Functional Reactivity: Eschewing the Imperative

An Overview of Functional Reactive Programming in the Context of Yampa

Henrik Nilsson

University of Nottingham, UK

Reactive programming (1)

Reactive systems:

Reactive programming (1)

Reactive systems:

- Input arrives ***incrementally*** while system is running.

Reactive programming (1)

Reactive systems:

- Input arrives ***incrementally*** while system is running.
- Output is generated in response to input in an interleaved and ***timely*** fashion.

Reactive programming (1)

Reactive systems:

- Input arrives ***incrementally*** while system is running.
- Output is generated in response to input in an interleaved and ***timely*** fashion.

Contrast ***transformational systems***.

Reactive programming (1)

Reactive systems:

- Input arrives ***incrementally*** while system is running.
- Output is generated in response to input in an interleaved and ***timely*** fashion.

Contrast ***transformational systems***.

The notions of

- time
- time-varying values, or ***signals***

are inherent and central to reactive systems.

Reactive Programming (2)

Reactive systems are

- generally concurrent
- often parallel
- often distributed

Reactive Programming (2)

Reactive systems are

- generally concurrent
- often parallel
- often distributed

Thus, besides timeliness, difficulties related to development of concurrent, parallel, and distributed programming are also inherent.

The Synchronous Approach (1)

The “synchronous realisation” (France, 1980s):

> If we heed the observation that time-varying values are central to reactive programming and

The Synchronous Approach (1)

The “synchronous realisation” (France, 1980s):

> If we heed the observation that time-varying values are central to reactive programming and
>
> - express systems directly as ***transformations*** of such entities

The Synchronous Approach (1)

The “synchronous realisation” (France, 1980s):

> If we heed the observation that time-varying values are central to reactive programming and
>
> - express systems directly as ***transformations*** of such entities
> - adopt system-wide ***logical*** time, abstracting away processing delays (hence ***synchronous***)

. . .

The Synchronous Approach (2)

. . . then:

The Synchronous Approach (2)

. . . then:

- systems can be described declaratively at a very high level of abstraction

The Synchronous Approach (2)

. . . then:

- systems can be described declaratively at a very high level of abstraction
- simple, deterministic semantics, facilitates reasoning

The Synchronous Approach (2)

. . . then:

- systems can be described declaratively at a very high level of abstraction
- simple, deterministic semantics, facilitates reasoning
- many problems related to imperative idioms for concurrency and synchronisation simply vanishes.

The Synchronous Approach (2)

. . . then:

- systems can be described declaratively at a very high level of abstraction
- simple, deterministic semantics, facilitates reasoning
- many problems related to imperative idioms for concurrency and synchronisation simply vanishes.

Contrast programming with values at isolated points in time in a fundamentally temporally agnostic setting.

The Synchronous Approach (3)

The synchronous languages were invented in France in the 1980s. The first ones were:

- Esterel
- Lustre
- Signal

Have been very successful; e.g. lots of industrial applications.

Many new languages and variations since then.

Functional Reactive Programming

Functional Reactive Programming (FRP):

- Paradigm for reactive, concurrent programming in purely declarative (functional) setting.

Functional Reactive Programming

Functional Reactive Programming (FRP):

- Paradigm for reactive, concurrent programming in purely declarative (functional) setting.
- Originated from Functional Reactive Animation (Fran) (Elliott & Hudak).

Functional Reactive Programming

Functional Reactive Programming (FRP):

- Paradigm for reactive, concurrent programming in purely declarative (functional) setting.
- Originated from Functional Reactive Animation (Fran) (Elliott & Hudak).
- Has evolved in a number of directions and into different concrete implementations.

Functional Reactive Programming

Functional Reactive Programming (FRP):

- Paradigm for reactive, concurrent programming in purely declarative (functional) setting.
- Originated from Functional Reactive Animation (Fran) (Elliott & Hudak).
- Has evolved in a number of directions and into different concrete implementations.
- (Usually) continuous notion of time and additional support for discrete events.

FRP applications

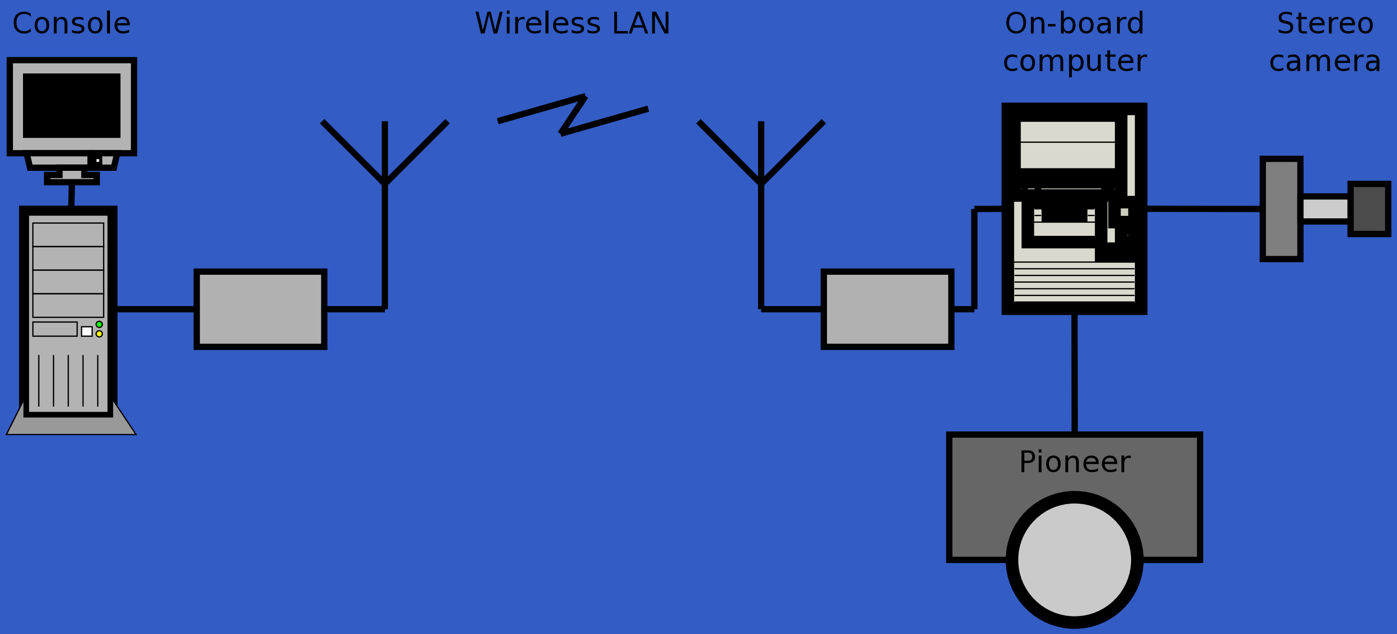
Some domains where FRP or FRP-like ideas have been used:

- Graphical Animation
- Robotics
- Vision
- GUIs
- Hybrid modeling
- Video games
- Sensor networks
- Audio processing and generation
- Financial, event-based systems

Example: Robotics (1)

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

Hardware setup:



Example: Robotics (2)

Related approaches

FRP related to:

- Synchronous languages, like Esterel, Lucid Synchrone.
- Modeling languages, like Simulink, Modelica.

