

Yampa: Functional Reactive Programming for Systems with Dynamic Structure

Henrik Nilsson

University of Nottingham, UK

Reactive programming

Reactive systems:

- input arrives incrementally while system is running
- output is generated in response to input in an interleaved fashion

(Contrast *transformational systems*.)

The notions of

- time
- time-varying entities, *signals*

are inherent.

Functional Reactive Programming

Functional Reactive Programming (FRP)

- Framework for reactive programming in a functional setting.
- Systems described by mapping *signals* to *signals*.
- Supports hybrid systems (continuous and discrete time).
- Supports systems with evolving structure.
- Originated from Functional Reactive Animation (Fran) (Elliott & Hudak).

Related languages

FRP related both to modeling and synchronous dataflow languages:

- Modeling Languages:
 - Simulink
 - Ptolemy II
 - Modelica
- Synchronous languages:
 - Esterel
 - Lustre
 - Lucid Synchrone

FRP applications

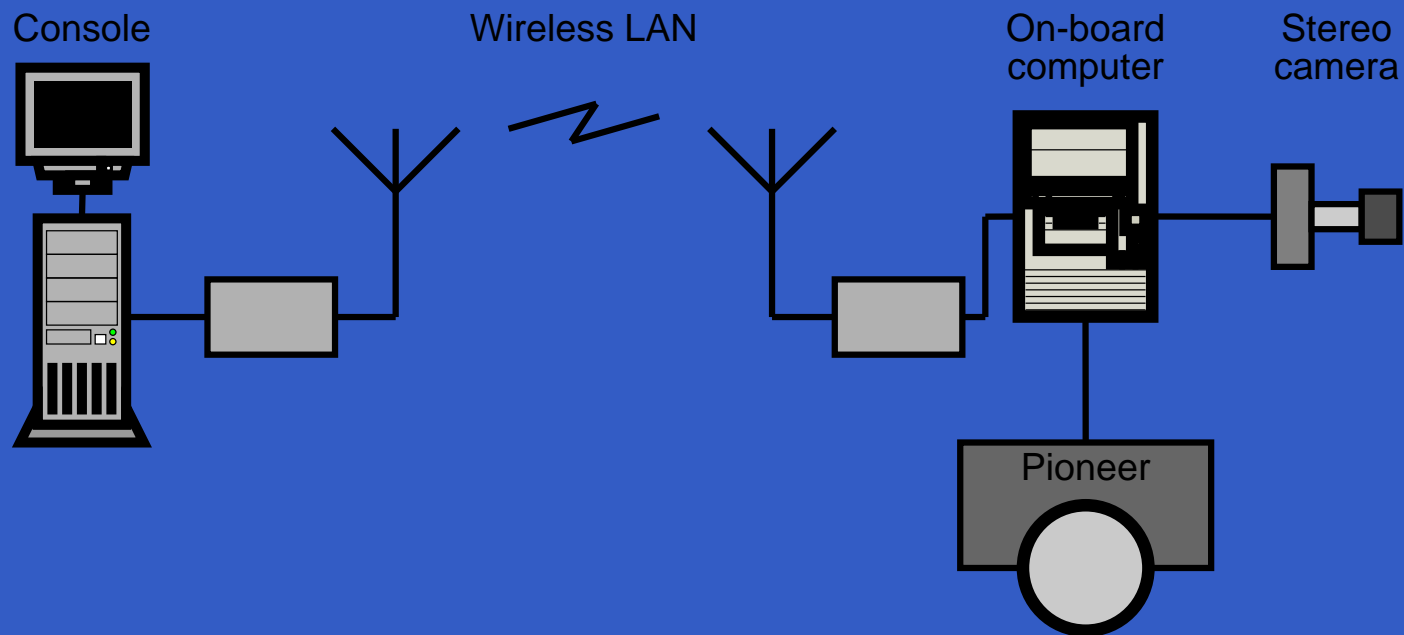
Some domains where FRP has been used:

- Graphical Animation (Fran: Elliott, Hudak)
- Robotics (Frob: Peterson, Hager, Hudak, Elliott, Pembeci, Nilsson)
- Vision (FVision: Peterson, Hudak, Reid, Hager)
- GUIs (Fruit: Courtney)
- Hybrid modeling (Nilsson, Hudak, Peterson)

Example: Robotics (1)

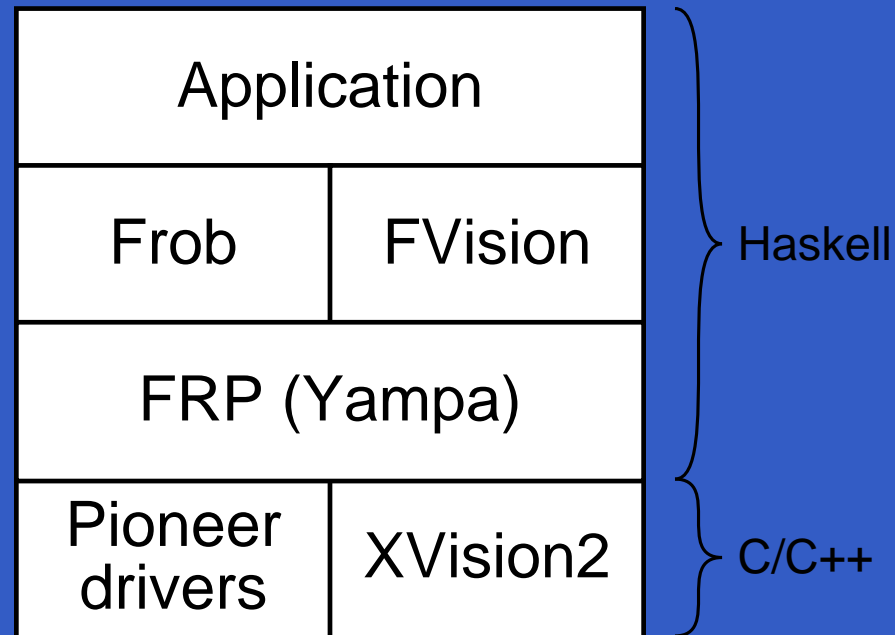
[PPDP'02, with Izzet Pembeci and Greg Hager,
Johns Hopkins University]

Hardware setup:

