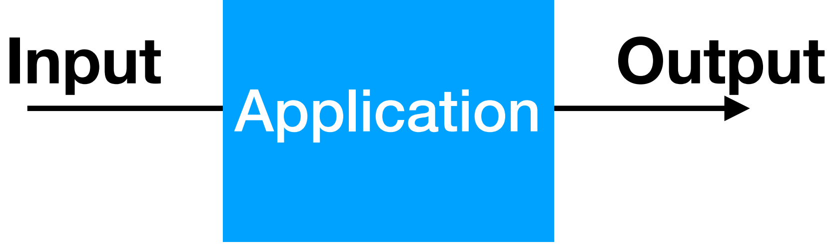
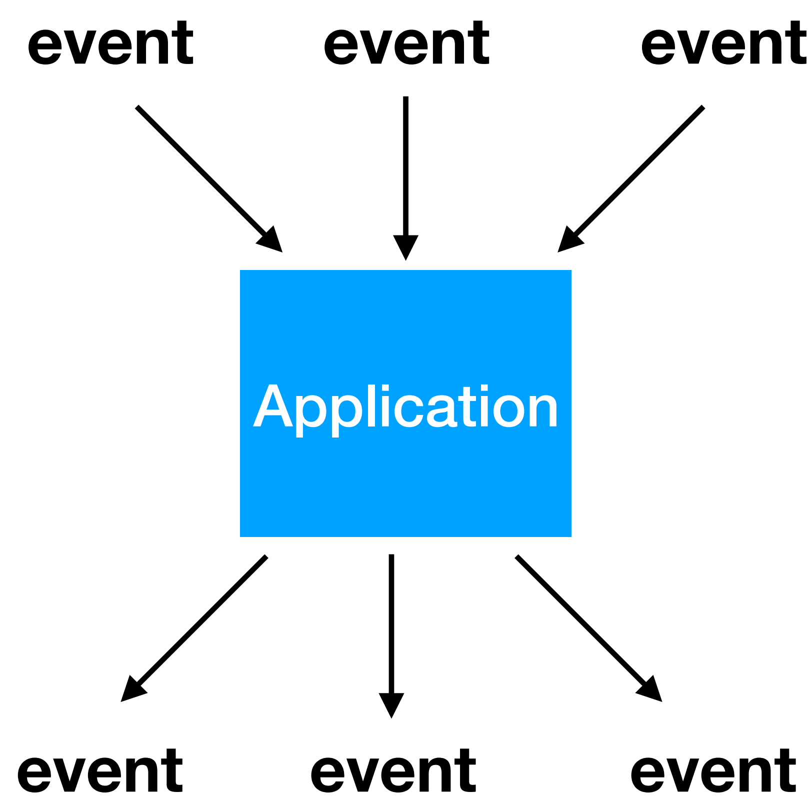
Reactive Programming, Bananas & Robots

Christophe Scholliers
Christophe.Scholliers@UGent.be

GHENT UNIVERSITY

Event Driven Applications





Event Driven Applications

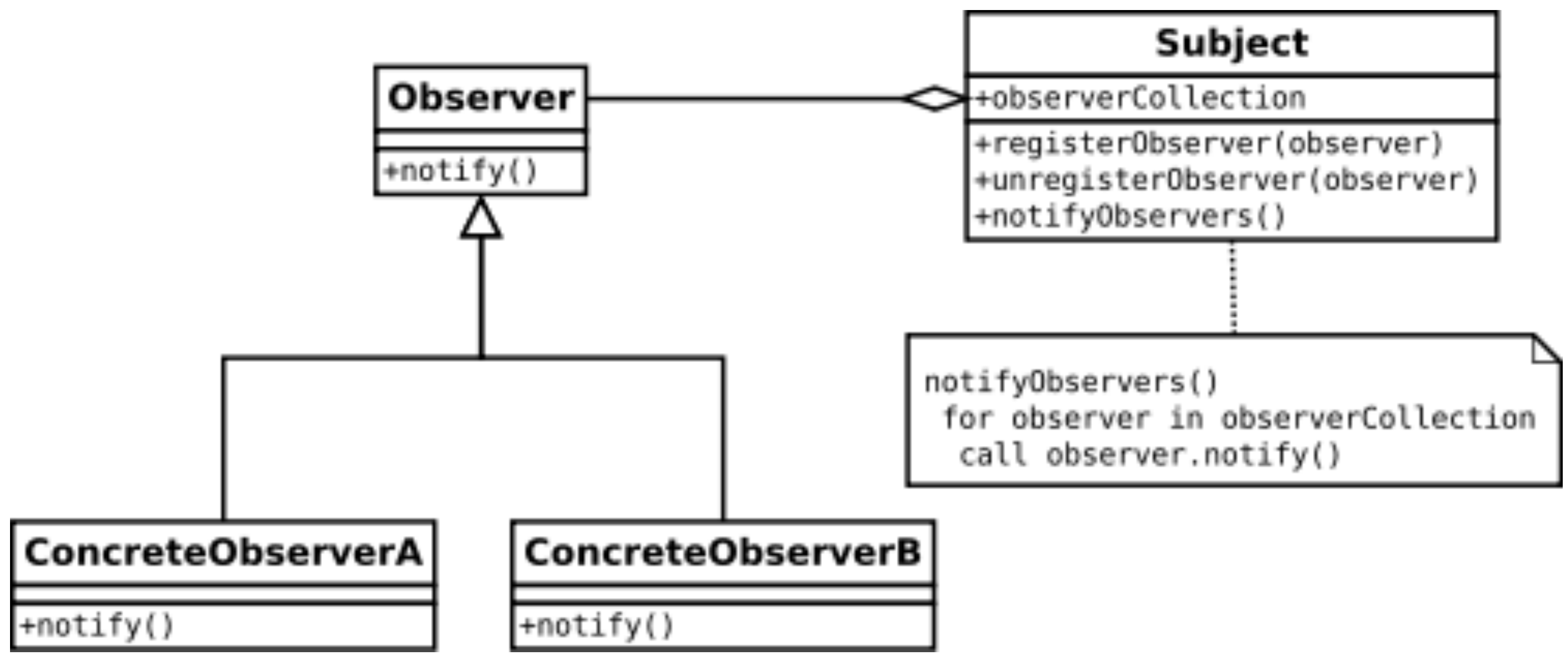
Adobe

33% of all code is event handling code

50% of the bugs are in event handling code

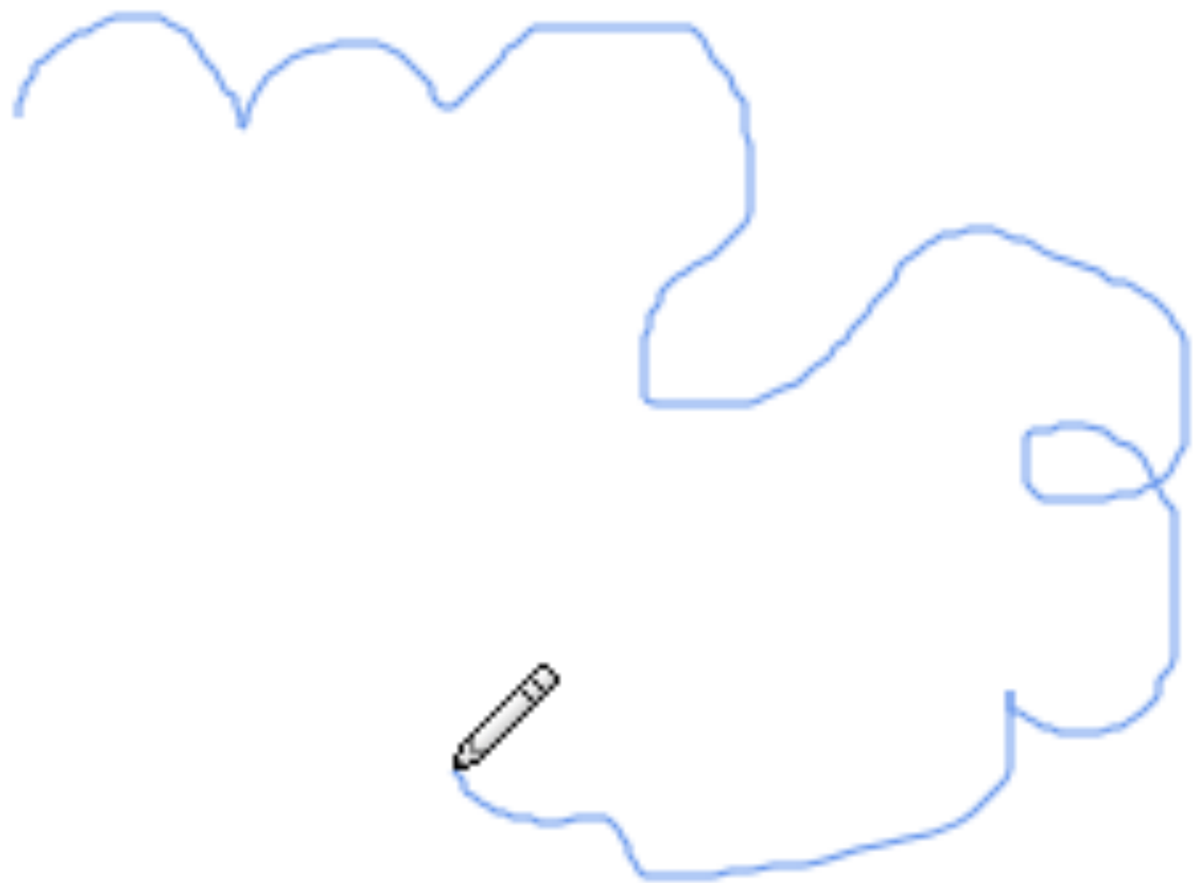
Sean Parent. A possible future of software development. 2008.

Observer Pattern



Ingo Maier and Martin Odersky. Deprecating the observer pattern with Scala.React. Technical Report EPFL-REPORT-176887, École Polytechnique Fédérale de Lausanne, May 2012.

Drawing with a Mouse



Drawing with a Mouse

```
var path: Path = null

val moveObserver = {  (event: MouseEvent) =>
      path.lineTo(event.position)
      draw(path)
}

control.addMouseDownObserver { event =>
      path = new Path(event.position)
      control.registerMouseMoveObserver(moveObserver)
}

control.addMouseUpObserver { event =>
      control.unregisterMouseMoveObserver(moveObserver)
      path.close()
      draw(path)
}
```

Modularity

```
var path: Path = null

val moveObserver = {  (event: MouseEvent) =>
      path.lineTo(event.position)
      draw(path)
}

control.addMouseDownObserver { event =>
        path = new Path(event.position)
        control.registerMouseMoveObserver(moveObserver)
}

control.addMouseUpObserver { event =>
        control.unregisterMouseMoveObserver(moveObserver)
        path.close()
        draw(path)
}
```

Control Flow

```
var path: Path = null

val moveObserver = {  (event: MouseEvent) =>
      path.lineTo(event.position)
        draw(path)
}

control.addMouseDownObserver { event =>
        path = new Path(event.position)
        control.registerMouseMoveObserver(moveObserver)
}

control.addMouseUpObserver { event =>
        control.unregisterMouseMoveObserver(moveObserver)
        path.close()
        draw(path)
}
```

The control flow of the application is no longer dictated by the program text. Each event handler is an entry point.