Related approaches

FRP related to:

- Synchronous languages, like Esterel, Lucid Synchrone.
- Modeling languages, like Simulink, Modelica.

Distinguishing features of FRP:

- First class reactive components.
- Allows highly dynamic system structure.
- Supports hybrid (mixed continuous and discrete) systems.

Yampa

- An FRP system originating at Yale

Yampa

- An FRP system originating at Yale
- ***Embedding*** in Haskell (a Haskell library).

Yampa

- An FRP system originating at Yale
- ***Embedding*** in Haskell (a Haskell library).
- ***Arrows*** used as the basic structuring framework.

Yampa

- An FRP system originating at Yale
- ***Embedding*** in Haskell (a Haskell library).
- ***Arrows*** used as the basic structuring framework.
- Notionally ***continuous time***.

Yampa

- An FRP system originating at Yale
- ***Embedding*** in Haskell (a Haskell library).
- ***Arrows*** used as the basic structuring framework.
- Notionally ***continuous time***.
- Discrete-time signals modelled by continuous-time signals and an option type, allowing for ***hybrid*** systems.

Yampa

- An FRP system originating at Yale
- ***Embedding*** in Haskell (a Haskell library).
- ***Arrows*** used as the basic structuring framework.
- Notionally ***continuous time***.
- Discrete-time signals modelled by continuous-time signals and an option type, allowing for ***hybrid*** systems.
- Advanced ***switching constructs*** allows for highly dynamic system structure.

Yampa?

Yampa?

Y et

A nother

M ostly

P ointless

A cronym

Yampa?

Y et

A nother

M ostly

P ointless

A cronym

???

Yampa?

Y et

A nother

M ostly

P ointless

A cronym

???

No . . .

Yampa?

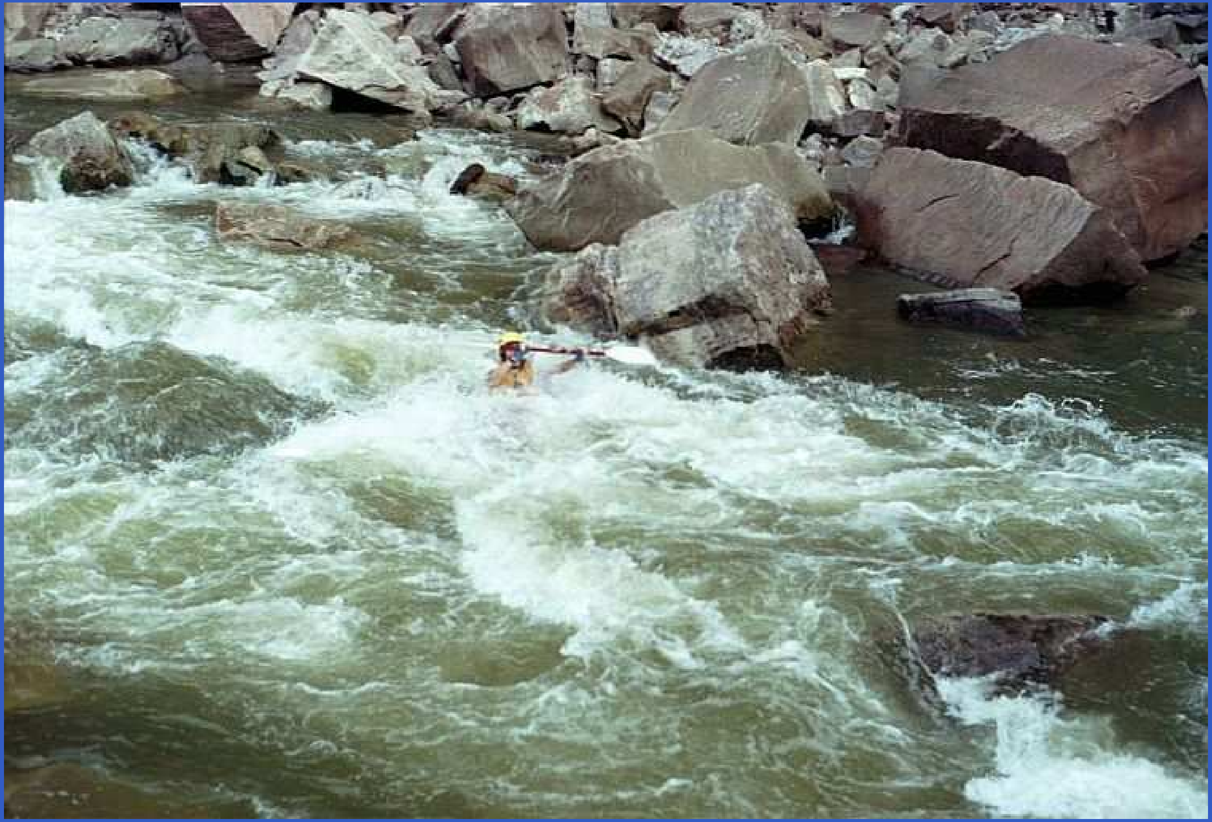
Yampa is a river . . .

Yampa?

. . . with long calmly flowing sections . . .

Yampa?

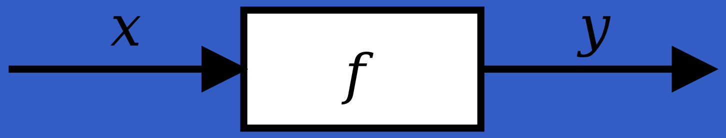
. . . and abrupt whitewater transitions in between.



A good metaphor for hybrid systems!

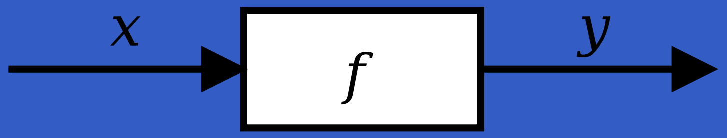
Signal functions

Key concept: ***functions on signals***.



Signal functions

Key concept: ***functions on signals***.

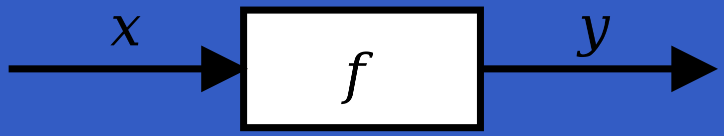


Intuition:

$$\texttt{Signal}\ \alpha \approx \texttt{Time} \rightarrow \alpha$$
$$x :: \texttt{Signal T1}$$
$$y :: \texttt{Signal T2}$$
$$f :: \texttt{Signal T1} \rightarrow \texttt{Signal T2}$$

Signal functions

Key concept: ***functions on signals***.



