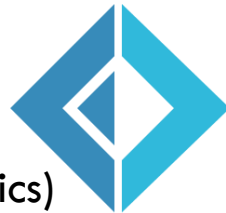
Continuous Time

Discrete Time

Virtual Time

Foundation of FRP - Behavior

(precise & simple semantics)



```
type Time = { Time : float }
```

```
type 'a Behavior = Behavior of (Time -> 'a)
```




Foundation of FRP - Behavior

```
type Time = { Time : float }
```

```
type 'a Behavior = Behavior of (Time -> 'a)
```

```
// The time itself
```

```
let time = Behavior (fun t -> t)
```

```
// Behavior constant over time
```

```
let constantBeh = Behavior (fun _ -> "Hello FRP!")
```

```
// Behavior that increase at 2.5 the rate of time
```

```
let incrSpeedBeh = Behavior (fun t -> t * 7.5)
```

Behavior API – Original Implementation



```
let lift0 'a = Behavior 'a
```

```
let lift1 ('a -> 'b) = Behavior 'a -> Behavior 'b
```

```
let lift2 ('a -> 'b -> 'c) =  
    Behavior 'a -> Behavior 'b -> Behavior 'c
```

```
let map f('a -> 'b) (Behavior(fv)) : Behavior 'b =  
    Behavior(fun t -> f (fv t))
```

Behavior API – Original Implementation



```
let time = Behavior (fun t -> t)

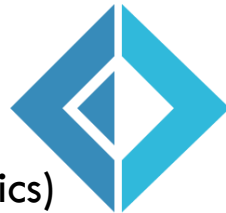
let squared = time |> map (fun t -> t * t)

let time7_5 = lift1 ((* 7.5) time)

let createBehavior f:(Time -> 'a) = (lift1 f) time
```

Foundation of FRP - Event

(precise & simple semantics)



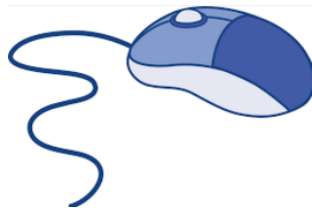
```
type Time = { Time : float }
```

```
type 'a Behavior = Behavior of (Time -> 'a)
```

```
type 'a Event = Event of [(Time * 'a option)] - no decreasing time
```

```
// When the Event passes 3 secs increase its speed  
let event = Event (fun t -> if (t > 3.) then Some(t*2.5) else None)
```

FRP - Mouse Position



Event Based view

`MouseMovedEvent` (position: `Position`)



FRP view - at any point in time represents the current mouse position

`mousePosition` = `Behavior` [`Position`]

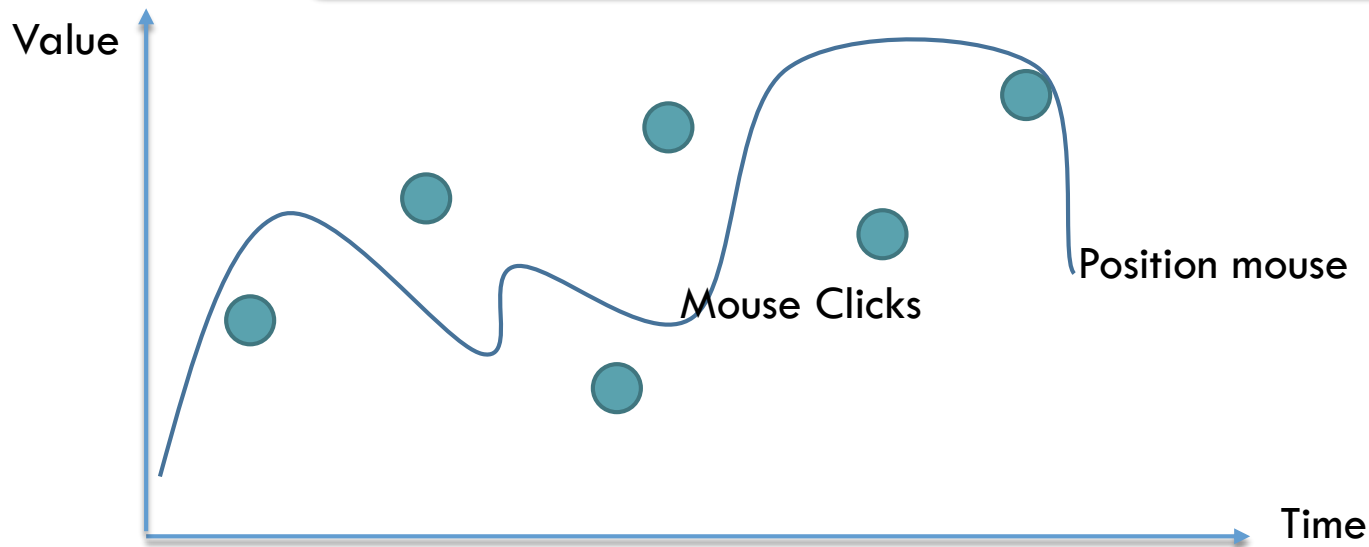
```
inRectangleBeh(ul: Position , lr:Position) : Behavior [bool] =  
    let position = mousePosition()  
    Behavior [ul <= position && position <= lr]
```