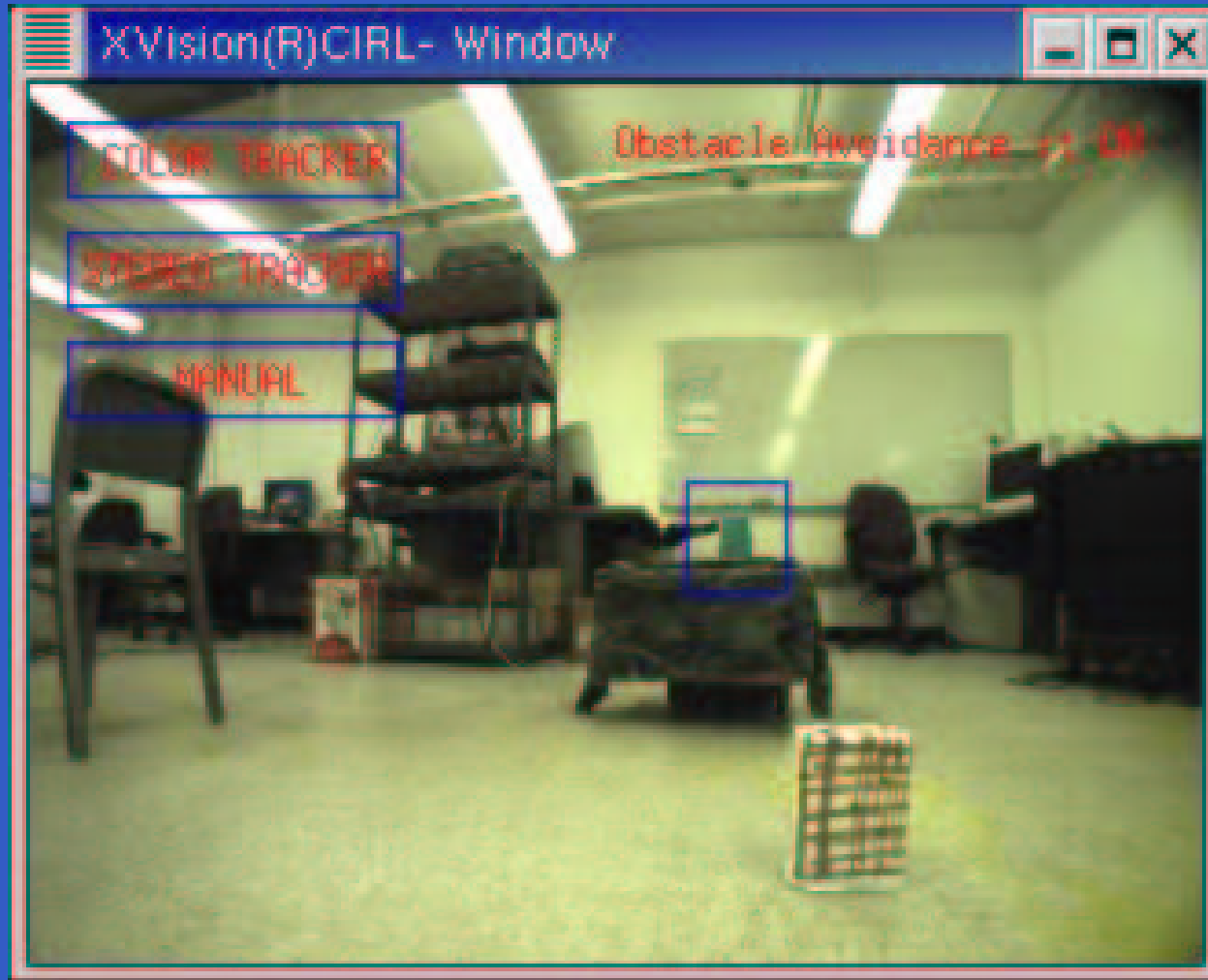


Example: Robotics (2)

Software architecture:



Example: Robotics (3)



Yampa

The most recent Yale FRP implementation is called *Yampa*:

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

-
-
-

Yampa?

Yampa?

Yet
Another
Mostly
Pointless
Acronym

Yampa?

Yet
Another
Mostly
Pointless
Acronym

???

Yampa?

Yet
Another
Mostly
Pointless
Acronym

???

No ...

-
-
-

Yampa?

Yampa is a river . . .



Yampa?

... with long calmly flowing sections ...



-
-
-

Yampa?

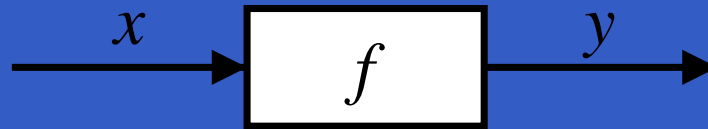
... and abrupt whitewater transitions in between.



A good metaphor for hybrid systems!

Signal functions

Key concept: functions on signals.



Intuition:

Signal $\alpha \approx \text{Time} \rightarrow \alpha$

$x :: \text{Signal } T1$

$y :: \text{Signal } T2$

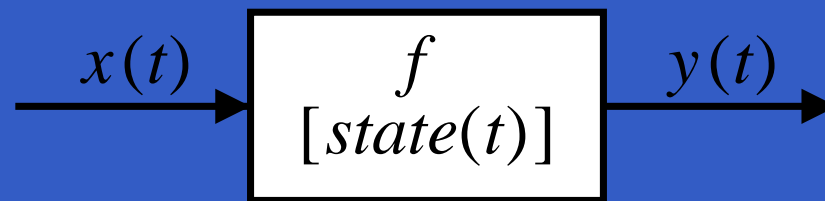
$f :: \text{Signal } T1 \rightarrow \text{Signal } T2$

Additionally: **causality** requirement.

Signal functions and state

Alternative view:

Functions on signals can encapsulate state.



$state(t)$ 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 $x(t)$

Signal functions in Yampa

- Signal functions are *first class entities*.

Intuition: $SF\ \alpha\ \beta \approx Signal\ \alpha \rightarrow Signal\ \beta$

Signal functions in Yampa

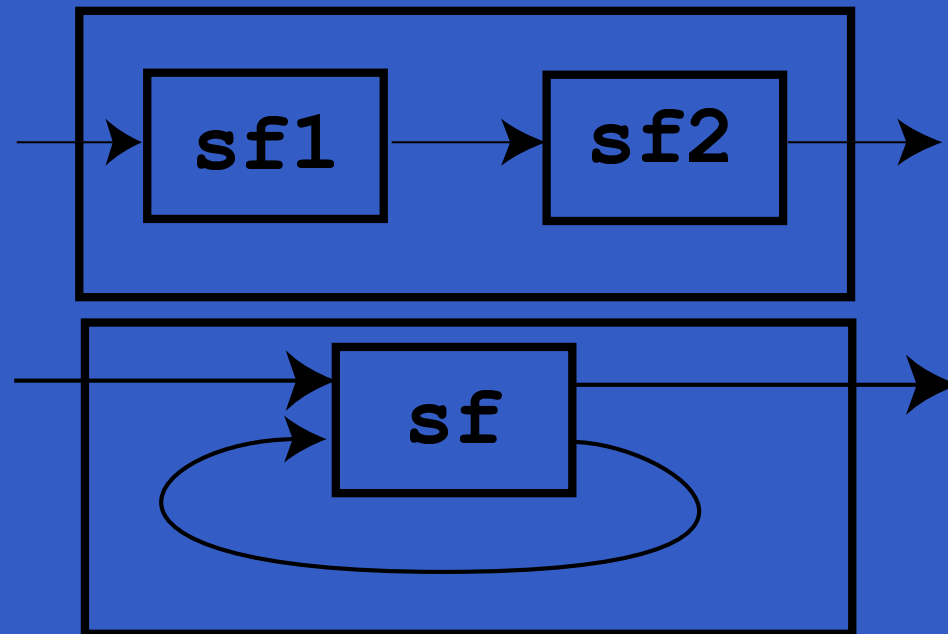
- Signal functions are *first class entities*.
Intuition: $SF\ \alpha\ \beta \approx Signal\ \alpha \rightarrow Signal\ \beta$
- Signals are *not* first class entities: they only exist indirectly through signal functions.

