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

# Week 4
# A DSL for animation

## CS4012

**Topics in Functional Programming**
**Michaelmas Term 2020**

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

# Animation

The final step with our Shape DSL is to introduce *animation*.

- We will model ways for a drawing to change over time
- To do this we will dip into the world of *Functional Reactive Programming* (FRP)
- This is an approach that tries to integrate (notionally) *continuous* functions, with *time flow* and *events*, into the functional style.
  - FRAN, Reactive Banana, and so on are larger frameworks that support this style.
  - We will keep it simple for our first foray, and consider only continuous functions.

# Animation

**Application domains for FRP include:**

- **Animation**
- **Robotics**
- **Computer Vision**
- **UI Programming**
- **Simulation**

**The original paper for this field is "**Functional Reactive Animation**" by Conal Elliott and Paul Hudak.**

# Signal functions

For our animation we will take the idea of time-varying functions (and leave out the FRP notion of "events" for now).

A "signal" is a function that emits values over time.

```
type Time           = Double

newtype Signal a = Signal {at :: Time -> a}
```

# Signal functions

**It might be that the signal produces the same value at all times. That's the simplest sort of signal function and we call it "constant".**

```
constant :: a -> Signal a
constant x = Signal $ const x
```

**"const" is a standard library function that always returns it's first argument (so in this case it discards the time component of the signal function).**

```
const :: a -> b -> a
```

# Signal functions

**Another basic signal function is one that reveals the current time**

```
timeS :: Signal Time
timeS = Signal id
```

# Signal functions

**A basic transformation function for signals would be one that applies a function to the signal at every moment in time.**

**Think of it as transforming the values in the stream that the signal produces**

```
mapS :: (a -> b) -> Signal a -> Signal b
```

**Actually… that's the function we would need to implement a Functor instance. So why not do it that way instead?**

```
instance Functor Signal where
  fmap = mapS
```

# Signal functions

Another scenario that arises is that we might have a signal that we want to apply different functions to at different times.

For example, in an animation there might be a function to translate shapes in one direction for a while, then a different signal to translate them in a different direction.

```
applyS :: Signal (a -> b) -> Signal a -> Signal b
```

You might recognise this as well…

# Signal functions

```
instance Applicative Signal where
  pure  = constant
  (<*>) = applyS
```

**We can finish the implementations now, I was just waiting for pure…**

```
instance Applicative Signal where
  pure x      = Signal $ const x
  fs <*> xs = Signal $ \t -> (fs `at` t) (xs `at` t)
```

```
instance Functor Signal where
  map f xs = pure f <*> xs
```

# Signal functions example

**Let's first look at drawing a simple sinusoid. Here's the basic data: a signal that contains a sinusoid:**

```
sinS :: Signal Double
sinS = fmap sin timeS
```

**I'd like to visualise this signal function as text, so I will create a few utility functions to help with that:**

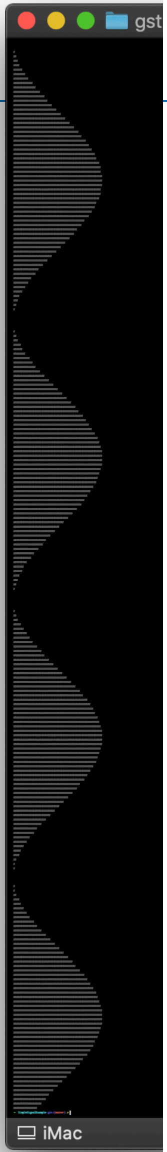
```
scale :: Num a =>  Signal a -> Signal a
scale = fmap ((30*) . (1+))

discretize :: Signal Double -> Signal Int
discretize  = fmap round

toBars :: Signal Int -> Signal String
toBars  = fmap (`replicate` '#')
```

# Signal functions example

By composing these operations I can get a series of snapshots of the sine wave, so by selecting carefully a sampling frequency (I choose intervals of 0.01) I get a believable view:




```
display :: Signal String -> IO ()
display ss = forM_ [0, 0.1 .. displayLength] $
               \x -> putStrLn (ss `at` x)

run :: IO ()
run = display . toBars . discretize . scale $ sinS
```

# Signal functions and Animation

We can use this approach to make some animations with our drawing.

Assuming we have our terminal drawing functions from the static version, we need a function of this type:

```
animate :: Window -> Time -> Time -> Signal Drawing -> IO ()
```

Loosely, this says: "Sample this signal between these two intervals and draw what you find into a window"

# Signal functions and Animation

**Drawing signals cal be very simple:**

```
staticDisc :: Signal Drawing
staticDisc = pure disc
     where disc = [(scale (point 0.5 0.5) <+>
                   translate (point 1.2 0.4), circle)]
```

**“Whenever you sample this signal you see the same thing: a small circle at (1.2, 0.4) on the plane”**

# Signal functions and Animation

**More interesting would be a signal that had something different at different times.**

**Start with a signal that has the Y coordinates of a ball that's bouncing up and down:**

```
bounceY :: Signal Double
bounceY = fmap (sin . (*3)) timeS
```

**Which I can make into a signal that gives displacement vectors:**

```
posS :: Signal Point
posS = pure point <*> pure 0.0 <*> bounceY
```

**Which I can make into a signal that gives Transformations:**

```
ts :: Signal Transform
ts = fmap translate posS
```

# Signal functions and Animation

**Now take a signal containing a single element of a drawing (a Transform, Shape pair):**

```
shapeS :: Signal (Transform, Shape)
shapeS = pure (scale (point 0.3 0.3), circle)
```

**To produce a signal containing this disc translated we can apply the transformation signal. We need a helped here, because transforms can’t be applied directly into (Transform,Shape) pairs.**

```
applyT :: Transform -> (Transform,Shape) -> (Transform,Shape)
applyT t (ts,s) = (ts <+> t, s)

ats = fmap addT tsS
```

# Signal functions and Animation

**Finally, we assemble a signal of drawings. First, apply the various transformations:**

```
movingShapeS :: Signal (Transform, Shape)
movingShapeS = ats <*> shapeS
```

**Now we need to convert the single drawing element "Signal (Transform, Shape)" to a Signal [(Transform,Shape)] to get our drawing:**

```
drawingS :: Signal Drawing
drawingS = fmap (:[]) movingShapeS
```

# Signal functions and Animation

## Another example

```
rotatingSquare :: Signal Drawing
rotatingSquare = fmap (:[]) $ fmap sq rs
     where
           rs :: Signal Transform
           rs = fmap rotate timeS

           sqs :: Transform -> Signal (Transform,Shape)
           sqs t = fmap sq rs

           sq :: Transform -> (Transform, Shape)
           sq t = ( scale (point 0.5 0.5) <+>
                    translate (point 1.2 0.4) <+> t, square)
```

# Signal functions and Animation

**A different animated shape:**

```
bouncingBall :: Signal Drawing
bouncingBall = fmap (:[]) $ fmap ball ( fmap translate pos )
       where bounceY = fmap (sin . (3*)) timeS
             bounceX = fmap (sin . (2*)) timeS
             pos = pure point <*> bounceX <*> bounceY
             ball t = ( t <+> scale (point 0.3 0.3), circle )
```

**I can combine the two drawings to produce a signal of compound drawings:**

```
joinDS :: Signal [a] -> Signal [a] -> Signal [a]
joinDS s0 s1 = (fmap ( (++) ) s0) <*> s1

example = bouncingBall `joinDS` rotatingSquare
```

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/