Declarative Game Programming

PPDP 2014
Distilled Tutorial

Henrik Nilsson and Ivan Perez

School of Computer Science

University of Nottingham, UK

Declarative Game Programming?

Video games are not a major application area for declarative programming . . .

Declarative Game Programming?

Video games are not a major application area for declarative programming . . . or even a niche one.

Declarative Game Programming?

Video games are not a major application area for declarative programming . . . or even a niche one.

- Many historical and pragmatical reasons

Declarative Game Programming?

Video games are not a major application area for declarative programming . . . or even a niche one.

- Many historical and pragmatical reasons
- More principled objection:

  With state and effects being pervasive in video games, is declarative programming a good fit?

Take-home Message # 1

Take-home Message # 1

Video games can be programmed declaratively by describing ***what*** game entities are ***over*** time, not just at a point in time.

Take-home Message # 1

Video games can be programmed declaratively by describing ***what*** game entities are ***over*** time, not just at a point in time.

(We focus on the core game logic in the following: there will often be code around the "edges" (e.g., rendering, interfacing to input devices) that may not be very declarative, at least not in the sense above.)

Take-home Message # 2

You too can program games declaratively . . .

Take-home Message # 2

You too can program games declaratively . . . today!

This Tutorial

We will implement a Breakout-like game using:

- Functional Reactive Programming (FRP): a paradigm for describing time-varying entities
- Simple DirectMedia Layer (SDL) for rendering etc.

Focus on FRP as that is what we need for the game logic. We will use Yampa:

`http://hackage.haskell.org/package/Yampa-0.9.6`

Functional Reactive Programming

What is Functional Reactive Programming (FRP)?

Functional Reactive Programming

What is Functional Reactive Programming (FRP)?

- Paradigm for reactive programming in a functional setting.

Functional Reactive Programming

What is Functional Reactive Programming (FRP)?

- Paradigm for reactive programming in a functional setting.
- Idea: programming with time-varying entities.

Functional Reactive Programming

What is Functional Reactive Programming (FRP)?

- Paradigm for reactive programming in a functional setting.
- Idea: programming with time-varying entities.
- Originated from Functional Reactive Animation (Fran) (Elliott & Hudak).

Functional Reactive Programming

What is Functional Reactive Programming (FRP)?

- Paradigm for reactive programming in a functional setting.
- Idea: programming with time-varying entities.
- Originated from Functional Reactive Animation (Fran) (Elliott & Hudak).
- Has evolved in a number of directions and into different concrete implementations.

Functional Reactive Programming

What is Functional Reactive Programming (FRP)?

- Paradigm for reactive programming in a functional setting.
- Idea: programming with time-varying entities.
- Originated from Functional Reactive Animation (Fran) (Elliott & Hudak).
- Has evolved in a number of directions and into different concrete implementations.
- Often realised as an ***Embedded Domain-Specific Language (EDSL)***.

FRP Applications

Some domains where FRP or FRP-inspired approaches have been used:

- Graphical Animation
- Robotics
- Vision
- Sound synthesis
- GUIs
- Virtual Reality Environments
- Games

FRP Applications

Some domains where FRP or FRP-inspired approaches have been used:

- Graphical Animation
- Robotics
- Vision
- Sound synthesis
- GUIs
- Virtual Reality Environments
- ***GAMES***

Key FRP Features

Combines conceptual simplicity of the synchronous data flow approach with the flexibility and abstraction power of higher-order functional programming:

Key FRP Features

Combines conceptual simplicity of the synchronous data flow approach with the flexibility and abstraction power of higher-order functional programming:

- Synchronous

Key FRP Features

Combines conceptual simplicity of the synchronous data flow approach with the flexibility and abstraction power of higher-order functional programming:

- Synchronous
- First class temporal abstractions

Key FRP Features

Combines conceptual simplicity of the synchronous data flow approach with the flexibility and abstraction power of higher-order functional programming:

- Synchronous
- First class temporal abstractions
- Hybrid: mixed continuous and discrete time

Key FRP Features

Combines conceptual simplicity of the synchronous data flow approach with the flexibility and abstraction power of higher-order functional programming:

- Synchronous
- First class temporal abstractions
- Hybrid: mixed continuous and discrete time
- Dynamic system structure

Key FRP Features

Combines conceptual simplicity of the synchronous data flow approach with the flexibility and abstraction power of higher-order functional programming:

- Synchronous
- First class temporal abstractions
- Hybrid: mixed continuous and discrete time
- Dynamic system structure

Good fit for typical video games

Key FRP Features

Combines conceptual simplicity of the synchronous data flow approach with the flexibility and abstraction power of higher-order functional programming:

- Synchronous
- First class temporal abstractions
- Hybrid: mixed continuous and discrete time
- Dynamic system structure

Good fit for typical video games
(but not everything labelled “FRP” supports them all).

Yampa

Yampa

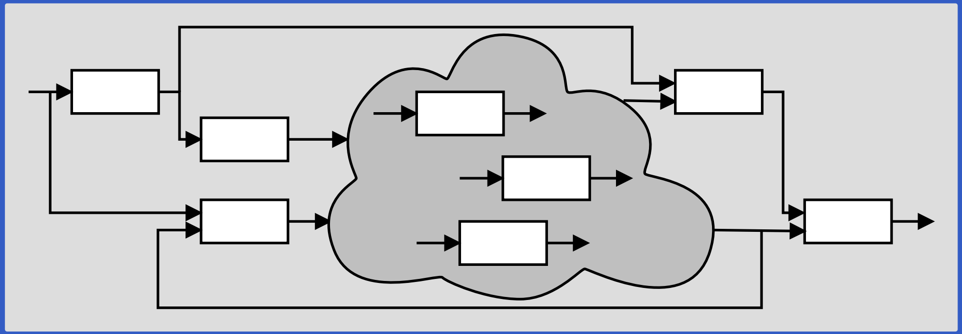
- FRP implemenattion embedded in Haskell

Yampa

- FRP implemenattion embedded in Haskell
- Key concepts:
  - ***Signals***: time-varying values
  - ***Signal Functions***: functions on signals
  - ***Switching*** between signal functions