Signal functions in Yampa

- Signal functions are *first class entities*.
Intuition: $SF\ \alpha\ \beta \approx Signal\ \alpha \rightarrow Signal\ \beta$
- 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.

Describing systems

Systems are described by combining signal functions into larger signal functions:



Yampa and arrows

Yampa uses John Hughes' *arrow* framework:
Signal functions are arrows.

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:

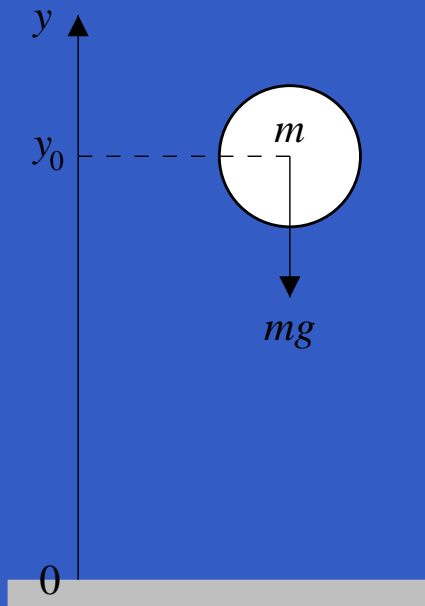
```
proc pat -> do [ rec ]  
    pat1 <- sfexp1 -< exp1  
    pat2 <- sfexp2 -< exp2  
    ...  
    patn <- sfexpn -< expn  
    returnA -< exp
```

Also: $\text{let } pat = exp \equiv pat -< \text{arr id} -< exp$

Some basic signal functions

- `identity :: SF a a`
`identity = arr id -- semantics`
- `constant :: b -> SF a b`
`constant b = arr (const b) -- semant`
- `integral :: VectorSpace a s -> SF a a`
- `time :: SF a Time`
`time = constant 1.0 >>> integral`
- `(^<<) :: (b -> c) -> SF a b -> SF a c`
`f (^<<) sf = sf >>> arr f`

A bouncing ball



$$y = y_0 + \int \dot{y} dt$$

$$\dot{y} = \int -9.81$$

On impact:

$$\dot{y} = -\dot{y}(t-)$$

(fully elastic collision)

Modelling the bouncing ball: part 1

Free-falling ball:

```
type Pos = Double
```

```
type Vel = Double
```

```
fallingBall ::
```

```
    Pos -> Vel -> SF () (Pos, Vel)
```

```
fallingBall p0 v0 = proc () -> do
```

```
    v <- (v0 +) ^<< integral -< -9.81
```

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

```
    returnA -< (p, v)
```

Events

Conceptually, *discrete-time* signals are only defined at discrete points in time, often associated with the occurrence of some *event*.

Yampa models discrete-time signals by lifting the *range* of continuous-time signals:

```
data Event a = NoEvent | Event a
```

Discrete-time signal = `Signal (Event a)`.

We often want to associate information with an event occurrence:

```
tag :: Event a -> b -> Event b
```

Some basic event sources

- `never :: SF a (Event b)`
- `now :: b -> SF a (Event b)`
- `after :: Time -> b -> SF a (Event b)`
- `repeatedly ::
Time -> b -> SF a (Event b)`
- `edge :: SF Bool (Event ())`

Stateful event suppression

- `notYet :: SF (Event a) (Event a)`
- `once :: SF (Event a) (Event a)`

Modelling the bouncing ball: part 2

Detecting when the ball goes through the floor:

```
fallingBall' ::  
  Pos -> Vel  
  -> SF () ((Pos, Vel), Event (Pos, Vel))  
fallingBall' p0 v0 = proc () -> do  
  pv@(p, _) <- fallingBall p0 v0 -< ()  
  hit      <- edge          -< p <= 0  
  returnA -< (pv, hit `tag` pv)
```

Switching

Q: How and when do signal functions “start”?

Switching

Q: How and when do signal functions “start”?

- A:
- **Switchers** “apply” a signal functions to its input signal at some point in time.
 - This creates a “running” signal function **instance**, which often replaces the previously running instance.

Switching

Q: How and when do signal functions “start”?

A:

- **Switchers** “apply” a signal functions to its input signal at some point in time.
- This creates a “running” signal function **instance**, which often replaces the previously running instance.

Switchers thus allow systems with **varying structure** to be described.

The basic switch

Idea:

- Allows one signal function to be replaced by another.
- Switching occurs on the first occurrence of the switching event source.

```
switch ::
```

```
  SF a (b, Event c)
```

```
-> (c -> SF a b)
```

```
-> SF a b
```

The basic switch

Idea:

- Allows one signal function to be replaced by another.
- Switching occurs on the first occurrence of the switching event source.

```
switch ::
```

```
SF a (b, Event c)
```

```
-> (c -> SF a b)
```

```
-> SF a b
```

Initial SF with event source



The basic switch

Idea:

- Allows one signal function to be replaced by another.
- Switching occurs on the first occurrence of the switching event source.

```
switch ::
```

```
  SF a (b, Event c)
```

```
-> (c -> SF a b)
```

```
-> SF a b
```

Function yielding SF to switch into



Modelling the bouncing ball: part 3

Making the ball bounce:

```
bouncingBall :: Pos -> SF () (Pos, Vel)
```

```
bouncingBall p0 = bbRec p0 0.0
```

