Introduction to FRP and Yampa through Games and Music

Henrik Nilsson, Ivan Perez, and George Giorgidze

School of Computer Science

University of Nottingham, UK

Aims and Overview (1)

- Introduction to ***Functional*** Reactive Programming
- Demonstrating that a pure, declarative, reactive approach covers more applications areas than what one might think.
- Through the above, give you a hopefully somewhat different and useful perspective on the reactive programming technologies that you are already using.

Aims and Overview (2)

Much will be familiar from reactive frameworks like Cycle.js or XStream:

- Circuit-like programming metaphor.
- Transforming streams by e.g. mapping or accumulation.
- Similar primitive streams, e.g. never, periodic.

Aims and Overview (3)

However, much is also different:

- Time (notionally) ***continuous***.
- But ***events*** (discrete time), also supported, allowing for ***hybrid*** systems.
- Synchronous (no race conditions)
- Declarative (emphasis on what, not how; fewer operational concerns)

Aims and Overview (4)

And specific to Yampa:

- "Stream processors" is the central, first-class, abstraction, while "streams" are secondary (very close to the circuit metaphor).
- High-level support for highly dynamic system structure (generalisation of XStream's `flatten`, no need for low-level attaching/detaching of listeners).
- Statically typed

Example: Feedback

Arranging feedback in XStream necessitates taking operational concerns, ***how***, into account:

```
var secondProxy$ = xs.create();
var first$ = secondProxy$.map ...;
var second$ = first$.map ...;
secondProxy$.imitate(second$);
```

Of course, the intent, ***what***, is really:

```
var first$ = second$.map ...;
var second$ = first$.map ...;
```

with "=" denoting equality rather than assignment.

Why Program Games Declaratively?

Video games are not a major application area for declarative programming . . . or even a niche one.

Perhaps not so surprising:

- Many pragmatical reasons: performance, legacy issues, . . .
- State and effects are pervasive in video games: Is declarative programming even a conceptually good fit?

But Why NOT, Really?

Many eloquent and compelling cases for functional programming in general:

- John Backus, 1977 ACM Turing Award Lecture: Can Programming Be Liberated from the von Neumann Style?
- John Hughes, recent retrospective: Why Functional Programming Matters (on YouTube, recommended)

One key point: Program with whole values, not a word-at-a-time. Which, of course, is the point also of frameworks like Cycle.js and XStreams.

Possible Gains (1)

With his Keera Studios hat on, Ivan's top three reasons:

- Reliability.
- Lower long-term maintenance cost.
- Lower production cost and fast time-to-prototype.

Possible Gains (2)

High profile people in the games industry have pointed out potential benefits:

- John D. Carmack, id Software: Wolfenstein 3D, Doom, Quake
- Tim Sweeney, Epic Games: The Unreal Engine

E.g. pure, declarative code:

- promotes parallelism
- eliminates many sources of errors

"Whole Values" for Games?

How should we go about writing video games "declaratively"?

In particular, what should those "whole values" be?

- Could be conventional entities like vectors, arrays, lists and aggregates of such.
- Could even be things like pictures.

But we are going to go one step further and consider programming with ***time-varying entities***.

Functional Reactive Programming

- Key idea: Don't program one-time-step-at-a-time, but describe an evolving entity as whole.
- FRP originated in Conal Elliott and Paul Hudak's work on Functional Reactive Animation (Fran). Highly cited 1997 ICFP paper; ICFP award for most influential paper in 2007.
- FRP has evolved in a number of directions and into different concrete implementations.
- We will use Yampa: an FRP system embedded in Haskell.

Take-home Message # 1

Video games can be programmed declaratively
by describing ***what*** entities are ***over*** time.

Our whole values are things like:

- The totality of input from the player
- The animated graphics output
- The entire life of a game object

We construct and work with ***pure*** functions on these:

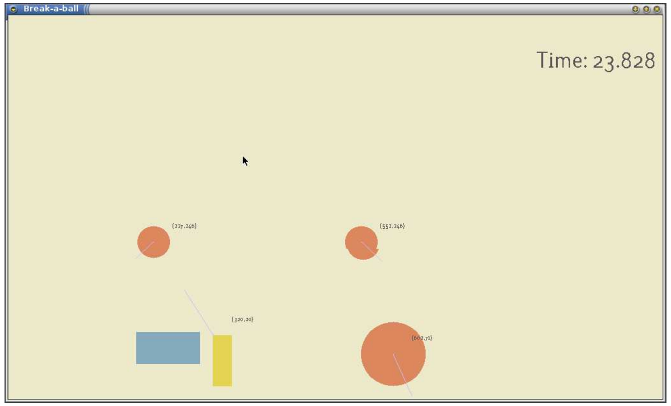
- The game: function from input to animation
- In the game: fixed point of function on collection of game objects

Take-home Message # 2

You too can program games declaratively . . . today!