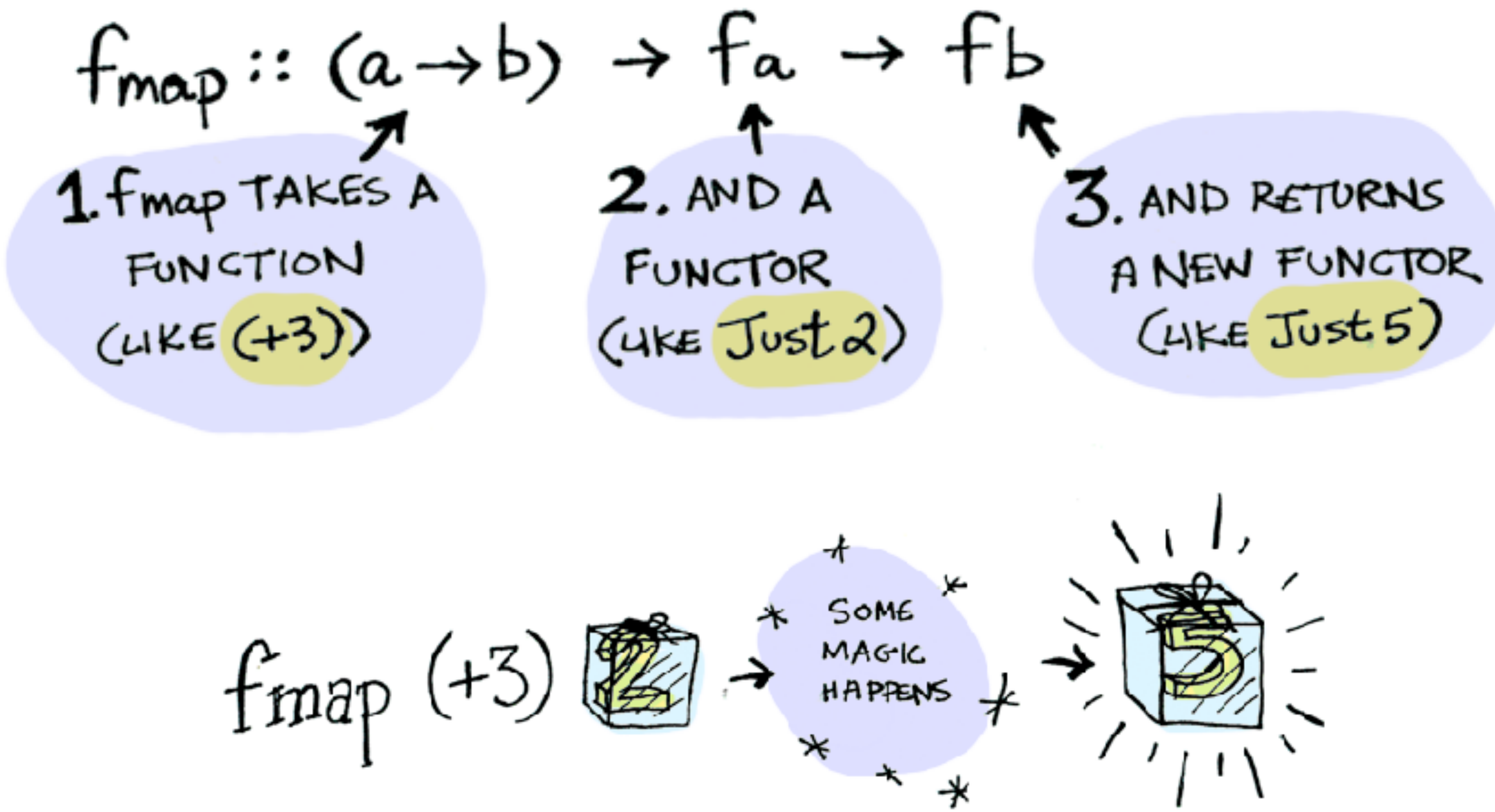
Separation of concerns

```
var path: Path = null

val moveObserver = {  (event: MouseEvent) =>
      path.lineTo(event.position)
        draw(path)
}

control.addMouseDownObserver { event =>
        path = new Path(event.position)
        control.registerMouseMoveObserver(moveObserver)
}

control.addMouseUpObserver { event =>
        control.unregisterMouseMoveObserver(moveObserver)
        path.close()
        draw(path)
}
```

The observers from our example not only trace the mouse path but also call a drawing command, or more generally, include two different concerns in the same code location. It is often preferable to separate the concerns of constructing the path and displaying it, e.g., as in the model-view-controller (MVC) pattern.

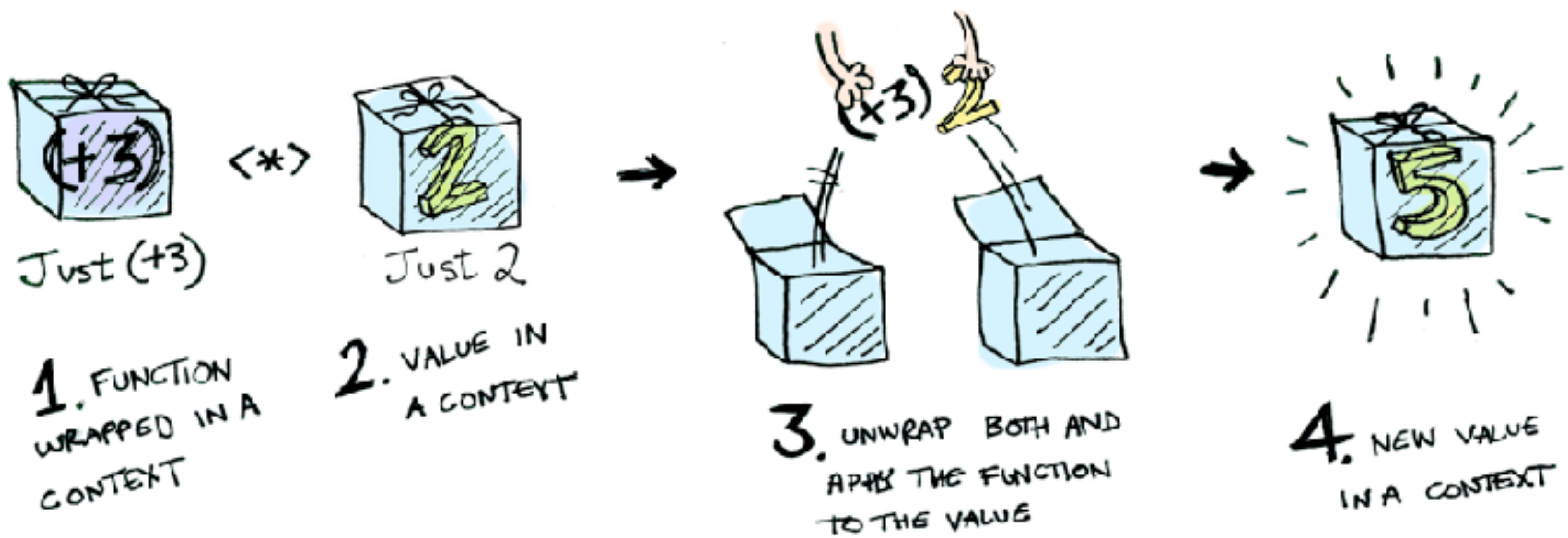
Just in Case

Functors



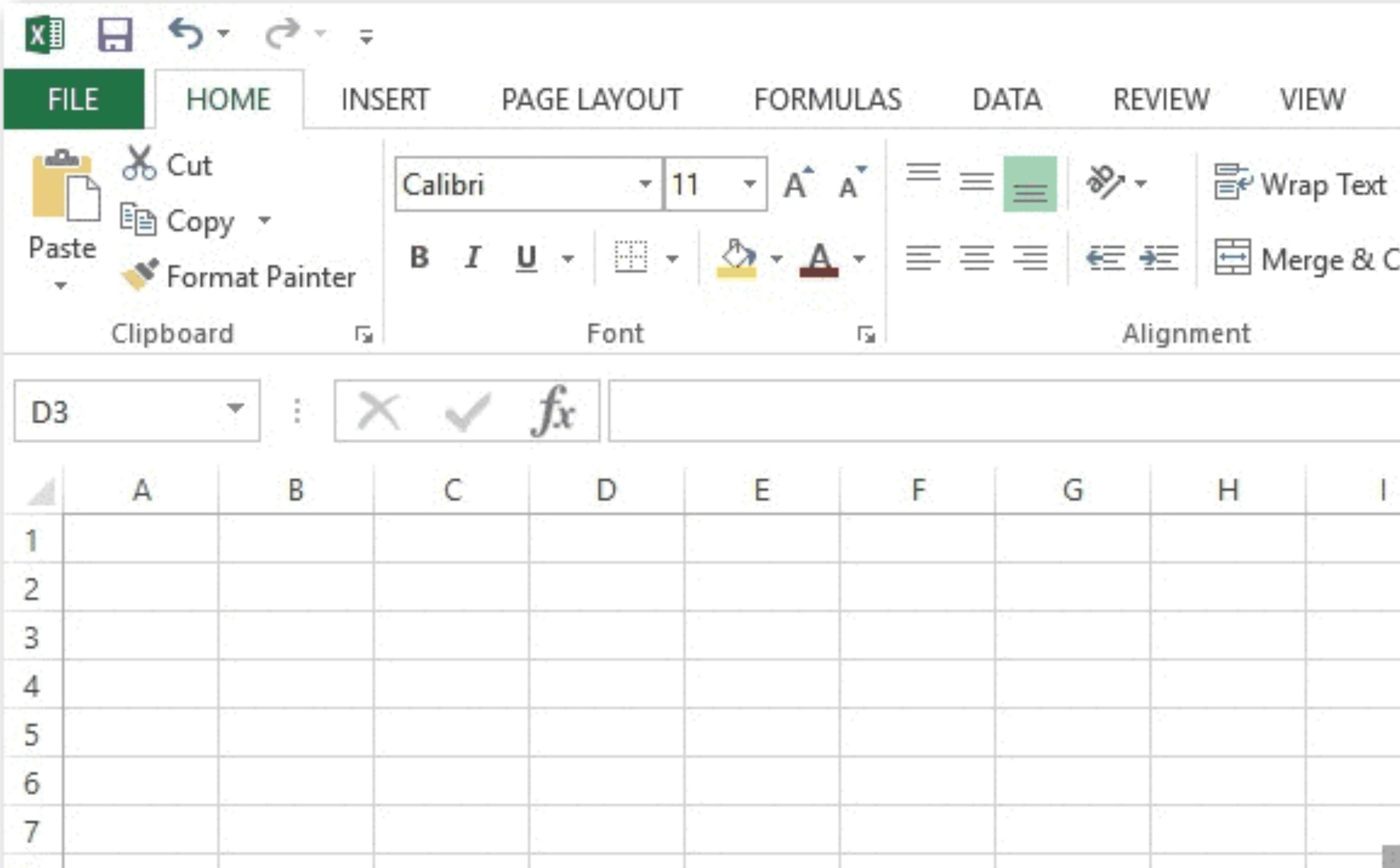
http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html

Applicative



http://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html

Functional Reactive Programming


FILE
HOME
INSERT
PAGE LAYOUT
FORMULAS
DATA
REVIEW
VIEW
Cut
Copy
Paste
Format Painter
Clipboard
Calibri
11
B
I
U
Font
Wrap Text
Merge & Ce
Alignment
D3
A
B
C
D
E
F
G
H
I
1
2
3
4
5
6
7

The Essence !

```
type Behavior a = Time -> a
type Event    a = [(Time,a)]
```

Variation in Time as First Class Value

Elliott, Conal; Hudak, Paul (1997), "Functional Reactive Animation", ICFP.