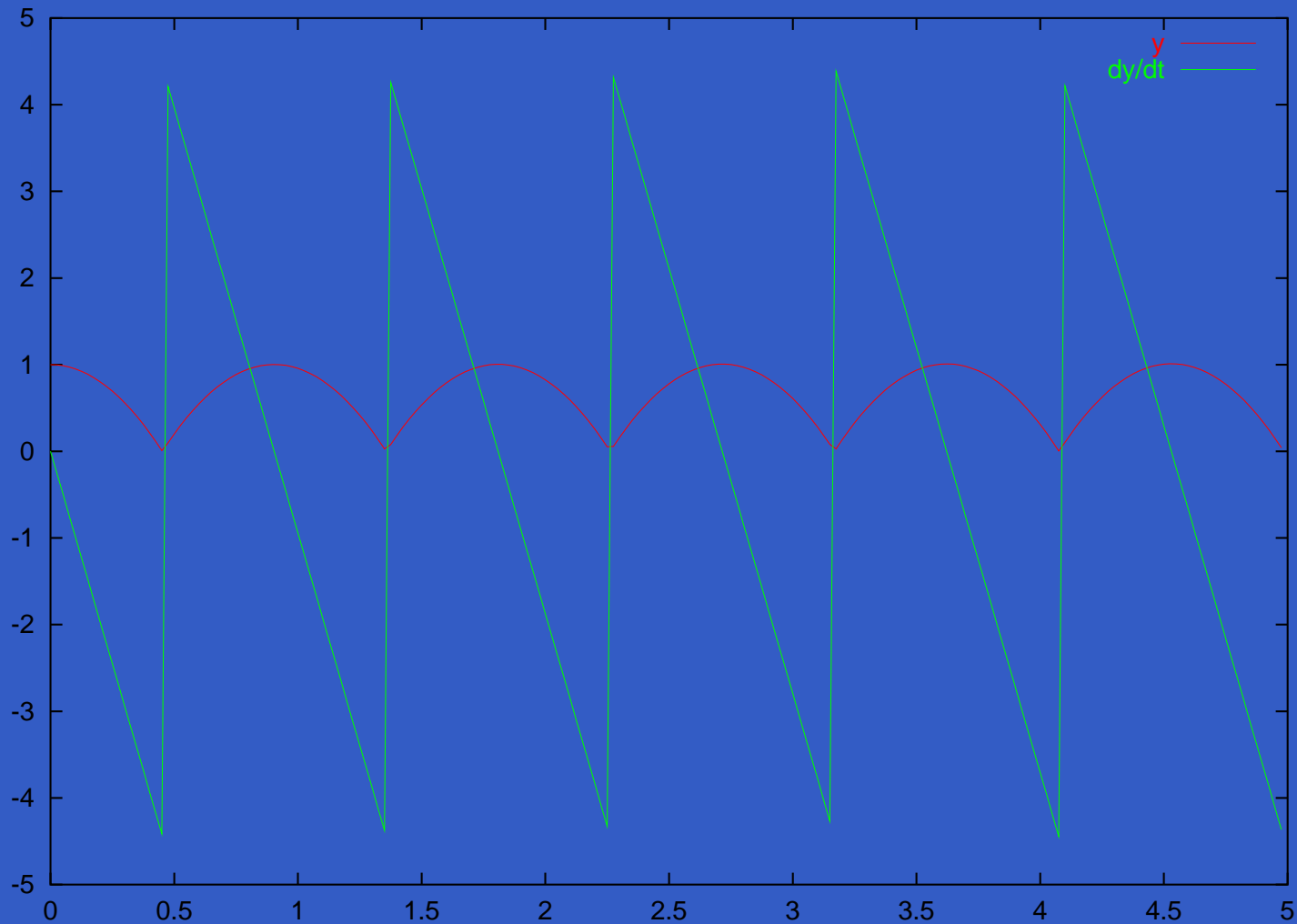
where

```
bbRec p0 v0 =
```

```
  switch (fallingBall' p0 v0) $ \(p,v) ->
```

```
  bbRec p (-v)
```

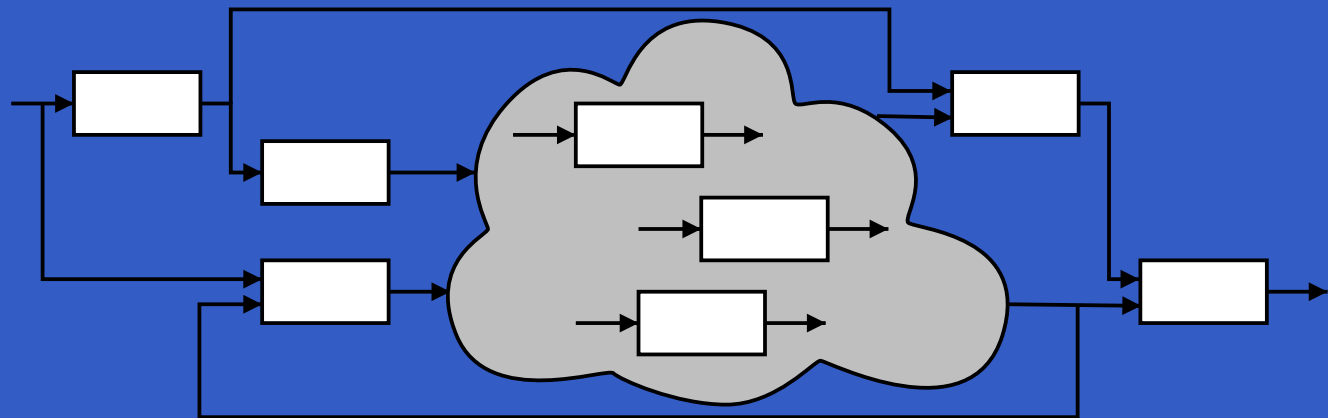
Simulation of bouncing ball



Highly dynamic system structure?

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:

... 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, like dynamic collections of signal functions.

Yampa attempts to address this [Haskell Workshop '02]

- Not many examples of good FRP code around.

[Haskell Workshop '03]

The game



Describing the alien behavior (1)

```
type Object = SF ObjInput ObjOutput
```

```
alien :: RandomGen g =>
```

```
  g -> Position2 -> Velocity -> Object
```

```
alien g p0 vvd = proc oi -> do
```

```
  rec
```

```
    -- Pick a desired horizontal position
```

```
    rx    <- noiseR (xMin, xMax) g -< ()
```

```
    smp1  <- occasionally g 5 ()    -< ()
```

```
    xd    <- hold (point2X p0) -< smp1 `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,
```