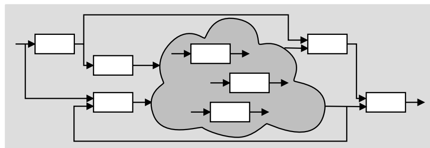
Take-home Game!

Or download one for free to your Android device!



Play Store: Pang-a-lambda (Keera Studios)

Yampa

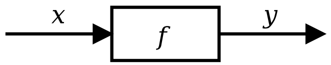
- FRP implementation embedded in Haskell
- Key notions:
  - ***Signals***: time-varying values (cf. streams)
  - ***Signal Functions***: pure functions on signals
  - ***Switching***: temporal composition of signal functions (cf. XStream's `flatten`)
- Programming model: 

Yampa?

Yampa is a river with long calmly flowing sections and abrupt whitewater transitions in between.

A good metaphor for hybrid systems!

Signal Functions



Intuition:

$$Time \approx \mathbb{R}$$
$$Signal\ a \approx Time \rightarrow a$$
$$x :: Signal\ T1$$
$$y :: Signal\ T2$$
$$SF\ a\ b \approx Signal\ a \rightarrow Signal\ b$$
$$f :: SF\ T1\ T2$$

Additionally, ***causality*** required: output at time t must be determined by input on interval $[0, t]$.

Some Basic Signal Functions

$$identity :: SF\ a\ a$$

$$constant :: b \rightarrow SF\ a\ b$$

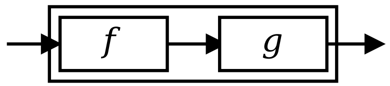
$$integral :: VectorSpace\ a\ s \Rightarrow SF\ a\ a$$

$$y(t) = \int_0^t x(\tau)\,\mathrm{d}\tau$$

Composition

In Yampa, systems are described by combining signal functions (forming new signal functions).

For example, serial composition:



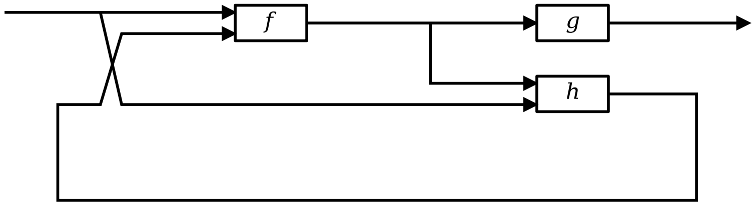
A ***combinator*** that captures this idea:

$$(\ggg) :: SF\ a\ b \rightarrow SF\ b\ c \rightarrow SF\ a\ c$$

Signal functions are the primary notion; signals a secondary one, only existing indirectly.

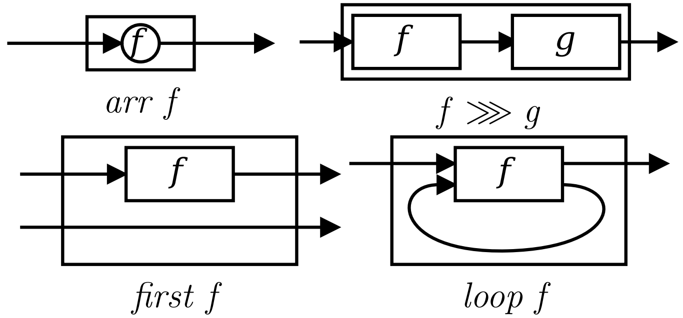
Systems

What about larger, more complicated networks?
How many combinators are needed?



John Hughes's ***Arrow*** framework provides a good answer!

The Arrow framework



$$\begin{aligned} &arr &&:: (a \rightarrow b) \rightarrow SF\ a\ b \\ &(\ggg) &&:: SF\ a\ b \rightarrow SF\ b\ c \rightarrow SF\ a\ c \\ &first &&:: SF\ a\ b \rightarrow SF\ (a, c)\ (b, c) \\ &loop &&:: SF\ (a, c)\ (b, c) \rightarrow SF\ a\ b \end{aligned}$$

Oscillator from Pang-a-lambda

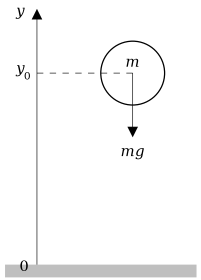
This oscillator determines the movement of blocks:

$$\begin{aligned} &osci\ ampl\ period = \mathbf{proc}\ \_ \rightarrow \mathbf{do} \\ &\quad \mathbf{rec} \\ &\qquad \mathbf{let}\ a = -(2.0 * pi\ /\ period) \uparrow 2 * p \\ &\qquad v \leftarrow \qquad\qquad\qquad integral \prec a \\ &\qquad p \leftarrow (ampl+)\hat{}\ll integral \prec v \\ &\quad returnA \prec p \end{aligned}$$

Direct transliteration of standard equations.

