Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Week 12
GUI Development
Part 3 - Functional Reactive Programming

Glenn.Strong@scss.tcd.ie
https://scss.tcd.ie/Glenn.Strong/

The FRP approach

The FRP style emphasises composing functions.

Specifically, we will compose a number of *signal functions* that carry information about events and "behaviours".

The idea is to get away from mutation, callbacks and all that

Important concepts:

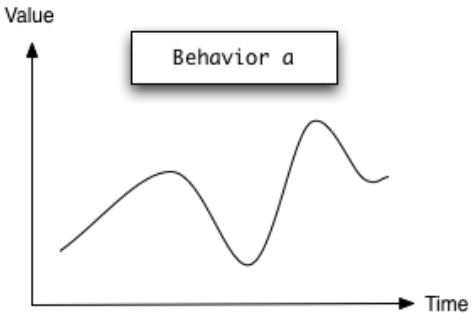
- **"Events" : things that happen at certain moments in time (for example, mouse clicks, key presses)**
- **"Behavours": continuously occurring things (the content of an input field, for example)**

Behaviours

Remember Signal functions?

Behaviours are a bit like that. Think of it as being this type:

```
type Behavior a = Time -> a
```



We'd like to be able to:

- **Apply functions to one or more Behaviors**
- **Create constant behaviours**

So Behavior should be an instance of Functor & Applicative.

Behaviours

We can use Functor and Applicative operations to modify behaviours. For example, if you have a behaviour that tells you how much money you have

```
balance :: Behavior Int
```

And another that tells you the current price of something:

```
price :: Behavior Int
```

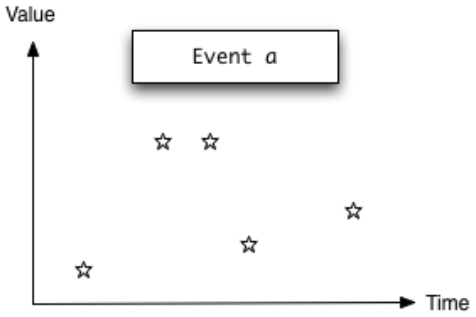
Then you can produce a behavior that tells you whether you can afford, at any given moment in time, to buy something:

```
affordable :: Behavior Bool
affordable = (>=) <$> balance <*> price
```

Events

Events are things that happen at certain times; at any given moment there's at most one event taking place (and usually there are none).

```
type Event a = [ (Time,a) ]
```



Events are things like key presses, mouse clicks, and so on. The only one that comes from the framework is the one that never fires; all the rest come from the UI library (in threepenny that means: from the javascript UI library).

```
never :: Event a
never = []
```

Events

There's a set of functions to combine events.

First, we have a Functor instance:

```
fmap :: (a -> b) -> Event a -> Event b
```

Event streams can be *joined:*

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

Only one event is permitted at a time; the first argument has the job of combining the two events in that case.

Events

Utilities

We'll often want to filter event streams to ignore some events, keeping only the ones that match some condition:

```
filterE :: (a -> Bool) -> Event a -> Event a
```

A common case is merging many event streams that contain functions

```
unions :: [Event (a -> a)] -> Event (a -> a)
```

Events

Accumulation

Finally, *accumulation*:

```
accumE :: MonadIO m => a -> Event (a -> a) -> m (Event a)
```

The idea here is to accumulate events. For example, say we have an input event that produces numbers:

```
newItemsArrived :: Event Int   -- gives number of new items
```

We can transform this into an event that fires when new items arrive, but which gives is the *total* accumulated number of items instead:

```
totalItems :: UI (Event Int)   -- gives number of total items
totalItems = accumE 0 ((+) <$> newItemsArrived)
```

Combining Events and Behaviors

One combination it to take a Behavior that has a function which varies over time and apply it. Think of it as an extension of fmap.

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

Here's a related notion - makes an Event that fires when the supplied event does, but which takes it's value from the behavior:

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

Combining Events and Behaviors

Behaviors can be involved in filtering events with these functions:

```
filterApply :: Behavior (a -> Bool) -> event a -> event a
```

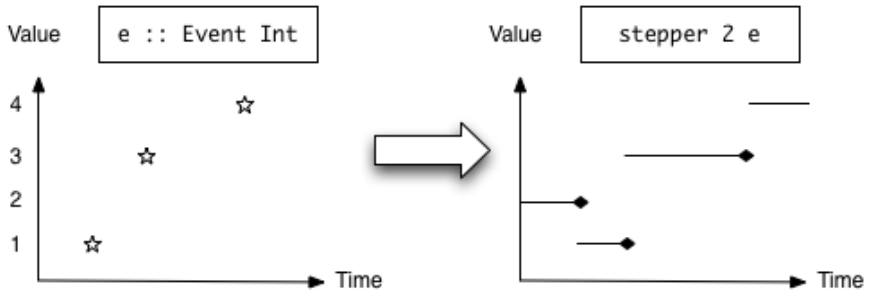
```
whenE :: Behavior Bool -> Event a -> Event a
```