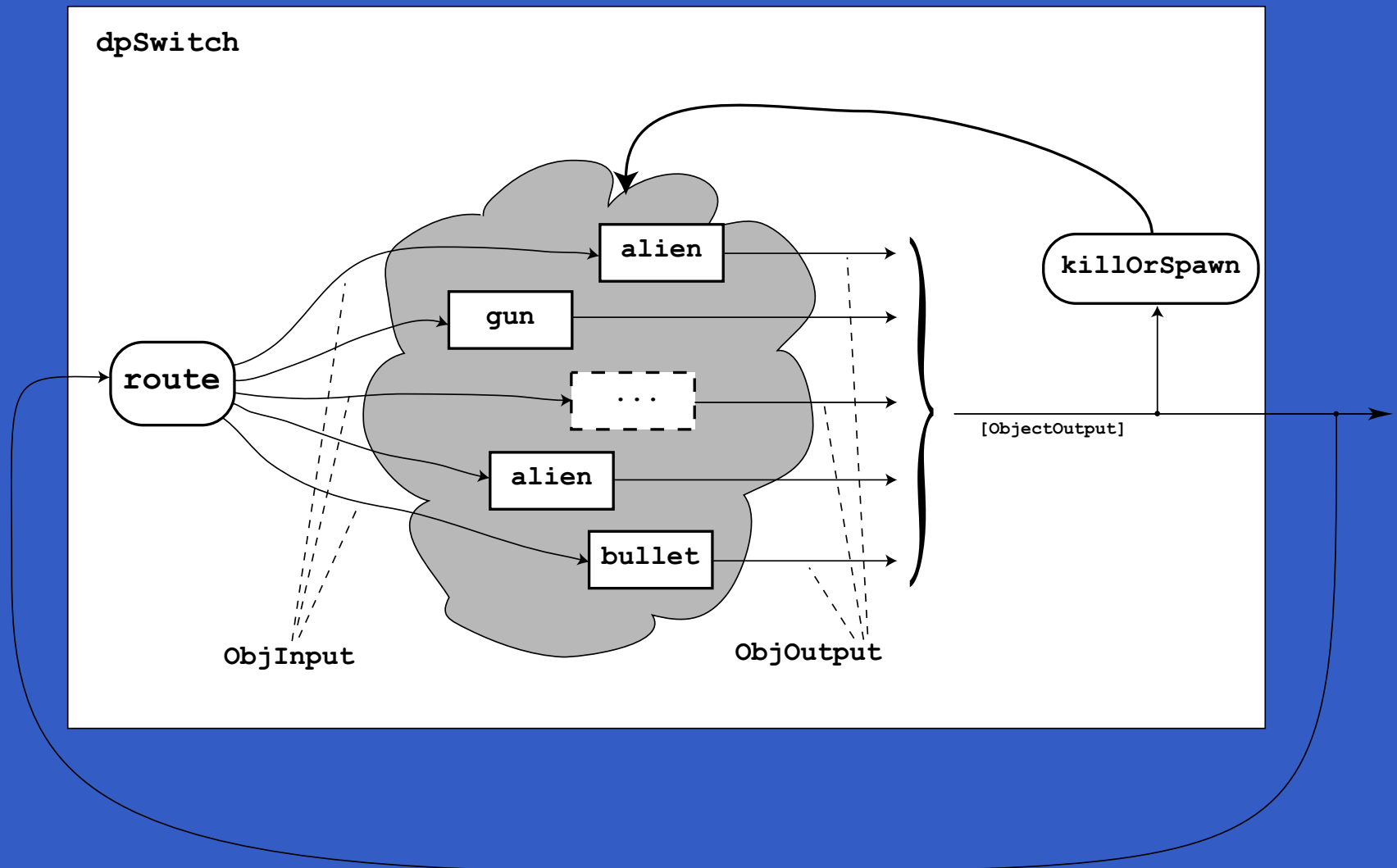
```
    ooKillReq     = die,
```

```
    ooSpawnReq    = noEvent
```

```
where
```

```
    v0 = zeroVector
```

Overall game structure



Dynamic signal function collections

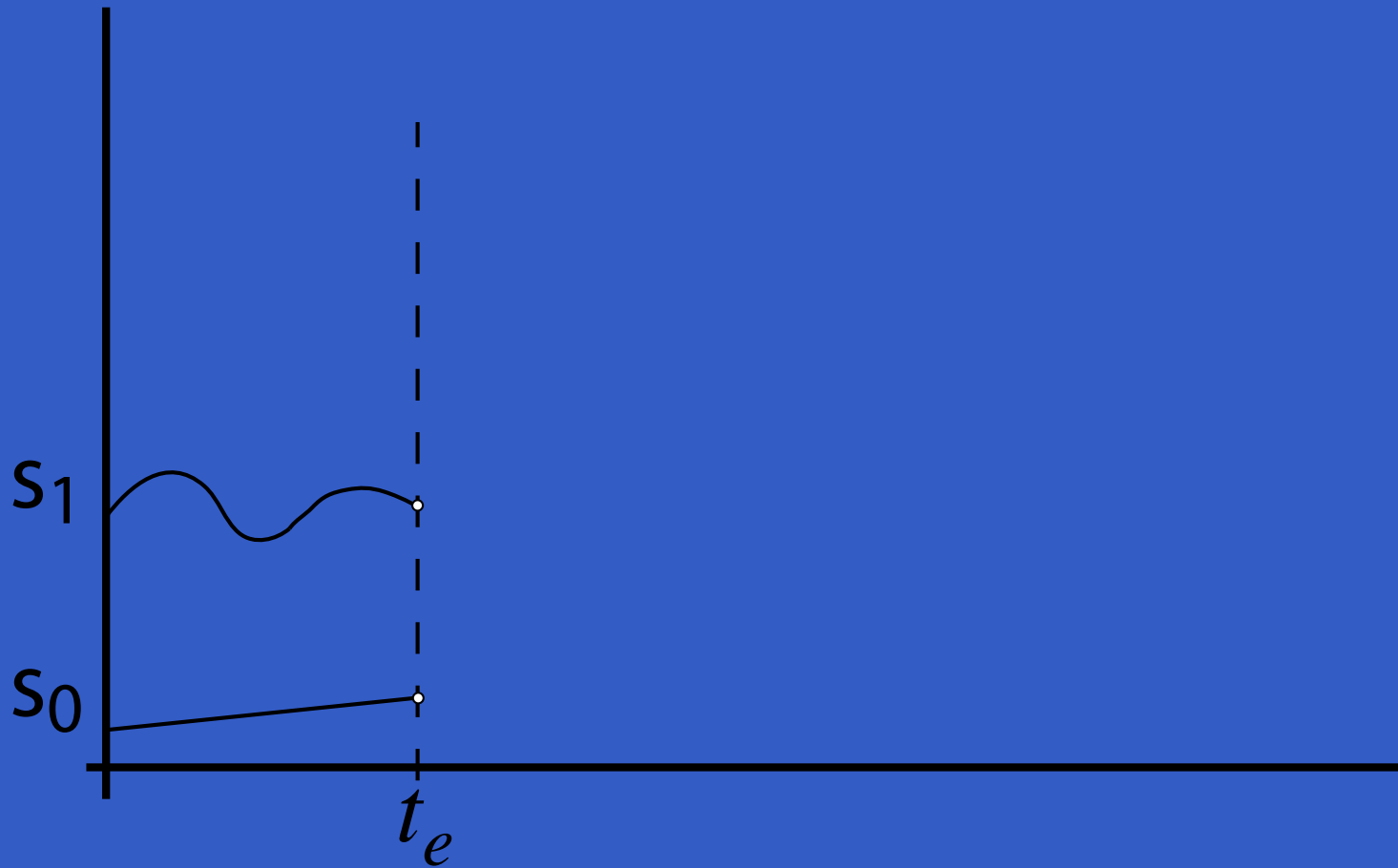
Idea:

- 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.

