ITU FRP 2010

Lecture 4: Dynamic System Structure

Henrik Nilsson

School of Computer Science and Information Technology

University of Nottingham, UK

Outline

- Describing systems with highly dynamic structure: a generalized `switch`-construct.
- Example: Space Invaders

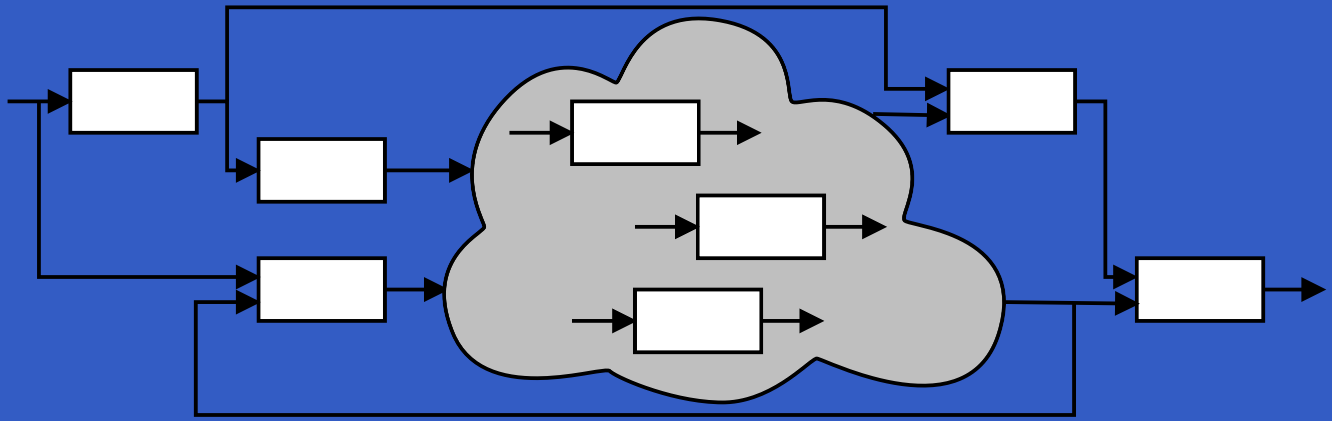
Highly dynamic system structure?

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.

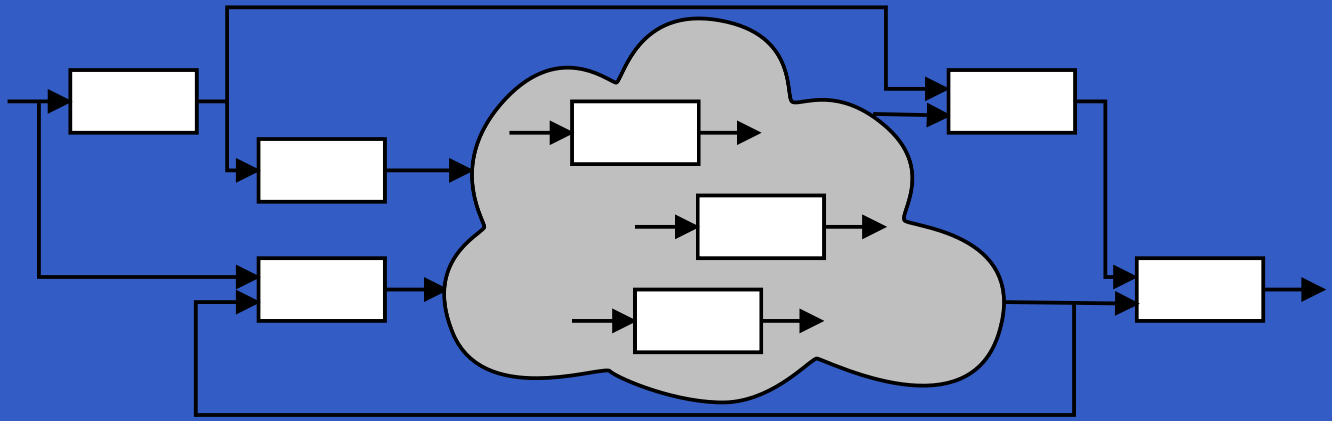
- What about more general structural changes?



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. . . .