Foundation of FRP - Behavior

```
type 'a Behavior = Behavior of (Time -> 'a)
```

```
type 'a Event = Event of [Time -> 'a]
```



“So, what is FRP? You could have invented it yourself, start with these ideas:”

<http://stackoverflow.com/questions/1028250/what-is-functional-reactive-programming> - Conal Elliot

□ *Temporal modeling*

- *Composable Behavior first class values*

□ *Event modeling*

- *Composable Event first class values*

□ *Declarative reactivity*

- *Semantic in terms of temporal composition*

□ *Polymorphic media*

- *Set of combinators applicable to any types of time-varying values*

DEMO

A 3D white figure, resembling a mannequin or a simple human model, is standing and pointing its right arm towards a large, light gray rectangular screen. The screen displays the word "Demo" in a dark red, serif font. Overlaid on the entire scene is a large, red, rectangular stamp with rounded corners and a distressed, ink-like texture. The word "DEMO" is written in a bold, red, serif font within this stamp, tilted slightly upwards from left to right. The background is plain white.

Push-Pull Functional Reactive Programming



Push-Pull Functional Reactive Programming

Conal Elliott

LambdaPix

conal@conal.net

Abstract

Functional reactive programming (FRP) has simple and powerful semantics, but has resisted efficient implementation. In particular, most past implementations have used demand-driven sampling, which accommodates FRP's continuous time semantics and fits well with the nature of functional programming. Consequently, values are wastefully recomputed even when inputs don't change, and reaction latency can be as high as the sampling period.

more composable than their finite counterparts, because they can be scaled arbitrarily in time or space, before being clipped to a finite time/space window.

While FRP has simple, pure, and composable semantics, its efficient implementation has not been so simple. In particular, past implementations have used demand-driven (pull) sampling of reactive behaviors, in contrast to the data-driven (push) evaluation typically used for reactive systems, such as GUIs. There are at least two strong reasons for choosing pull over push for FRP:

Chains of simple phases - Reactivity



```
type 'a Behavior =  
    Behavior of (Time -> 'a * ReactBeh<'a>))
```

```
and 'a ReactBeh = unit -> 'a Behavior
```



Chains of simple phases

```
type 'a Behavior = Behavior of (Time -> 'a * ReactBeh<'a>)
and 'a ReactBeh = unit -> 'a Behavior
```

```
let rec pureBeh value = Behavior(fun time ->
    (value, fun () -> pureBeh value))
```

```
let rec timeBeh = Behavior(fun time ->
    (time, fun () -> timeBeh ))
```



Chains of simple phases

```
type 'a Event = Event of  
    (Time -> Option<'a> * ReactEvent<'a>)  
and 'a ReactEvent = unit -> 'a Event
```

```
let rec pureEvt value = Event(fun time ->  
    (Some(value), fun () -> pureEvt value))
```



FRP Behavior can compose

```
fmap :: ('a -> 'a) -> Behavior 'a -> Behavior 'a
```

```
pure :: 'a -> Behavior 'a
```

```
(<*>) :: Behavior ('a -> 'a) -> Behavior 'a -> Behavior 'a
```

- Less learning and more leverage
- Specifications and laws for “free”



FRP Behavior can compose

```
fmap :: ('a -> 'a) -> Behavior 'a -> Behavior 'a
```

```
pure :: 'a -> Behavior 'a
```

```
(<*>) :: Behavior ('a -> 'a) -> Behavior 'a -> Behavior 'a
```

```
type Position = Position of (float*float)
```

```
let inRectangleBeh (ul:Position, lr:Position) : bool Behavior =  
  pureBeh (fun (position:Position) ->  
    if ul <= position && lr <= position then true  
    else false) <*> mousePositionBeh // Position Behavior
```



