Functional Reactive Programming (FRP)

- Framework for reactive programming in a functional setting
- Systems described by composing *signal functions*: functions mapping *signals* to *signals*
- Originated from Functional Reactive Animation (Fran) (Elliott & Hudak)

*Yampa* is our latest implementation of FRP.
George Russel said on the Haskell GUI list:

“I have to say I’m very sceptical about things like Fruit which rely on reactive animation, ever since I set our students an exercise implementing a simple space-invaders game in such a system, and had no end of a job producing an example solution. . . .
The Challenge

George Russel said on the Haskell GUI list:

... Things like getting an alien spaceship to move slowly downward, moving randomly to the left and right, and bouncing off the walls, turned out to be a major headache. Also I think I had to use 'error' to get the message out to the outside world that the aliens had won. ...
The Challenge

George Russel said on the Haskell GUI list:

My suspicion is that reactive animation works very nicely for the examples constructed by reactive animation folk, but not for my examples.”
What was wrong?

Possible reasons for George Russel’s reaction:

- Original reactive animation systems like Fran and FAL lacked crucial features. *Yampa attempts to address this [Haskell Workshop ’02]*
- Not many examples of good FRP code around. *The present paper attempts to address that*
This talk tries to convey that FRP/Yampa

- is a reasonable approach for this kind of applications
- has some unique advantages over other approaches
The Game
Our most recent FRP implementation is called **Yampa**:

- Embedding in Haskell; i.e. a Haskell library.
- Arrows used as the basic structuring framework.
- Advanced switching constructs allows for description of systems with highly dynamic structure.
Key concept: functions on signals.

Intuition:

\[ \text{Signal } \alpha \approx \text{Time } \rightarrow \alpha \]
\[ x :: \text{Signal T1} \]
\[ y :: \text{Signal T2} \]
\[ f :: \text{Signal T1} \rightarrow \text{Signal T2} \]

Additionally: \textit{causality} requirement.
Signal Functions in Yampa

- Signal Functions are *first class entities*. Intuition: $SF \alpha \beta \approx \text{Signal } \alpha \rightarrow \text{Signal } \beta$
- Signals are *not* first class entities: they only exist indirectly through signal functions.
- The strict separation between signals and signal functions distinguishes Yampa from earlier FRP implementations.
Signal Functions and State

Alternative view:

Functions on signals can encapsulate state.

\[ f \left[ state(t) \right] \]

\( state(t) \) summarizes input history \( x(t'), t' \in [0, t] \).

Functions on signals are either:

- **Stateful**: \( y(t) \) depends on \( x(t) \) and \( state(t) \)
- **Stateless**: \( y(t) \) depends only on \( x(t) \)
Describing Systems

Systems are described by combining signal functions into more complex signal functions:
Yampa and Arrows

Yampa uses John Hughes’ *arrow* framework. Core Signal Function combinators:

- `arr :: (a -> b) -> SF a b`
- `>>> :: SF a b -> SF b c -> SF a c`
- `first :: SF a b -> SF (a,c) (b,c)`
- `loop :: SF (a,c) (b,c) -> SF a b`

Enough to express any conceivable “wiring”.
The Arrow Syntactic Sugar

Using the basic combinators directly is often very cumbersome. Ross Paterson’s syntactic sugar for arrows provides a convenient alternative:

```
proc pat -> do [ rec ]
    pat₁ <- sfexp₁ <- exp₁
    pat₂ <- sfexp₂ <- exp₂
    ...
    patₙ <- sfexpₙ <- expₙ
    returnA <- exp
```

Also: `let pat = exp ≡ pat <- arr id <- exp`
Describing the Alien Behavior (1)

type Object = SF ObjInput ObjOutput

alien :: RandomGen g =>
    g -> Position2 -> Velocity -> Object
alien g p0 vyd = proc oi -> do
  rec
    -- Pick a desired horizontal position
    rx <- noiseR (xMin, xMax) g <- ()
    smpl <- occasionally g 5 () <- ()
    xd <- hold (point2X p0) <- smpl `tag` rx
    ...

Describing the Alien Behavior (2)

...  

```javascript
-- Controller
let axd = 5 * (xd - point2X p)
    - 3 * (vector2X v)
ayd = 20 * (vyd - (vector2Y v))
ad  = vector2 axd ayd
h   = vector2Theta ad
...
```
Describing the Alien Behavior (3)

...  

--- Physics

let a = vector2Polar

(min alienAccMax

(vector2Rho ad))

h

vp <- iPre v0 <- v

ffi <- forceField <- (p, vp)

v <- (v0 ^+^) ^<< impulseIntegral

<- (gravity ^+^ a, ffi)

p <- (p0 .+^) ^<< integral <- v

...
Describing the Alien Behavior (4)

...