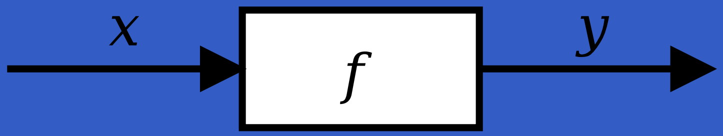
Intuition:

$$
\begin{array}{l}
\texttt{Signal}\ \alpha \approx \texttt{Time}\rightarrow\alpha \\
x \ \texttt{::} \ \ \texttt{Signal T1} \\
y \ \texttt{::} \ \ \texttt{Signal T2} \\
f \ \texttt{::} \ \ \texttt{Signal T1} \rightarrow \texttt{Signal T2}
\end{array}
$$

Additionally: ***causality*** requirement.

Signal functions

Key concept: ***functions on signals***.



Intuition:

$$\begin{array}{lll} \texttt{Signal}\ \alpha & \approx & \texttt{Time}\rightarrow\alpha \\ x & :: & \texttt{Signal T1} \\ y & :: & \texttt{Signal T2} \\ f & :: & \texttt{Signal T1}\ \rightarrow\texttt{Signal T2} \end{array}$$

Additionally: ***causality*** requirement.

Signal functions are ***first class entities*** in Yampa:

$\texttt{SF}\ \alpha\ \beta \approx \texttt{Signal}\ \alpha\ \rightarrow \texttt{Signal}\ \beta$

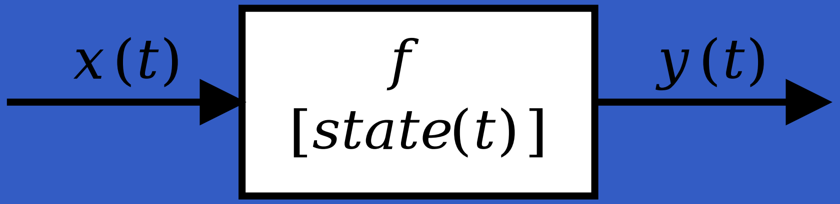
Signal functions and state

Alternative view:

Signal functions and state

Alternative view:

Signal functions can encapsulate ***state***.

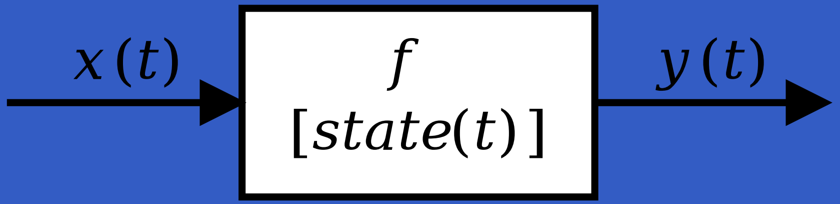


$state(t)$ summarizes input history $x(t')$, $t' \in [0, t]$.

Signal functions and state

Alternative view:

Signal functions can encapsulate ***state***.



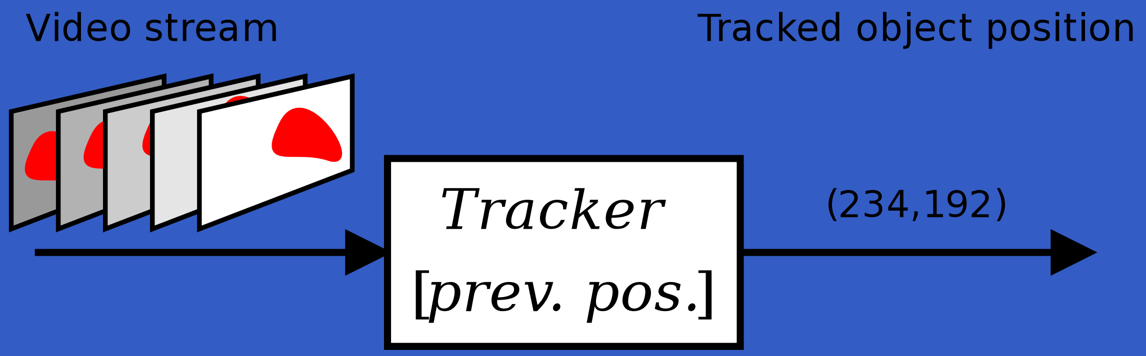
$state(t)$ summarizes input history $x(t')$, $t' \in [0, t]$.

Functions on signals are either:

- ***Stateful***: $y(t)$ depends on $x(t)$ and $state(t)$
- ***Stateless***: $y(t)$ depends only on $x(t)$

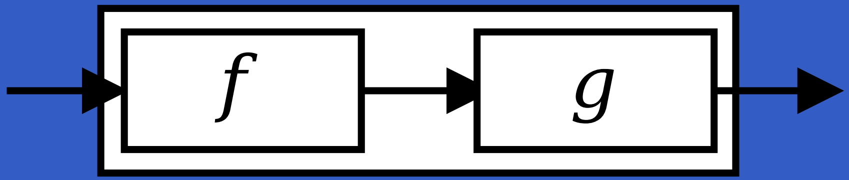
Example: Video tracker

Video trackers are typically stateful signal functions:



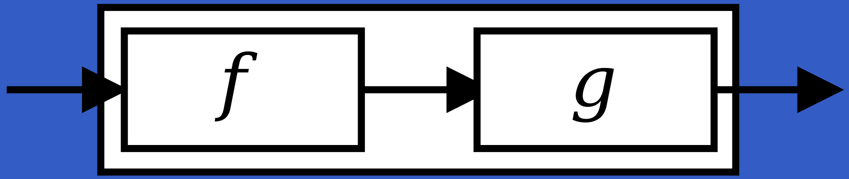
Building systems (1)

How to build systems? Think of a signal function as a ***block***. Blocks have inputs and outputs and can be combined into larger blocks. For example, serial composition:



Building systems (1)

How to build systems? Think of a signal function as a ***block***. Blocks have inputs and outputs and can be combined into larger blocks. For example, serial composition:

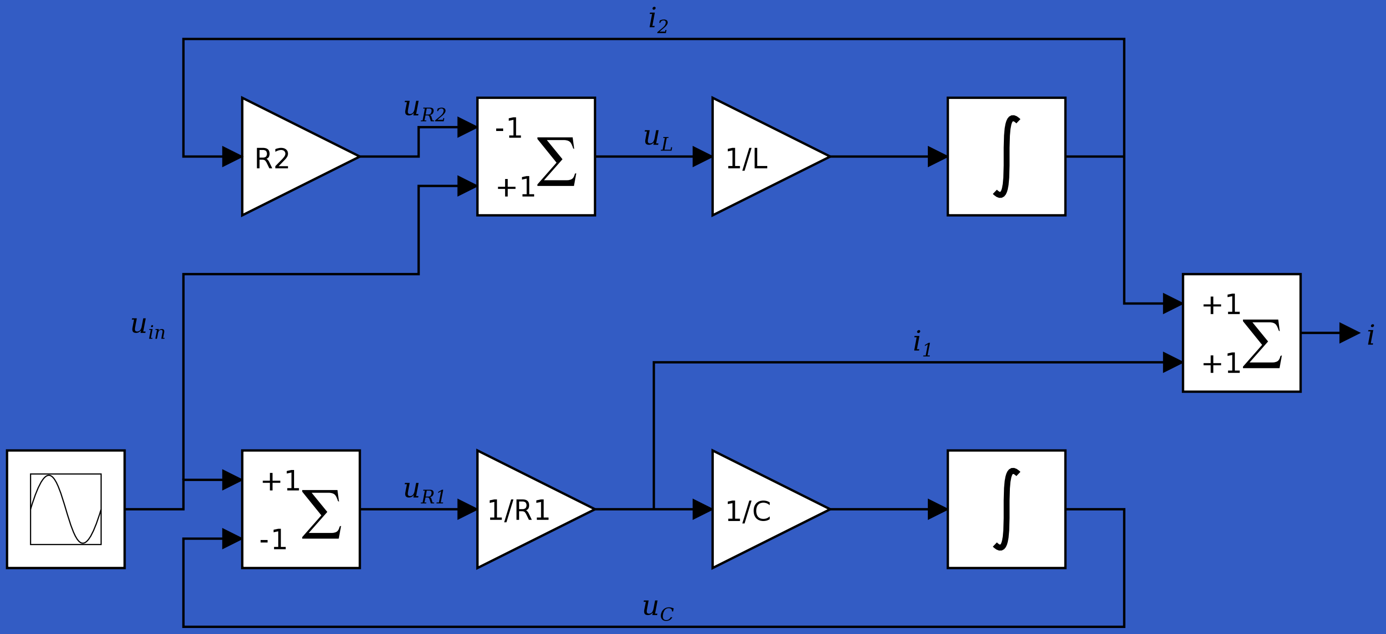


A *combinator* can be defined that captures this idea:

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

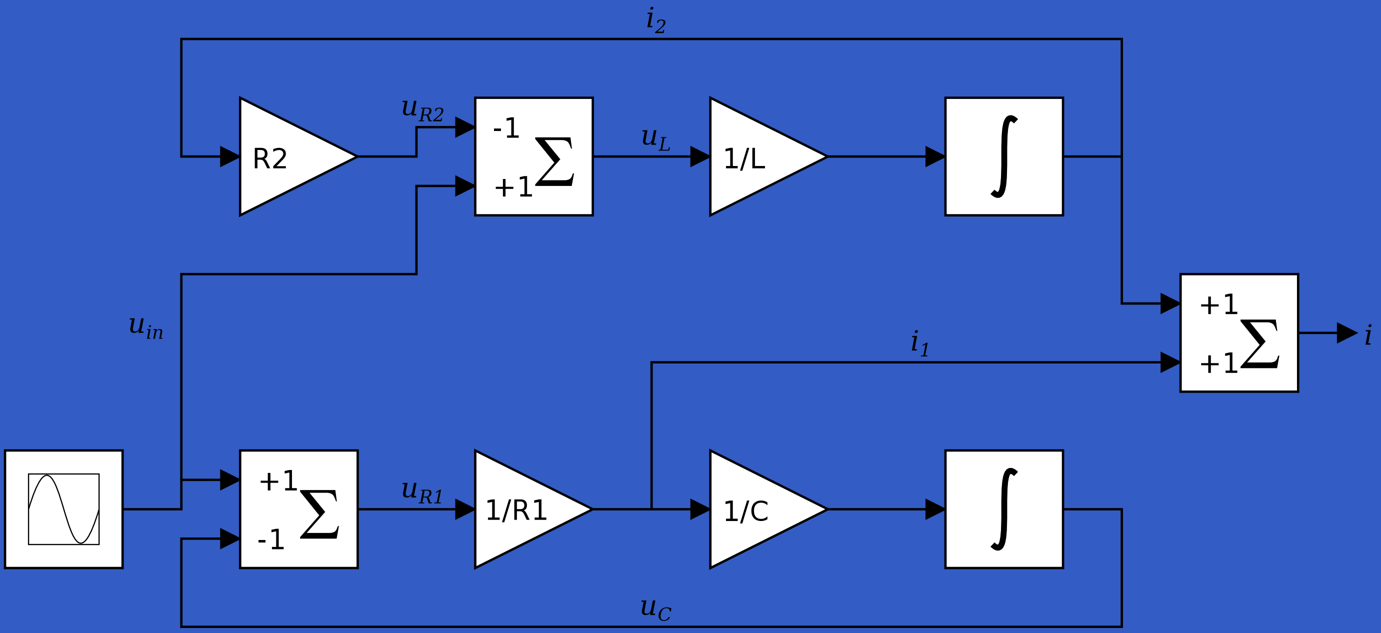
Building systems (2)

But systems can be complex:



Building systems (2)

But systems can be complex:



How many and what combinators do we need to be able to describe arbitrary systems?

Arrows

Yampa uses John Hughes’ ***arrow*** framework:

- Abstract data type interface for function-like types (or “blocks”, if you prefer).

Arrows

Yampa uses John Hughes' ***arrow*** framework:

- Abstract data type interface for function-like types (or "blocks", if you prefer).
- Particularly suitable for types representing process-like computations.

Arrows

Yampa uses John Hughes’ ***arrow*** framework:

- Abstract data type interface for function-like types (or “blocks”, if you prefer).
- Particularly suitable for types representing process-like computations.
- Provides a minimal set of “wiring” combinators.

What is an arrow? (1)

- A ***type constructor*** a of arity two.