FRP Event API

`never :: 'a Event`

`(.|.) :: 'a Event -> 'a Event -> 'a Event`

`whenEvent :: bool Behavior -> unit Event`

`whileEvent :: bool Behavior -> unit Event`

`(.&.) :: 'a Event -> 'b Event -> ('a * 'b) Event`

`(=>>) :: 'a Event -> ('a -> 'b) -> 'b Event`



FRP – Combinators

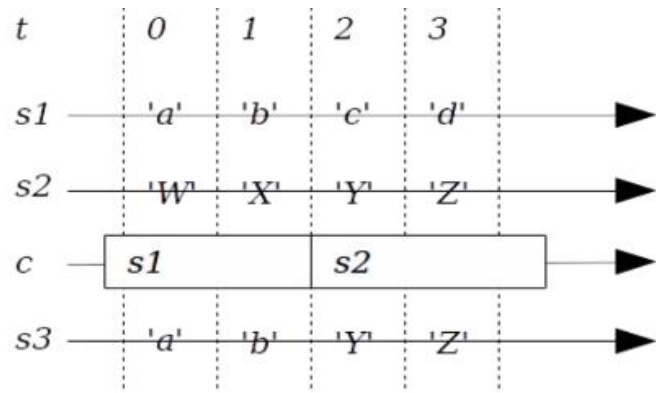
```
type Event ::
// ('a -> 'b) -> Event 'a -> Event 'b
let map(f : 'a -> 'b) : Event<'b> = // ...
// ('a -> bool) -> Event 'a
let filter(f : 'a -> bool) : Event<'a> = // ...
// (Event 'a * Event 'a) -> Event 'a
let merge(ea : Event<'a>, eb : Event<'a>) : Event<'a> = // ...
let (.|. ) = merge
// 'a -> Event<'a -> 'a> -> Behavior 'a
let accum (value:'a) (evt:Event<'a->'a>) : Behavior<'a> = // ...
```




FRP – Behavior switch

```
type Behavior
// Behavior 'a -> Event<Behavior<'a>> -> Behavior<'a>
let switchBeh (beh:Behavior<'a>) (evt:Event<Behavior<'a>>) = //..
```

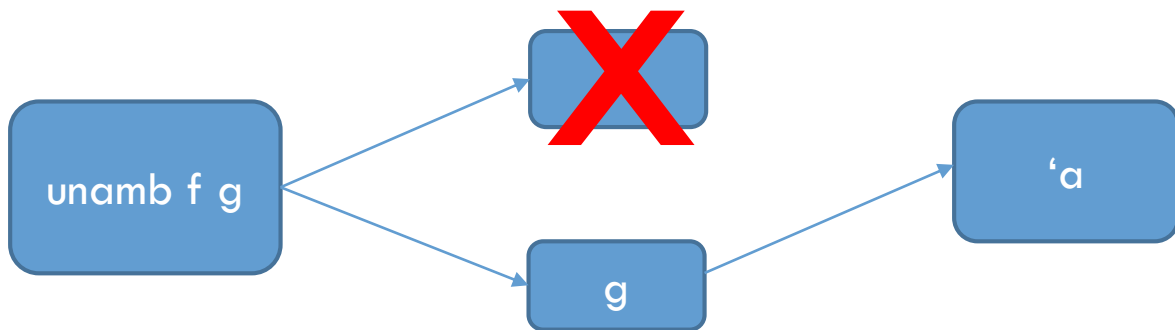
```
let s1 = MkStream [([0], 'a'), ([1], 'b'), ([2], 'c')]
let s2 = MkStream [([0], 'W'), ([1], 'X'), ([2], 'Y')]
// hold :: 'a -> 'a Event -> 'a Behavior
let c = hold s1 (MkStream[([1], s2)]) [0]
let s3 = Switch c
```





Unamb - *Unambiguous choice operator*

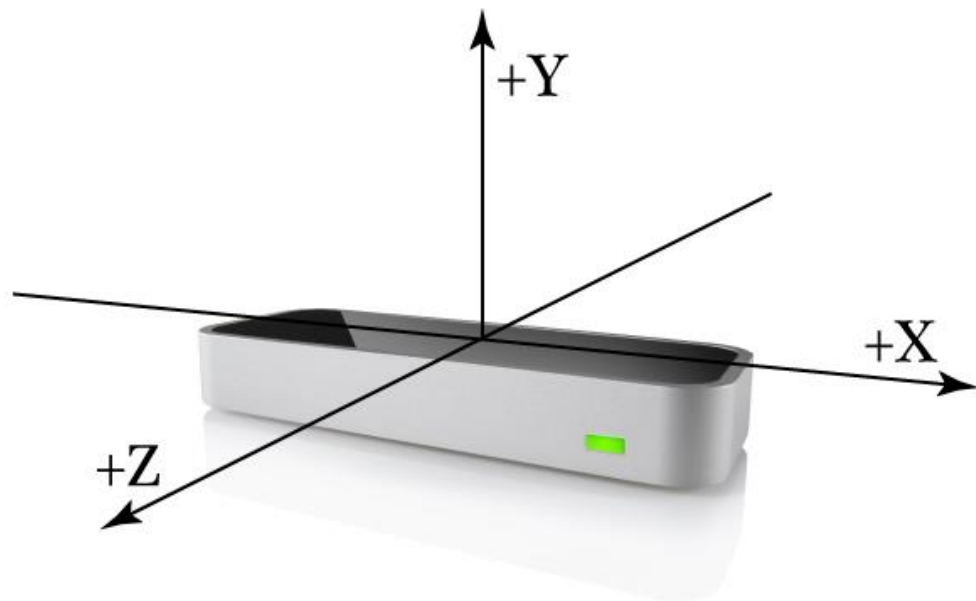
```
let unamb (f:unit -> 'a) (g:unit -> 'a) : 'a =
```



