MGS 2006: AFP Lecture 4

Functional Reactive Programming and Arrows

Henrik Nilsson

University of Nottingham, UK

Reactive programming

Reactive systems:

Reactive programming

Reactive systems:

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

Reactive programming

Reactive systems:

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

Reactive programming

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

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 for reactive systems.

Functional Reactive Programming (1)

Functional Reactive Programming (FRP):

- Paradigm for reactive programming in a functional setting.

Functional Reactive Programming (1)

Functional Reactive Programming (FRP):

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

Functional Reactive Programming (1)

Functional Reactive Programming (FRP):

- Paradigm for reactive programming in a 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 (1)

Functional Reactive Programming (FRP):

- Paradigm for reactive programming in a 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.

Functional Reactive Programming (2)

Yampa:

- The most recent Yale FRP implementation.

Functional Reactive Programming (2)

Yampa:

- The most recent Yale FRP implementation.
- ***Embedding*** in Haskell (a Haskell library).

Functional Reactive Programming (2)

Yampa:

- The most recent Yale FRP implementation.
- ***Embedding*** in Haskell (a Haskell library).
- ***Arrows*** used as the basic structuring framework.

Functional Reactive Programming (2)

Yampa:

- The most recent Yale FRP implementation.
- ***Embedding*** in Haskell (a Haskell library).
- ***Arrows*** used as the basic structuring framework.
- ***Continuous time***.

Functional Reactive Programming (2)

Yampa:

- The most recent Yale FRP implementation.
- ***Embedding*** in Haskell (a Haskell library).
- ***Arrows*** used as the basic structuring framework.
- ***Continuous time***.
- Discrete-time signals modelled by continuous-time signals and an option type.

Functional Reactive Programming (2)

Yampa:

- The most recent Yale FRP implementation.
- ***Embedding*** in Haskell (a Haskell library).
- ***Arrows*** used as the basic structuring framework.
- ***Continuous time***.
- Discrete-time signals modelled by continuous-time signals and an option type.
- Advanced ***switching constructs*** allows for highly dynamic system structure.

Related languages

FRP related to:

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

Related languages

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.

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)

Yampa?

Yampa?

Yet
Another
Mostly
Pointless
Acronym

Yampa?

Yet
Another
Mostly
Pointless
Acronym

???

Yampa?

Yet
Another
Mostly
Pointless
Acronym

???

No . . .

Yampa?