Sneak Preview

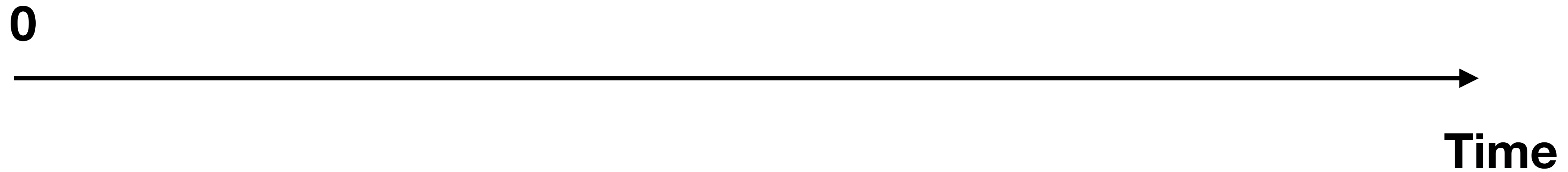
```haskell
seconds      :: Behavior Integer
printSeconds :: Behavior (IO ())
printSeconds = putStrLn . show  <$> seconds

main = runB printSeconds
```

```
Monadic Party xtofs$ ./slides
0
1
2
3
4
5
6
```

FRP adoption

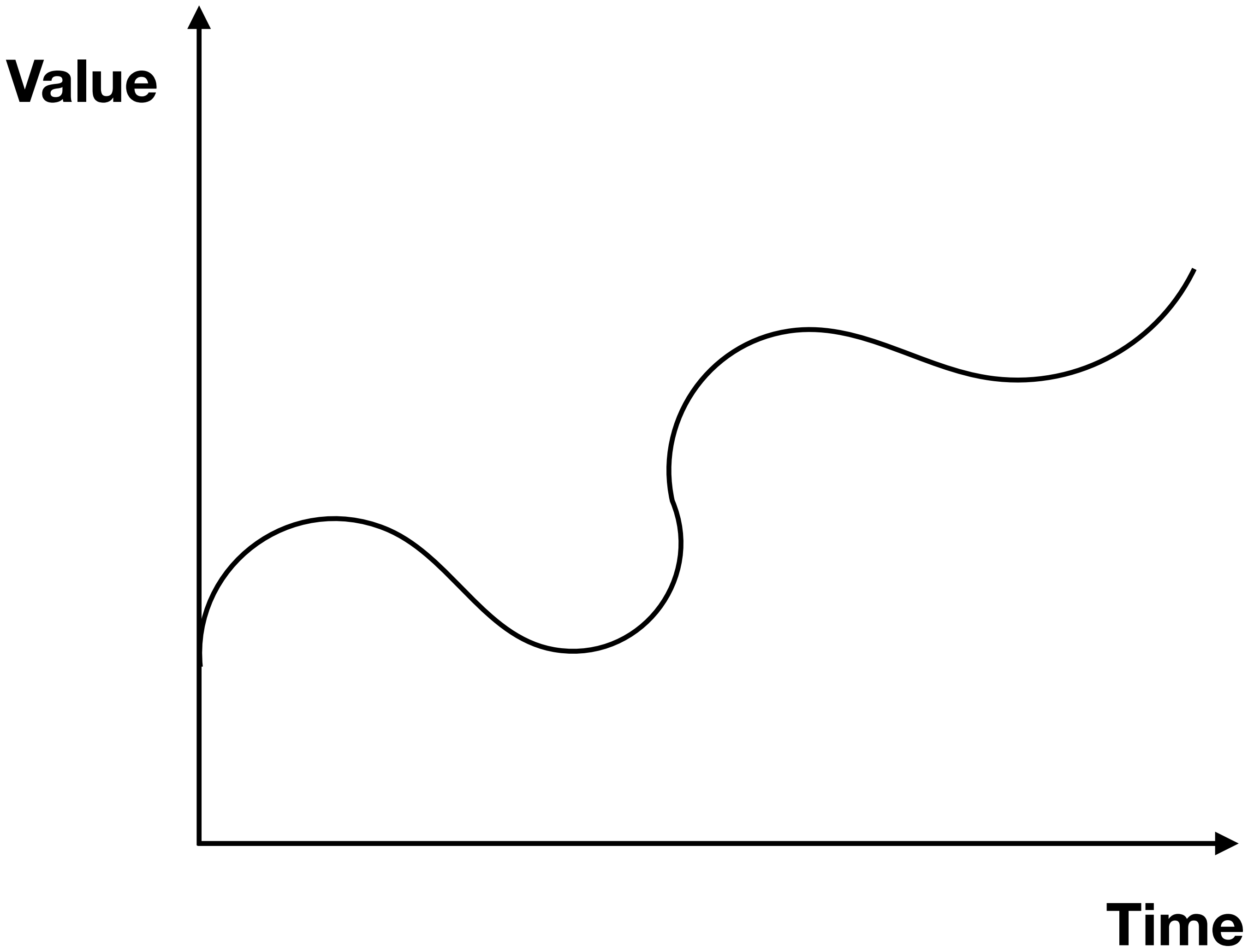
Time

```
type Time = Float -- begins at t = 0
```



We take as assumption that time is in seconds

Behaviour



Time varying value

```
data Behavior a = Behavior { run    :: Time -> a  }
```

Behavior is a Functor

```
instance Functor Behavior where
…
```

Behavior is a Functor

```
instance Functor Behavior where
   fmap f (Behavior g) = Behavior $ f . g
```

“Apply the function at every moment in time”

Behavior is an Applicative

```
instance Applicative Behavior where
    …
    …
```

Behavior is an Applicative

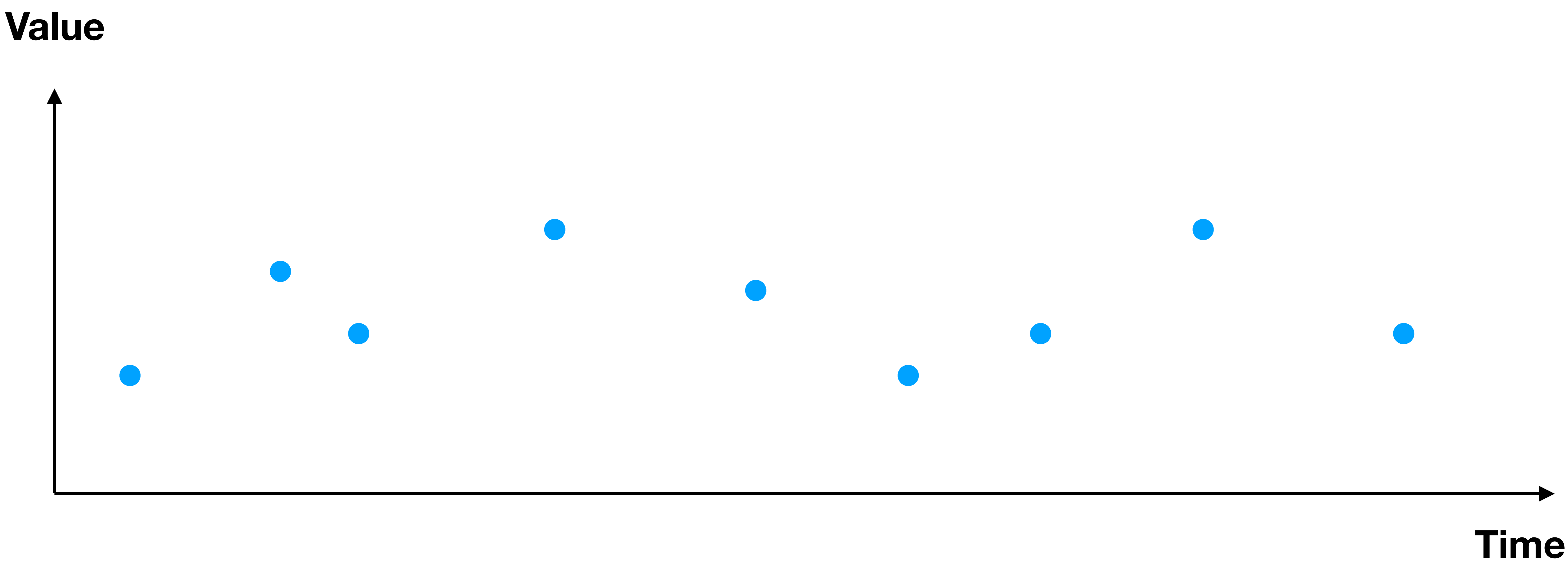
```
instance Applicative Behavior where
   pure a                          = …
   Behavior bf <*> Behavior ba     = …
```

Behavior is an Applicative

```
instance Applicative Behavior where
   pure a                          = Behavior $ \t -> a
   Behavior bf <*> Behavior ba     = Behavior $ \t -> bf t $ ba t
```

Events

```
data Event a = Event { values :: ⟦(Time,a)⟧ }
```



Event is a Functor (1)

```
instance Functor Event where
  fmap f (Event vs)  = …
```

Event is a Functor (1)

```
instance Functor Event where
  fmap f (Event vs)  = Event $ map (\(t,a)->(t,f a)) vs
```

Pairs are functors

```
Prelude> (+1)<$>(1,2)
(1,3)
```

Why !