Yampa

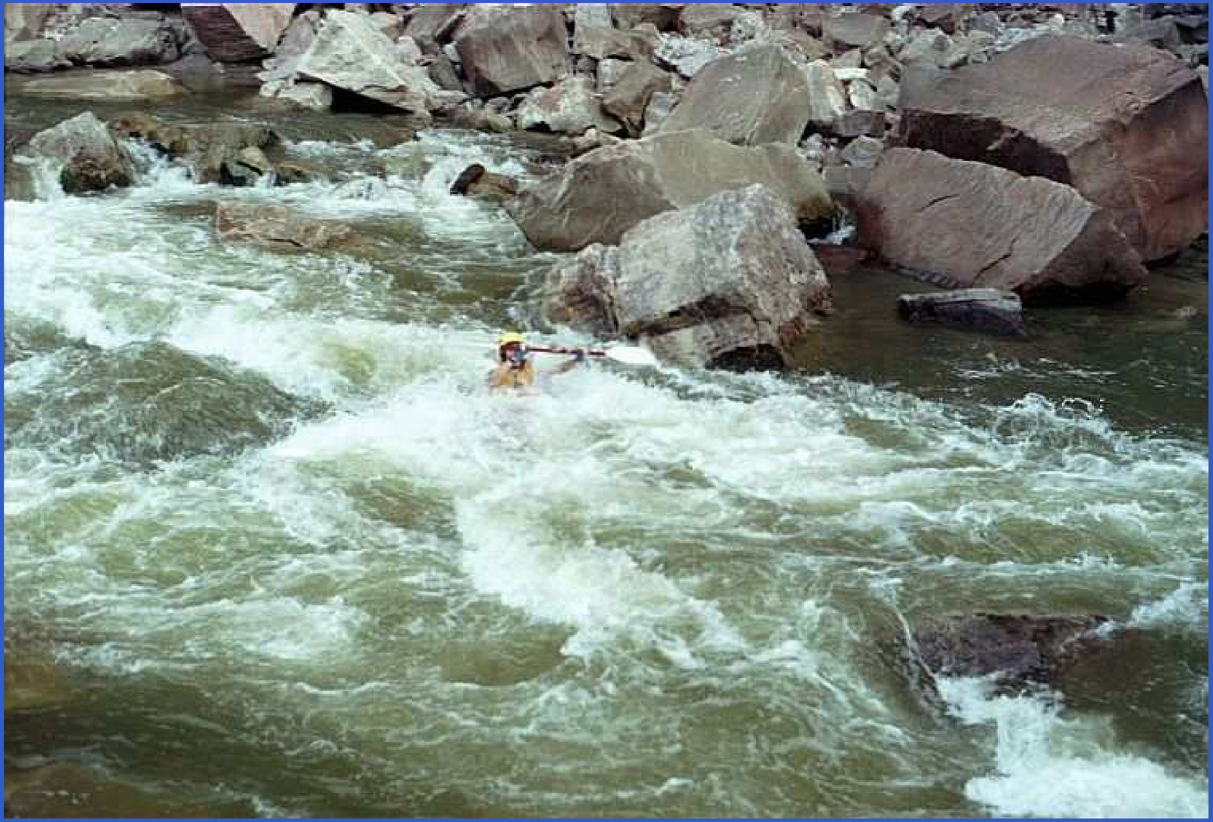
- FRP implemenattion embedded in Haskell
- Key concepts:
  - ***Signals***: time-varying values
  - ***Signal Functions***: functions on signals
  - ***Switching*** between signal functions
- Programming model:



Yampa?

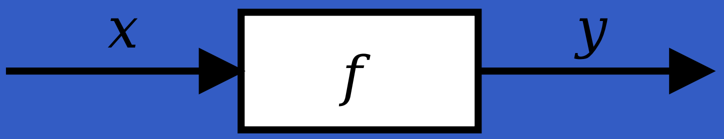
Yampa?

Yampa is a river with long calmly flowing sections and abrupt whitewater transitions in between.

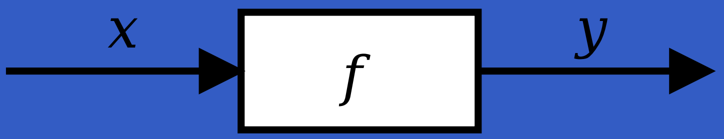


A good metaphor for hybrid systems!

Signal Functions

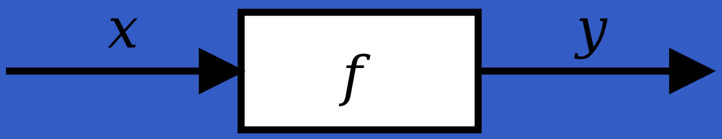


Signal Functions



Intuition:

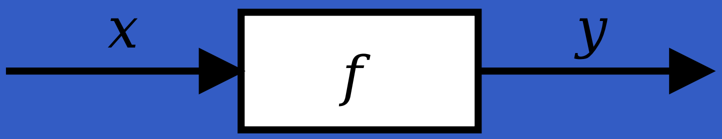
Signal Functions



Intuition:

$$Time \approx \mathbb{R}$$

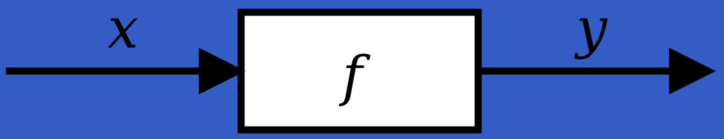
Signal Functions



Intuition:

$$Time \approx \mathbb{R}$$
$$Signal\ a \approx Time \rightarrow a$$
$$x :: Signal\ T1$$
$$y :: Signal\ T2$$

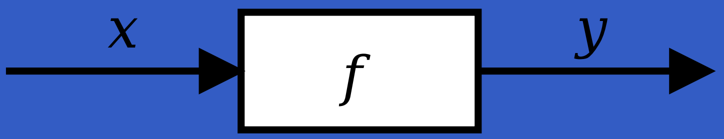
Signal Functions



Intuition:

$$
\begin{aligned}
&Time \approx \mathbb{R}\\
&Signal\ a \approx Time \rightarrow a\\
&x :: Signal\ T1\\
&y :: Signal\ T2\\
&SF\ a\ b \approx Signal\ a \rightarrow Signal\ b\\
&f :: SF\ T1\ T2
\end{aligned}
$$

Signal Functions



Intuition:

$$Time \approx \mathbb{R}$$
$$Signal\ a \approx Time \rightarrow a$$
$$x :: Signal\ T1$$
$$y :: Signal\ T2$$
$$SF\ a\ b \approx Signal\ a \rightarrow Signal\ b$$
$$f :: SF\ T1\ T2$$

Additionally, ***causality*** required: output at time t must be determined by input on interval $[0, t]$.