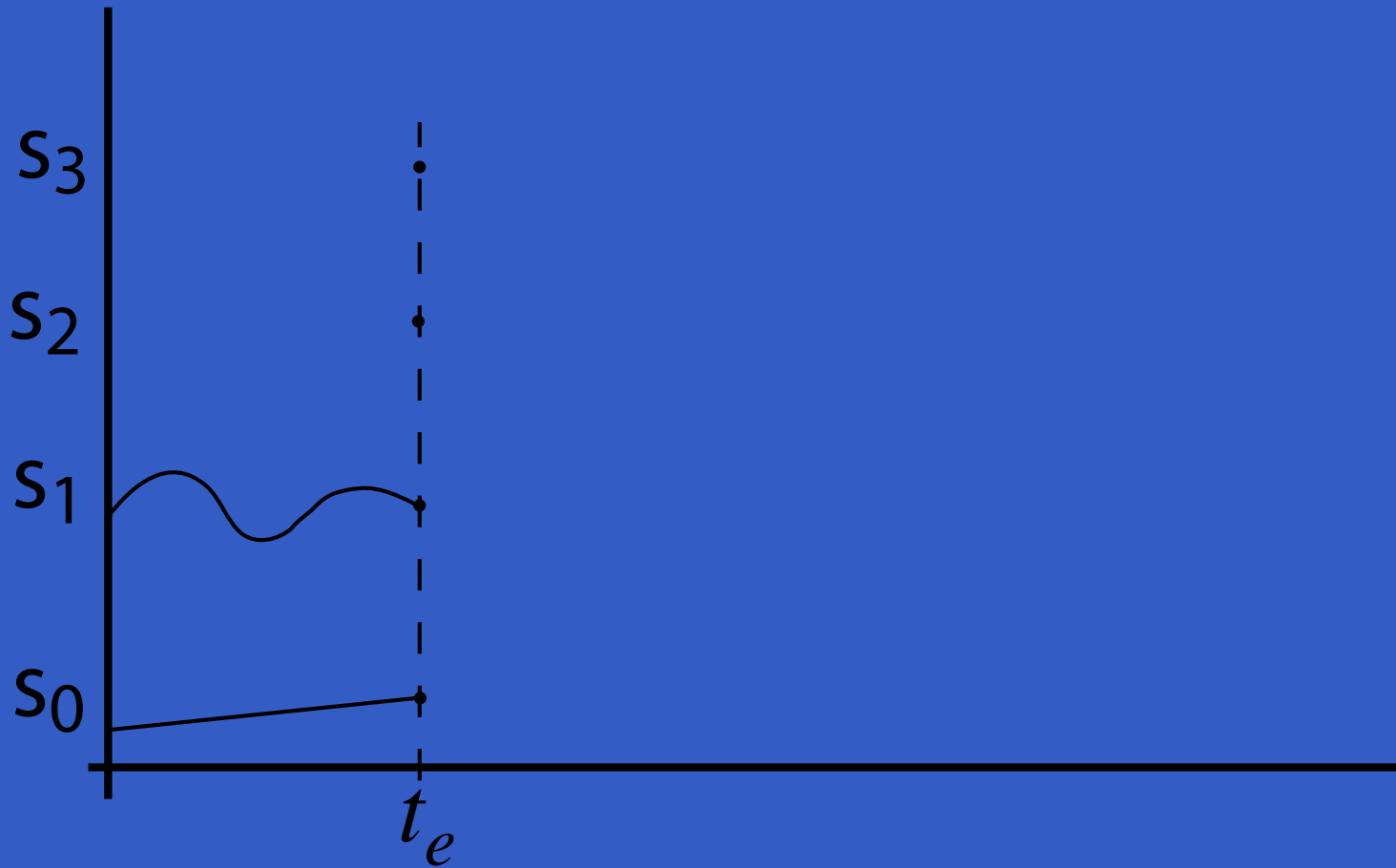
Dynamic signal function collections



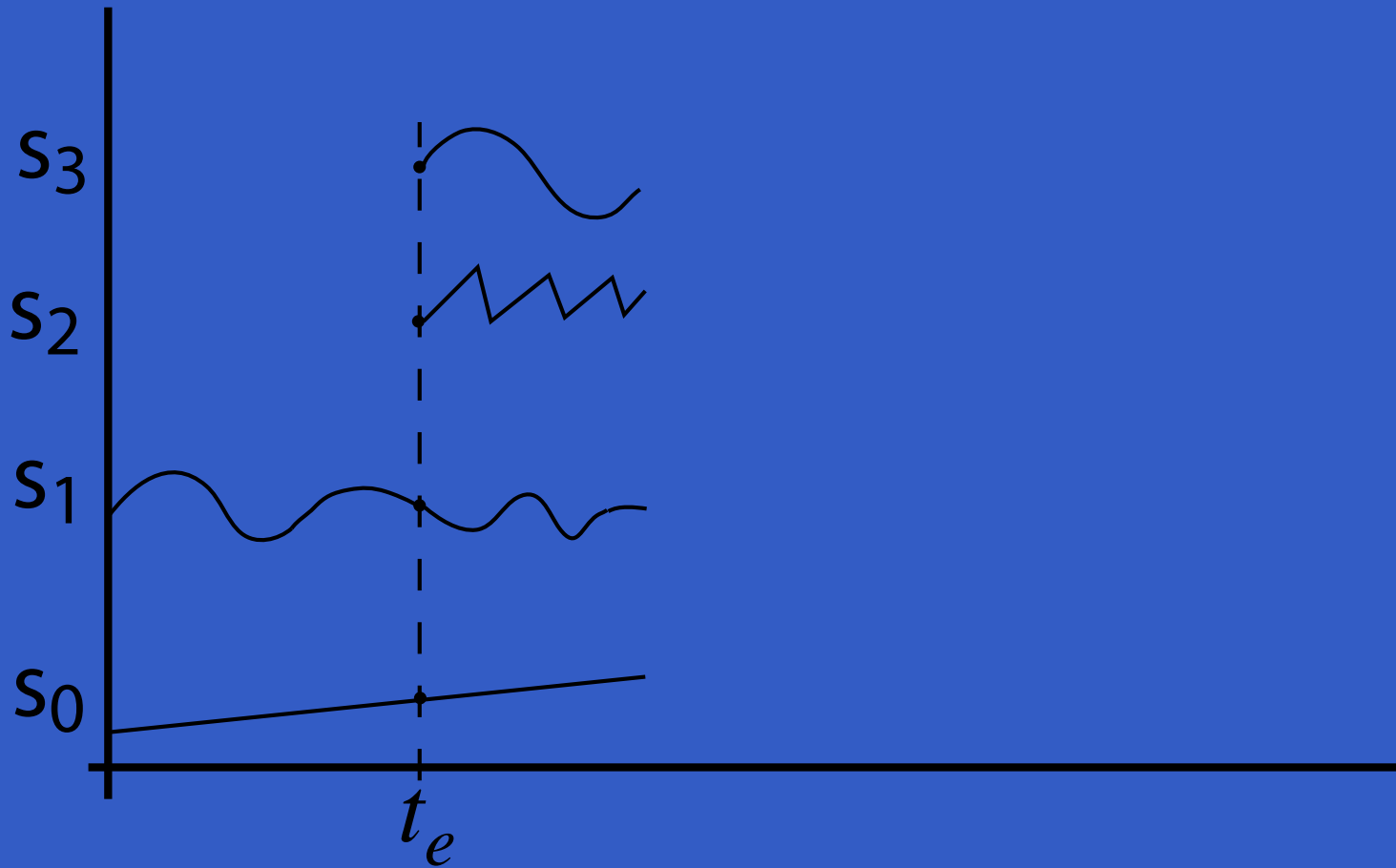
Dynamic signal function collections



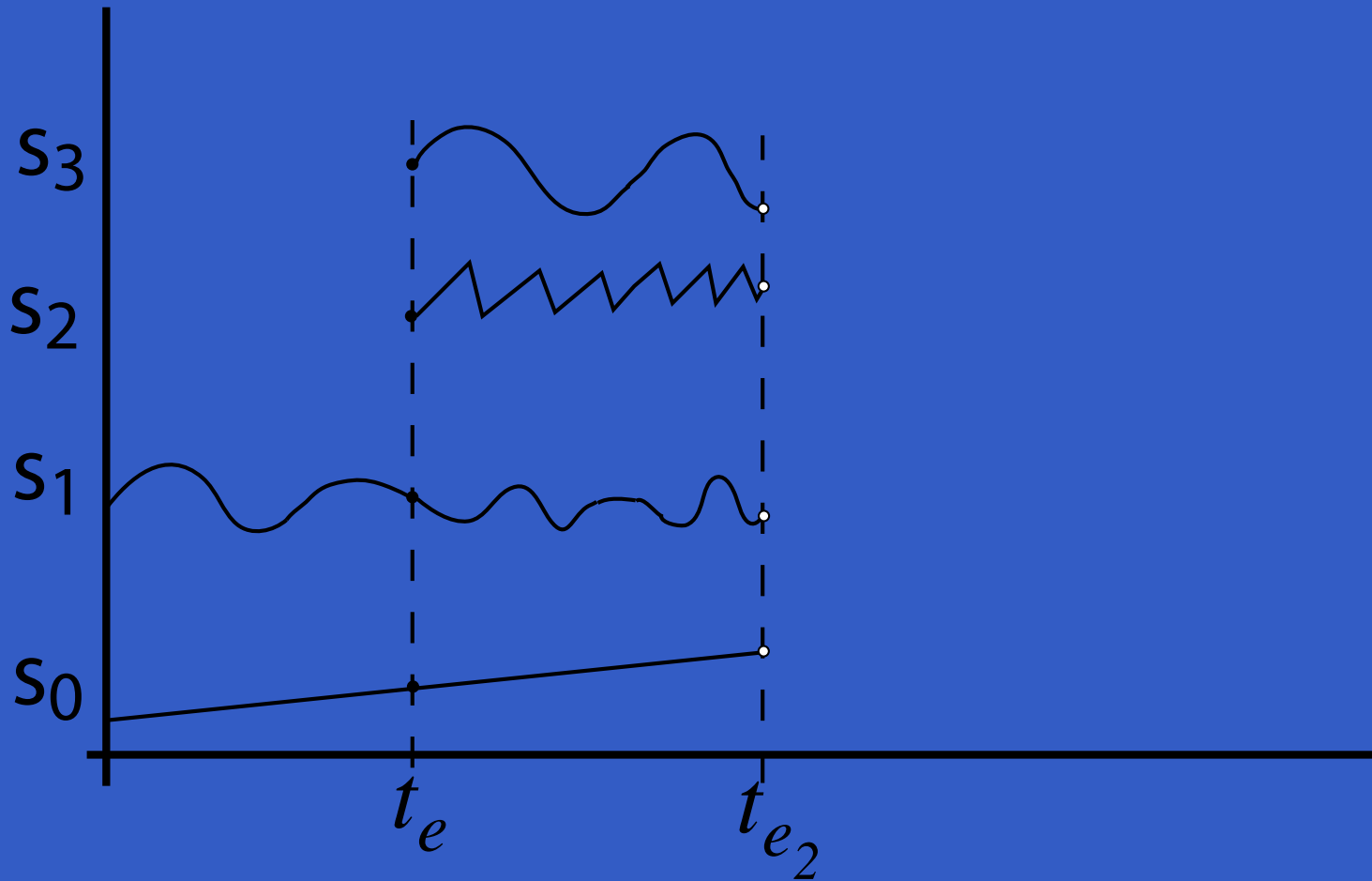
Dynamic signal function collections



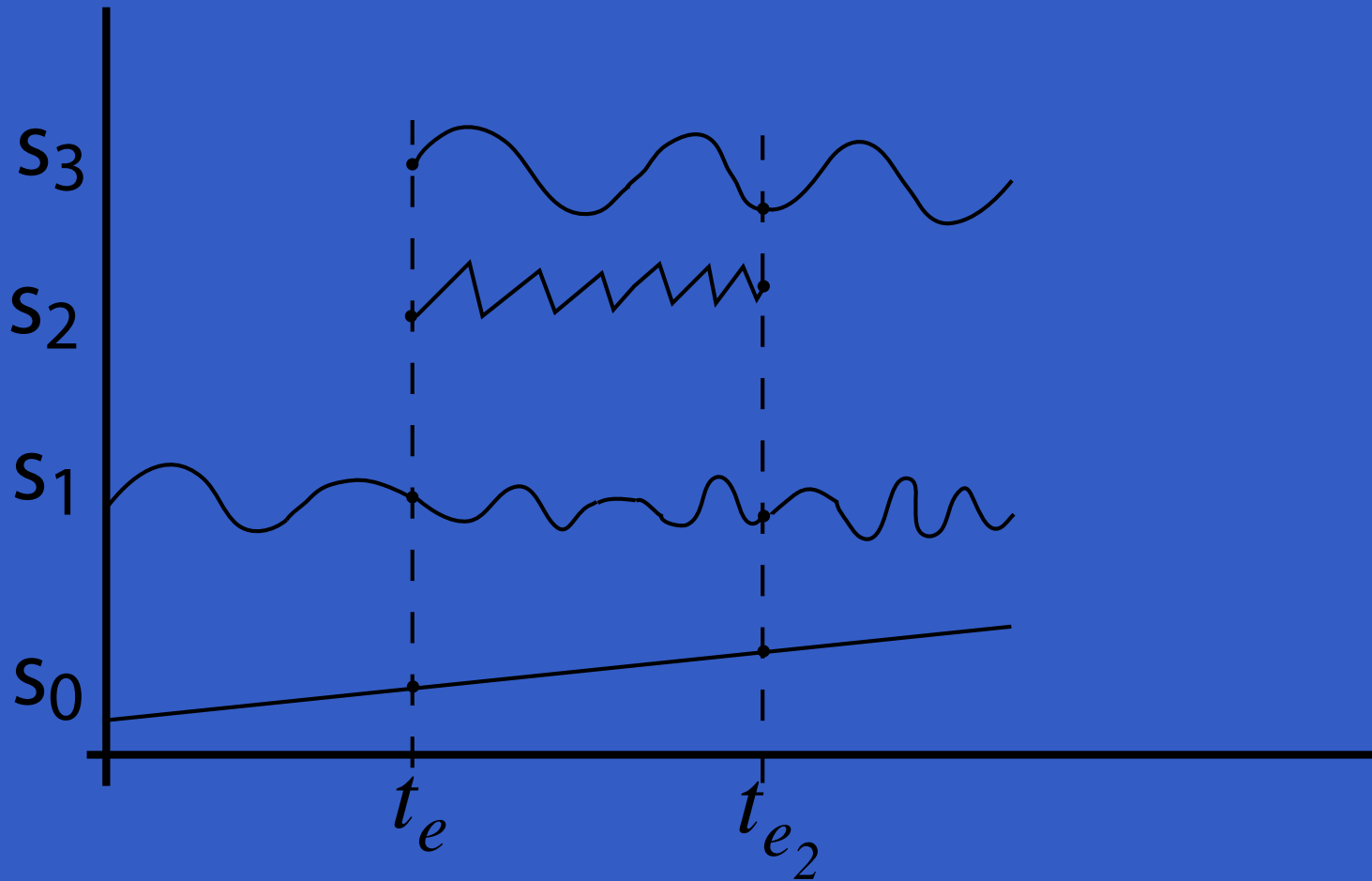
Dynamic signal function collections



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.

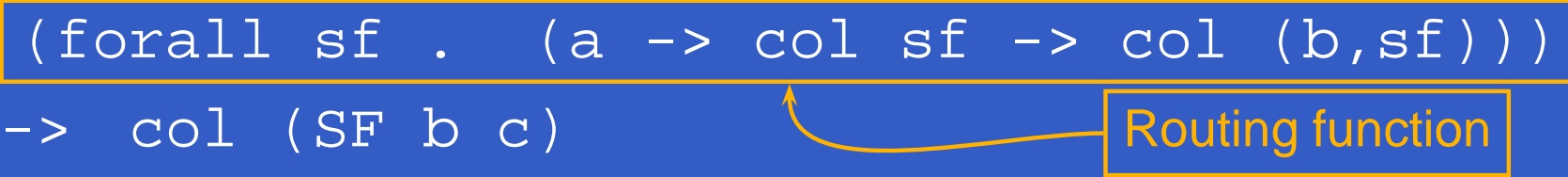
```
dpSwitch :: Functor col =>
  (forall sf . (a -> col sf -> col (b,sf)))
-> col (SF b c)
-> SF (a, col c) (Event d)
-> (col (SF b c) -> d -> SF a (col c))
-> SF a (col c)
```

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)