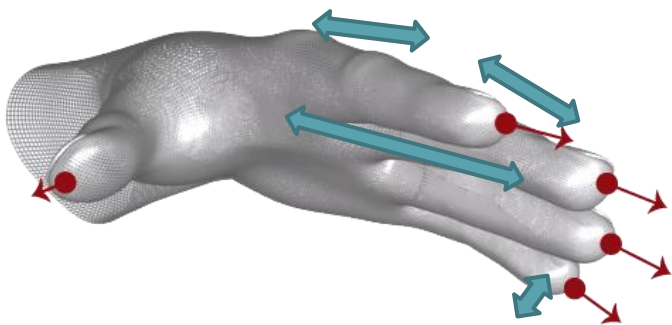
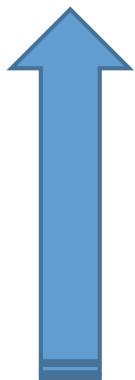
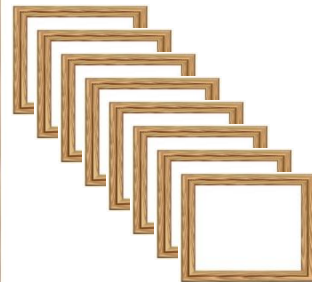
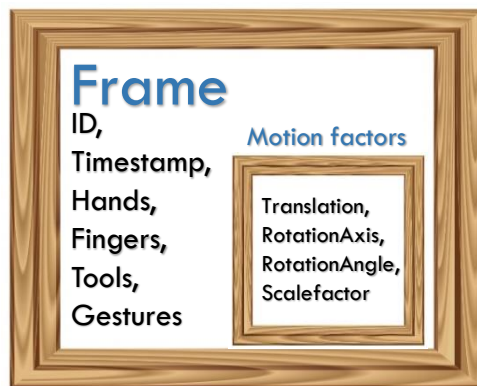
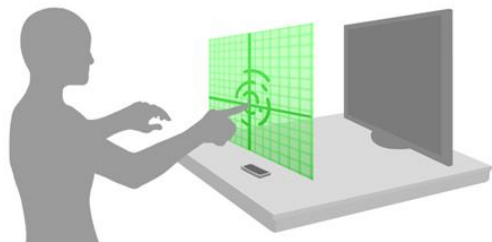
DEMO

A 3D white figure, resembling a mannequin or a simple human model, is standing and pointing its right arm towards a large, light gray rectangular screen. The screen displays the word "Demo" in a dark red, serif font. Overlaid on the entire scene is a large, red, distressed-style stamp that reads "DEMO" in a bold, serif font. The stamp is tilted slightly upwards from left to right and has a rough, ink-like texture.

