Functional Reactive Programming

Lecture 6,
Designing and Using Combinators
John Hughes

What is FRP?

- A DSEL designed for describing *behaviour which varies over time*.
- Functional "Reactive" Programs can react to events in their environment.
- First developed in the context of *Functional Reactive Animation* (Fran).
- Not a monad in sight...

Behaviours

The most basic concept in FRP:

```
Behaviour a ≅ Time -> a
```

Examples

```
time :: Behaviour Time
wiggle :: Behaviour Double
wiggle = sin (pi*time)
```

Overloading lets us treat behaviours as numbers.

Behaviours

The most basic concept in FRP:

```
Behaviour a ≅ Time -> a
```

Examples

```
time :: Behaviour Time
wiggle :: RealB
wiggle = sin (pi*time)
```

Abbreviate behaviour types

Behaviours and Animations

Behaviours need not be numeric.

```
clock :: StringB
clock = lift1 show time

clockImage :: ImageB
clockImage = stringBIm clock
```

Image behaviours are animations!

Moving Pictures

Moving pictures can be created by moveXY:

```
anim = moveXY wiggle 0 mary
  where mary = importBitmap "maryface.bmp"
```

```
moveXY x y = move (vector2XY x y)
```

Works on vectors

Stretching Pictures

Images can be rescaled, by a behaviour:

```
stretch wiggle mary
```

Delaying Behaviours

Behaviours can be delayed using later:

```
waggle = later 0.5 wiggle
```

Out of phase

In fact, we can transform the time in any way!

```
wiggle ‘timeTransform‘ (time/2)
```

Runs at half speed

Orbiting Mary

```
orbitingMary = moveXY wiggle waggle mary
```

More Orbiting Fun

```
orbit  b = moveXY wiggle waggle b

pic = orbit (stretch 0.5 (faster 3 (orbit mary)))
```

Combining Pictures

We can combine pictures with over:

```
pic = orbit (stretch 0.5 mary) ‘over‘ mary
```

Reactive Animations

So far, these animations ignore their environment. How can we make them *react* to the user?

```
displayU :: (User -> ImageB) -> IO ()
```

Can extract information about the user

Following the Mouse

```
mouseMotion :: User -> Vector2B
```

Follow the mouse:

```
move (mouseMotion u) mary
```

Follow the mouse with a delay:

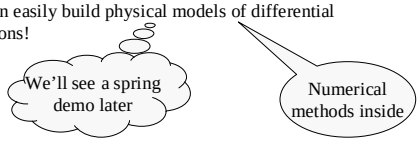
```
later 1 $ move (mouseMotion u) mary
```

Differential Calculus

We can even *differentiate* and *integrate* behaviours!

```
accel u = mouseMotion u
velocity u = integral (accel u) u
position u = initpos + integral (velocity u) u
```

We can easily build physical models of differential equations!



Reacting to Events

- Behaviours are *continuous*, but sometimes we should react to discrete *events*.
- Conceptually, events are Maybe a-behaviours!
- Implemented as a separate type.

```
Event a ≅ [(Time,a)]
```

Reacting to Events

```
untilB :: Behaviour a ->
          Event (Behaviour a) ->
            Behaviour a

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

Example: stop on mouse click

```
orbit mary `untilB` lbp u -=> mary
```

Mouse Button Events

```
lbp, lbr, rbp, rbr :: User -> Event ()
```

Left/right | button | Press/release

Let's make mary bigger while the mouse button is pressed!

```
size u = 0.5 `untilB` nextUser_ lbp u ==> \u'->
         1.0 `untilB` nextUser_ lbr u' ==> \u"->
         size u"
```

Event generates the next user state

Multiple Events

We can *combine* events, to wait for whichever happens first

```
updown n u =
  n `untilB` (nextUser_ lbp u ==> updown (n+1)
          .|. nextUser_ rbp u ==> updown (n-1))
```

```
stretch (0.3*updown 3 u) mary
```

Generating Events from Behaviours

Suppose we want to model a bouncing ball.

We must detect *collisions* -- when the position reaches the ground!

```
predicate :: BoolB -> User -> Event ()
```

Modelling a Bouncing Ball

```
accel u = -1
speed u = 1+integral (accel u) u
height u = integral (speed u) u `untilB`
        nextUser_ collision u ==> height
  where collision u =
        predicate (height u <* 0 &&*
                speed u <* (0::RealB)) u

ball = stretch 0.1 circle
```

Starred operators work on behaviours

Time for Conal Elliott's Demos...

Assessment

- Fran provides a small number of *composable* operations on behaviours and events.
- With these a rich variety of animations can be expressed
- Performance is good, since rendering is done by standard software
- FRP works in many other contexts:
  - *Frob* for robotics
  - *Fruit* for graphical user interfaces

How Does it Work? Representing Behaviours

```
Behaviour a = Time -> a
```

would be much too inefficient. We would need to recompute the entire history to do an integral!

```
Behaviour a = Time -> (a, Behaviour a)
```

Simplified (faster) behaviour, useable at later times.

How Does it Work? Detecting Predicate Events

```
predicate :: (Time->Bool) -> Event ()
```

would be far too inefficient!

We would need to try *every* time (double precision floats!) to be sure to detect events!

How Does it Work? Detecting Predicate Events

Key Idea: Interval analysis

```
data Ival a = a 'UpTo' a
```

Behaviours become:

```
data Behaviour a =
        Behaviour (Time -> (a, Behaviour a))
                  (Ival Time -> (Ival a, Behaviour a))
```

If f (t1'UpTo't2) = False'UpTo'False, the event does not occur between t1 and t2.

Summary

- FRP is a non-monadic DSEL which makes time-dependent behaviour very simple to express.
- Excellent example of capturing the *semantics of the application*.
- It's fun! Download Fran and try it out!

Summary

- FRP is a non-monadic DSEL which makes time-dependent behaviour very simple to express.
- Excellent example of capturing the *semantics of the application*.
- It's fun! Download Fran and try it out!

Now for Conal Elliott's latest: a quick Pan demo!