A Bouncing Ball

Lots of bouncing balls in Pang-a-lambda!



$$\begin{aligned} y &= y_0 + \int v\,\mathrm{dt} \\ v &= v_0 + \int -9.81 \end{aligned}$$

On impact:

$$v = -v(t-)$$

(fully elastic collision)

Modelling the Bouncing Ball: Part 1

Free-falling ball:

$$\begin{aligned}
&\mathbf{type}\ Pos = Double\\
&\mathbf{type}\ Vel = Double\\
&fallingBall :: Pos \rightarrow Vel \rightarrow SF\ ()\ (Pos, Vel)\\
&fallingBall\ y0\ v0 = \mathbf{proc}\ () \rightarrow \mathbf{do}\\
&\quad v \leftarrow (v0+) \hat{} \ll integral \prec -9.81\\
&\quad y \leftarrow (y0+) \hat{} \ll integral \prec v\\
&\quad returnA \prec (y, v)
\end{aligned}$$

Events

Yampa models discrete-time signals by lifting the ***co-domain*** of signals using an option-type:

$$\mathbf{data}\ Event\ a = NoEvent \mid Event\ a$$

Discrete-time signal = `Signal (Event` α`)`.

Some functions and event sources:

$$\begin{aligned}
&tag :: Event\ a \rightarrow b \rightarrow Event\ b\\
&after :: Time \rightarrow b \rightarrow SF\ a\ (Event\ b)\\
&edge :: SF\ Bool\ (Event\ ())
\end{aligned}$$

Modelling the Bouncing Ball: Part 2

Detecting when the ball goes through the floor:

$$\begin{aligned}
&fallingBall' ::\\
&\quad Pos \rightarrow Vel \rightarrow SF\ ()\ ((Pos, Vel), Event\ (Pos, Vel))\\
&fallingBall'\ y0\ v0 = \mathbf{proc}\ () \rightarrow \mathbf{do}\\
&\quad yv@(y, _) \leftarrow fallingBall\ y0\ v0 \prec ()\\
&\quad hit \leftarrow edge \prec y \leqslant 0\\
&\quad returnA \prec (yv, hit\ `tag`\ yv)
\end{aligned}$$

Switching

Q: How and when do signal functions “start”?

A:
- ***Switchers*** apply a signal functions to its input signal at some point in time.
- This is ***temporal composition*** of signal functions.

Switchers thus allow systems with ***varying structure*** to be described.

Generalised switches allow composition of ***collections*** of signal functions. Can be used to model e.g. varying number of objects in a game.

The Basic Switch

Idea:

- Allows one signal function to be replaced by another.
- Switching takes place on the first occurrence of the switching event source.

```
switch ::
  SF a (b, Event c)
  -> (c -> SF a b)
  -> SF a b
```


Modelling the Bouncing Ball: Part 3

Making the ball bounce:

```
bouncingBall :: Pos -> SF () (Pos, Vel)
bouncingBall y0 = bbAux y0 0.0
  where
    bbAux y0 v0 =
      switch (fallingBall' y0 v0) $ \(y, v) ->
      bbAux y (-v)
```


Game Objects

```
data Object = Object { objectName :: ObjectName
                     , objectKind :: ObjectKind
                     , objectPos  :: Pos2D
                     , objectVel  :: Vel2D
                     ...
                     }

data ObjectKind = Ball ... | Player ... | ...

data ObjectInput = ObjectInput
  { userInput  :: Controller
  , collisions :: Collisions
  }
```


Overall Game Structure

```
gamePlay :: [ListSF ObjectInput Object]
              -> SF Controller ([Object], Time)
gamePlay objs = loopPre [] $
  proc (input, cs) -> do
    let oi = ObjectInput input cs
    ol <- dlSwitch objs -< oi
    let cs' = detectCollisions ol
    tLeft <- time -< ()
    returnA -< ((ol, tLeft), cs')
```