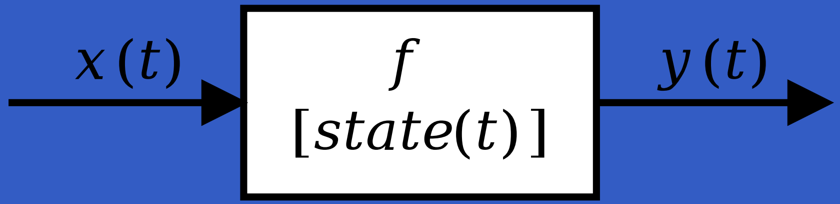
Signal Functions and State

Alternative view:

Signal Functions and State

Alternative view:

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

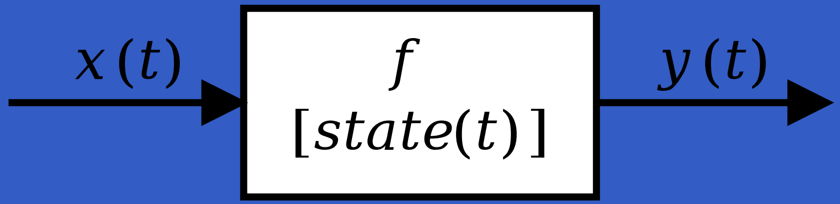


$state(t)$ summarizes input history $x(t')$, $t' \in [0, t]$.

Signal Functions and State

Alternative view:

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



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

From this perspective, signal functions are:

- ***stateful*** if $y(t)$ depends on $x(t)$ and $state(t)$
- ***stateless*** if $y(t)$ depends only on $x(t)$

Some Basic Signal Functions

$$identity :: SF\ a\ a$$

Some Basic Signal Functions

$$identity :: SF\ a\ a$$

$$constant :: b \rightarrow SF\ a\ b$$

Some Basic Signal Functions

$$identity :: SF\ a\ a$$

$$constant :: b \rightarrow SF\ a\ b$$

$$iPre :: a \rightarrow SF\ a\ a$$

Some Basic Signal Functions

$$identity :: SF\ a\ a$$

$$constant :: b \rightarrow SF\ a\ b$$

$$iPre :: a \rightarrow SF\ a\ a$$

$$integral :: VectorSpace\ a\ s \Rightarrow SF\ a\ a$$

$$y(t) = \int_0^t x(\tau)\,\mathrm{d}\tau$$

Some Basic Signal Functions

$$identity :: SF\ a\ a$$

$$constant :: b \rightarrow SF\ a\ b$$

$$iPre :: a \rightarrow SF\ a\ a$$

$$integral :: VectorSpace\ a\ s \Rightarrow SF\ a\ a$$

$$y(t) = \int_0^t x(\tau)\,\mathrm{d}\tau$$

Which are stateless and which are stateful?

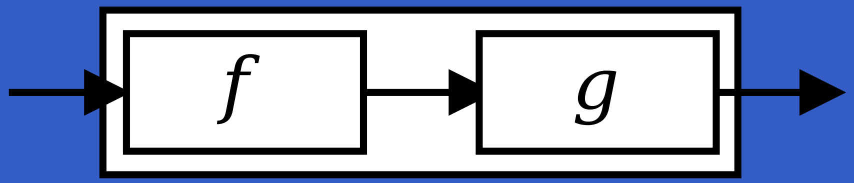
Composition

In Yampa, systems are described by combining signal functions (forming new signal functions).

Composition

In Yampa, systems are described by combining signal functions (forming new signal functions).

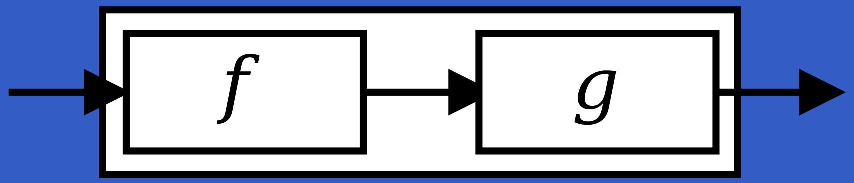
For example, serial composition:



Composition

In Yampa, systems are described by combining signal functions (forming new signal functions).

For example, serial composition:



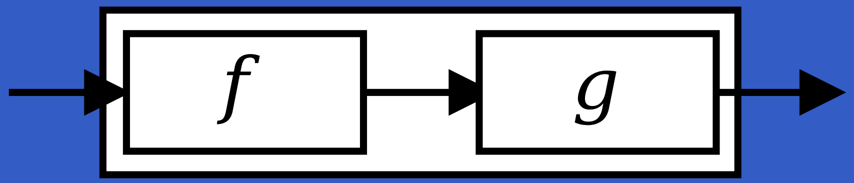
A *combinator* that captures this idea:

$$(\ggg) :: SF\ a\ b \rightarrow SF\ b\ c \rightarrow SF\ a\ c$$

Composition

In Yampa, systems are described by combining signal functions (forming new signal functions).

For example, serial composition:



A *combinator* that captures this idea:

$$(\ggg) :: SF\ a\ b \rightarrow SF\ b\ c \rightarrow SF\ a\ c$$

Signal functions are the primary notion; signals a secondary one, only existing indirectly.

Time

Quick exercise: Define time!

$$time :: SF\ a\ Time$$

Time

Quick exercise: Define time!

$$time :: SF\ a\ Time$$

$$time = constant\ 1.0 \ggg integral$$

Time

Quick exercise: Define time!