Implementation of functor for pairs

```
instance Functor ((,)a) where
      b->c (a,b)    (a,c)
  fmap f (x,y) = (x, f y)
```

Event is a Functor (2)

```
instance Functor Event where
         fmap f (Event vs)  = Event $ map (f<$>) vs
```

Overview

```haskell
type Time = Float

data Behavior a = Behavior { run    :: Time -> a   }
data Event    a = Event    { values :: [(Time,a)] }

instance Functor Behavior where
        fmap f (Behavior g) = Behavior $ f . g

instance Applicative Behavior where
        pure a                          = Behavior $ \t -> a
        Behavior bf <*> Behavior ba     = Behavior $ \t -> bf t $ ba t

instance Functor Event where
        fmap f (Event vs)  = Event $ map (f<$>) vs
```

Basic Events

```
never  =    ...
```

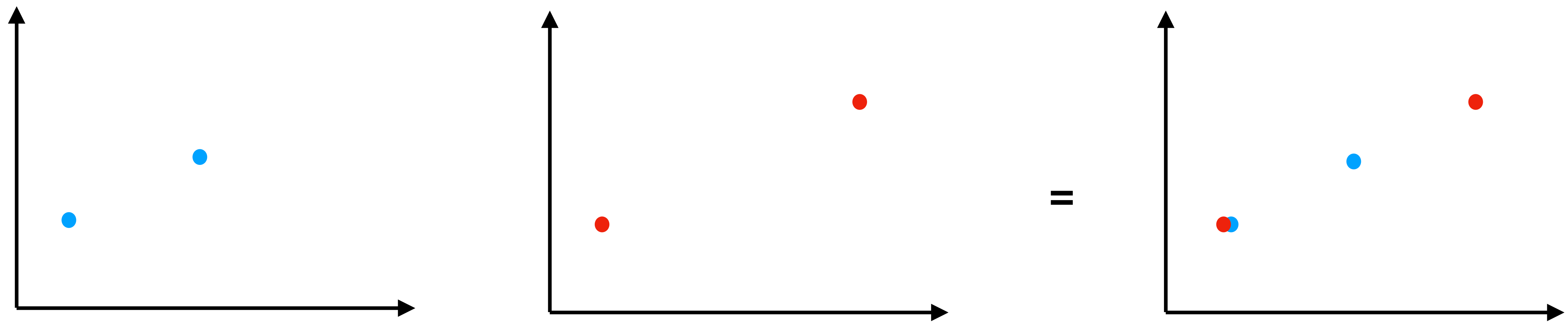
Basic Events

```
never = Event ⟦⟧
```

Filtering Events

```
filterE :: (a->Bool) -> Event a -> Event a
filterE f (Event e) = Event $ filter (f . snd) e
```

UnionWith




```
unionWith :: (a->a->a) -> Event a -> Event a -> Event a
```

UnionWith

```haskell
unionWith :: (a->a->a) -> Event a -> Event a -> Event a
unionWith f (Event va) (Event vb) = Event $ combine f va vb

combine :: (a->a->a) -> [(Time,a)] -> [(Time,a)] -> [(Time,a)]
combine f []  b                   = b
combine f a   []                  = a
combine f ((t1,a):ea) ((t2,b):eb) | t1 == t2   = (t1,f a b):(combine f ea eb)
                                  | t1 <  t2   = (t1,a):(combine f ea ((t2,b):eb))
                                  | otherwise  = (t2,b):(combine f ((t1,a):ea) eb)
```

UnionWith

```
unionWith :: (a->a->a) -> Event a -> Event a -> Event a
unionWith f (Event va) (Event vb) = Event $ combine f va vb
```

UnionWith

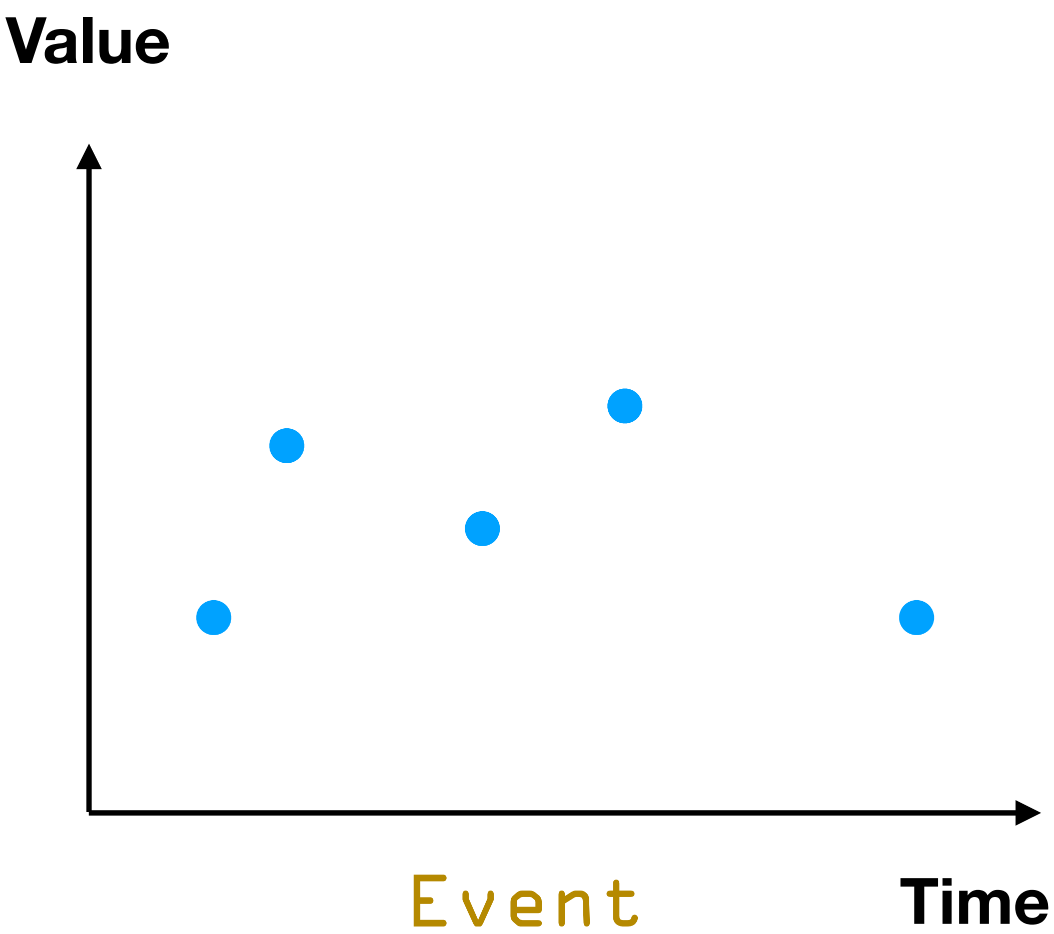
```
combine :: (a->a->a) -> [(Time,a)] -> [(Time,a)] -> [(Time,a)]
combine f []  b                  = b
combine f a   []                 = a
…
```

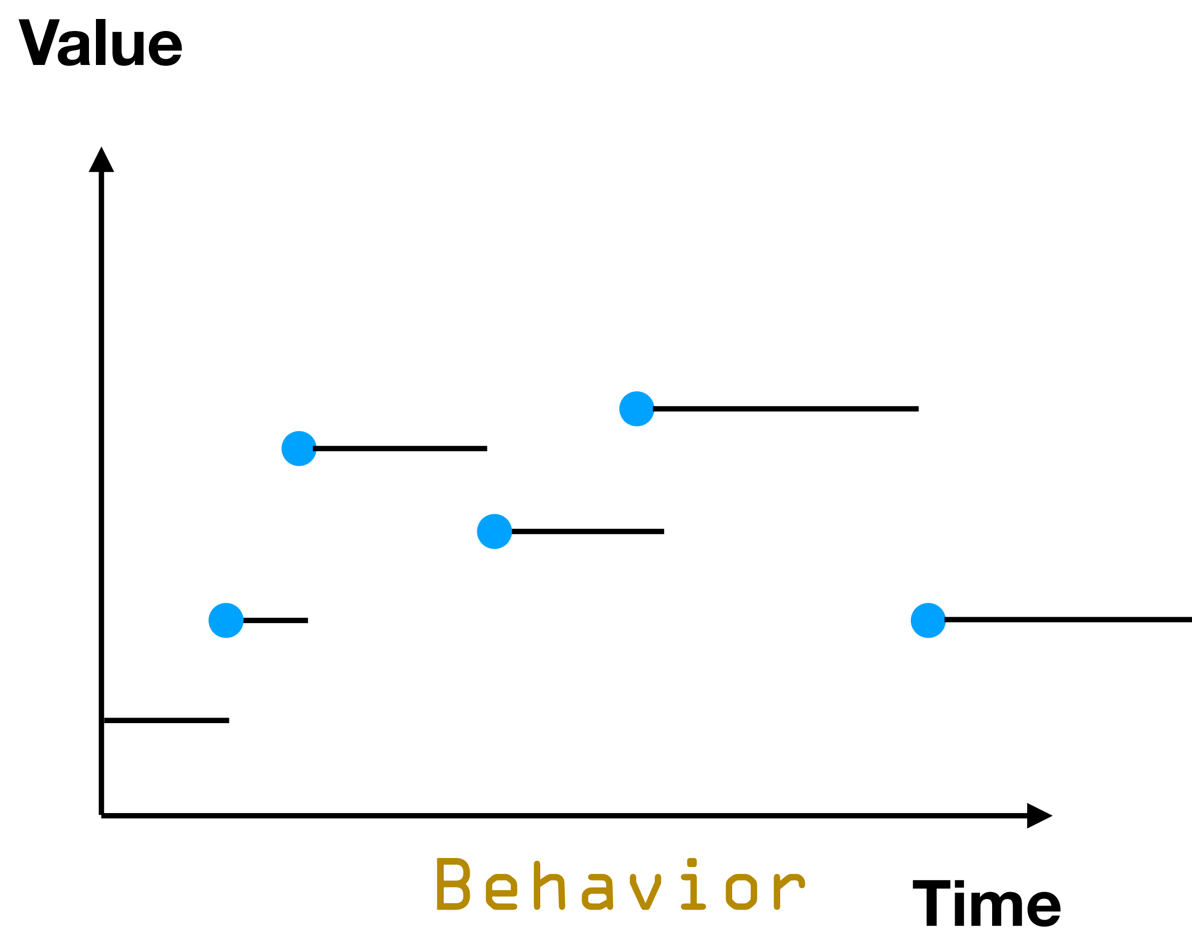
UnionWith

```
...
combine f ((t1,a):ea) ((t2,b):eb)
     | t1 == t2    = (t1,f a b):(combine f ea eb)
     | t1 <  t2    = (t1,a):(combine f ea ((t2,b):eb))
     | otherwise   = (t2,b):(combine f ((t1,a):ea) eb)
```

Stepper

```
stepper :: a -> Event a -> Behavior a
```





Stepper

```
dropUntil t1 l@((_,_):(t2,v):rest)
   | t2>=t1      = l
   | otherwise = dropUntil t1 ((t2,v):rest)

dropUntil t1 l = l

findStep a t ((t0,_):rest) | t < t0 = a

findStep a t e  = takeValue rest
 where rest = dropUntil t e
       takeValue ((t,a):_)  = a
```

Behavior Combinators

```
(<@>) :: Behavior (a->b) -> Event a -> Event b
(Behavior b) <@> (Event e) = …
```

```
(<@) :: (Behavior b) -> Event a -> Event b
(Behavior b) <@  (Event e) = …
```