Leap Motion Sensor



"In just one hand, you have 29 bones, 29 joints, 123 ligaments, 48 nerves, and 30 arteries. That's sophisticated, complicated, and amazing technology (times two). Yet it feels effortless. The Leap Motion Controller has come really close to figuring it all out."



```
Hand has 5 fingers, average finger tip position: (8.05207, 69.3328, -17.8252)
Frame id: 2222, timestamp: 10401134, hands: 1, fingers: 5, tools: 0, gestures: 0

Hand has 5 fingers, average finger tip position: (8.29357, 69.3763, -17.896)
Frame id: 2226, timestamp: 10419846, hands: 1, fingers: 5, tools: 0, gestures: 0

Hand has 5 fingers, average finger tip position: (8.44423, 69.4465, -17.9379)
Frame id: 2230, timestamp: 10438557, hands: 1, fingers: 5, tools: 0, gestures: 0

Hand has 5 fingers, average finger tip position: (8.4602, 69.4943, -17.9635)
Frame id: 2234, timestamp: 10457268, hands: 1, fingers: 5, tools: 0, gestures: 0

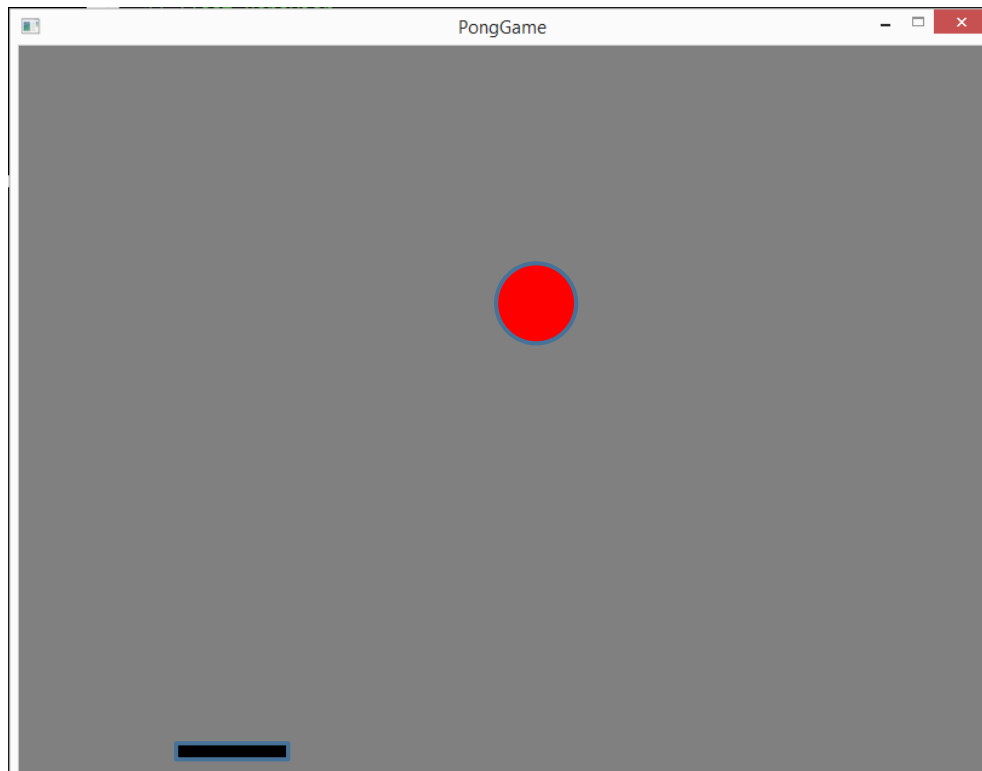
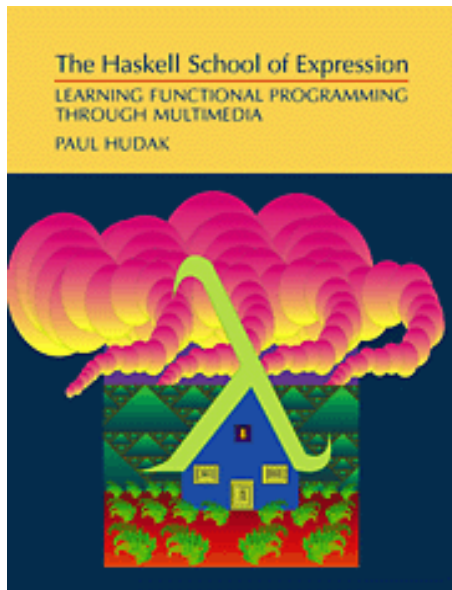
Hand has 5 fingers, average finger tip position: (8.36989, 69.5165, -17.9892)
Frame id: 2238, timestamp: 10475979, hands: 1, fingers: 5, tools: 0, gestures: 0

Hand has 5 fingers, average finger tip position: (8.19734, 69.4871, -18.0453)
Frame id: 2242, timestamp: 10494690, hands: 1, fingers: 5, tools: 0, gestures: 0

Hand has 5 fingers, average finger tip position: (7.98751, 69.4126, -18.1288)
Frame id: 2246, timestamp: 10513402, hands: 1, fingers: 5, tools: 0, gestures: 0

Hand has 5 fingers, average finger tip position: (7.73525, 69.3156, -18.2251)
Frame id: 2250, timestamp: 10532113, hands: 1, fingers: 5, tools: 0, gestures: 0
```