What is an arrow? (1)

- A ***type constructor*** a of arity two.
- Three operators:

What is an arrow? (1)

- A ***type constructor*** a of arity two.
- Three operators:
  - ***lifting***:
    `arr ::  (b->c) -> a b c`

What is an arrow? (1)

- A ***type constructor*** a of arity two.
- Three operators:
  - ***lifting***:
    `arr ::  (b->c) -> a b c`
  - ***composition***:
    `(>>>) ::  a b c -> a c d -> a b d`

What is an arrow? (1)

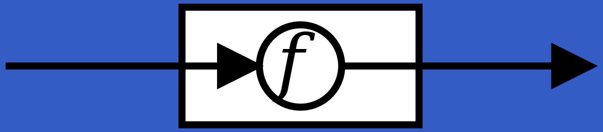
- A ***type constructor*** a of arity two.
- Three operators:
  - ***lifting***:
    `arr :: (b->c) -> a b c`
  - ***composition***:
    `(>>>) :: a b c -> a c d -> a b d`
  - ***widening***:
    `first :: a b c -> a (b,d) (c,d)`

What is an arrow? (1)

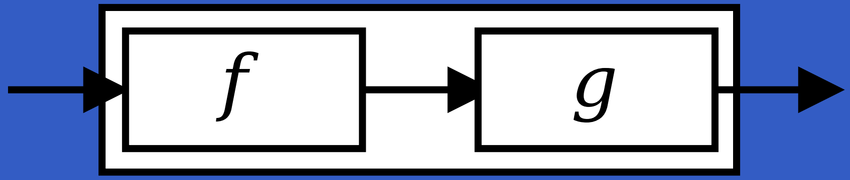
- A ***type constructor*** a of arity two.
- Three operators:
  - ***lifting***:
    ```
    arr ::  (b->c) -> a b c
    ```
  - ***composition***:
    ```
    (>>>) ::  a b c -> a c d -> a b d
    ```
  - ***widening***:
    ```
    first ::  a b c -> a (b,d) (c,d)
    ```
- A set of ***algebraic laws*** that must hold.

What is an arrow? (2)

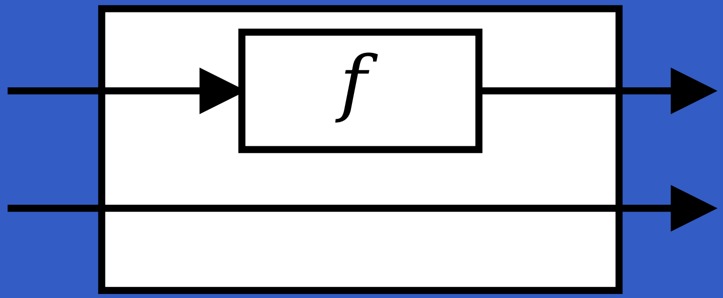
These diagrams convey the general idea:



`arr` f



f `>>>` g



`first` f

Some arrow laws

```
(f >>> g) >>> h = f >>> (g >>> h)
  arr (f >>> g) = arr f >>> arr g
   arr id >>> f = f
              f = f >>> arr id
  first (arr f) = arr (first f)
first (f >>> g) = first f >>> first g
```

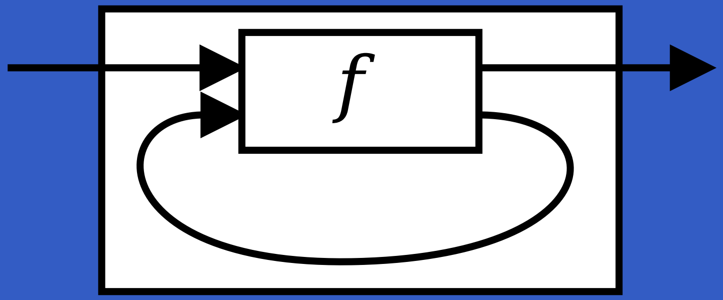
Some arrow laws

```
(f >>> g) >>> h = f >>> (g >>> h)
  arr (f >>> g) = arr f >>> arr g
   arr id >>> f = f
              f = f >>> arr id
  first (arr f) = arr (first f)
first (f >>> g) = first f >>> first g
```

Being able to use simple algebraic laws like these greatly facilitates reasoning about programs.

The `loop` combinator

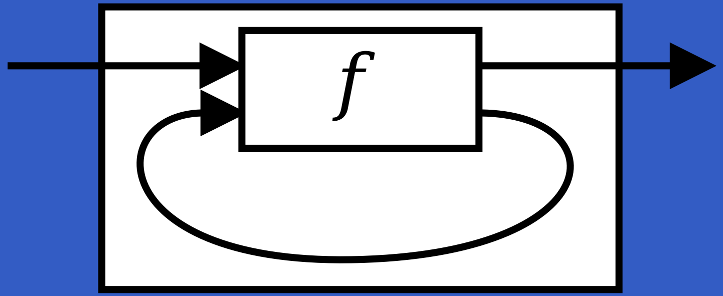
Another important operator is `loop`: a fixed-point operator used to express recursive arrows or ***feedback***:



`loop` f

The `loop` combinator

Another important operator is `loop`: a fixed-point operator used to express recursive arrows or ***feedback***:



`loop` f

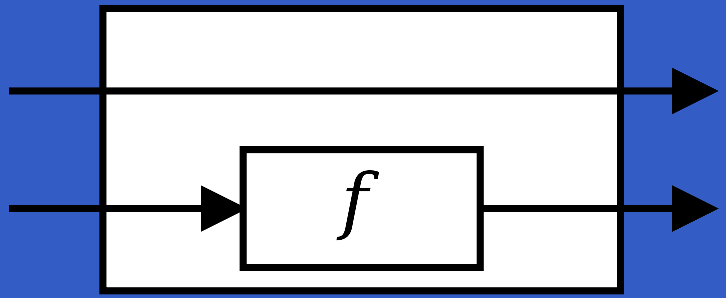
Remarkably, the four combinators `arr`, `>>>`, `first`, and `loop` suffice for expressing ***any conceivable wiring!***

Some more arrow combinators (1)

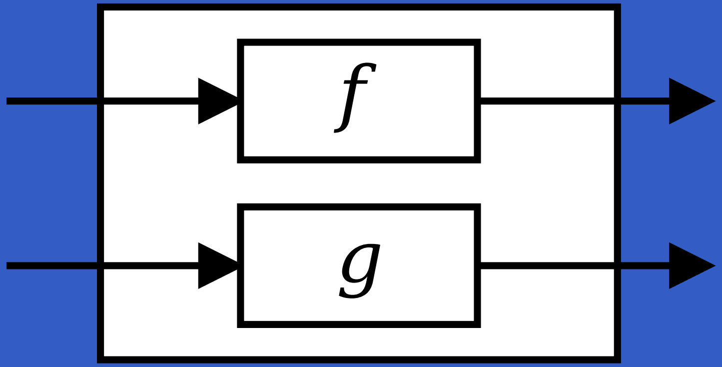
```
second :: Arrow a =>
    a b c -> a (d,b) (d,c)

(***) :: Arrow a =>
    a b c -> a d e -> a (b,d) (c,e)

(&&&) :: Arrow a =>
    a b c -> a b d -> a b (c,d)
```

Some more arrow combinators (2)