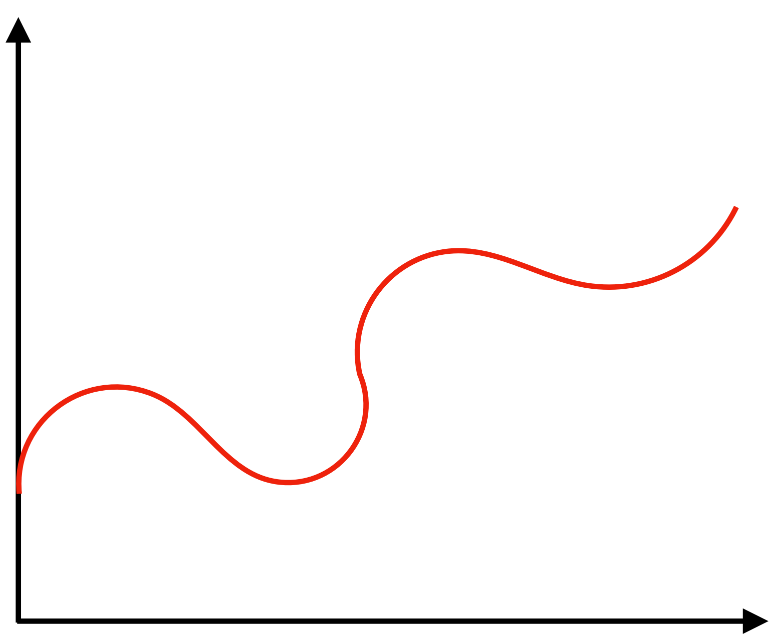
Behavior Combinators

```haskell
(<@>) :: Behavior (a->b) -> Event a -> Event b
(Behavior b) <@> (Event e) = Event $ map (\(t,a)->(t,b t a)) e
```

```haskell
(<@) :: (Behavior b) -> Event a -> Event b
(Behavior b) <@  (Event e) = Event $ map (\(t,a) -> (t,b t)) e
```

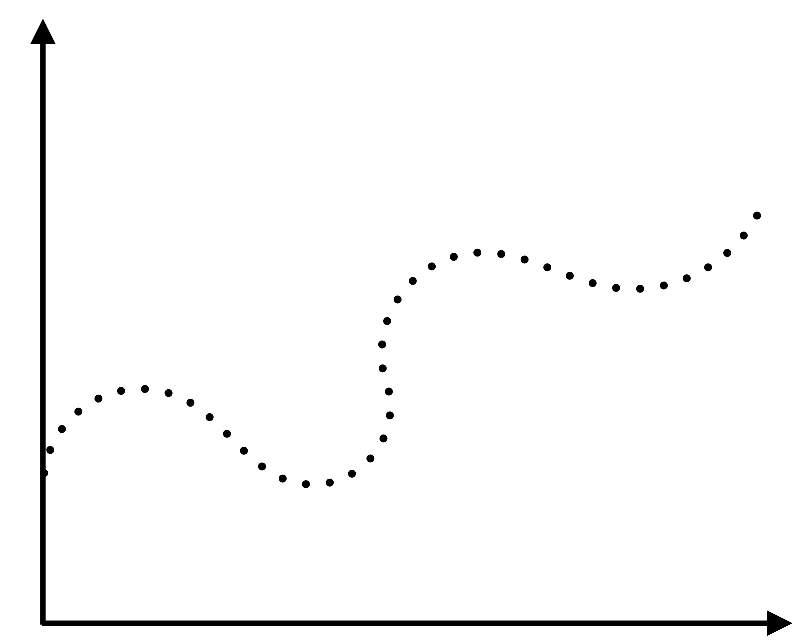
Behavior Combinators

```
(<@) :: (Behavior b) -> Event a -> Event b
(Behavior f) <@  (Event e) = Event $ map (\(t,a) -> (t,f t)) e
```



Behavior b

Event e



b <@ e

Back to our Mini-Example

Seconds

```
seconds      :: Behavior Integer
printSeconds :: Behavior (IO ())
printSeconds = putStrLn . show  <$> seconds

main = runB printSeconds
```

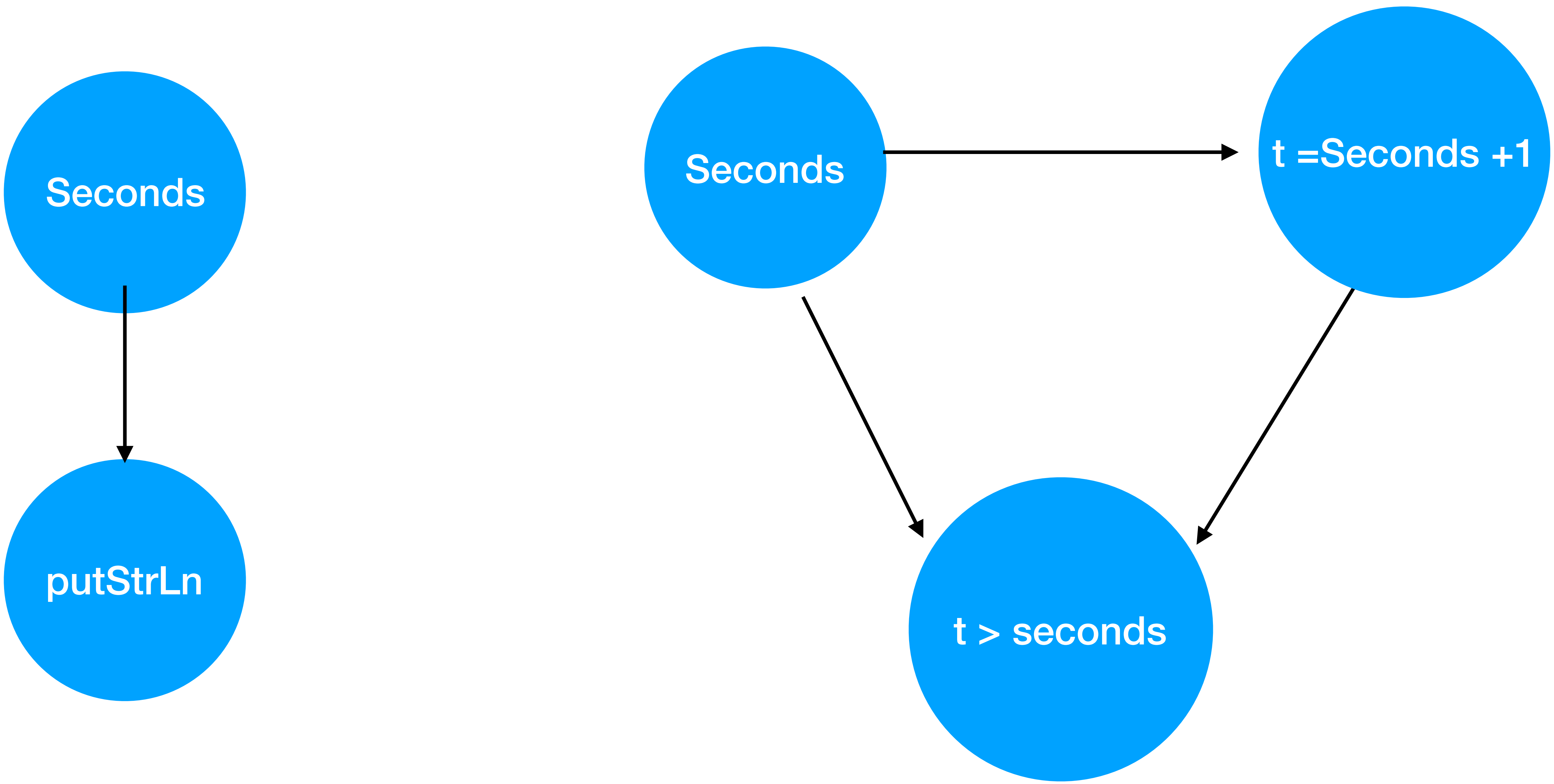
RunB

```
second = 1000000

runB :: Behavior (IO a) -> IO()
runB (Behavior s) = mapM_ (wait . s) [0..]
                    where wait = (>> (threadDelay second))
```

RunB

```
runB :: Behavior (IO a) -> IO(a)
runB (Behavior s) = loop 0
                    where loop x = do o <- (s x)
                                      threadDelay 1000000
                                      loop (x+1)
```