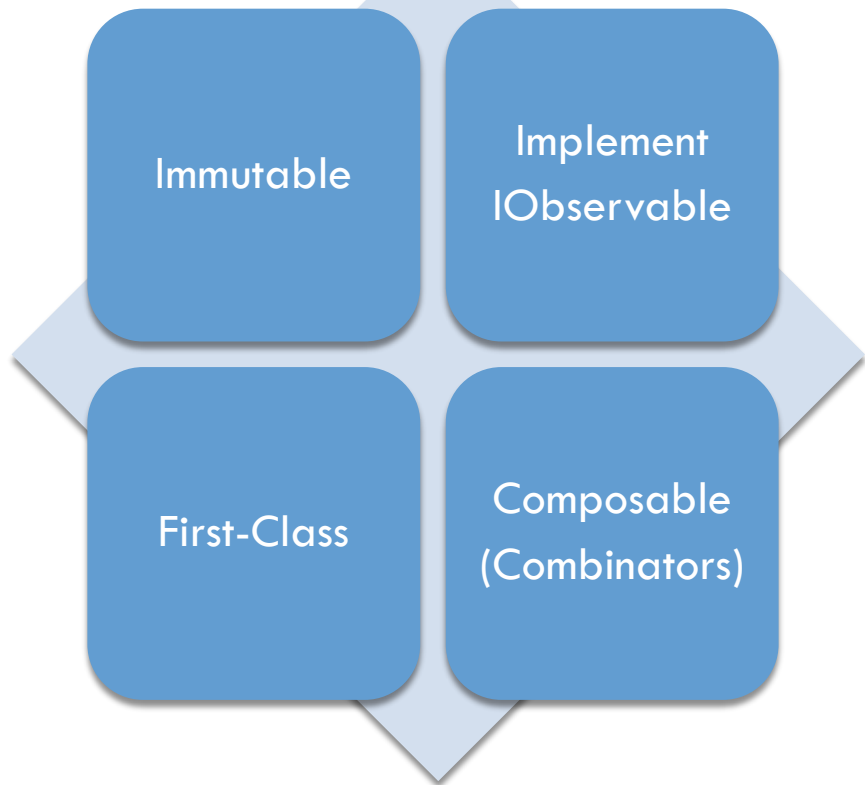
Paddle-ball Game with Leap Motion



Event Processing in F#



SAY GOOD BYE TO THE CALLBACK HELL



Data Processing in F#



```
let myList = [1;2;3]
```

```
let myList = List.Cons(1, List.Cons(2, List.Cons(3,  
List.Empty)))
```


Data Processing in F#



```
let myList = [1;2;3]
```

```
let myList = List.Cons(1, List.Cons(2, List.Cons(3, List.Empty)))
```

```
let rec filter lst pred acc =  
    match lst with  
    | [] -> acc  
    | h::t -> if pred h then filter t pred (h::acc)  
               else filter t pred acc  
filter myList (fun x -> x % 2 = 0) []
```

Data Processing in F#



```
let myList = [1;2;3]
```

```
let myList = List.Cons(1, List.Cons(2, List.Cons(3, List.Empty)))
```

```
myList |> List.filter(fun x -> x % 2 = 0)
```

```
myList |> List.map(fun x -> x * 2)
```

Data Processing in F#



```
let maxLen (text:string list) =  
    text  
    |> Seq.map (fun s -> s.Length)  
    |> Seq.reduce max
```

```
let searchForPosOrNegWords (text:string list) =  
    text  
    |> Seq.fold(fun s word -> Set.add word s) Set.empty  
    |> Seq.map(fun word ->  
        if Set.contains word positive then 1  
        elif Set.contains word negative then -1  
        else 0)  
    |> Seq.sum
```

Event Processing in F#



```
let twitter = Twitter.AuthenticateAppOnly(key, secret)
let sample = twitter.Streaming.SampleTweets()
```

sample.TweetReceived

```
|> Event.filter(fun tweet -> tweet.lang = "en")
|> Event.choose(fun tweet -> tweet.Text) // Text is option
|> Event.map(fun text ->
    if Set.contains text positive then 1
    elif Set.contains text negative then -1
    else 0)
|> Event.scan (+)
|> Event.add(fun n -> printfn "Mood=%d" n)
```



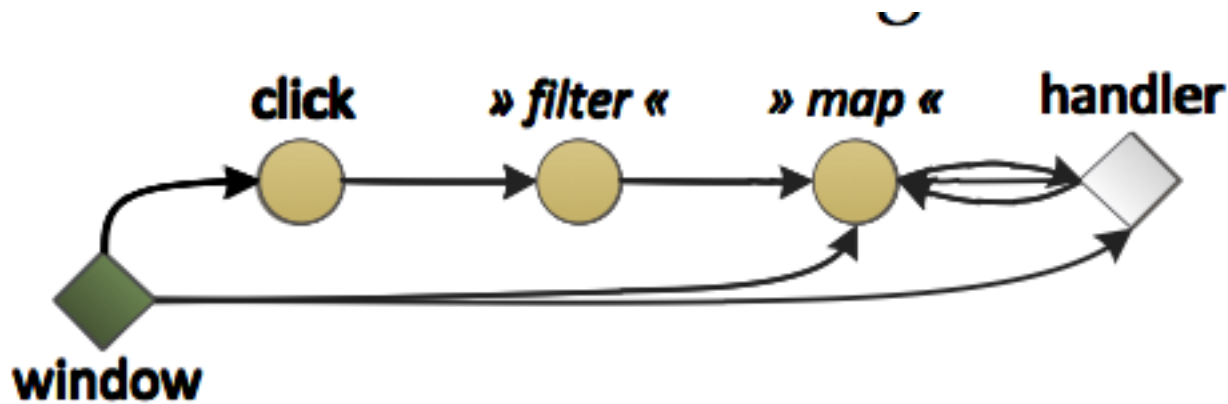
Few Event HOF

```
Event.map      : ('T -> 'R)    -> IEvent<'T> -> IEvent<'R>
Event.filter   : ('T -> bool) -> IEvent<'T> -> IEvent<'T>
Event.add      : ('T -> unit) -> IEvent<'Del,'T> -> unit
Event.merge    : IEvent<'T> -> IEvent<'T> -> IEvent<'T>
Event.scan     : ('St -> 'T -> 'St) -> 'St -> IEvent<'T> ->
```