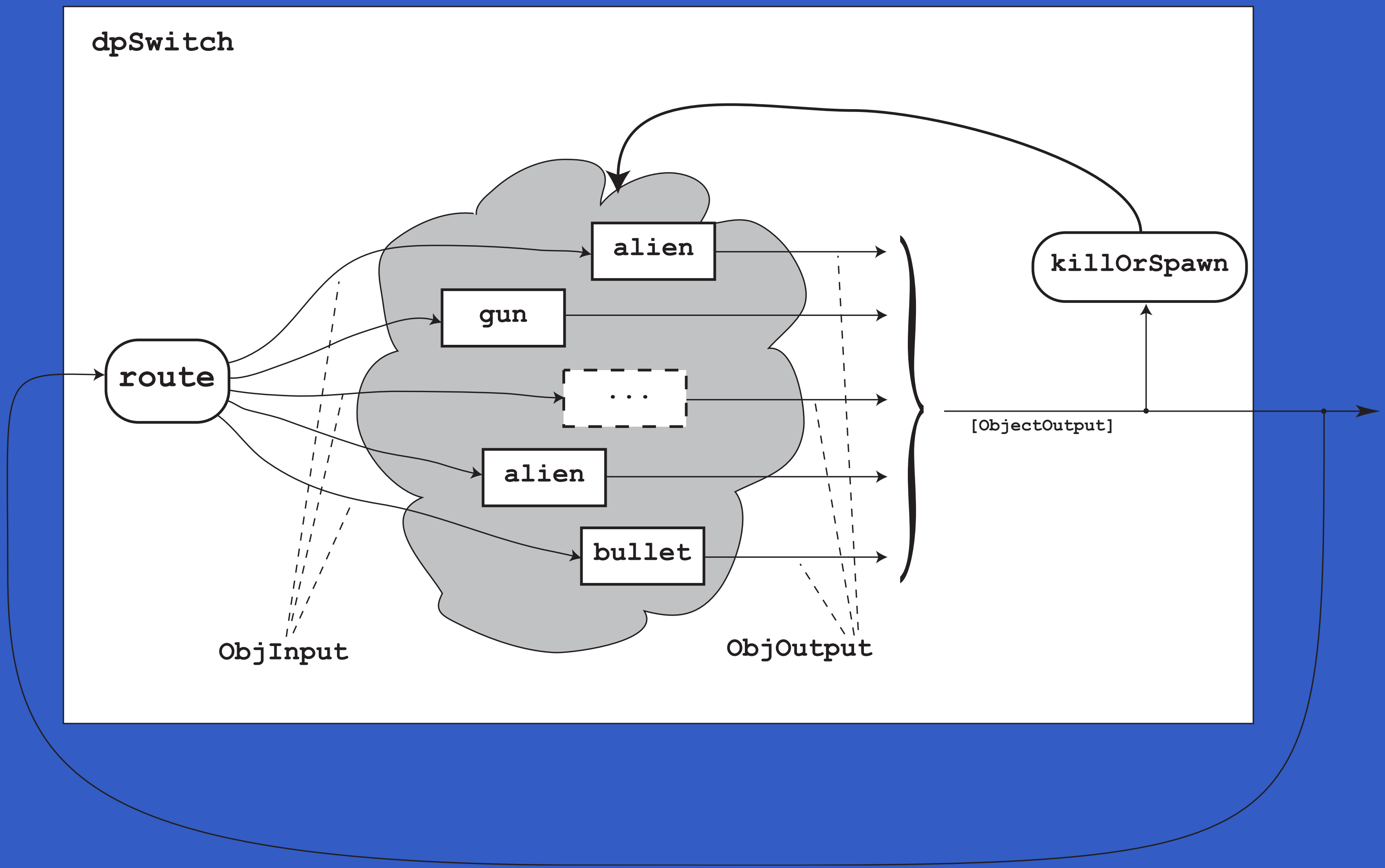
The challenge

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."

Example: Space Invaders

Overall game structure



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***.