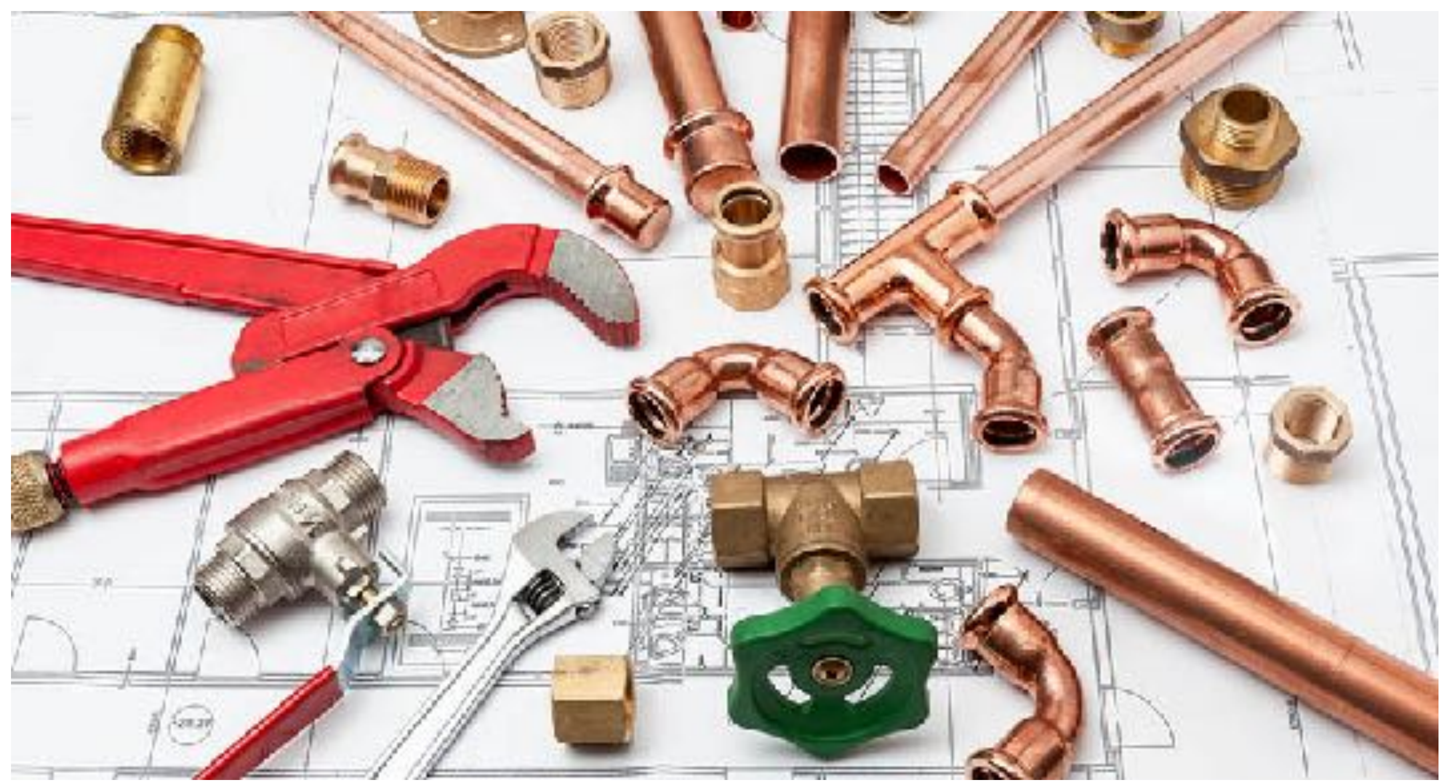
Evaluation of Reactive Programs



When using the wrong evaluation order this might lead to a glitch

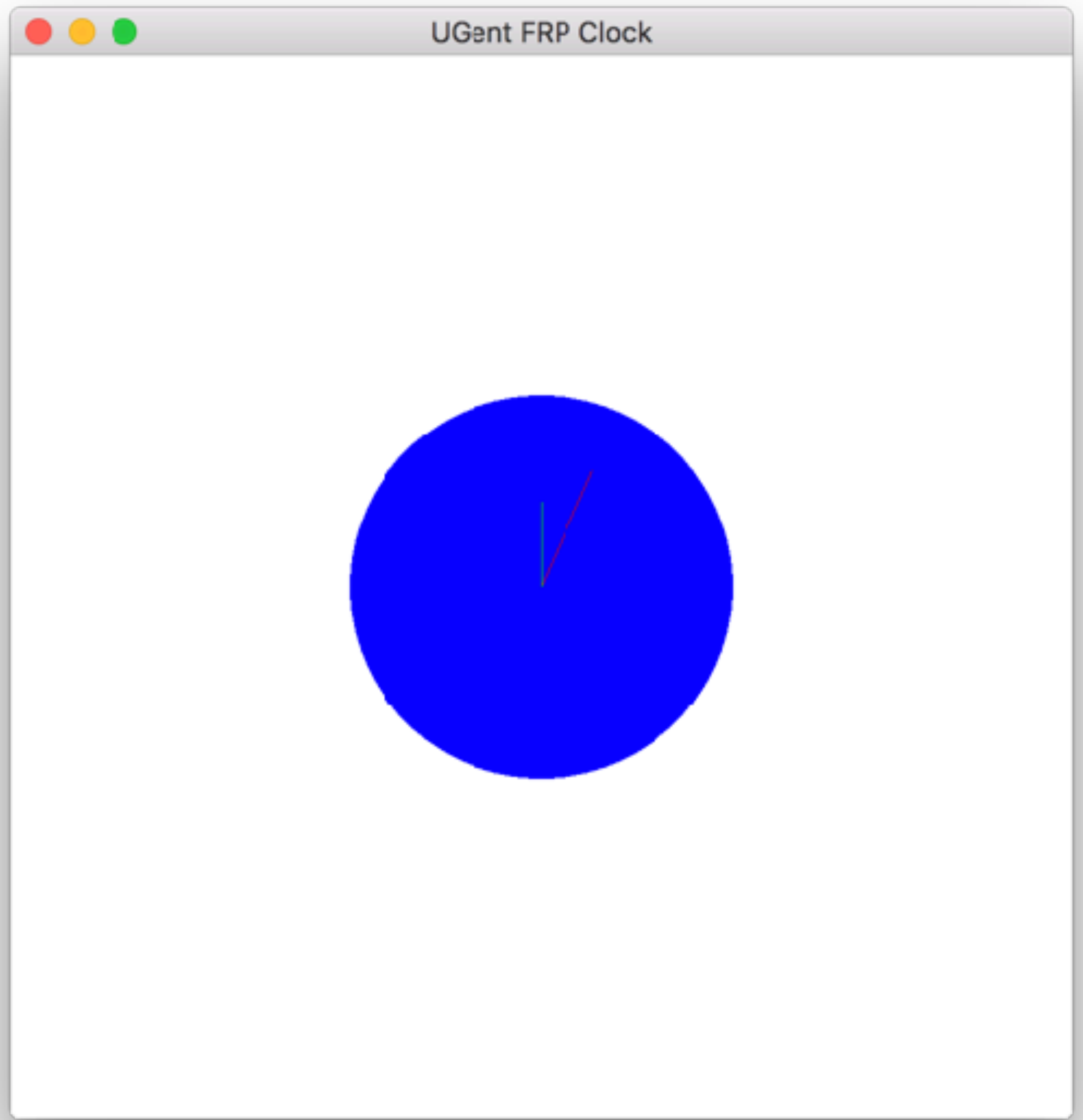
Reactive Banana

1) construct the dependency graph
2) feed events to the decency graph

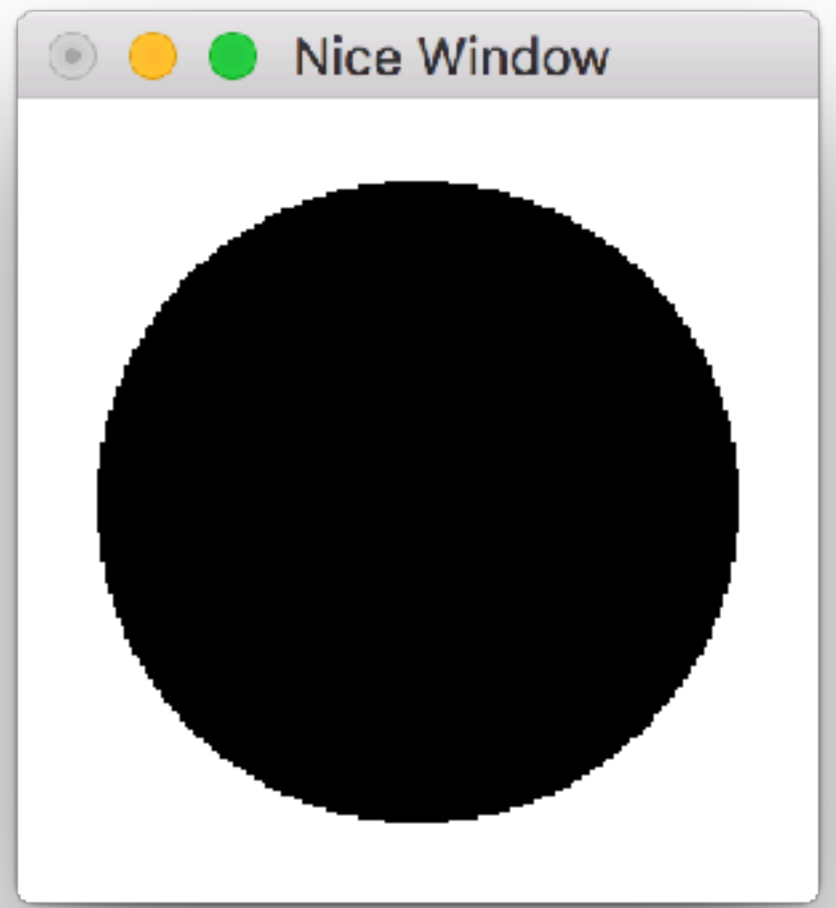


https://wiki.haskell.org/Reactive-banana

Animating a Clock



Gloss



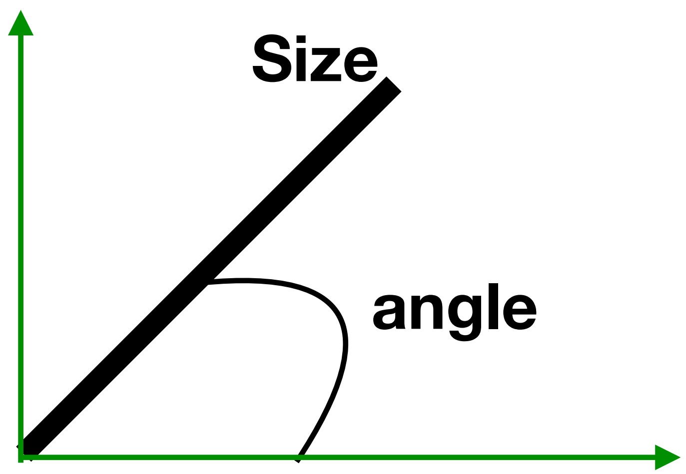

```
import Graphics.Gloss
main = display (InWindow
                   "Nice Window"
                    (200, 200)
                    (10, 10))
               white
               (circleSolid 80)
```

```
*Main> :t (circleSolid 80)
(circleSolid 80) :: Picture
```

ClockHand

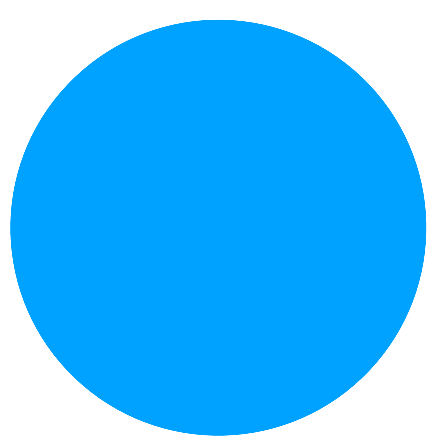
Size **angle**

```
data ClockHand = ClockHand Float Color Float
```

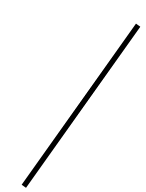


Rendering (Gloss)

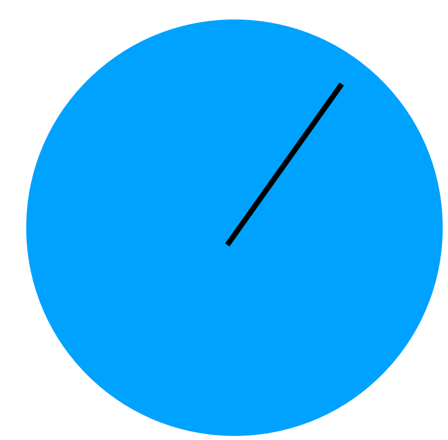
```
clockFace              = color blue $ circleSolid clockSize

renderHand (ClockHand l c angle) = color c  $ Line [center, (x,y) ]
     where x = (fst center) + l * cos angle
           y = (snd center) + l * sin angle

renderClock  hands = Pictures   $ clockFace :  (renderHand <$> hands)
```



clockFace



renderHand



renderClock

Constructing the network

```
let net :: MomentIO ()
     net = do seconds   <- getSecondEvents  win

              let sClockHand = singleton
                                <$>
                                (ClockHand 60 red) .
                                (convert 60) .
                                (fromIntegral)
                                <$>
                                seconds

              reactimate $ (drawPicture win) <$>
                            (renderClock <$> sClockHand)
```

Seconds

```
let net :: MomentIO ()
    net = do seconds   <- getSecondEvents  win
```

```
getSecondEvents :: RWindow -> MomentIO (Event Int)
```

Converting events

```
sClockHand = singleton <$>
             (ClockHand 60 red) . (convert 60) . (fromIntegral)
              <$>
             seconds
```