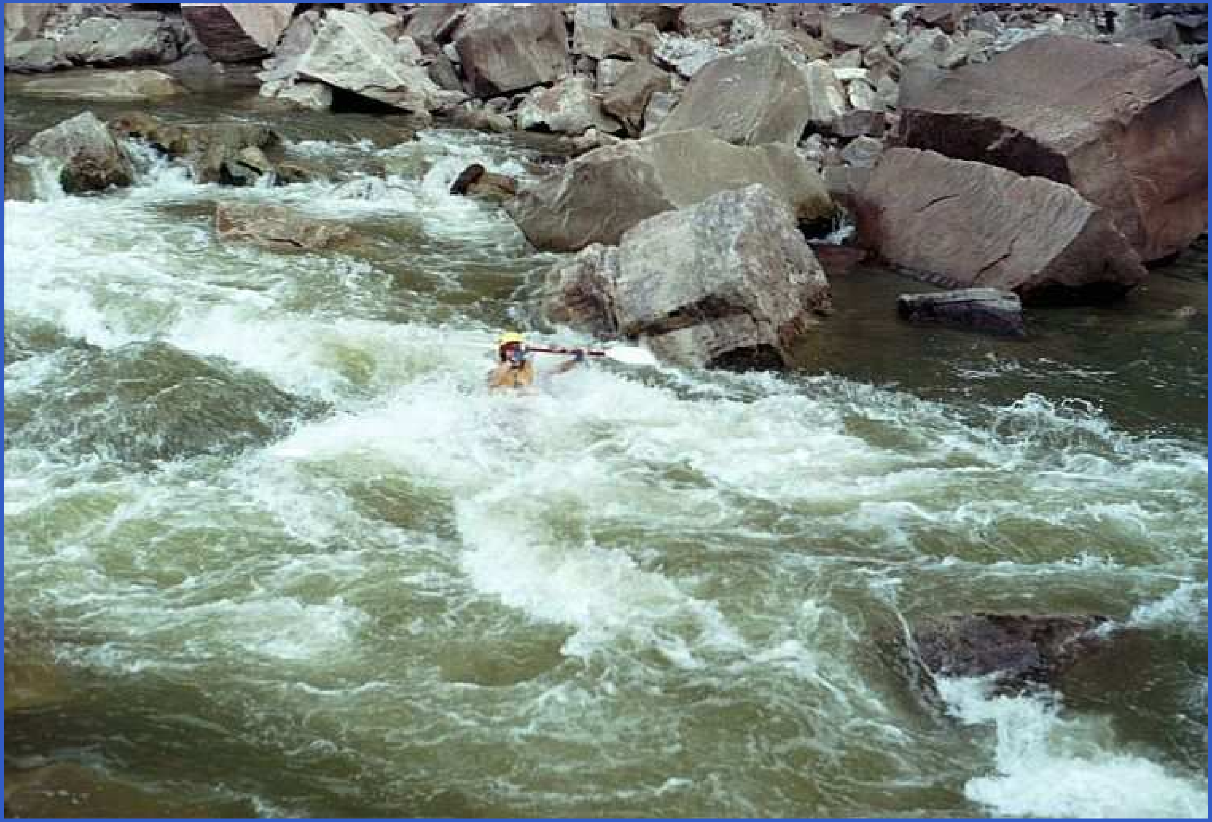
Yampa is a river . . .

Yampa?

. . . with long calmly flowing sections . . .

Yampa?

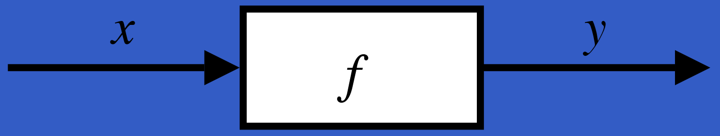
. . . and abrupt whitewater transitions in between.



A good metaphor for hybrid systems!

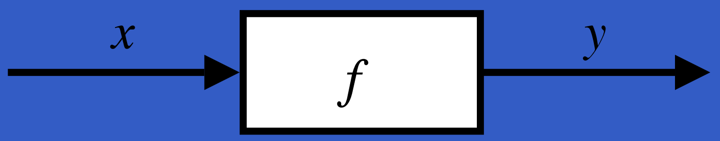
Signal functions

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



Signal functions

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

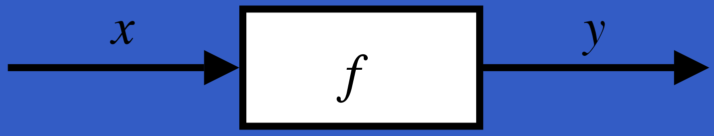


Intuition:

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

Signal functions

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



Intuition:

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

Additionally: ***causality*** requirement.

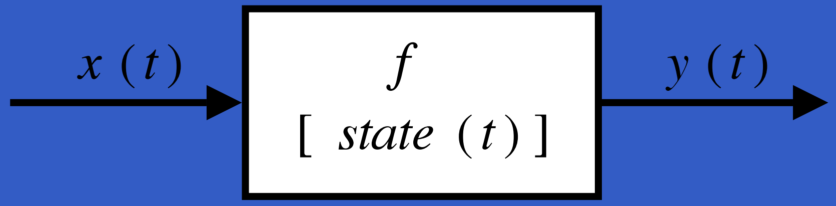
Signal functions and state

Alternative view:

Signal functions and state

Alternative view:

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

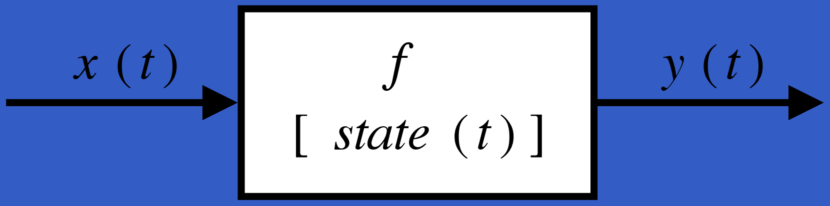


$state(t)$ summarizes input history $x(t')$, $t' \in [0, t]$.