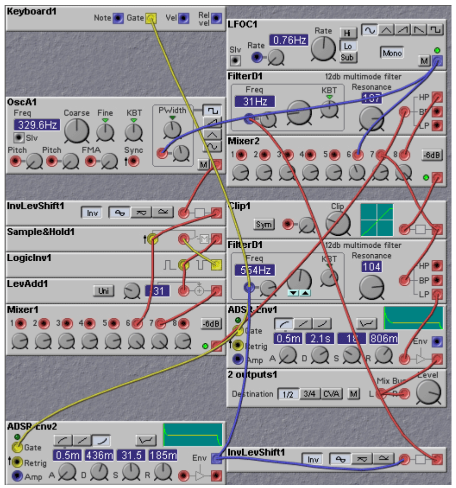
ListSF and *dlSwitch* are related abstractions that allow objects to die or spawn new ones.

And now for something different . . .

Switched-on Yampa: Programming Modular Synthesizers in Haskell

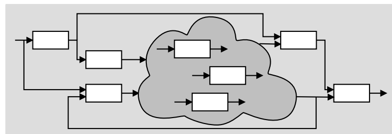
Modular synthesizers?

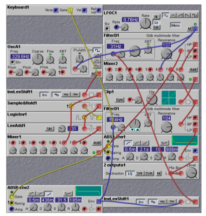
Modern Modular Synthesizers



Where does Yampa enter the picture?

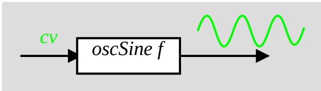
- Music can be seen as a hybrid phenomenon. Thus interesting to explore a hybrid approach to programming music and musical applications.
- Yampa's programming model is very reminiscent of programming modular synthesizers:





- Fun application! Useful for teaching?

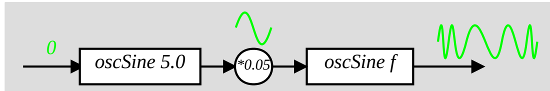
Example 1: Sine oscillator



$$
\begin{array}{l}
oscSine :: Frequency \rightarrow SF\ \ CV\ \ Sample \\
oscSine\ f0 = \mathbf{proc}\ cv \rightarrow \mathbf{do} \\
\quad \mathbf{let}\ f = f0 * (2 ** cv) \\
\quad phi \leftarrow integral \prec 2 * pi * f \\
\quad returnA \prec sin\ phi
\end{array}
$$

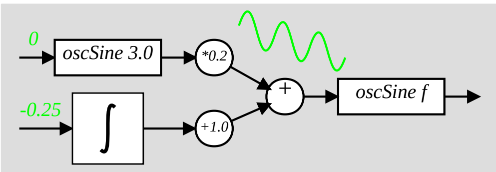
$$constant\ 0 \ggg oscSine\ 440$$

Example 2: Vibrato



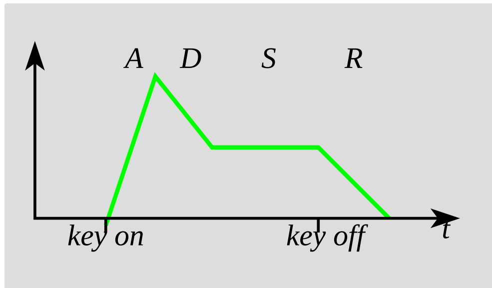
$$
\begin{array}{l}
constant\ 0 \\
\ggg oscSine\ 5.0 \\
\ggg arr\ (*0.05) \\
\ggg oscSine\ 440
\end{array}
$$

Example 3: 50’s Sci Fi



$$
\begin{array}{lll}
sciFi :: SF\ ()\ Sample & & \\
sciFi = \mathbf{proc}\ () \rightarrow \mathbf{do} & & \\
\quad und \leftarrow arr\ (*0.2) \lll oscSine\ 3.0 & \prec & 0 \\
\quad swp \leftarrow arr\ (+1.0) \lll integral & \prec & -0.25 \\
\quad audio \leftarrow oscSine\ 440 & \prec & und + swp \\
\quad returnA \prec audio & &
\end{array}
$$

Envelope Generators (1)



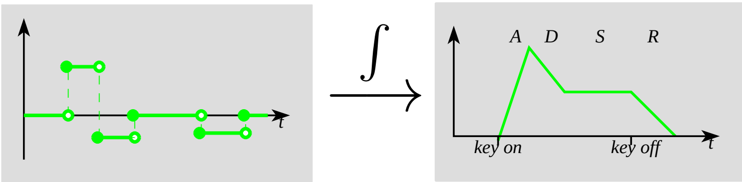
$$
\begin{array}{l}
envGen :: CV \rightarrow [(Time, CV)] \rightarrow (Maybe\ Int) \\
\qquad\qquad \rightarrow SF\ (Event\ ())\ (CV, Event\ ())
\end{array}
$$

$$
\begin{array}{l}
envEx = envGen\ 0\ [(0.5, 1), (0.5, 0.5), (1.0, 0.5), (0.7, 0)] \\
\qquad\qquad\qquad (Just\ 3)
\end{array}
$$