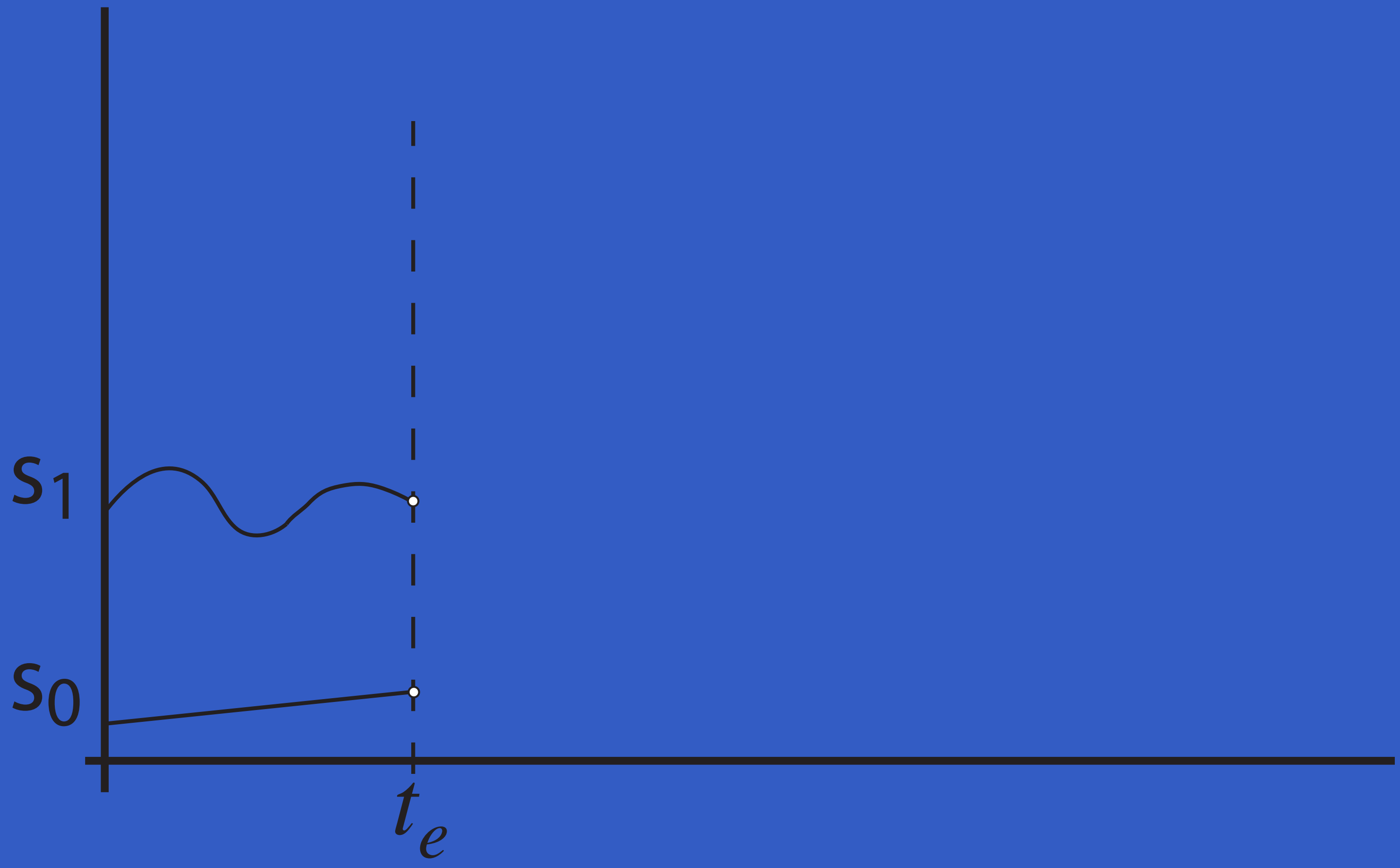
Dynamic signal function collections

Idea:

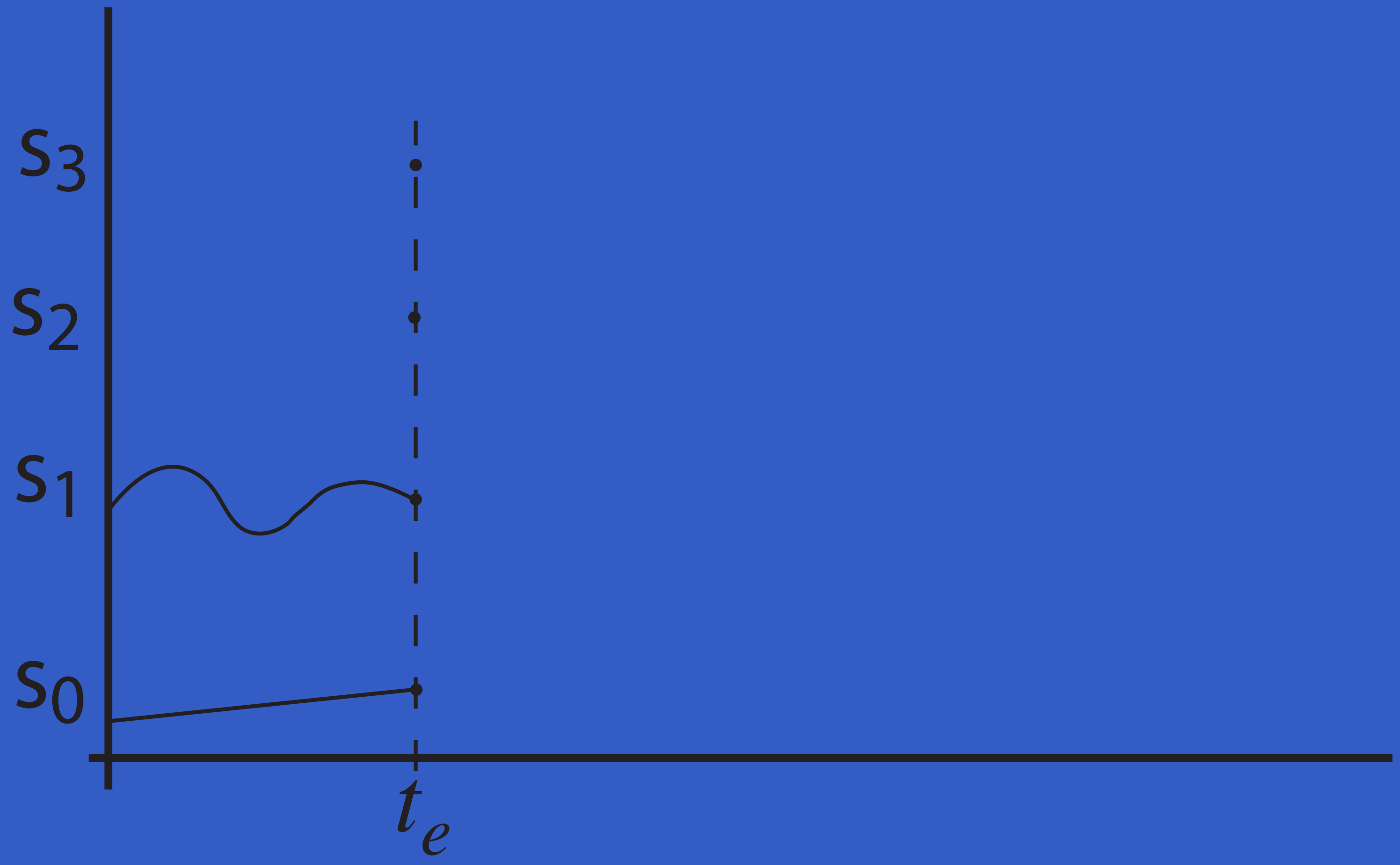
- Switch over ***collections*** of signal functions.
- On event, “freeze” running signal functions into collection of signal function ***continuations***, preserving encapsulated ***state***.
- Modify collection as needed and switch back in.