Event are Observable... *almost*

- Memory leak ☹️
 - **IEvent** does not support *removing* event handlers
(*RemoveHandler* on the resulting event, it leaves some handlers attached... leak!)
 - Observable is able to remove handlers - **IDisposable**





Observable in F#

- `Event<'T>` interface inherits from `IObservable<'T>`
 - ▣ We can use the same standard F# Events functions for working with Observable

```
Observable.filter : ('T -> bool) -> IObservable<'T> -> IObservable<'T>
Observable.map    : ('T -> 'R)   -> IObservable<'T> -> IObservable<'R>
Observable.add    : ('T -> unit) -> IObservable<'T> -> unit
Observable.merge  : IObservable<'T> -> IObservable<'T> -> IObservable<'T>
```

`Observable.subscribe` : `IObservable<'T>` -> [`IDisposable`](#)



IEnumerable is the Dual of IObservable

```
type IEnumerable<'a> =  
    interface IEnumerable
```

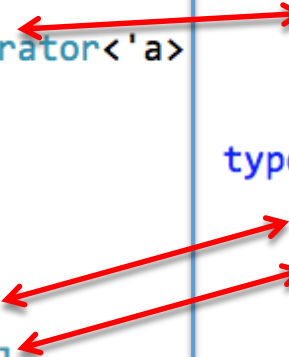
```
        abstract GetEnumerator : IEnumerator<'a>
```

```
type IEnumerator<'a> =  
    interface IDisposable  
    interface IEnumerator
```

```
        abstract Current : 'a with get  
        abstract MoveNext : unit -> bool
```

```
type IObservable<'a> =  
    abstract Subscribe : IObservable<'a>  
        -> IDisposable
```

```
type IObservable<'a> =  
    abstract OnNext : 'a with set  
    abstract OnCompleted : bool -> unit  
    abstract OnError : Exception -> unit
```



Reversing arrows
The input becomes output and <->



IObserver & IObservable

```
let observable = { new IObservable<string> with
    member x.Subscribe(observer:IObserver<string>) =
        { new IDisposable with
            member x.Dispose() = ()
        }
}

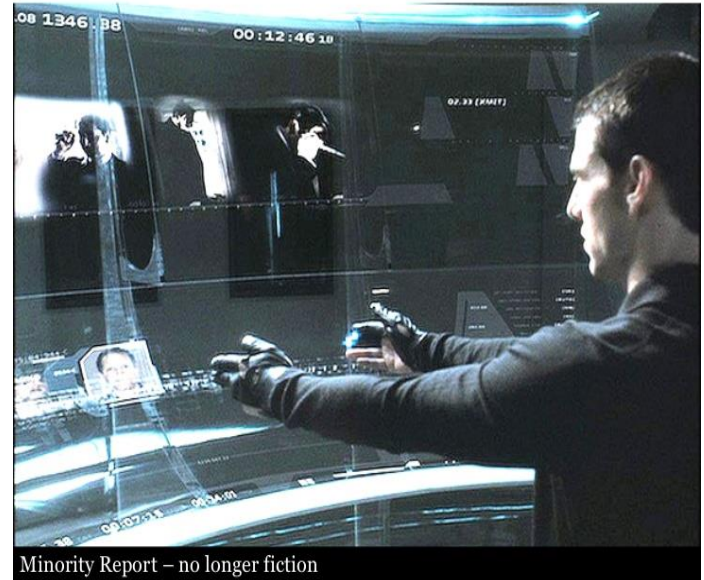
let observer = { new IObserver<string> with
    member x.OnNext(value) = ()
    member x.OnCompleted() = ()
    member x.OnError(exn) = () }
```

DEMO

A 3D white figure, resembling a mannequin or a simple human model, is standing and pointing its right arm towards a large, light gray rectangular screen. The screen displays the word "Demo" in a dark red, serif font. Overlaid on the entire scene is a large, red, distressed rectangular stamp with rounded corners, containing the word "DEMO" in a bold, red, serif font. The stamp is tilted slightly upwards from left to right.