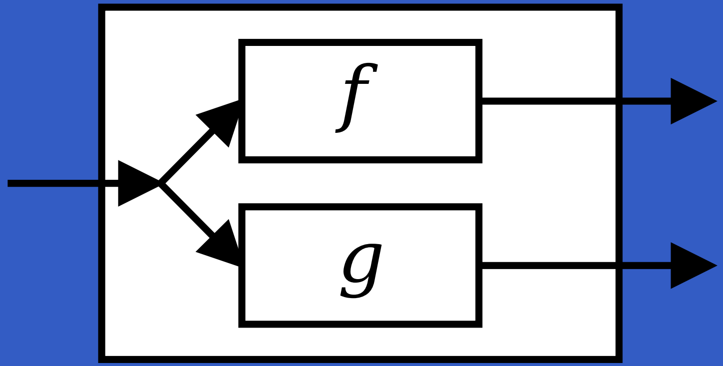
As diagrams:



second f



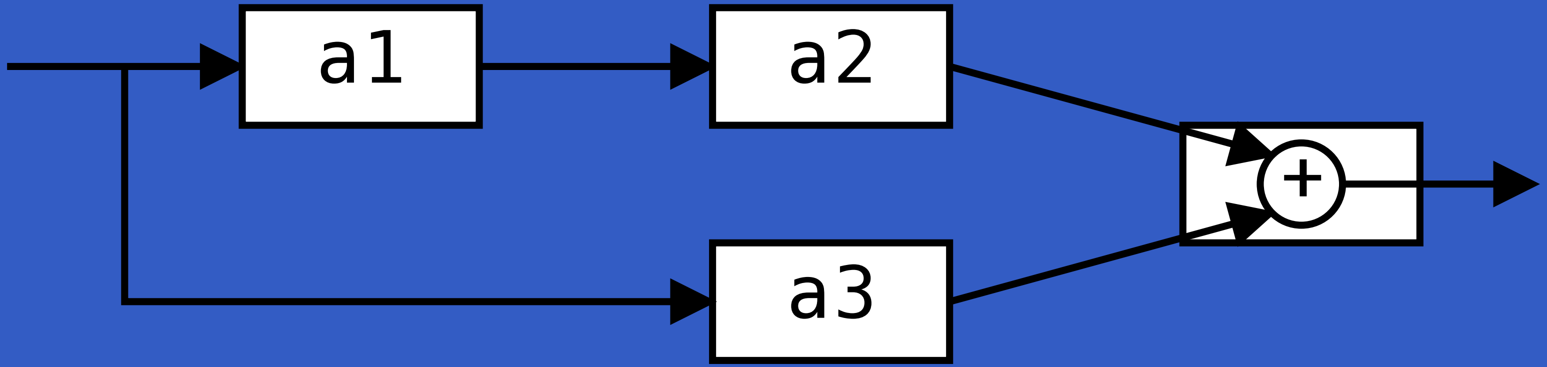
f *** g



f &&& g

Example: A Simple Network

A simple network:



```
a1, a2, a3 ::  A Double Double
```

One way to express it using arrow combinators:

```
circuit_v1 ::  A Double Double
circuit_v1 = (a1 &&& arr id)
             >>> (a2 *** a3)
             >>> arr (uncurry (+))
```

The arrow do notation (1)

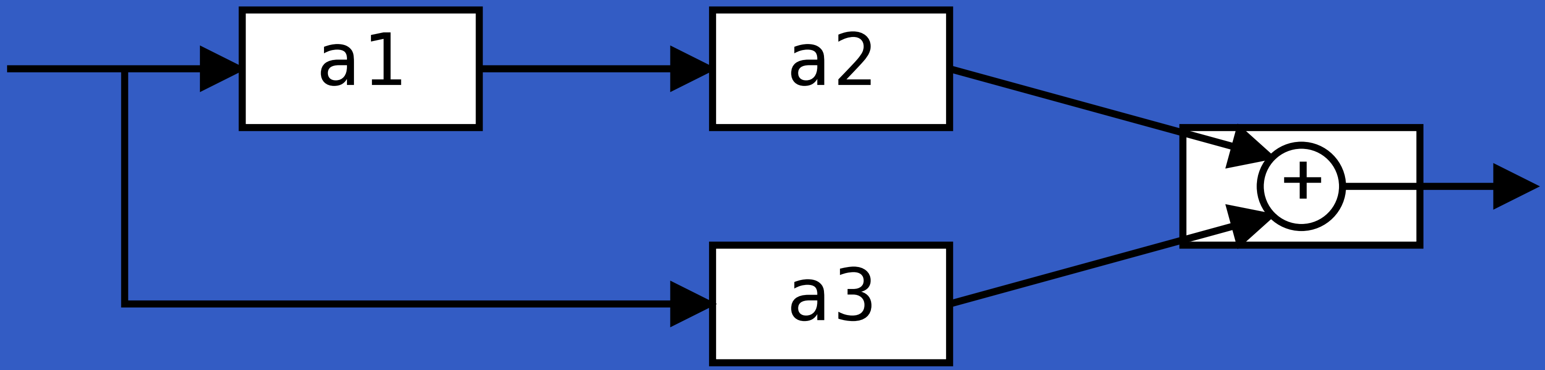
Using the basic combinators directly can be cumbersome. Ross Paterson's ***do***-notation for arrows provides a convenient alternative. Only ***syntactic sugar***!

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

Also: `let` pat = exp $\equiv$ pat `<- arr id -<` exp

The arrow do notation (2)

Let us redo the example using this notation:



```
circuit_v4 :: A Double Double
circuit_v4 = proc x -> do
    y1 <- a1 -< x
    y2 <- a2 -< y1
    y3 <- a3 -< x
    returnA -< y2 + y3
```

Yampa and Arrows

The Yampa signal function type is an arrow.

Signal function instances of the core 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`

Some further basic signal functions

-
  ```
  identity :: SF a a
  identity = arr id
  ```

Some further basic signal functions

- `identity :: SF a a`
  `identity = arr id`
- `constant :: b -> SF a b`
  `constant b = arr (const b)`

Some further basic signal functions

- `identity :: SF a a`
  `identity = arr id`
- `constant :: b -> SF a b`
  `constant b = arr (const b)`
- `integral :: VectorSpace a s=>SF a a`

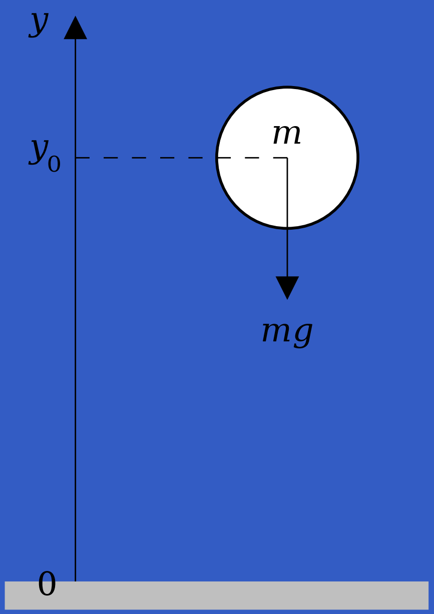
Some further basic signal functions

- `identity :: SF a a`
  `identity = arr id`
- `constant :: b -> SF a b`
  `constant b = arr (const b)`
- `integral :: VectorSpace a s=>SF a a`
- `time :: SF a Time`
  `time = constant 1.0 >>> integral`

Some further basic signal functions

- `identity :: SF a a`
  `identity = arr id`
- `constant :: b -> SF a b`
  `constant b = arr (const b)`
- `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



$$\begin{aligned} y &= y_0 + \int v \,\mathrm{d}t \\ v &= v_0 + \int -9.81 \end{aligned}$$

On impact:

$$v = -v(t-)$$

(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 y0 v0 = proc () -> do
    v <- (v0 +) ˆ<< integral -< -9.81
    y <- (y0 +) ˆ<< integral -< v
    returnA -< (y, v)
```

Events

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

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
```

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 $= \texttt{Signal}\,(\texttt{Event}\,\alpha)$.