Dynamic signal function collections

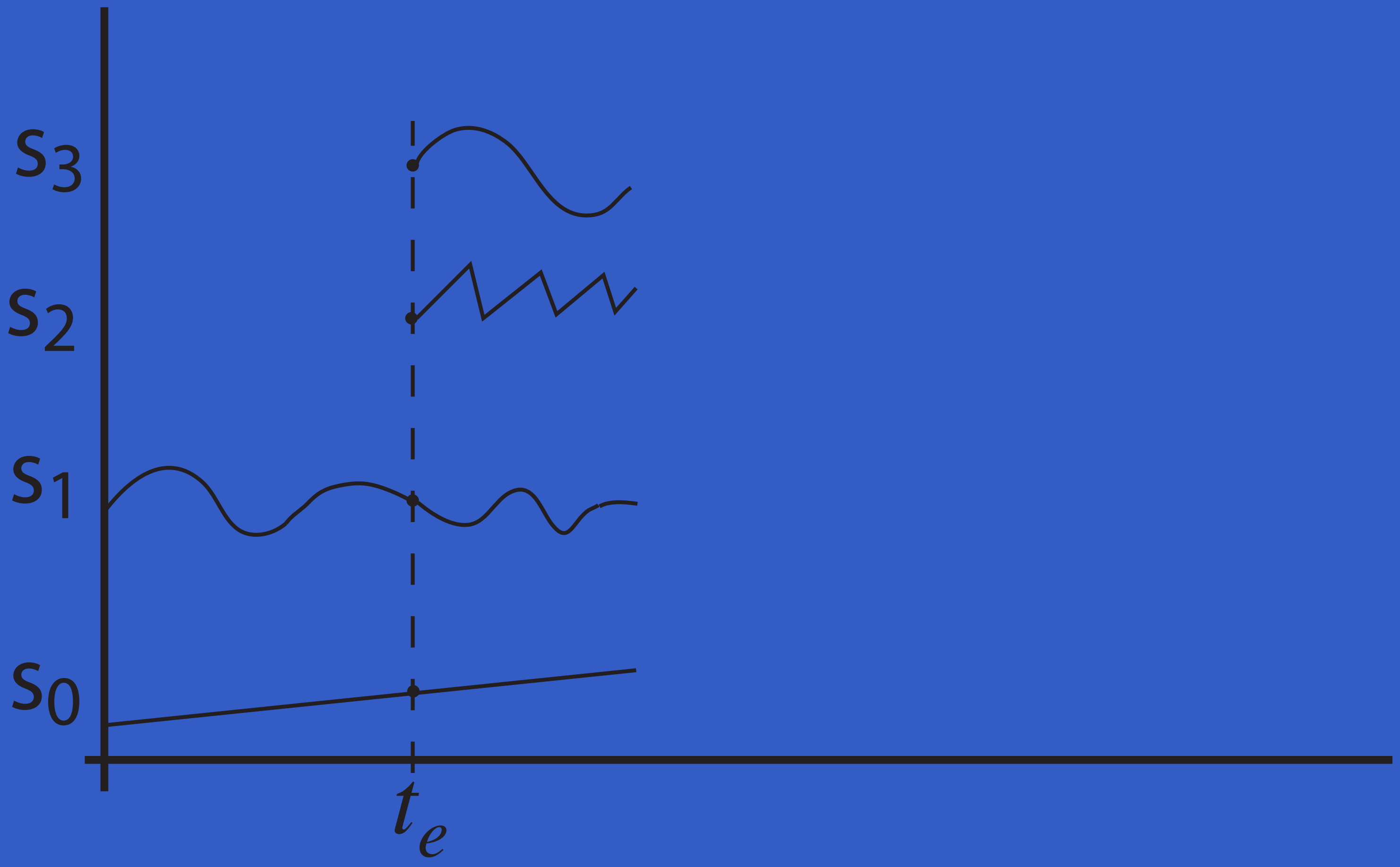
Dynamic signal function collections



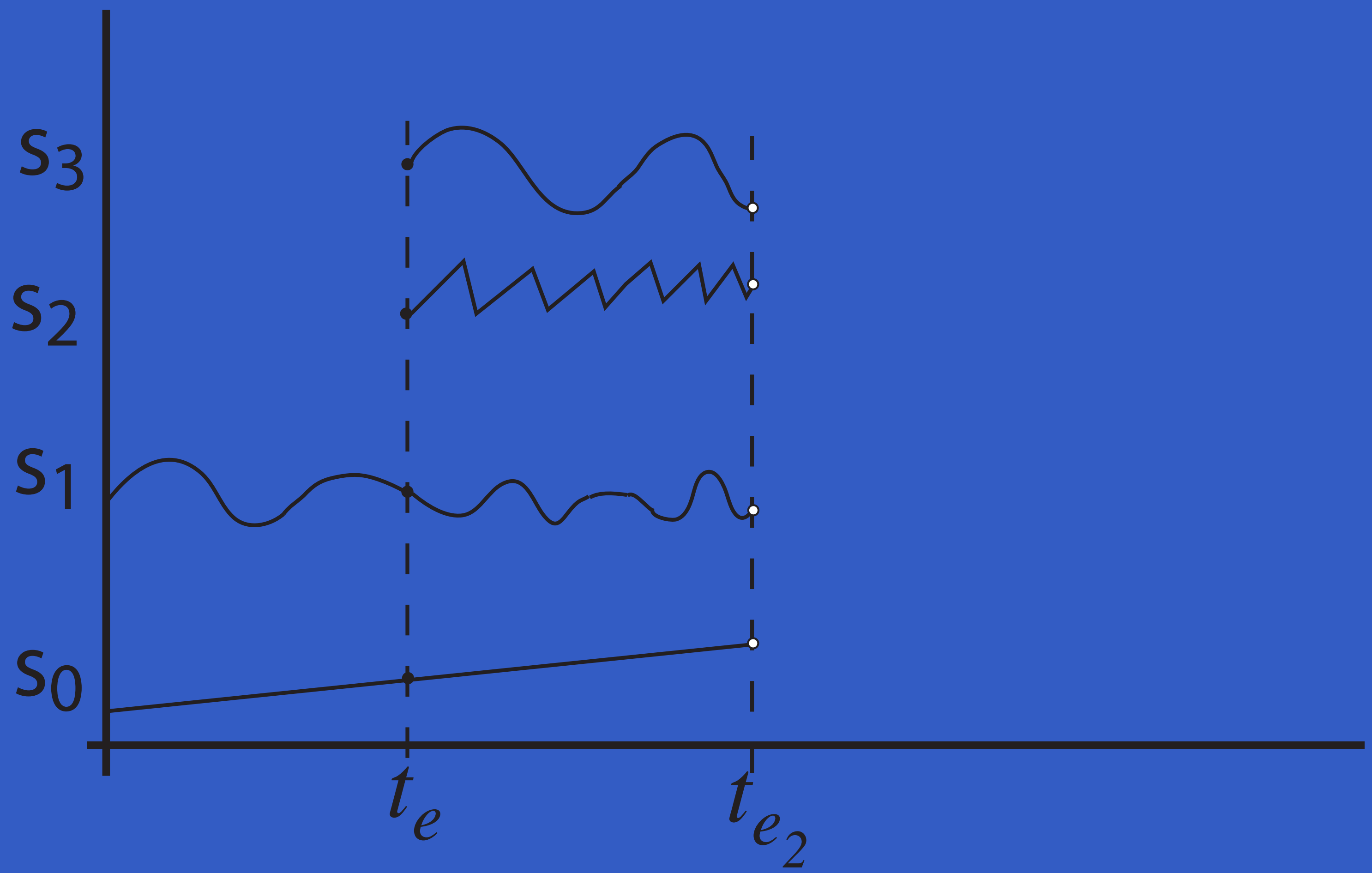
Dynamic signal function collections



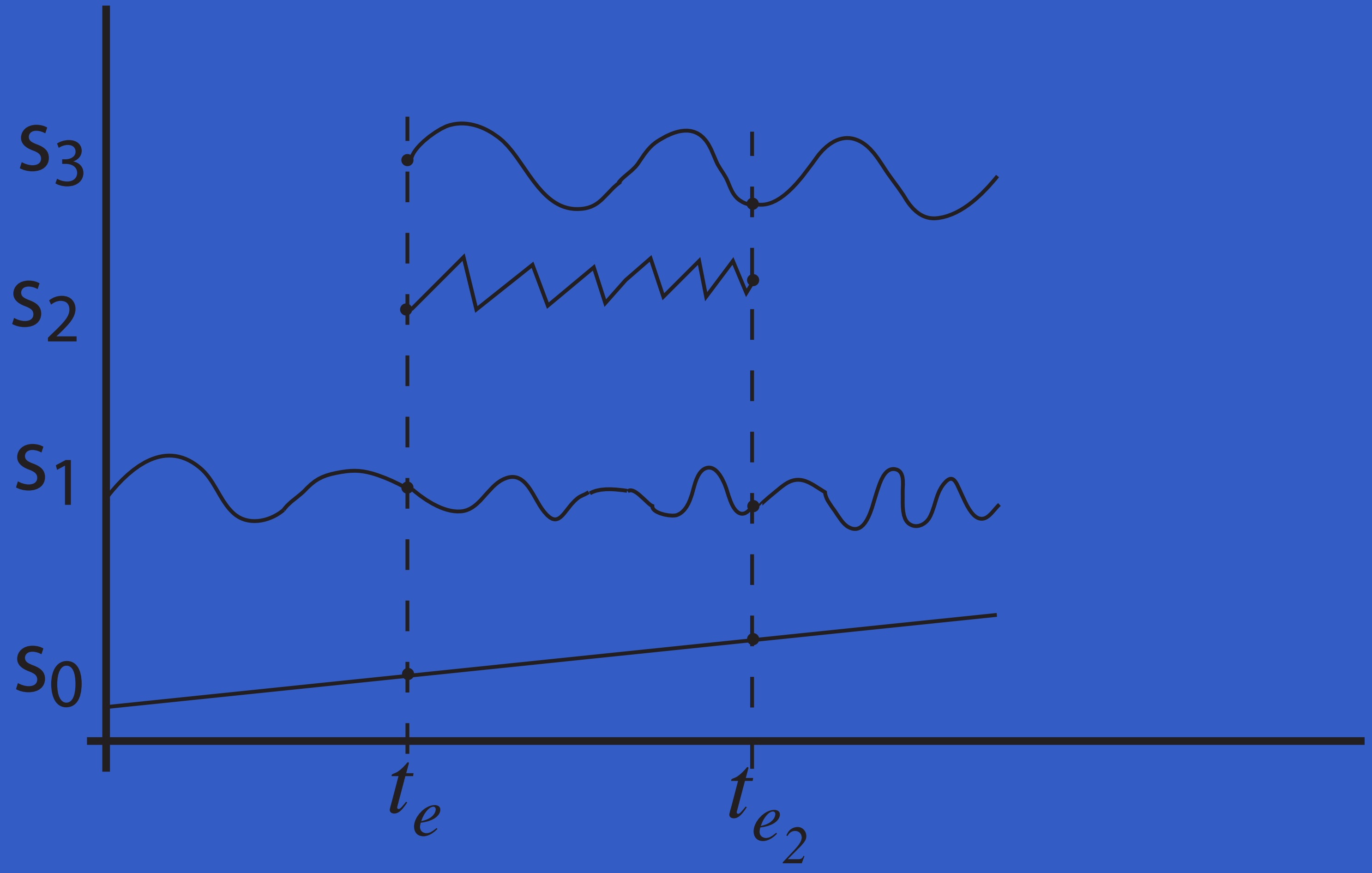
Dynamic signal function collections



Dynamic signal function collections



Dynamic signal function collections



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 function

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)
```

Initial collection

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)                        Event source
   -> 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)
```