0 1 2

[(ClockHand 60 red 1.57)] [(ClockHand 60 red 1.46)] [(ClockHand 60 red 1.36)]

Reactimate

```haskell
reactimate $ (drawPicture win) <$> (renderClock <$> sClockHand)
```

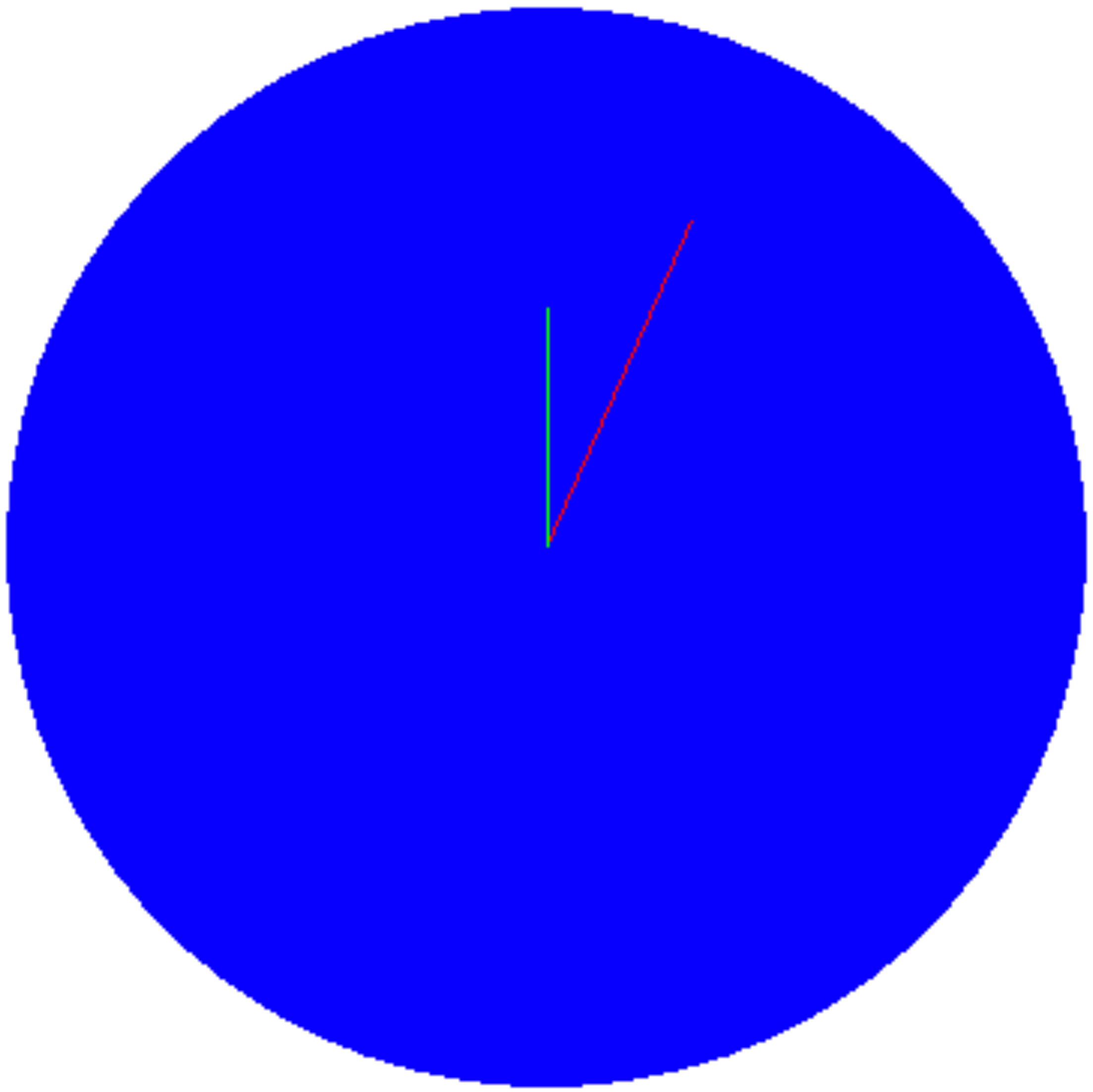
```haskell
drawPicture :: RWindow -> Picture -> IO ()
reactimate :: Event (IO ()) -> MomentIO ()
```

Running the network

```
main = do
  win  <- newWindow 500 500 "UGent FRP Clock"

  let net :: MomentIO ()
      net = do seconds   <- getSecondEvents  win
               let sClockHand = (singleton) <$>
                                ((ClockHand 60 red   ) .
                                 (convert 60)    .
                                 (fromIntegral)  <$>
                                seconds)
               reactimate $ (drawPicture win) <$>
                            (renderClock <$> sClockHand)

  runRWindow net win
```

```
runRWindow :: MomentIO () -> RWindow -> IO ()
```

Extending the clock



- **Hour handle**
- **keyboard to change the hour handle**

Adding the hour handle

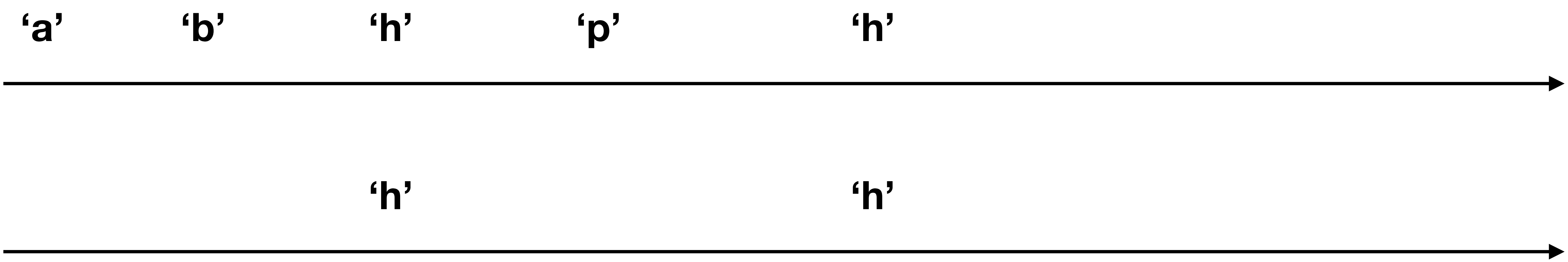
```
main = do
  win  <- newWindow 500 500 "UGent FRP Clock"
  let net :: MomentIO ()
      net = do seconds   <- getSecondEvents  win
               let sClockHand =(singleton) <$> ((ClockHand 60 red   ) .
                                                (convert 60)    .
                                                (fromIntegral)    <$>
                                                 seconds)

               let mClockHand =(singleton) <$> ((ClockHand 40 green) .
                                                (convert 43200) .
                                                (fromIntegral)  <$>
                                                 seconds)

               reactimate $ (drawPicture win) <$>
                            (renderClock <$>
                             unionWith (++)  sClockHand  mClockHand)
  runRWindow net win

 unionWith :: (a->a->a) -> Event a -> Event a -> Event a
```

Adding a keyboard

```
allKeys <- getKeyEvents      win
```

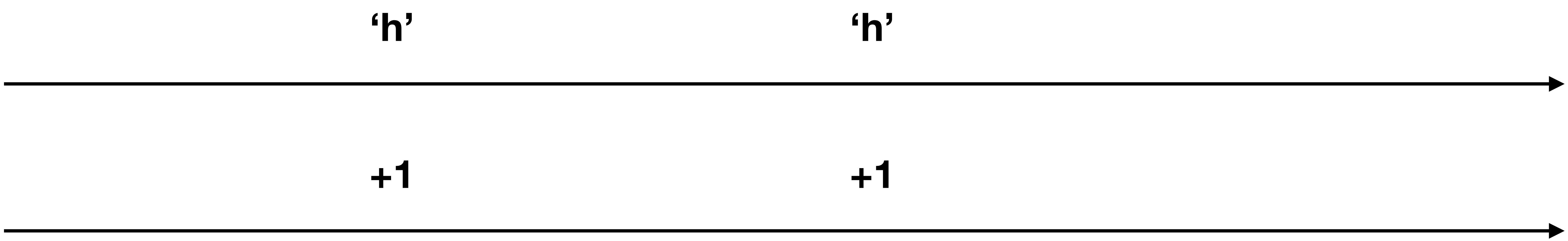
Adding a keyboard

```
allKeys <- getKeyEvents     win
timesH  <-                        (filterE (=='h') allKeys)
```



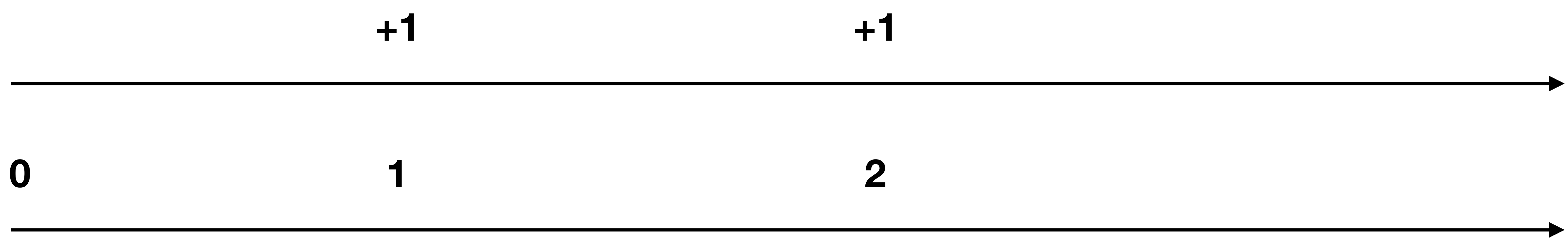
Adding a keyboard

```
allKeys <- getKeyEvents     win
timesH  <-                (+1) <$ (filterE (=='h') allKeys)
```