```



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) ← Initial collection
-> 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)
```



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) Function yielding SF to switch into
-> SF (a, col c) (Event d)
-> (col (SF b c) -> d -> SF a (col c))
-> SF a (col c)
```

Routing

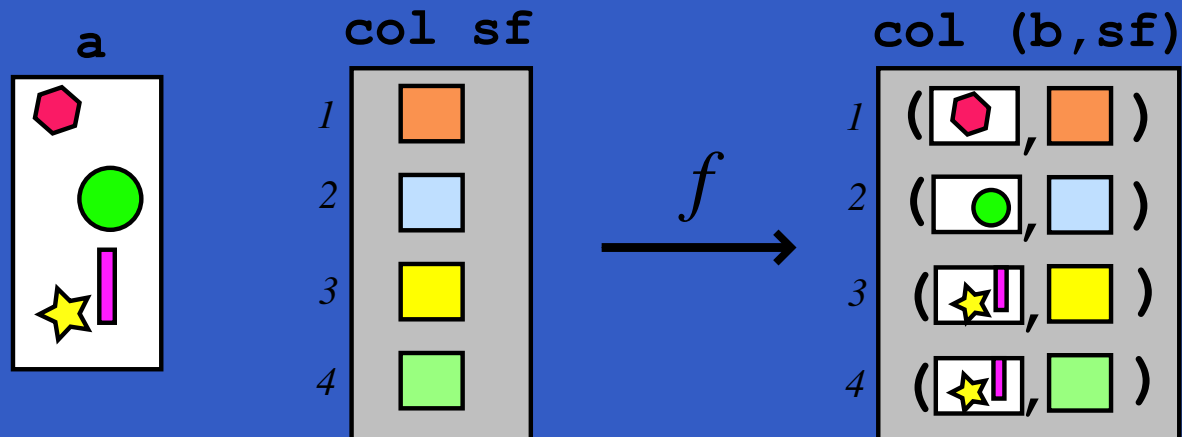
Idea:

- 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:

```
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.

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'))
```

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 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).

Related work (1)

- First-Order Systems: no dynamic collections
 - Esterel [Berry 92], Lustre [Caspi 87], Lucid Sychrone [Caspi 00], SimuLink, RT-FRP [Wan, Taha, Hudak 01]
- Fudgets [Carlsson and Hallgren 93, 98]
 - Continuation capture with `extractSP`
 - Dynamic Collections with `dynListF`
 - No synchronous bulk update

Related work (2)

- Fran [Elliott and Hudak 97, Elliott 99]
 - First class *signals*.
 - But dynamic collections?
- FranTk [Sage 99]
 - Dynamic collections, but only via IO monad.

Obtaining Yampa

Yampa 0.9 is available from

<http://www.haskell.org/yampa>