Function yielding SF to switch into

Routing

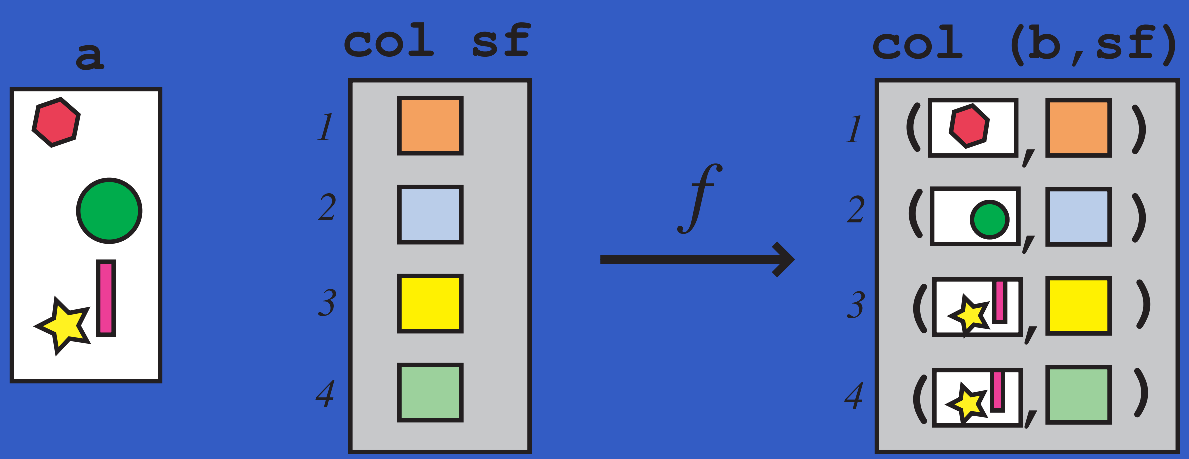
Idea:

- 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:

```
Functor col =>
  (forall sf . (a -> col sf -> col (b,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.

How many different kinds of switches?

There might seem to be quite a few different kinds of switches around:

```
switch, dSwitch, rSwitch,
drSwitch, pSwitch, dpSwitch,
...
```

In fact, they can all easily be defined in terms of, respectively, `switch` or `dSwitch`. But for the parallel (and other ***continuation-based***) switches, an additional notion is needed to provide the capability to freeze signal functions: ***ageing***.

Aging (1)

The primitive `age` continuously makes a frozen, aged, version of its argument signal function available:

```
age :: SF a b -> SF a (b, SF a b)
```

This is used to define the simple continuation-based switched, `kSwitch`:

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

Aging (2)

```
kSwitch :: SF a b -> SF (a,b) (Event c)
           -> (SF a b -> c -> SF a b) -> SF a b
kSwitch sf1 sfe k =
    switch sf (\(c, sf1') -> k sf1' c)
    where
        -- sf :: SF a (b, Event (c, SF a b))
        sf = (identity &&& age sf1)
             >>> arr (\(a, (b, sf1')) ->
                      ((a,b), (b, sf1')))
             >>> first sfe
             >>> arr (\(e, (b, sf1')) ->
                      (b, e `attach` sf1'))
```

Aging (3)

Aging of collections:

```
agePar :: Functor col =>
    (forall sf . (a -> col sf -> col (b, sf)))
    -> col (SF b c)
    -> SF a (col c, col (SF b c))
```

This can be used to define `pSwitch` and `dpSwitch` in terms of `switch` and `dSwitch`, respectively, in a similar way to `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 ***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