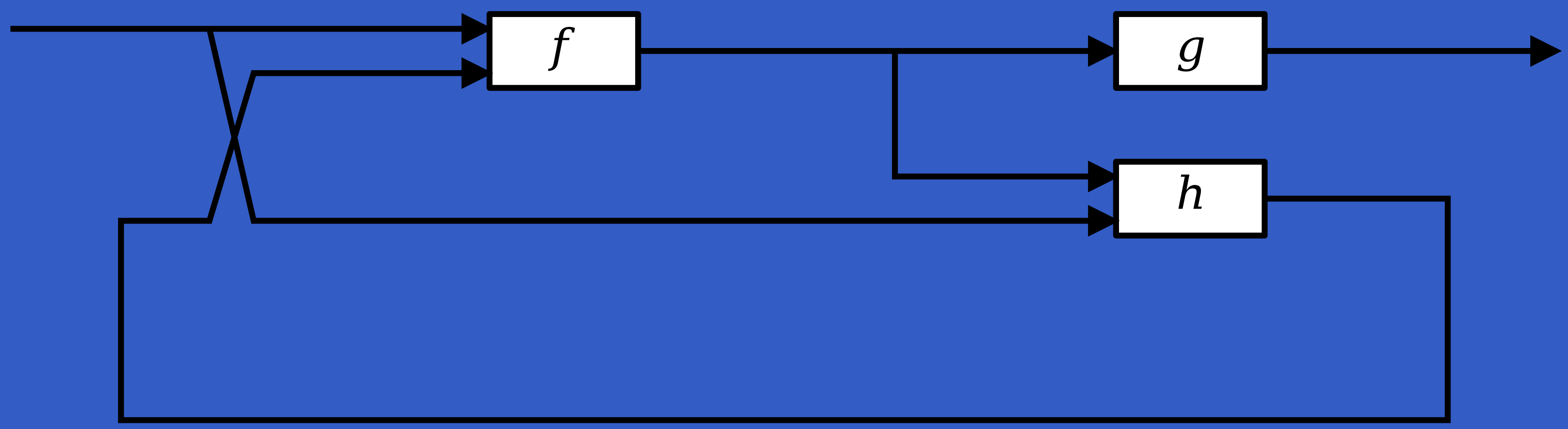
$$time :: SF\ a\ Time$$

$$time = constant\ 1.0 \ggg integral$$

Note: there is ***no*** built-in notion of global time in Yampa: time is always ***local***, measured from when a signal function started.

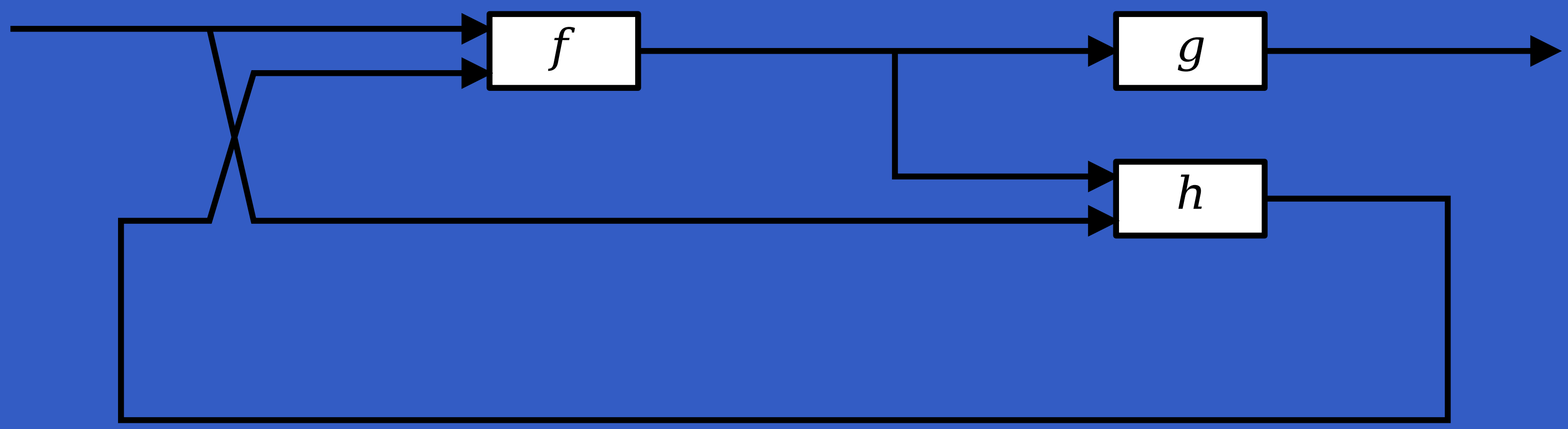
Systems

What about larger networks?
How many combinators are needed?



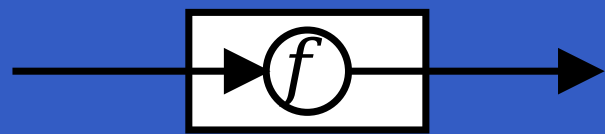
Systems

What about larger networks?
How many combinators are needed?

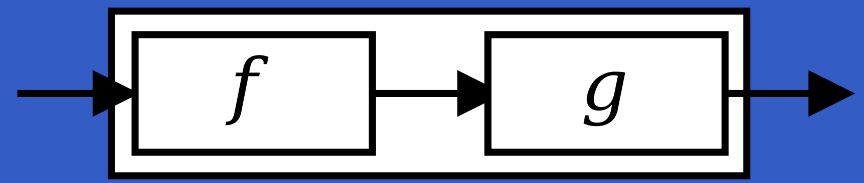


John Hughes's ***Arrow*** framework provides a good answer!

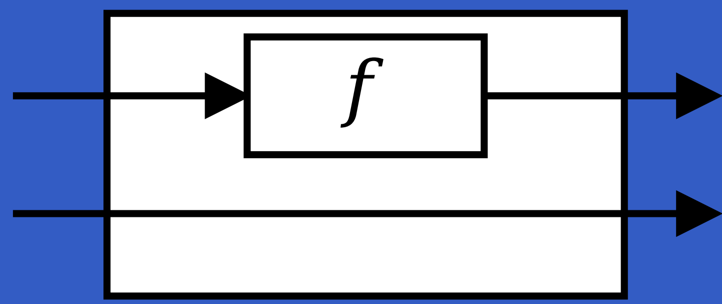
The Arrow framework (1)