Natural User Interface

Natural User Interface



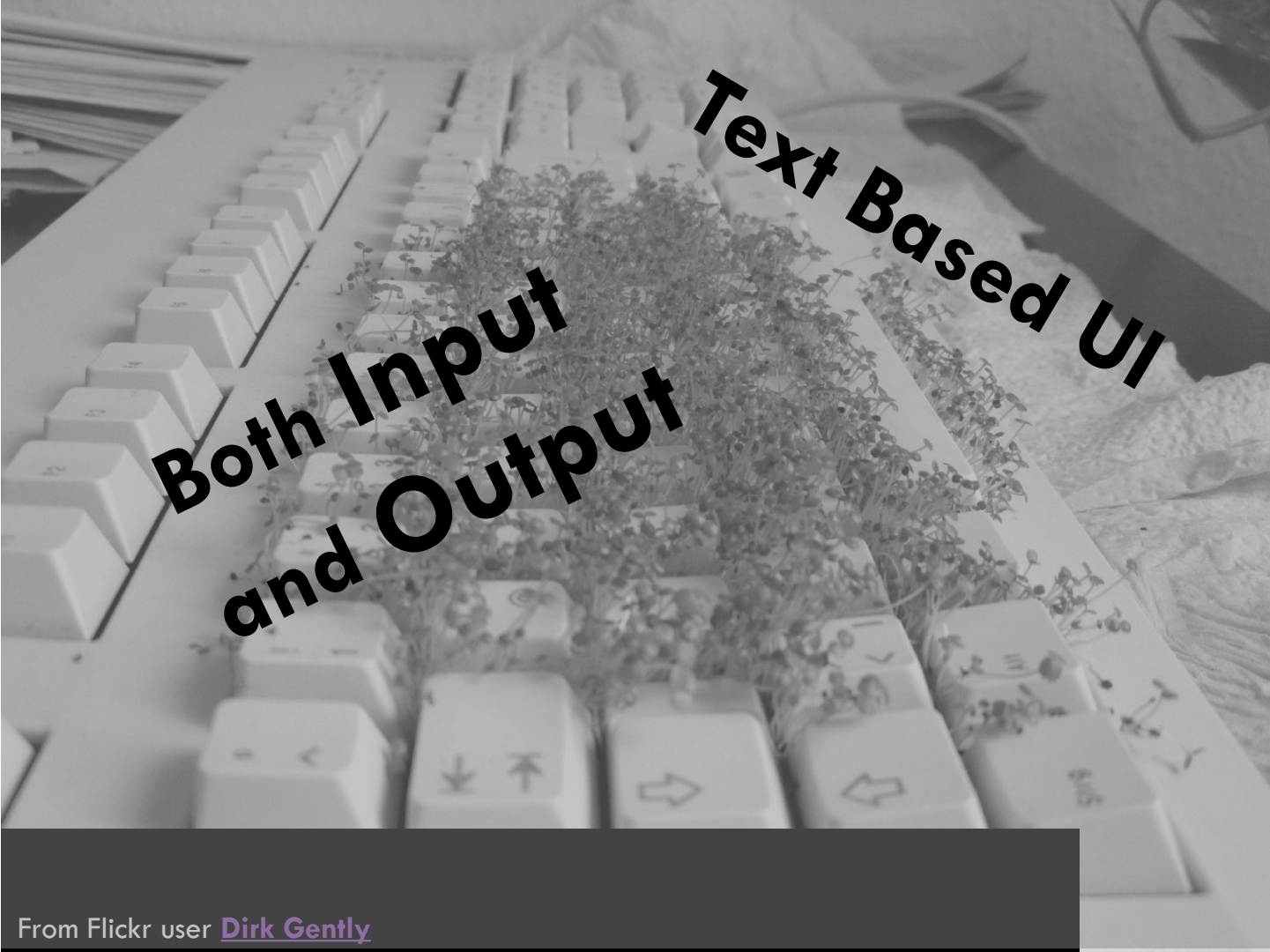
Minority Report – no longer fiction

NUI Evolution





From Flickr user [Dirk Gently](#)



**Both Input
and Output**

Text Based UI





Graphics Exploration

Mouse Drag

**Double
Click**

Right Click

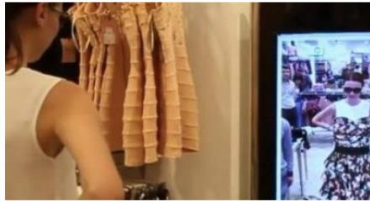




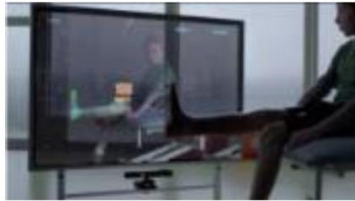
NUI User Cases



RETAIL



THERAPY



HEALTHCARE



EDUCATION



TRAINING



NUI WARS



Summary



- True FRP is about dynamic evolving values over time
 - ▣ Precise, simple denotation. (Elegant & rigorous)
 - ▣ Continuous time. (Natural & composable)
- Denotational Semantic leads to simpler designs and reusable abstraction
- FRP provide a declarative, composable and elegant programming style for animation, graphic and music (IMO - FRP will influence future NUI studies)
- Build your own FRP!

That's all Folks!



The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities.

-- Edsger Dijkstra

How to reach me



github.com/rikace/Presentations/FRP-NUI

@TRikace

tericcardo@gmail.com