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 $= \texttt{Signal}\,(\texttt{Event}\,\alpha)$.

Associating information with an event occurrence:

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

Modelling the bouncing ball: part 2

Detecting when the ball goes through the floor:

```
fallingBall’ ::
    Pos -> Vel
    -> SF () ((Pos,Vel), Event (Pos,Vel))
fallingBall’ y0 v0 = proc () -> do
    yv@(y, _) <- fallingBall y0 v0 -< ()
    hit       <- edge              -< y <= 0
    returnA -< (yv, hit ‘tag‘ yv)
```

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.

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***.

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***.
- The new signal function instance 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***.
- The new signal function instance 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 takes place 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 takes place 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 takes place on the first occurrence of the switching event source.

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

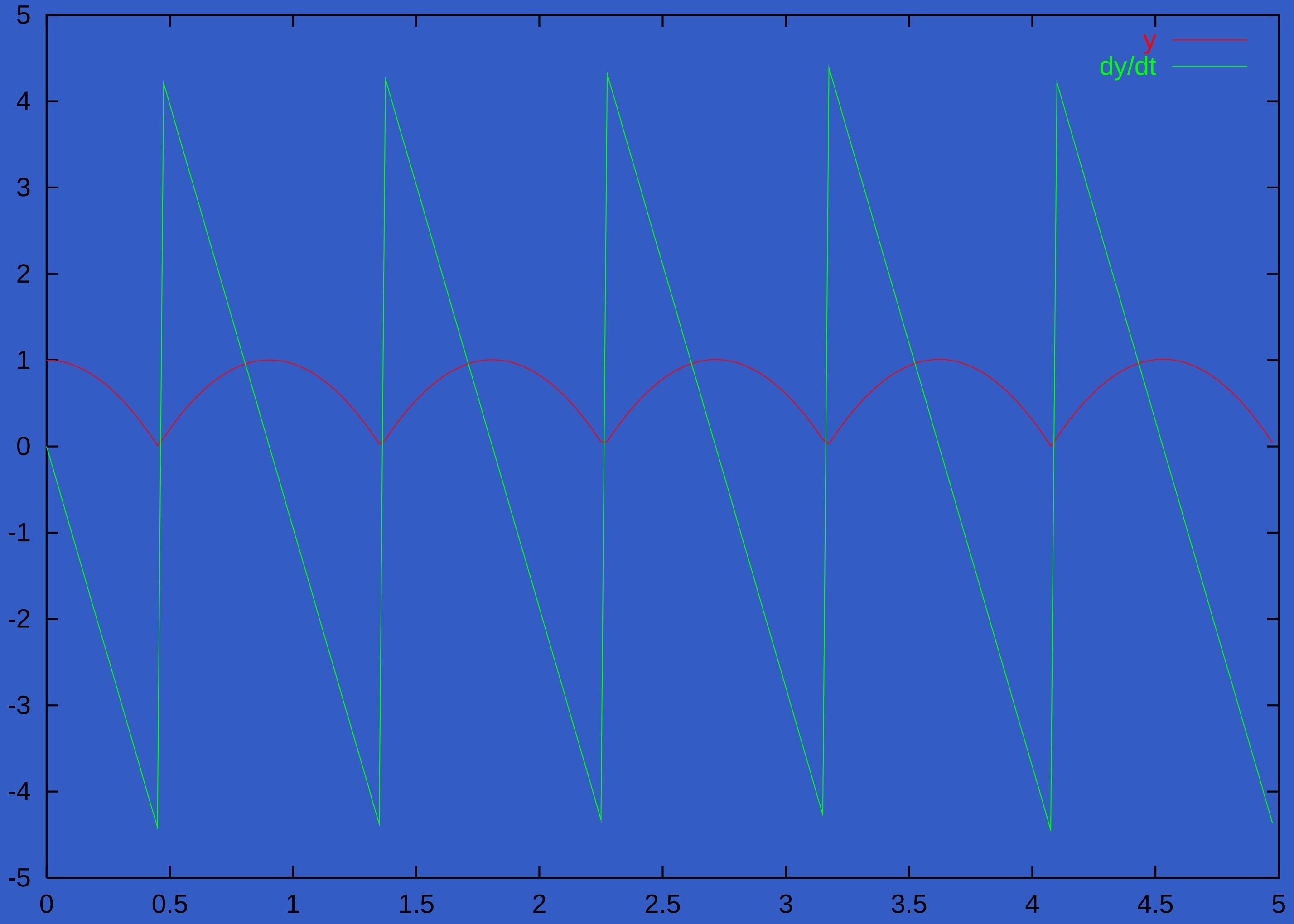
Function yielding SF to switch into

Modelling the bouncing ball: part 3

Making the ball bounce:

```
bouncingBall :: Pos -> SF () (Pos, Vel)
bouncingBall y0 = bbAux y0 0.0
  where
    bbAux y0 v0 =
      switch (fallingBall’ y0 v0) $ \(y,v) ->
      bbAux y (-v)
```

Simulation of bouncing ball



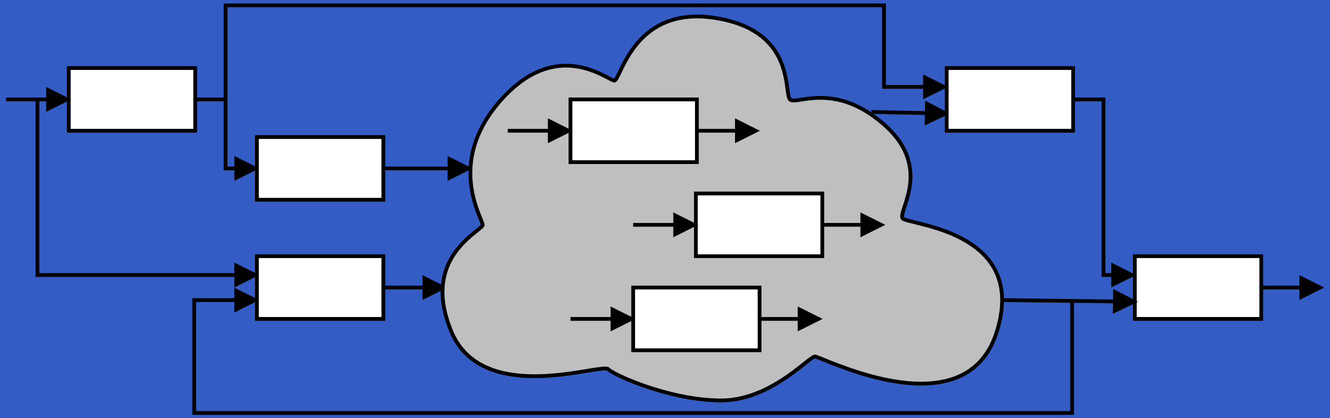
Highly dynamic system structure?