Combining Events and Behaviors

And most usefully, a way to treat Events as Behaviors:

```
stepper :: monadIO m => a -> Event a -> m (Behavior a)
```

```
stepper x0 ex = return $ \time ->
    last (x0 : [x | (timex, x) <- ex, timex < time])
```



stepper is often combined with accumE. For convenience, this exists as:

```
accumB :: MonadIO m => a -> Event (a -> a) -> m (Behavior a)
```

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Week 12
GUI Development
Part 4 - Example of FRP style

CS4012
Topics in Functional Programming
Michaelmas Term 2020

Glenn Strong <glenn.Strong@scss.tcd.ie>

FRP Example

The same program in threepenny, but now using the FRP style

Let's remove the "IORef" from the canvas program:

```
import qualified Graphics.UI.Threepenny as UI
import Graphics.UI.Threepenny.Core
import Reactive.Threepenny
```

FRP Example

Setup is the same…

```
setup :: Window -> UI ()
setup window = do
  return window # set title "Clickable canvas"

canvas <- UI.canvas
    # set UI.height canvasSize
    # set UI.width canvasSize
    # set UI.style [("border", "solid black 1px"), "background", "#eee")]

  fillMode  <- UI.button #+ [string "Fill"]
  emptyMode <- UI.button #+ [string "Hollow"]
  clear     <- UI.button #+ [string "Clear"]

  getBody window #+
    [column [element canvas]
    , element fillMode, element emptyMode, element clear]
```

FRP Example

After that things start to deviate from the IOref design…

```
let
    efs :: Event (Mode -> Mode)
    efs = const Fill <$ UI.click fillMode
    ehs :: Event (Mode -> Mode)
    ehs = const NoFill <$ UI.click emptyMode
    modeEvent :: Event (Mode -> Mode)
    modeEvent = unionWith const efs ehs
```

The first event fires on mouse clicks. We don't want the actual click information, so we convert it to an event that returns a mode-transforming function.

Then we perform the same trick on the "empty" button, and finally combine the two event streams so that there is one Event that fires when either button is pressed.

FRP Example

We want to track the mouse position. There is an event that fires whenever the mouse moves, but what I'd really like is to have a behaviour that *always* tells me where the mouse is.

```
mousePos <- stepper (0,0) $ UI.mousemove canvas
```

```
mousePos :: Behavior (Double,Double)
```

We can also turn the mouse event stream into a behavior that gives us the current drawing mode at any time.

```
drawingMode <- accumB Fill modeEvent
```

```
drawingMode :: Behavior Modes
```

FRP Example

OK, almost there. Now we combine the two Behaviors and we'll have a behavior that describes the program state at any time…

```
let bst = (,) <$> drawingMode <*> mousePos
```

```
bst :: Behavior (Modes, (Double,Double))
```

This relies on the Applicative expectation that:

```
do
  x <- g
  y <- h
  return f x y
```

≡

```
f <$> g <*> h
```

FRP Example

I want an event that fires when we click on the canvas, and I want it to do something based on the behavior we've just defined.

```
let bst :: Behavior (Modes, (Double,Double))
    bst = (,) <$> drawingMode <*> mousePos

    eDraw :: Event (Modes, (Double,Double))
    eDraw = bst <@ UI.click canvas
```

FRP Example

Time to tie it all together:

```
onEvent eDraw $ \e -> do drawShape e canvas

on UI.click clear $ const $
  canvas # UI.clearCanvas
```

```
drawShape :: (Modes, (Double,Double)) -> Element -> UI ()
drawShape (Fill, (x,y)) canvas = do
  canvas # set' UI.fillStyle   (UI.htmlColor "black")
  canvas # UI.fillRect (x,y) 100 100
drawShape (NoFill, (x,y)) canvas = do
  canvas # set' UI.fillStyle   (UI.htmlColor "white")
  canvas # UI.fillRect (x,y) 100 100
```

FRP Summary

- There are quite a few FRP frameworks out there
- The author of Threepenny GUI also has Reactive Banana which binds to backends (Wx, SDL), for making native applications.
- The design is very similar (but not identical)
- Another that might be worth a look is Reflex (bindings to browser/ DOM, SDL, Gloss)

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

End of part 4

Glenn.Strong@scss.tcd.ie
https://scss.tcd.ie/Glenn.Strong/

Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

Thank you

glenn.Strong@scss.tcd.ie
https://scss.tcd.ie/Glenn.Strong/