-- Shields

sl <- shield <- oiHit oi
die <- edge <- sl <= 0

returnA <- ObjOutput

  ooObsObjState = oosAlien p h v,
  ooKillReq = die,
  ooSpawnReq = noEvent

where

  v0 = zeroVector
Overall Game Structure

dpSwitch

route

ObjInput

alien

ObjOutput

gun

bullet

alien

killOrSpawn

[ObjectOutput]
Dynamic Signal Function Collections

Idea:

- Switch over *collections* of signal functions.
- On event, “freeze” running signal functions into collection of signal function *continuations*, preserving encapsulated *state*.
- Modify collection as needed and switch back in.
Need ability to express:

- How input routed to each signal function.
- When collection changes shape.
- How collection changes shape.

\[
dpSwitch :: \text{Functor} \ col \Rightarrow
\begin{align*}
& (\forall sf . (a \to \col sf \to \col (b,sf))) \\
& \to \col (SF b c) \\
& \to SF (a, \col c) \ (\text{Event} \ d) \\
& \to (\col (SF b c) \to d \to SF a (\col c)) \\
& \to SF a (\col c)
\end{align*}
\]
**dpSwitch**

Need ability to express:

- How input routed to each signal function.
- When collection changes shape.
- How collection changes shape.

\[
\text{dpSwitch :: Functor col =>} \\
\quad (\forall \text{sf . } (a \to \text{col sf} \to \text{col (b, sf)})) \\
\quad \to \text{col (SF b c)} \\
\quad \to \text{SF (a, col c) (Event d)} \\
\quad \to (\text{col (SF b c) \to d} \to \text{SF a (col c)}) \\
\quad \to \text{SF a (col c)}
\]
**dpSwitch**

Need ability to express:

- How input routed to each signal function.
- When collection changes shape.
- How collection changes shape.

\[
\text{dpSwitch :: Functor } \text{col} \Rightarrow \\
(\forall \text{sf} . \, (a \to \text{col}\, \text{sf} \to \text{col}\,(b,\text{sf}))) \\
\to \text{col}\,(\text{SF}\,b\,c) \\
\to \text{SF}\,(a, \text{col}\,c)\,(\text{Event}\,d) \\
\to (\text{col}\,(\text{SF}\,b\,c) \to d \to \text{SF}\,a\,(\text{col}\,c)) \\
\to \text{SF}\,a\,(\text{col}\,c)
\]
dpSwitch

Need ability to express:

- How input routed to each signal function.
- When collection changes shape.
- How collection changes shape.

dpSwitch :: Functor col =>
  (forall sf . (a -> col sf -> col (b,sf)))
  -> col (SF b c)
  -> SF (a, col c) (Event d)
  -> (col (SF b c) -> d -> SF a (col c))
  -> SF a (col c)
dpSwitch

Need ability to express:

- How input routed to each signal function.
- When collection changes shape.
- How collection changes shape.

$$\text{dpSwitch} :: \text{Functor } \text{col} \Rightarrow$$

$$(\forall \text{sf} . \ (\text{a} \to \text{col sf} \to \text{col} \ (\text{b}, \text{sf})))$$

$\to \ \text{col} \ (\text{SF b c})$

$\to \ \text{SF} \ (\text{a}, \ \text{col} \ \text{c}) \ (\text{Event} \ \text{d})$

$\to \ ((\text{col} \ (\text{SF b c}) \to \text{d} \to \text{SF} \ \text{a} \ (\text{col} \ \text{c})))$

$\to \ \text{SF} \ \text{a} \ (\text{col} \ \text{c})$
The Game Core

gameCore :: IL Object
    -> SF (GameInput, IL ObjOutput)
        (IL ObjOutput)
gameCore objs =
    dpSwitch route
        objs
        (arr killOrSpawn >>> notYet)
        (\sfs' f -> gameCore (f sfs'))
game :: RandomGen g =>
g -> Int -> Velocity -> Score ->
SF GameInput ((Int, [ObsObjState]),
    Event (Either Score Score))
game g nAliens vydAlien score0 = proc gi -> do
    rec
    oos <- gameCore objs0 <- (gi, oos)
    score <- accumHold score0
    <- aliensDied oos
    gameOver <- edge <- alienLanded oos
    newRound <- edge <- noAliensLeft oos
    ...
    ...
    ...
    ...

Closing the Feedback Loop (1)
returnA <- ((score,
    map ooObsObjState
    (elemsIL oos)),
  (newRound ‘tag’ (Left score))
  ‘lMerge’ (gameOver
          ‘tag’ (Right score)))

where
obj0 =
    listToIL
    (gun (Point2 0 50)
         : mkAliens g (xMin+d) 900 nAliens)
Other approaches?

Transition function operating on world model with explicit state (e.g. Asteroids by Lüth):

- Model snapshot of world with all state components.
- Transition function takes input and current world snapshot to output and the next world snapshot.

One could also use this technique within Yampa to avoid switching over dynamic collections.
Why use Yampa, then?

- Yampa provides a lot of functionality for programming with time-varying values:
  - captures common patterns
  - packaged in a way that makes reuse very easy

- Yampa allows state to be nicely encapsulated by signal functions:
  - avoids keeping track of all state globally
  - adding more state is easy and usually does not imply any major changes to type or code structure
State in *alien*

Each of the following signal functions used in *alien* encapsulate state:

- noiseR
- occasionally
- hold
- iPre
- forceField
- impulseIntegral
- integral
- shield
- edge
Drawbacks of Yampa?

- Choosing the right switch can be tricky.
- Subtle issues concerning when to use e.g. `iPre, notYet`.
- Syntax could be improved (with specialized pre-processor).