Signal functions and state

Alternative view:

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



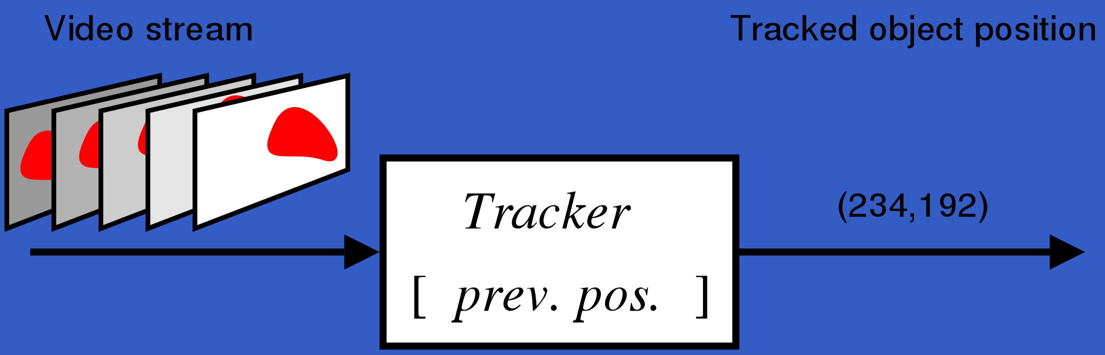
$state(t)$ summarizes input history $x(t')$, $t' \in [0, t]$.

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:



Signal functions in Yampa

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

Signal functions in Yampa

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

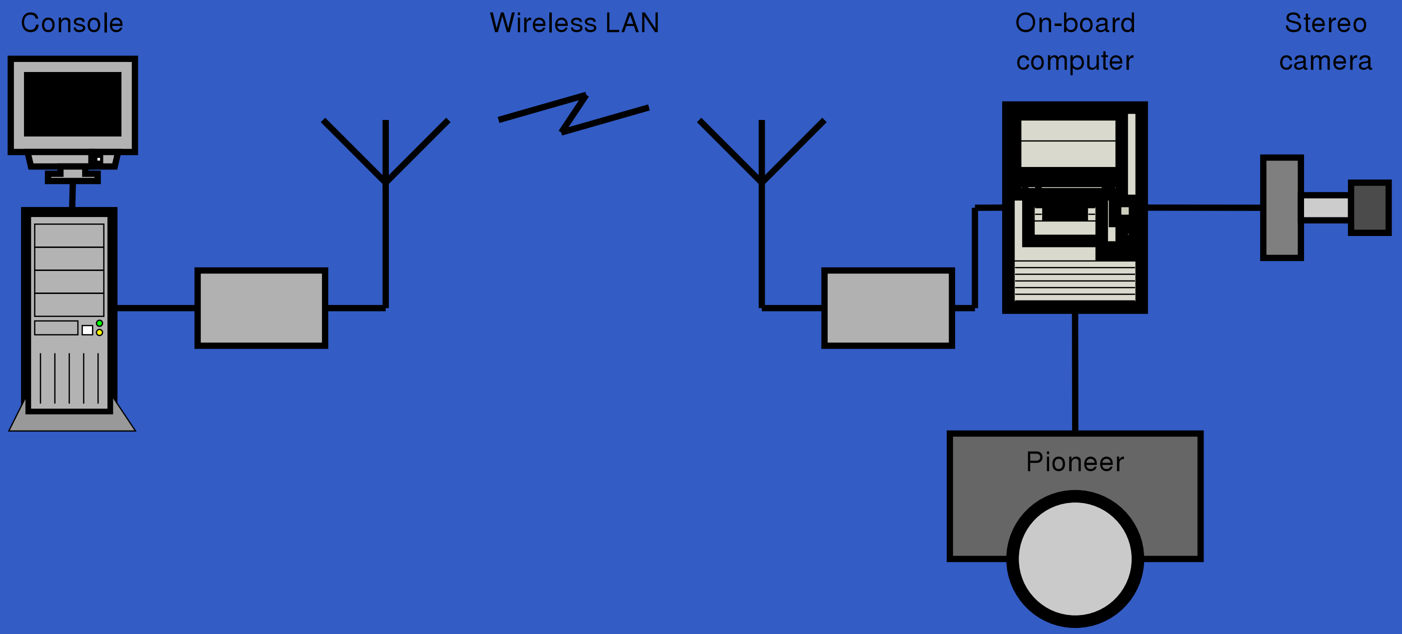
Signal functions in Yampa

- ***Signal functions*** are ***first class entities***. Intuition: $\texttt{SF}\ \alpha\ \beta \approx \texttt{Signal}\ \alpha \rightarrow \texttt{Signal}\ \beta$
- ***Signals*** are ***not*** first class entities: they only exist indirectly through signal functions.
- The second-class nature of signals allows causality to be exploited for an efficient implementation.

Example: Robotics (1)

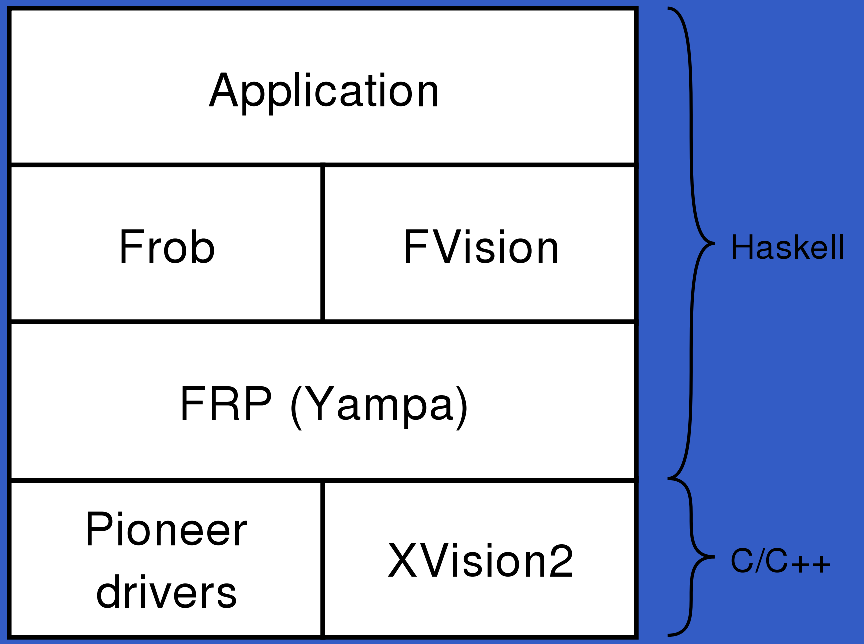
[PPDP'02, with Izzet Pembeci and Greg Hager, Johns Hopkins University]

Hardware setup:



Example: Robotics (2)

