Highly dynamic system structure?

The basic switch allows one signal function to be replaced by another.

- What about more general structural changes?
- What about state?

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.”
Example: Space Invaders

Overall game structure

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.
dpSwitch

Need ability to express:

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

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

The routing function type

Universal quantification over the collection members:

\[
\text{Functor} \ col \Rightarrow \\
(\forall sf . (a \to \text{col} \ sf \to \text{col} \ (b, sf)))
\]

Collection members thus \textit{opaque}:

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

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:

\[
\begin{align*}
\begin{array}{c}
a \\
\text{col sf} \\
\text{col (b, sf)}
\end{array}
\end{align*}
\]

\[
f
\]

The game core

\[
\text{gameCore} :: \text{IL Object} \\
\to \text{SF} \ (\text{GameInput}, \text{IL ObjOutput}) \\
\quad \quad \quad \quad \quad \quad (\text{IL ObjOutput})
\]

\[
\text{gameCore} \ \text{objs} = \\
\text{dpSwitch} \ \text{route} \\
\text{objs} \\
\quad (\text{arr killOrSpawn} >>> \text{notYet}) \\
\quad (\text{\textbackslash sfs' f} \to \text{gameCore} (f \ \text{sfs'}))
\]
Describing the alien behavior (1)

```haskell
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)

```haskell```

```
Describing the alien behavior (3)

```

```
Describing the alien behavior (4)

```haskell```

```
```
Recap: Overall game structure

Closing the feedback loop (1)

```haskell
game :: RandomGen g =>
g -> Int -> Int -> Velocity -> Score ->
SF GameInput ((Int, [ObsObjState]),
                Event (Either Score Score))
gevent 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)

```haskell
... returnA <- ((score,
                map ooObsObjState
                (elemIL oos)),
               (newRound 'tag' (Left score))
               'lMerge' (gameOver
                         'tag' (Right score)))

where
  objs0 =
    listToIL
    (gun (Point2 0 50)
     : mkAliens g (xMin+d) 900 nAliens)
```

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
- occasionally
- hold
- iPre
- forceField

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.

Yet some more reading

Obtaining Slides and Yampa

The lecture slides will be available from:
http://www.cs.nott.ac.uk/~nhn

Yampa 0.92 is available from
http://www.haskell.org/yampa