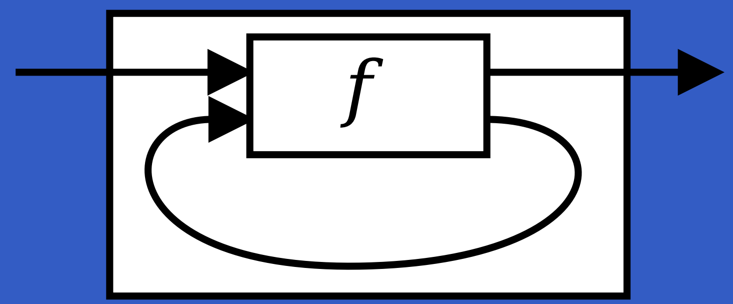
$arr\ f$



$f \ggg g$



$first\ f$



$loop\ f$

$$
\begin{aligned}
arr &:: (a \rightarrow b) \rightarrow SF\ a\ b \\
(\ggg) &:: SF\ a\ b \rightarrow SF\ b\ c \rightarrow SF\ a\ c \\
first &:: SF\ a\ b \rightarrow SF\ (a, c)\ (b, c) \\
loop &:: SF\ (a, c)\ (b, c) \rightarrow SF\ a\ b
\end{aligned}
$$

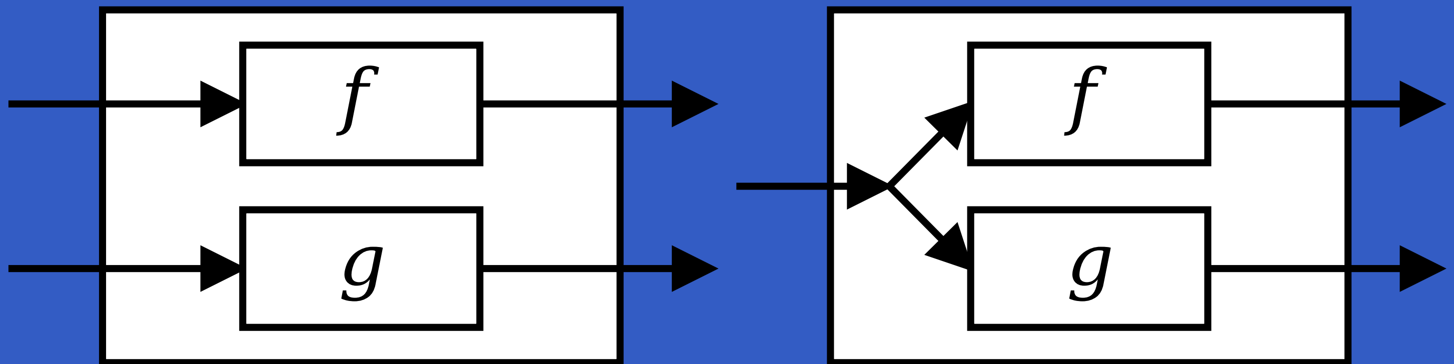
The Arrow framework (2)

Examples:

$$identity :: SF\ a\ a$$
$$identity = arr\ id$$
$$constant :: b \rightarrow SF\ a\ b$$
$$constant\ b = arr\ (const\ b)$$
$$\hat{}\ll :: (b \rightarrow c) \rightarrow SF\ a\ b \rightarrow SF\ a\ c$$
$$f\ \hat{}\ll\ sf = sf \ggg arr\ f$$

The Arrow framework (2)

Some derived combinators:

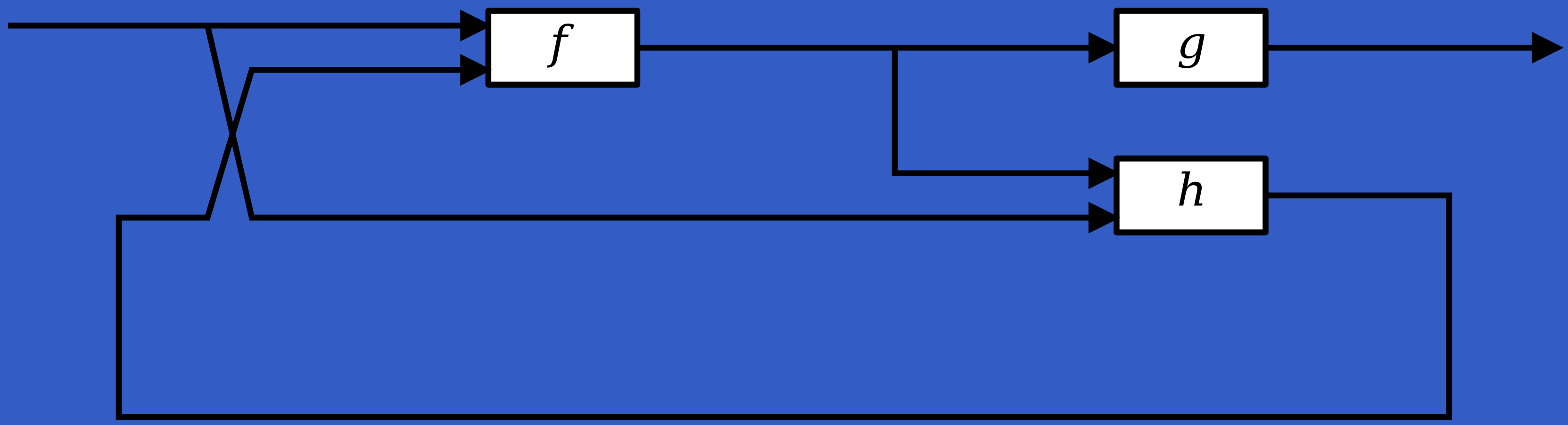


f *** g $\qquad$ f &&& g

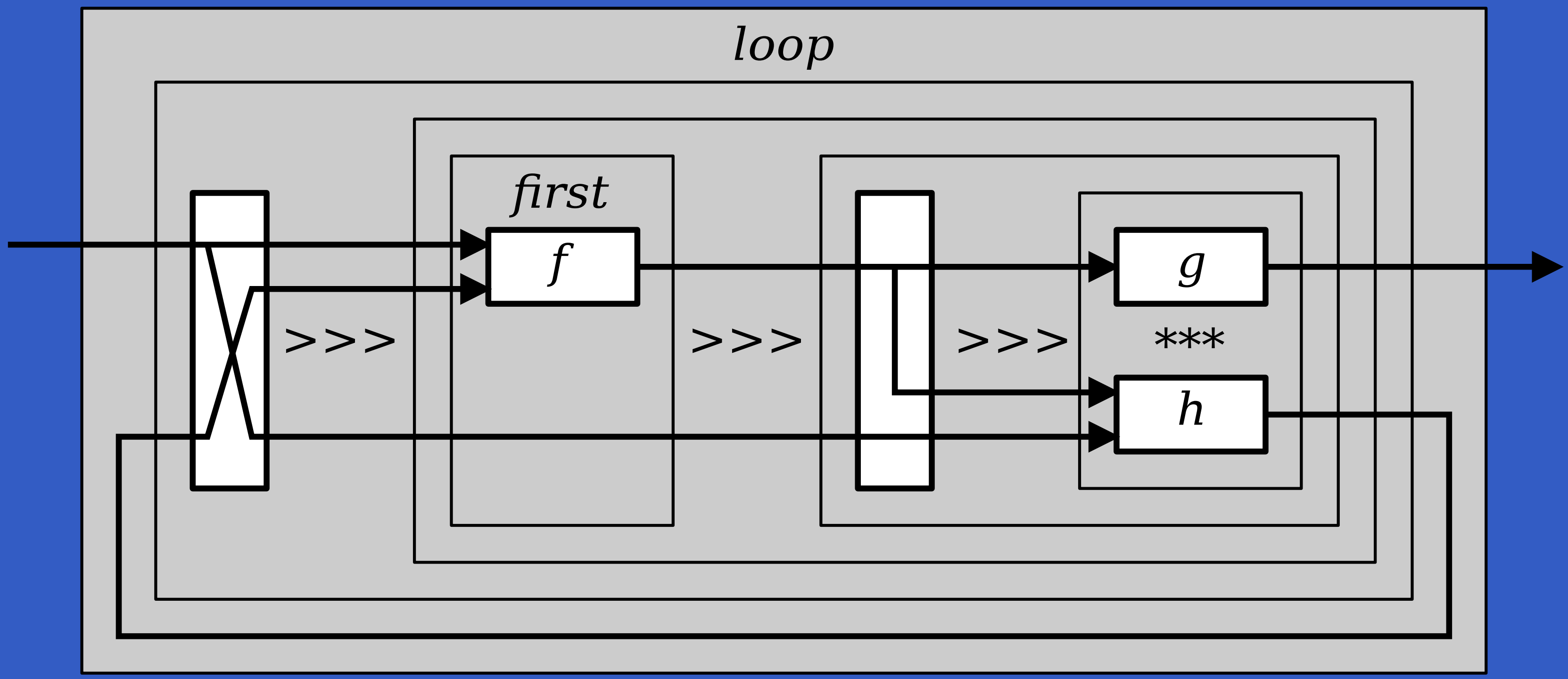
$$(\ast\!\ast\!\ast) :: SF\ a\ b \rightarrow SF\ c\ d \rightarrow SF\ (a, c)\ (b, d)$$