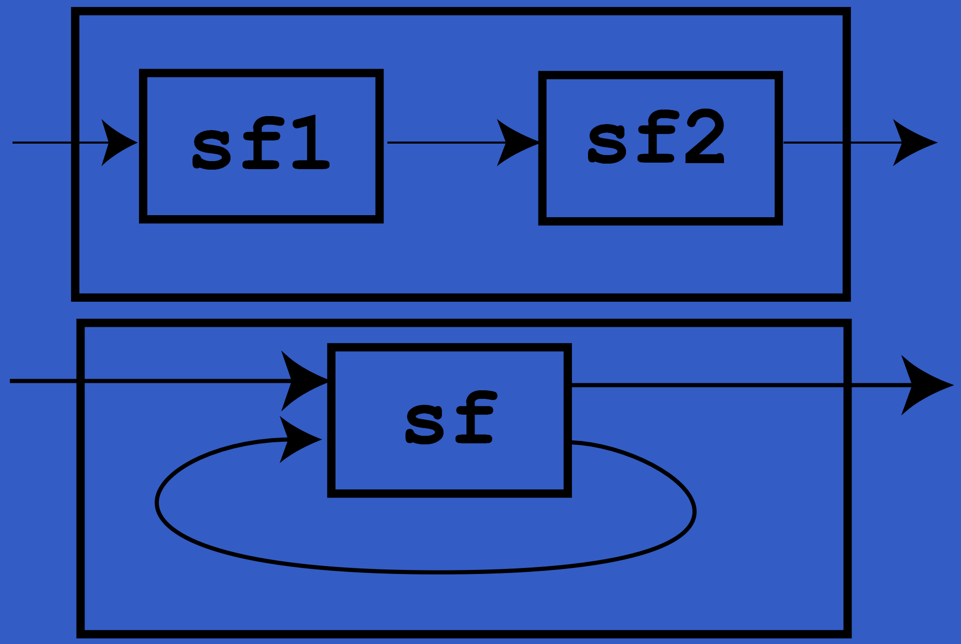
Software architecture:



Example: Robotics (3)

Yampa and Arrows (1)

Systems are described by combining signal functions (forming new signal functions):



Yampa and Arrows (2)

Yampa uses John Hughes' ***arrow*** framework: the signal function type is an arrow.

Signal function instances of 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`

Yampa and Arrows (2)

Yampa uses John Hughes' ***arrow*** framework: the signal function type is an arrow.

Signal function instances of 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`

Enough to express any conceivable "wiring".

Arrows, Monads, and FRP (1)

- Like monads, arrows represent a form of effectful computations.
- In fact, some arrows, those that support an ***apply*** operation, are also monads (but not vice versa).

Arrows, Monads, and FRP (2)

- Could Yampa be based on monads instead?

  NO! Essentially because

  ```
  (>>=) :: Monad m =>
               m a -> (a -> m b) -> m b
  ```

  implies that a new signal function would have to be computed at every point in time, depending on the result of the first computation. This does not make much sense in a dataflow setting.

- But possibly on ***co-monads*** (Uustalu, Vene 2005)

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 ]`
    pat_1 `<-` $sfexp_1$ `-<` exp_1
    pat_2 `<-` $sfexp_2$ `-<` exp_2
    `...`
    pat_n `<-` $sfexp_n$ `-<` exp_n
    `returnA -<` exp

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

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
  ```
  ```

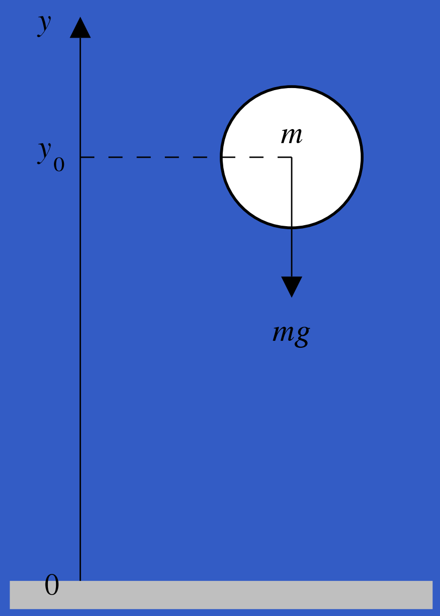
Some further basic signal functions

- ```
  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



$$y = y_0 + \int v \, \mathrm{d}t$$

$$v = v_0 + \int -9.81$$

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 +) ^<< integral -< -9.81
    y <- (y0 +) ^<< 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 $=$ `Signal (Event` α`)`.

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` α).

Associating 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 ())`
  ```