Basic switch allows one signal function to be replaced by another.

Highly dynamic system structure?

Basic switch allows one signal function to be replaced by another.

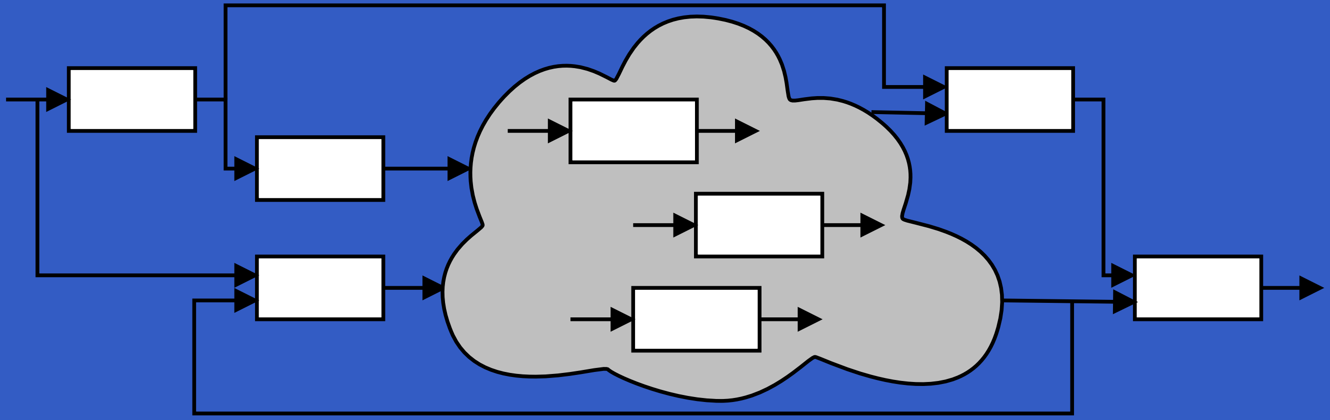
- What about more general structural changes?



Highly dynamic system structure?

Basic switch allows one signal function to be replaced by another.

- What about more general structural changes?



- What about state?

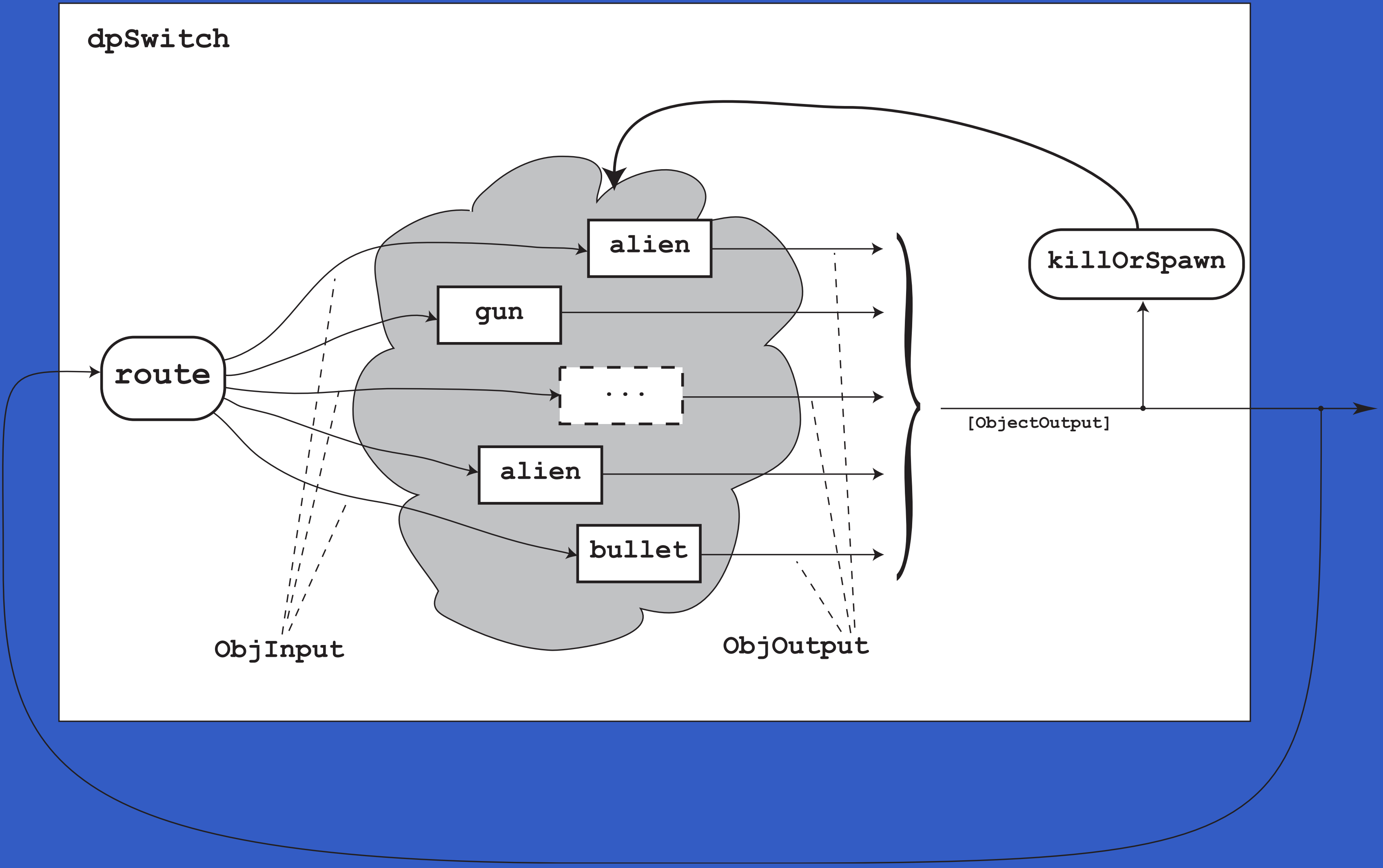
Dynamic signal function collections

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.

Example: Space Invaders

Overall game structure



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
```

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 is 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 is simpler.
- Synchronous approach avoids “event-call-back soup”, meaning robust, easy-to-understand semantics.