The challenge

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

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.

Example: Space Invaders

Overall game structure

```
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)
```
Routing

Idea:

- The routing function decides which parts of the input to pass to each running signal function instance.
- It achieves this by pairing a projection of the input with each running instance:

$$f : \text{col} \times \text{sf} \rightarrow \text{col} \times (\text{b}, \text{sf})$$

The routing function type

Universal quantification over the collection members:

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

Collection members thus opaque:

- Ensures only signal function instances from argument can be returned.
- Unfortunately, does not prevent duplication or discarding of signal function instances.

Aging (1)

The primitive \text{age} continuously makes a frozen, aged, version of its argument signal function available:

$$\text{age} :: \text{SF} \times \text{a} \times \text{b} \rightarrow \text{SF} \times (\text{a}, \text{b})$$

This is used to define the simple continuation-based switched, \text{kSwitch}:

$$\text{kSwitch} :: \forall \text{sf} . (\forall \text{a} . (\text{a} \rightarrow \text{SF} \times \text{b} \rightarrow \text{SF} \times (\text{a}, \text{b})))$$

Aging (2)

$$\text{kSwitch} \text{sf1 sfe k} = \text{switch sf} (\lambda (\text{c}, \text{sf1'}) \rightarrow \text{k sf1'} \text{ c})$$

where

$$\text{sf} \equiv (\forall \text{a} . (\text{a} \rightarrow \text{Event} \rightarrow \text{SF} \times \text{b} \rightarrow \text{SF} \times (\text{a}, \text{b})))$$

Aging of collections:

$$\text{agePar} :: \forall \text{col} . \forall \text{sf} . (\forall \text{a} . (\text{a} \rightarrow \text{SF} \times \text{b} \rightarrow \text{SF} \times (\text{a}, \text{b})))$$

This can be used to define \text{pSwitch} and \text{dpSwitch} in terms of \text{switch} and \text{dSwitch}, respectively, in a similar way to \text{kSwitch}.

Side note: Application of Freezing

The DIVE virtual reality environment (Blom 2009), implemented using an somewhat customised version of Yampa, allows objects in a virtual world to be continuously manipulated in a similar way to a real world.

But in such a setting, how to implement an \text{undo} facility?

DIVE used Yampa’s capability to age and freeze signal functions to good effect: whenever a point is reached one might want to return to, just capture the aged signal function representing the system and store it for possible later use.

The game core

In the game core:

$$\text{gameCore} :: \forall \text{Object} . \forall \text{ObjInput} . \forall \text{ObjOutput}$$

$$\text{alien g p0 v0} = \text{proc} \text{oi} \rightarrow \text{do}$$

Describing the alien behavior (1)

$$\text{type Object} = \forall \text{ObjInput} . \forall \text{ObjOutput}$$

$$\text{alien g p0} = \text{proc} \text{oi} \rightarrow \text{do}$$

$$\text{Pick a desired horizontal position}$$

$$\text{rx} \leftarrow \text{noiseR} (\text{xMin}, \text{xMax})$$

$$\text{smpl} \leftarrow \text{occasionally} \text{g} \rightarrow \text{Object}$$

$$\text{xd} \leftarrow \text{hold} (\text{point2X p0})$$

$$\text{...}$$
Describing the alien behavior (2)

```
-- 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
```

Recap: Overall game structure

---

Other functional 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.
  - Carefully designed to facilitate reuse.
- Yampa allows state to be nicely encapsulated by signal functions:
  - Avoids keeping track of all state globally.
  - Adding more state 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
- impulseIntegral
- occasionally
- integral
- hold
- shield
- iPre
- edge
- forceField

Closing the feedback loop (1)

```
game :: RandomGen g =>
    g -> Int -> Velocity -> Score ->
    SF GameInput ((Int, [ObsObjState]),
    Event (Either Score Score))
game g nAliens vydAlien score0 = proc gi -> do
    oos <- gameCore objs0 <- (gi, oos)
    score <- accumHold score0
                <- aliensDied oos
    gameOver <- edge <- alienLanded oos
    newRound <- edge <- noAliensLeft oos
    ```

Closing the feedback loop (2)

```
returnA <- {(score,
            map ooObsObjState
            (elemsIL oos)),
    (newRound 'tag' (Left score))
    'lMerge' (gameOver
                'tag' (Right score))
where
    objs0 = listToIL
        (gun (Point2 0 50)
        : mkAliens g (xMin+d) 900 nAliens)
```

ITU FRP 2010: Lecture 4 – p.19/31

ITU FRP 2010: Lecture 4 – p.20/31

ITU FRP 2010: Lecture 4 – p.21/31

ITU FRP 2010: Lecture 4 – p.22/31

ITU FRP 2010: Lecture 4 – p.23/31

ITU FRP 2010: Lecture 4 – p.24/31

ITU FRP 2010: Lecture 4 – p.25/31

ITU FRP 2010: Lecture 4 – p.26/31

ITU FRP 2010: Lecture 4 – p.27/31
**Why not imperative, then?**

If state is so important, why not stick to imperative/object-oriented programming where we have "state for free"?

- Advantages of declarative programming retained:
  - High abstraction level.
  - Referential transparency, algebraic laws: formal reasoning ought to be simpler.
- Synchronous approach avoids "event-call-back soup", meaning robust, easy-to-understand semantics.

**Elerea**

For an entirely different approach to dynamic collections of time-varying entities, see Elerea (Patai 2010):

- Elera has first class signals, and thus signals can carry collections of signals.
- A signal carrying a collection of signals is turned into a signal carrying a signal of a collection of values.
- A signal if signals is given meaning through a monadic join:

```haskell
join :: S (S a) -> S a
```

**Reading (1)**


**Reading (2)**