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
```

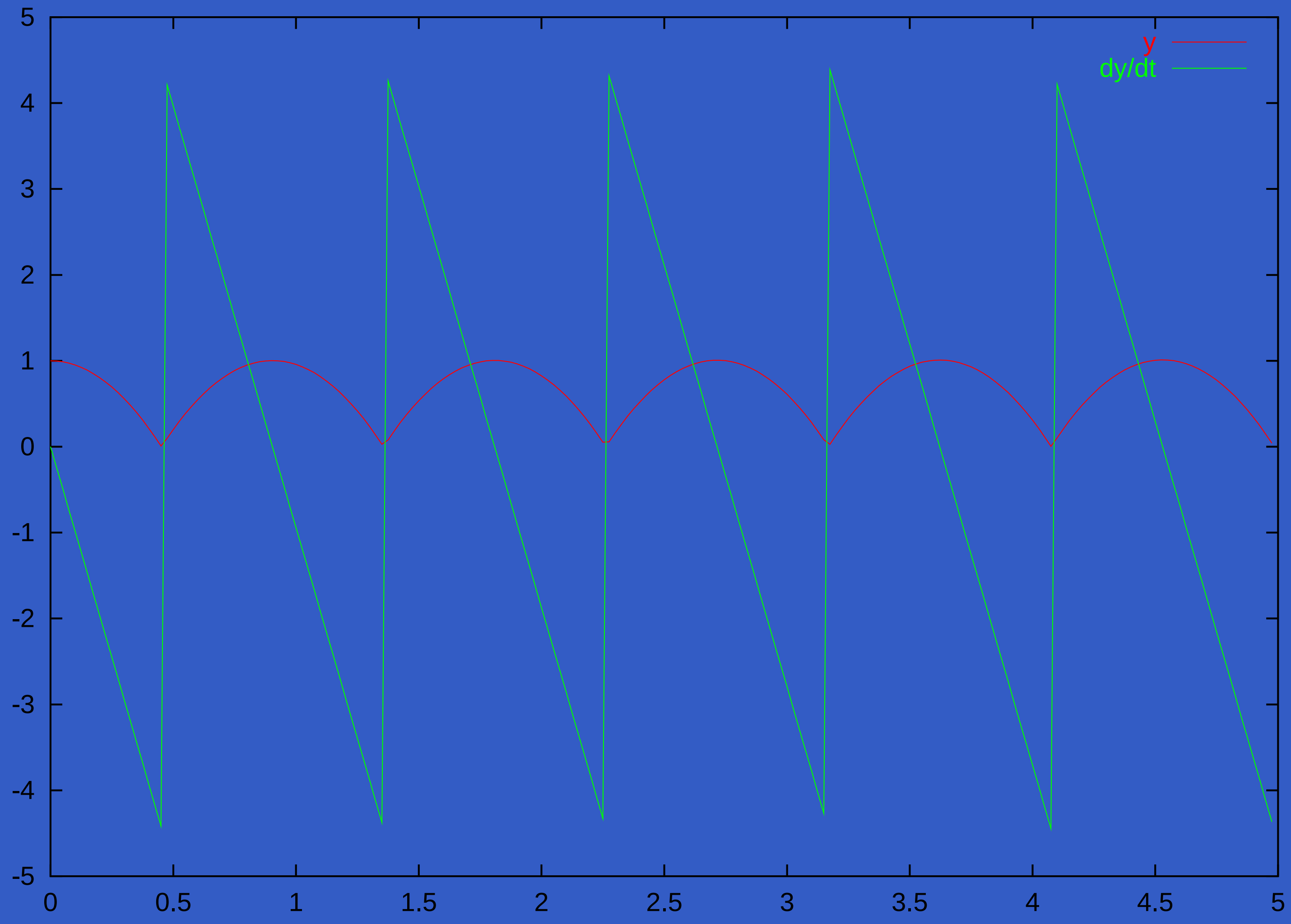
Function yielding SF to switch into

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



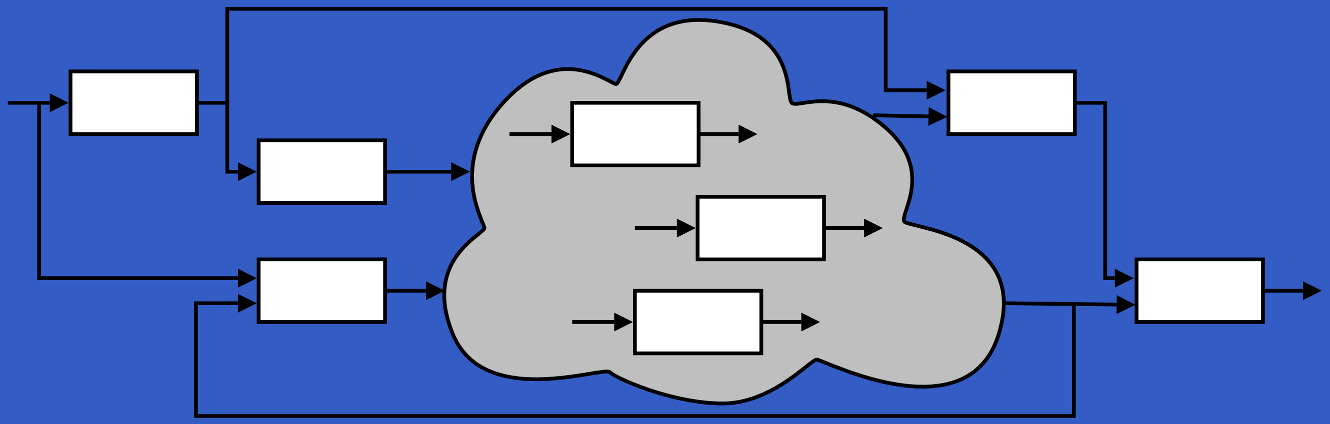
Highly dynamic system structure?

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.

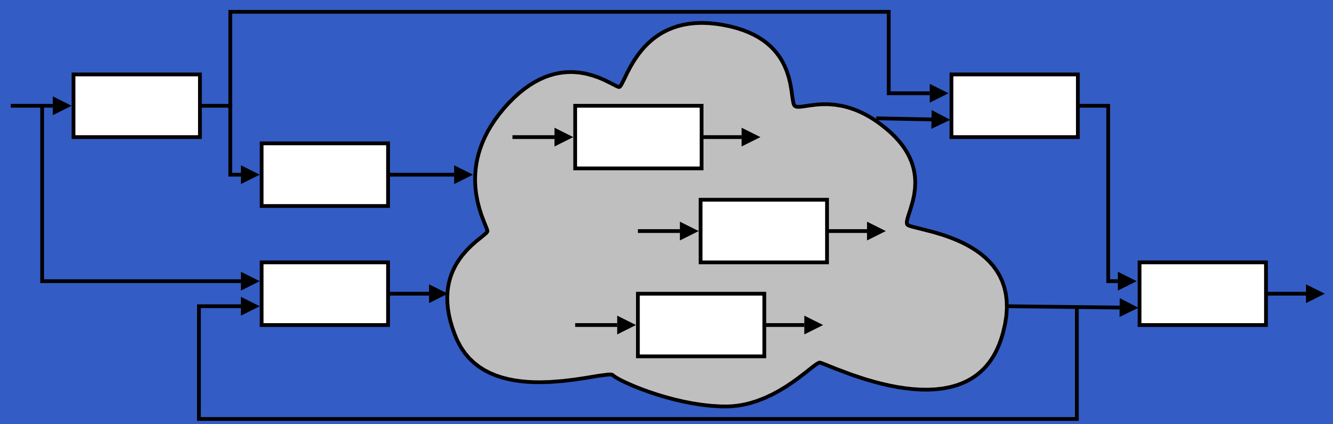
- What about more general structural changes?



Highly dynamic system structure?

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

