• Describing systems with highly dynamic structure: a generalized switch-construct.
• Example: Space Invaders
Highly dynamic system structure?

The basic switch allows one signal function to be replaced by another.
Highly dynamic system structure?

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

- What about more general structural changes?
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. . . .
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.”
Example: Space Invaders
Overall game structure

dpSwitch

route

alien

gun

alien

bullet

killOrSpawn

ObjInput

ObjOutput

[ObjectOutput]
Dynamic signal function collections

Idea:
Dynamic signal function collections

Idea:

- Switch over *collections* of signal functions.
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*.
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.
Dynamic signal function collections

\[ S_0 \]

\[ S_1 \]
Dynamic signal function collections

\( S_1 \)

\( S_0 \)

\( t_e \)
Dynamic signal function collections
Dynamic signal function collections

\[ S_0, S_1, S_2, S_3 \]

\[ t_e \]
Dynamic signal function collections

\[ S_0, S_1, S_2, S_3 \]

\[ t_e, t_{e_2} \]
Dynamic signal function collections

\[ S_3, S_2, S_1, S_0 \]

\[ t_e, t_{e_2} \]
dpSwitch

Need ability to express:

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

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

Need ability to express:

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

dpSwitch :: Functor col =>
(foreall 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)
Need ability to express:

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

dpSwitch :: Functor col =>

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

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.
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:
The routing function type

Universal quantification over the collection members:

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

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.
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'))
Describing the alien behavior (1)

type Object = SF ObjInput ObjOutput

describingalien randomGen g =>
g -> Position2 -> Velocity -> Object
describingalien 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)

...<br><br>```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
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
  rec
    oos <- gameCore objs0 <- (gi, oos)
    score <- accumHold score0
    <- aliensDied oos
    gameOver <- edge <- alienLanded oos
    newRound <- edge <- noAliensLeft oos
  ...
...
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)
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
- impulseIntegral
- integral
- shield
- edge
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.
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.
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