```
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

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)

```
...
-- 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)

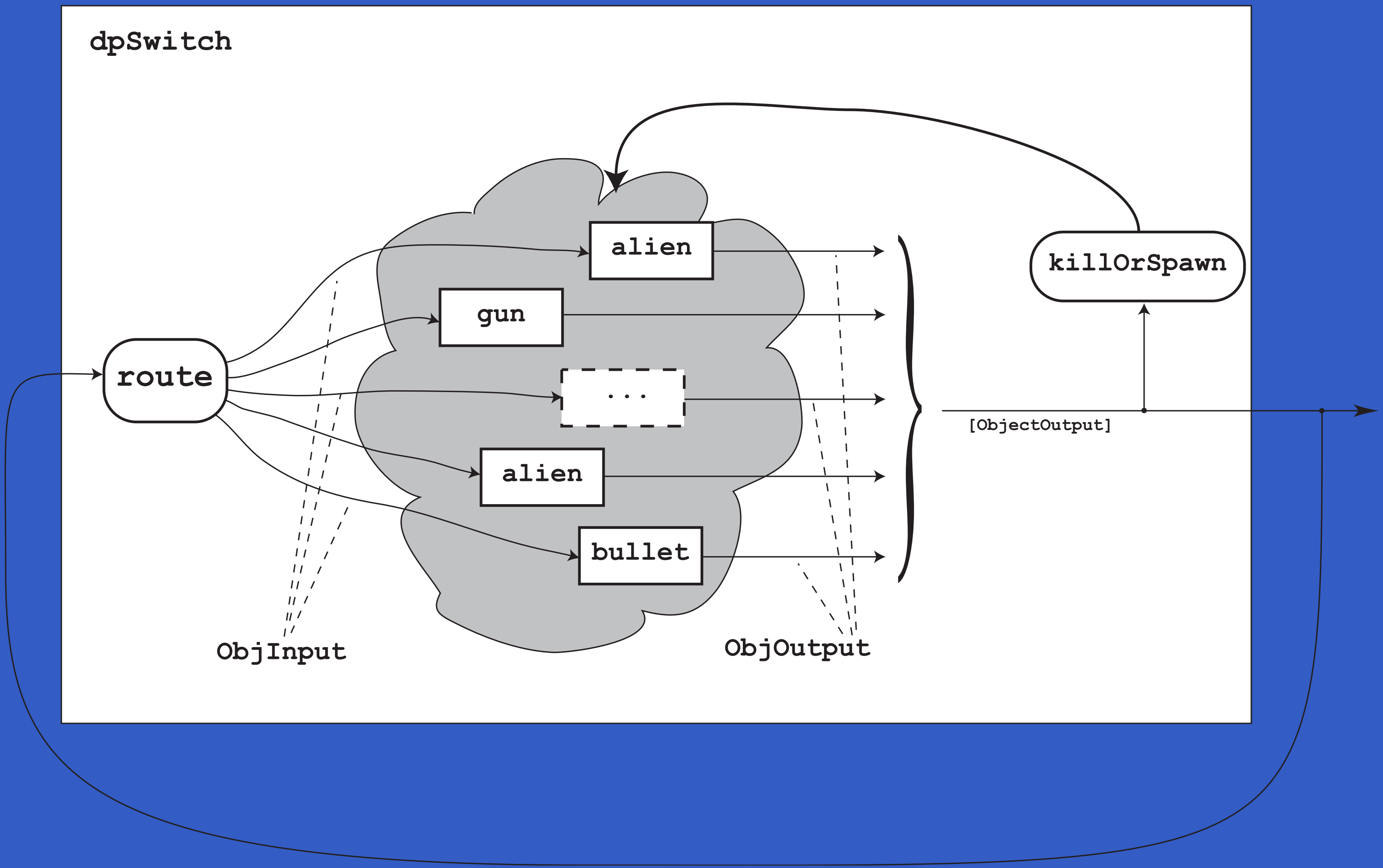
```
...
-- Physics
let a = vector2Polar
            (min alienAccMax
                 (vector2Rho ad))
            h
vp  <- iPre v0      -< v
ffi <- forceField -< (p, vp)
v   <- (v0 ^+^) ^<< impulseIntegral
          -< (gravity ^+^ a, ffi)
p   <- (p0 .+^) ^<< integral -< v
...
```

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
  ...
```

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)
```

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”?

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.

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:

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

Reading (1)

- Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 Haskell Workshop*, pp. 51–64, October 2002.
- Antony Courtney and Henrik Nilsson and John Peterson. The Yampa Arcade. In *Proceedings of the 2003 Haskell Workshop*, pp. 7–18, August 2003.

Reading (2)

- Kristopher J. Blom. *Dynamic Interactive Virtual Environments*. PhD Thesis, University of Hamburg, Department of Informatics, 2009
- Gergely Patai. Efficient and Compositional Higher-Order Streams. In *Proceedings of Functional and (Constraint) Logic Programming (WFLP) 2010*, Madrid, Spain, January 2010.