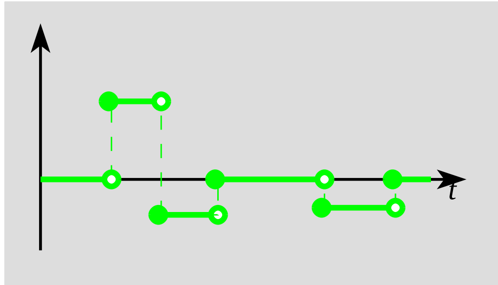
Envelope Generators (2)

How to implement?

Integration of a step function yields suitable shapes:

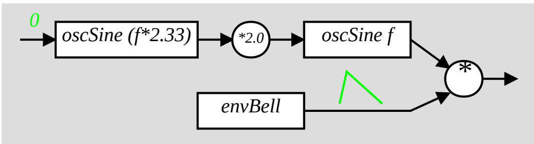


Envelope Generators (3)



$$\begin{aligned}
&\mathit{afterEach} :: [(\mathit{Time}, b)] \rightarrow \mathit{SF}\ a\ (\mathit{Event}\ b)\\
&\mathit{hold} \quad :: a \rightarrow \mathit{SF}\ (\mathit{Event}\ a)\ a\\
&\mathit{steps} = \mathit{afterEach}\ [(0.7, 2), (0.5, -1), (0.5, 0), (1, -0.7), (0.7, 0)]\\
&\qquad \ggg \mathit{hold}\ 0
\end{aligned}$$

Example 4: Bell



$$\begin{aligned}
&\mathit{bell} :: \mathit{Frequency} \rightarrow \mathit{SF}\ ()\ (\mathit{Sample}, \mathit{Event})\\
&\mathit{bell}\ f = \mathbf{proc}\ () \rightarrow \mathbf{do}\\
&\quad m \qquad\qquad\ \leftarrow \mathit{oscSine}\ (2.33 * f) \prec 0\\
&\quad \mathit{audio} \qquad\ \leftarrow \mathit{oscSine}\ f \qquad\quad \prec 2.0 * m\\
&\quad (\mathit{ampl}, \mathit{end}) \leftarrow \mathit{envBell} \qquad\quad\ \prec \mathit{noEvent}\\
&\quad \mathit{returnA} \prec (\mathit{audio} * \mathit{ampl}, \mathit{end})
\end{aligned}$$

Example 5: Tinkling Bell

$$\begin{aligned}
&\mathit{tinkle} :: \mathit{SF}\ ()\ \mathit{Sample}\\
&\mathit{tinkle} = (\mathit{repeatedly}\ 0.25\ 84\\
&\qquad\qquad \ggg \mathit{constant}\ ()\\
&\qquad\qquad\quad \&\&\&\ \mathit{arr}\ (\mathit{fmap}\ (\mathit{bell} \circ \mathit{midiNoteToFreq}))\\
&\qquad\qquad \ggg \mathit{rSwitch}\ (\mathit{constant}\ 0))
\end{aligned}$$

Example 6: Playing simultaneous notes

$$
\begin{array}{l}
\mathit{mysterySong} :: \mathit{SF}\ ()\ (\mathit{Sample}, \mathit{Event}\ ()) \\
\mathit{mysterySong} = \mathbf{proc}\ \_ \rightarrow \mathbf{do} \\
\quad t \ \leftarrow \mathit{tinkle} \quad \prec ()\\
\quad m \leftarrow \mathit{mystery} \prec () \\
\quad \mathit{returnA} \prec (0.4 * t + 0.6 * m)
\end{array}
$$

Switched-on Yampa?

Software and paper: `www.cs.nott.ac.uk/~ggg`

Conclusions

- FRP offers one way to write interactive games and similar software in a declarative way.
- It allows systems to be described in terms of whole values varying over time.
- Not covered in this talk:
  - Not everything fit easily into the FRP paradigm: e.g., interfacing to existing GUI toolkits, other imperative APIs.
  - But also such APIs can be given a “whole-value treatment” to improve the fit within a declarative setting. E.g. ***Reactive Values and Relations***.