$$(\&\&\&) :: SF\ a\ b \rightarrow SF\ a\ c \rightarrow SF\ a\ (b, c)$$

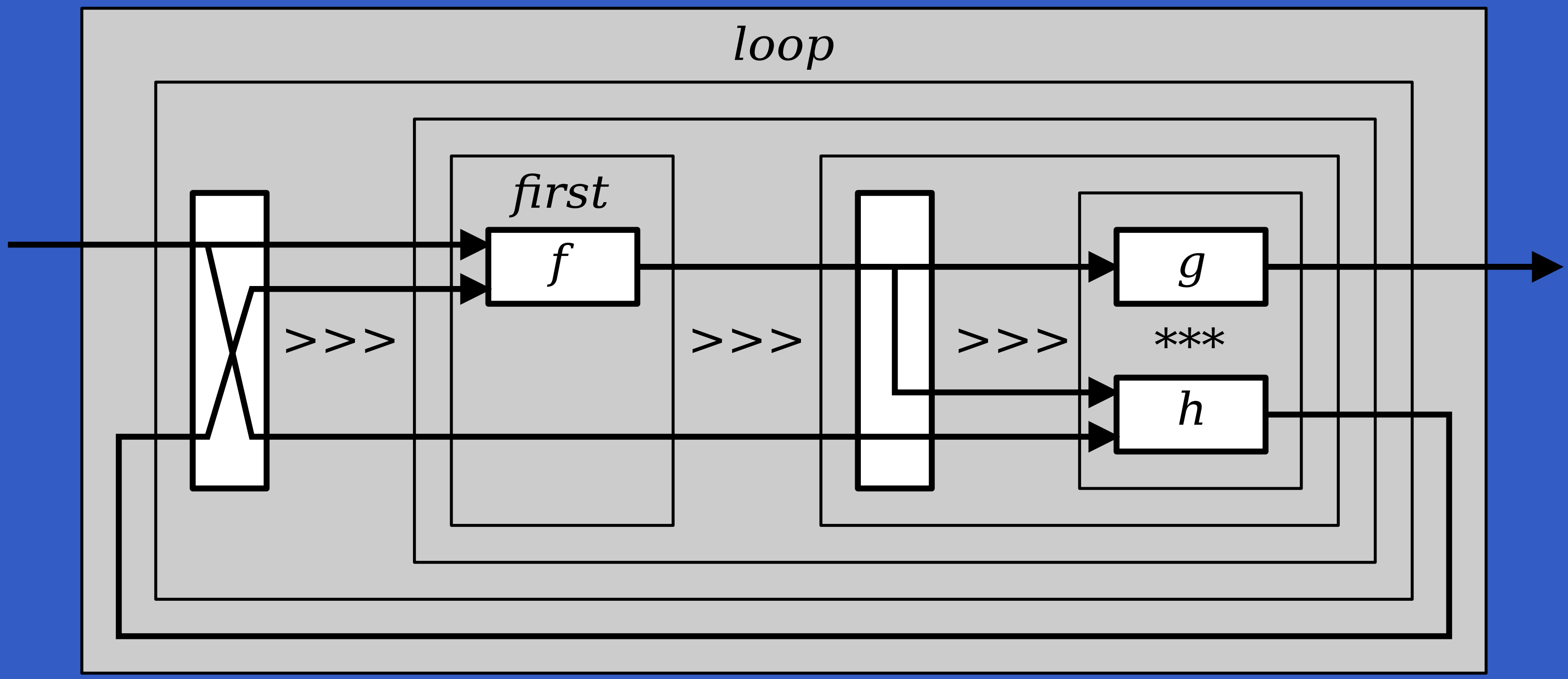
Constructing a network



Constructing a network

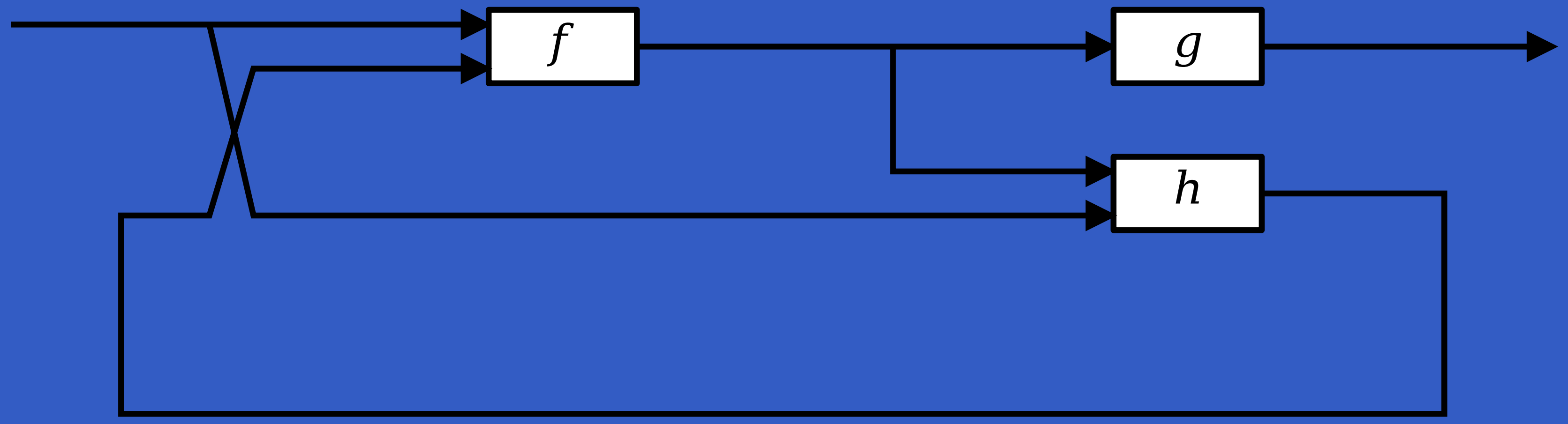


Constructing a network

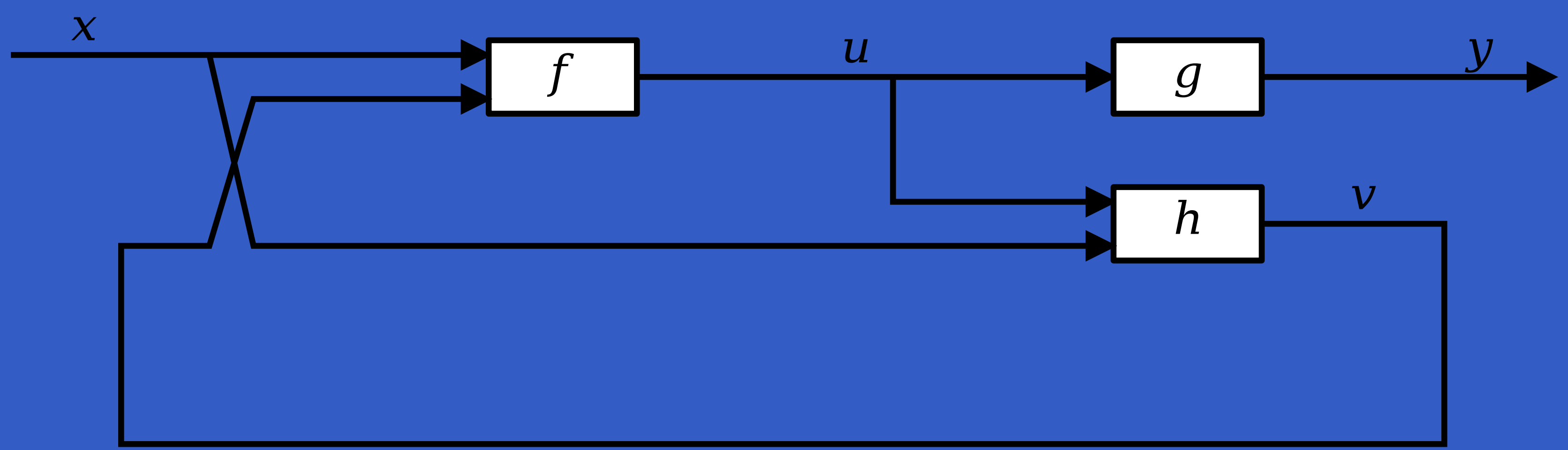


$$loop\ (arr\ (\lambda(x, y) \rightarrow ((x, y), x))$$
$$\ggg (first\ f$$
$$\ggg (arr\ (\lambda(x, y) \rightarrow (x, (x, y))) \ggg (g *** h))))$$