Adding a keyboard

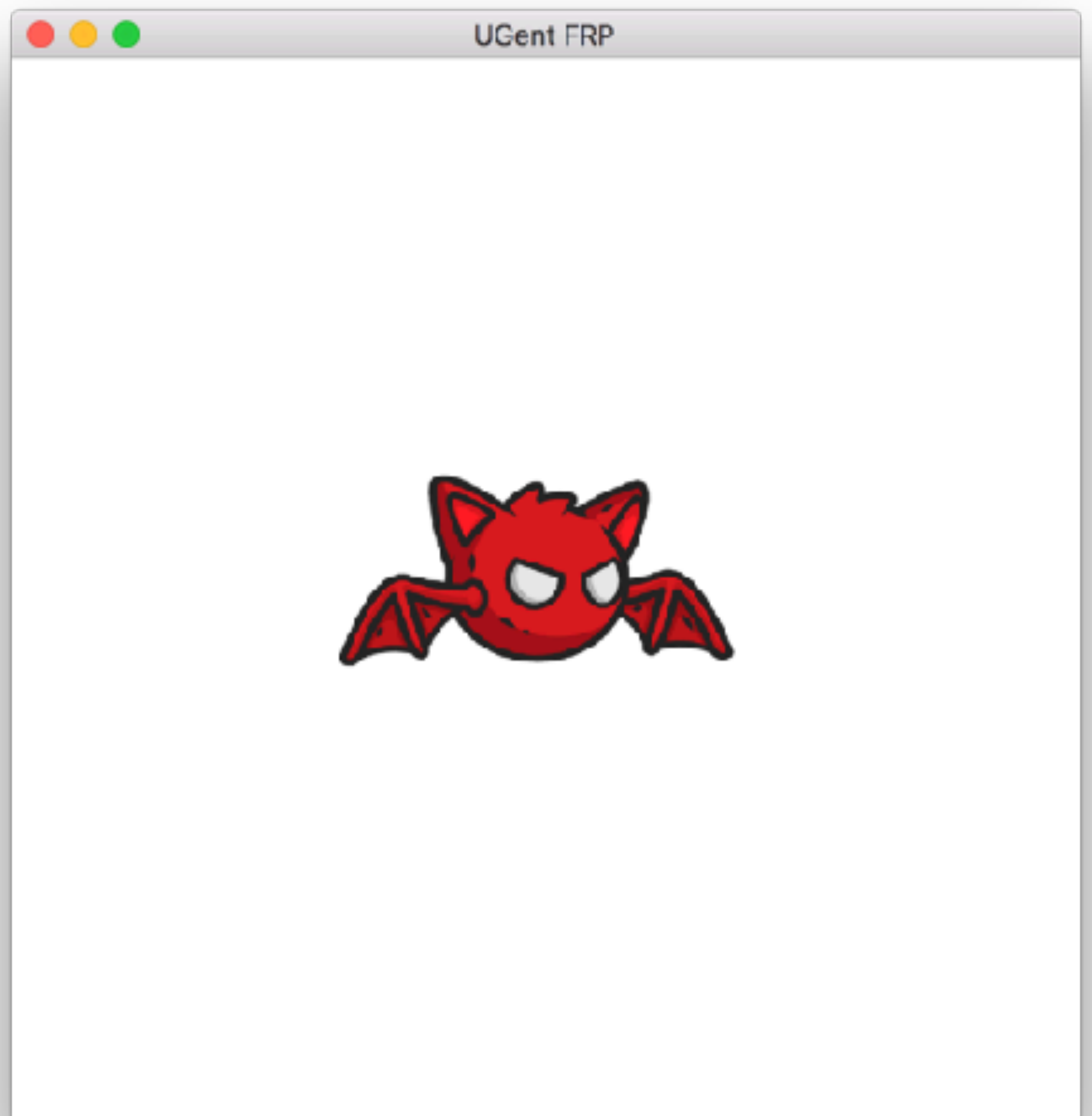
```
allKeys <- getKeyEvents     win
timesH  <- accumE 0    $ (+1) <$ (filterE (=='h') allKeys)
```



Adding a keyboard

```
allKeys   <- getKeyEvents     win
timesH    <- accumE 0    $ (+1) <$ (filterE (=='h') allKeys)
upKeyB    <- stepper(+0) $ (+)  <$> (*3600) <$>  timesH
```

Upscaling the Graphics (a little bit)



Flying Bat

```
main = do
  win  <- newWindow 500 500 "UGent FRP"
  bat1 <- newAnimation batFly
  let net :: MomentIO ()
      net = do seconds   <- getSecondEvents  win
               flyingBat <- (accumE bat1) $ (nextAnimation) <$ seconds
               reactimate $ (drawPicture win) <$> (renderAnimation <$>  flyingBat)
  runRWindow net win
```

Keyboard

```
main = do
  win  <- newWindow 500 500 "UGent FRP"
  bat1 <- newAnimation batFly
  bat2 <- newAnimation batDead
  let net :: MomentIO ()
      net = do seconds   <- getSecondEvents  win
               -- Event Char
               allKeys   <- getKeyEvents     win
               dOrL      <- stepper (pickAnimation 'l') $
                            --  (pickAnimation 'd')  (pickAnimation 'f')
                            pickAnimation <$>
                            --  'd'   'f'
                            filterE iskey allKeys
               setXB     <- setX <$> (stepper  0   $
                            --
                            (moveOverInterval) <$> seconds)
               flyingBat <- moveOnEvent bat1 seconds
               deadBat   <- moveOnEvent bat2 seconds
               reactimate $ (drawPicture win) <$>
                            (renderAnimation <$>
                            ((setXB <*>  (dOrL <*> flyingBat <*> deadBat) <@ seconds)))
  runRWindow net win
```

Moving animations

```
data Animation = Animation { pos     :: (Int,Int),
                             shapes  :: [Picture],
                             active  :: Int
                           }
```

Loading pictures

```
isPng s = snd (break (=='.') s) == “.png"

loadPictures :: String -> IO([Picture])
loadPictures path
=
do bmpPath  <- (++path) <$> System.Directory.getCurrentDirectory
   list     <- listDirectory bmpPath
   return $ (png) . (bmpPath++) <$> filter isPng list
```

Create a new animation

```
newAnimation :: String -> IO(Animation)
newAnimation path
=
do pictures <- loadPictures path
   return $ Animation (0,0) pictures 0
```

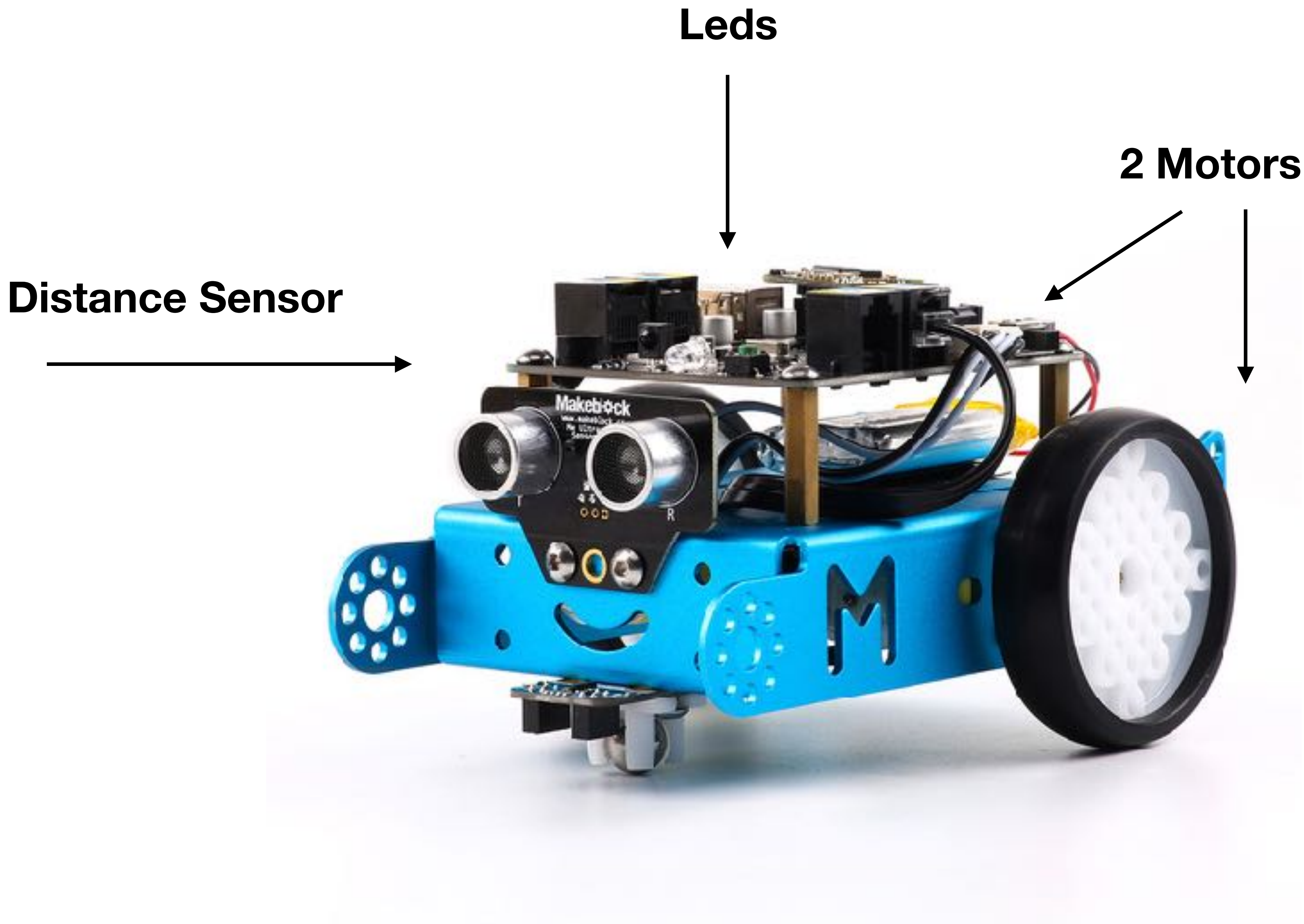
Cycle shapes

```
nextAnimation animation@(Animation { shapes = s, active = a })
=
animation { active = (a+1) `mod` length s }
```

Rendering

```
renderAnimation  :: Animation -> Picture
renderAnimation Animation { pos = (x,y), shapes = s, active = a }
=
Translate (fromIntegral x) (fromIntegral y) $ s!!a
```

Robots



Robots

```haskell
import MBot

main =  do
  putStrLn "My first mBot program in Haskell !"
  -- Open the connection with the mMbot
  d <- openMBot
  putStrLn "Opened a connection with the mBot"
  -- Turn on led 1 of the mBot and set the RGB value to (0,100,0)
  sendCommand d $ setRGB 1 0 100 0
  putStrLn "Look at all the pretty colors !"
  -- Turn on led 2 of the mBot and set the RGB value to (100,0,0)
  sendCommand d $ setRGB 2 100 0 0
  -- close the connection with the mBot
  closeMBot d
```

https://hackage.haskell.org/package/MBot

Reactive Library

```
module ReactiveMBot (
        RMBot,
        newMBot,
        runMBot,
        getDistance,
        left,
        forward,
        …
 ) where
```

Reactive Library

```haskell
import ReactiveMBot
import Reactive.Banana.Frameworks
import Reactive.Banana.Combinators


setLights mbot dist | dist > 30 = leftLight mbot green
                    | otherwise = leftLight mbot blue

detectDistance :: RMBot -> MomentIO ()
detectDistance mbot =
  do dist  <- getDistance mbot
     reactimate $ (setLights mbot) <$> dist

main = do
  mbot  <- newMBot
  runMBot (detectDistance mbot) mbot
```

Obstacle Avoidance

```
drive mbot dist | dist > 30 = forward mbot
                | otherwise = left mbot

setLights mbot dist | dist > 30 = leftLight mbot green
                    | otherwise = leftLight mbot blue

avoidWall :: RMBot -> MomentIO ()
avoidWall mbot =
  do dist  <- getDistance mbot
     reactimate $ (drive mbot)     <$> dist
     reactimate $ (setLights mbot) <$> dist

main = do
  mbot  <- newMBot
  runMBot (avoidWall mbot) mbot
```