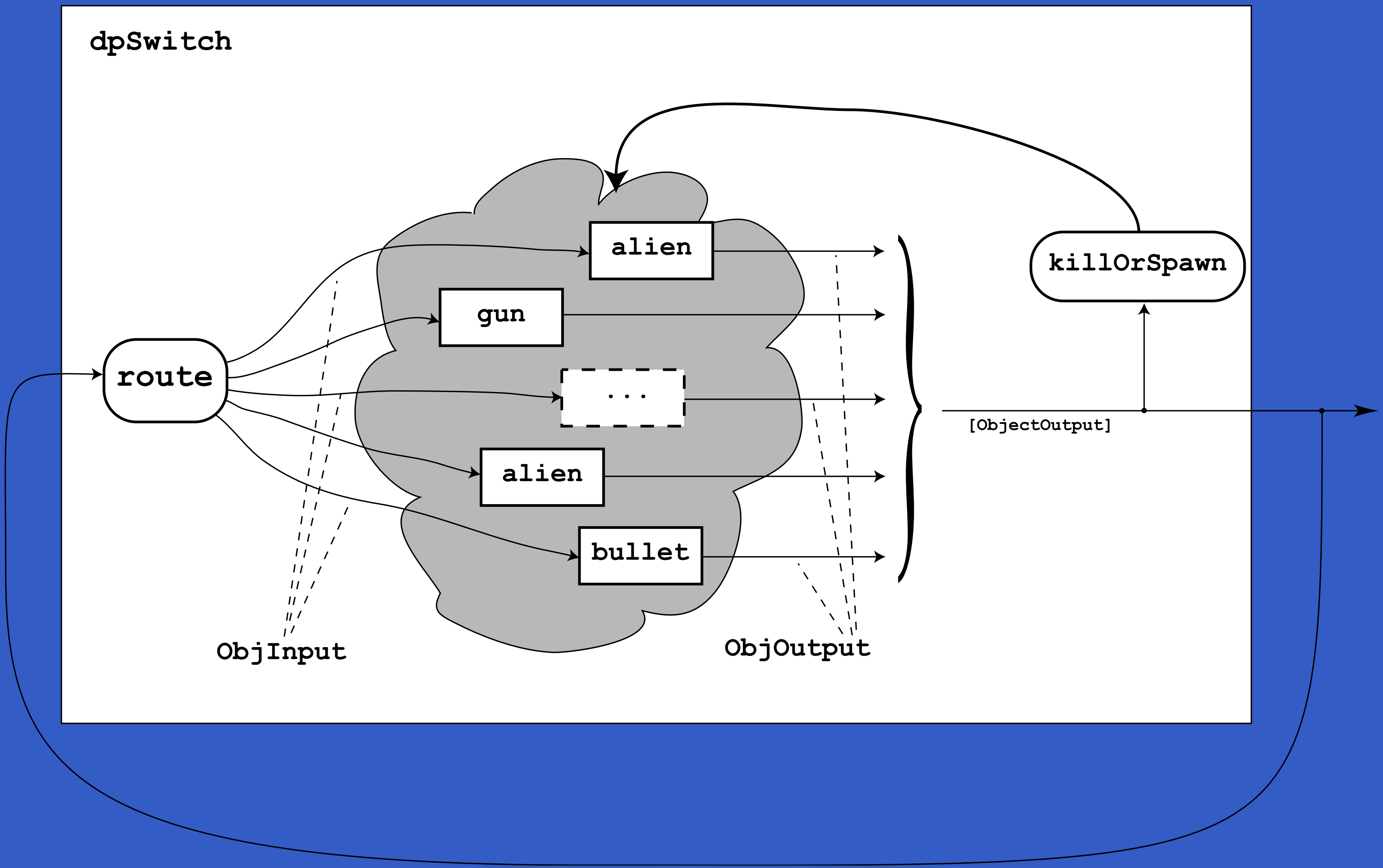
- What about more general structural changes?



- What about state?

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.

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)        <- 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)
```

Event source

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

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
```

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.

Obtaining Yampa

Yampa 0.92 is available from

`http://www.haskell.org/yampa`

Reading

- John Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, May 2000
- John Hughes. Programming with arrows. In *Advanced Functional Programming*, 2004. To be published by Springer Verlag.
- Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 Haskell Workshop*, pp. 51–64, October 2002.

Reading (2)

- Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In *Advanced Functional Programming*, 2002. LNCS 2638, pp. 159–187.
- Tarmo Uustalu and Varmo Vene. The Essence of Dataflow Programming. 2005