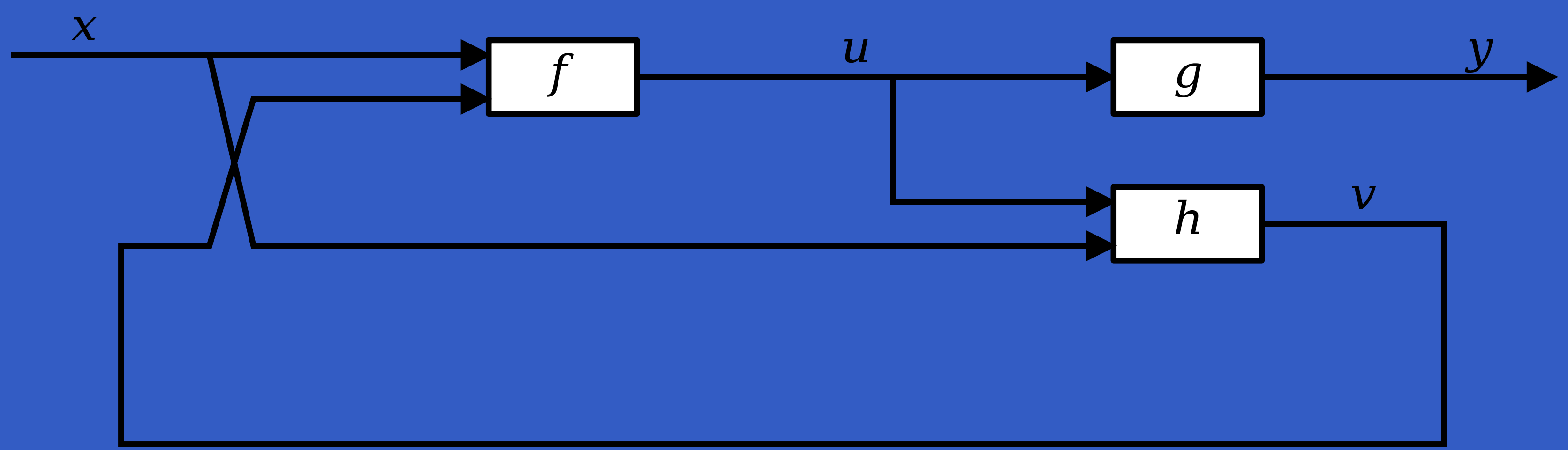
Arrow notation



Arrow notation



Arrow notation



proc $x \rightarrow$ **do**

rec

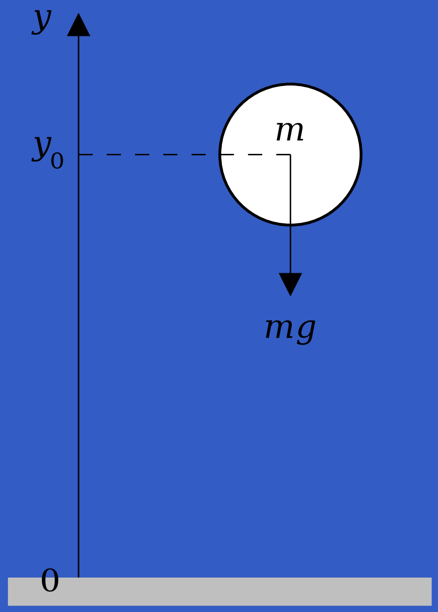
$u \leftarrow f \prec (x, v)$

$y \leftarrow g \prec u$

$v \leftarrow h \prec (u, x)$

$returnA \prec y$

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:

$$
\begin{array}{l}
\textbf{type}\ Pos = Double \\
\textbf{type}\ Vel = Double \\
fallingBall :: Pos \rightarrow Vel \rightarrow SF\ ()\ (Pos, Vel) \\
fallingBall\ y0\ v0 = \textbf{proc}\ () \rightarrow \textbf{do} \\
\quad v \leftarrow (v0+)\hat{}\ll integral \prec -9.81 \\
\quad y \leftarrow (y0+)\hat{}\ll integral \prec v \\
\quad returnA \prec (y, v)
\end{array}
$$

Discrete-time Signals or Events

Yampa's signals are conceptually ***continuous-time*** signals.

Discrete-time Signals or Events

Yampa's signals are conceptually ***continuous-time*** signals.

Discrete-time signals: signals defined at discrete points in time.

Discrete-time Signals or Events

Yampa's signals are conceptually ***continuous-time*** signals.

Discrete-time signals: signals defined at discrete points in time.

Yampa models discrete-time signals by lifting the ***co-domain*** of signals using an option-type:

$$\mathbf{data}\ Event\ a = NoEvent \mid Event\ a$$

Discrete-time Signals or Events

Yampa's signals are conceptually ***continuous-time*** signals.

Discrete-time signals: signals defined at discrete points in time.

Yampa models discrete-time signals by lifting the ***co-domain*** of signals using an option-type:

$$\mathbf{data}\ Event\ a = NoEvent \mid Event\ a$$

Discrete-time signal $=$ `Signal` (`Event` α).

Some Event Functions and Sources

$$tag :: Event\ a \rightarrow b \rightarrow Event\ b$$

$$never :: SF\ a\ (Event\ b)$$

$$now :: b \rightarrow SF\ a\ (Event\ b)$$

$$after :: Time \rightarrow b \rightarrow SF\ a\ (Event\ b)$$

$$repeatedly :: Time \rightarrow b \rightarrow SF\ a\ (Event\ b)$$

$$edge :: SF\ Bool\ (Event\ ())$$

$$notYet :: SF\ (Event\ a)\ (Event\ a)$$

$$once :: SF\ (Event\ a)\ (Event\ a)$$

Modelling the Bouncing Ball: Part 2

Detecting when the ball goes through the floor:

$$
\begin{aligned}
&\mathit{fallingBall'} :: \\
&\quad \mathit{Pos} \rightarrow \mathit{Vel} \rightarrow \mathit{SF}\ ()\ ((\mathit{Pos}, \mathit{Vel}), \mathit{Event}\ (\mathit{Pos}, \mathit{Vel})) \\
&\mathit{fallingBall'}\ \mathit{y0}\ \mathit{v0} = \mathbf{proc}\ () \rightarrow \mathbf{do} \\
&\quad \mathit{yv}@(y, \_) \leftarrow \mathit{fallingBall}\ \mathit{y0}\ \mathit{v0} \prec () \\
&\quad \mathit{hit} \qquad\quad\ \leftarrow \mathit{edge} \qquad\qquad\qquad\quad \prec y \leqslant 0 \\
&\quad \mathit{returnA} \prec (\mathit{yv}, \mathit{hit}\ `\mathit{tag}`\ \mathit{yv})
\end{aligned}
$$

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.

$$switch ::$$
$$SF\ a\ (b, Event\ c)$$
$$\rightarrow\ (c \rightarrow SF\ a\ b)$$
$$\rightarrow SF\ a\ b$$

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.

Initial SF with event source

$$switch :: \boxed{SF\ a\ (b, Event\ c)} \rightarrow (c \rightarrow SF\ a\ b) \rightarrow SF\ a\ b$$

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.

$$switch :: SF\ a\ (b, Event\ c) \rightarrow \boxed{(c \rightarrow SF\ a\ b)} \rightarrow SF\ a\ b$$

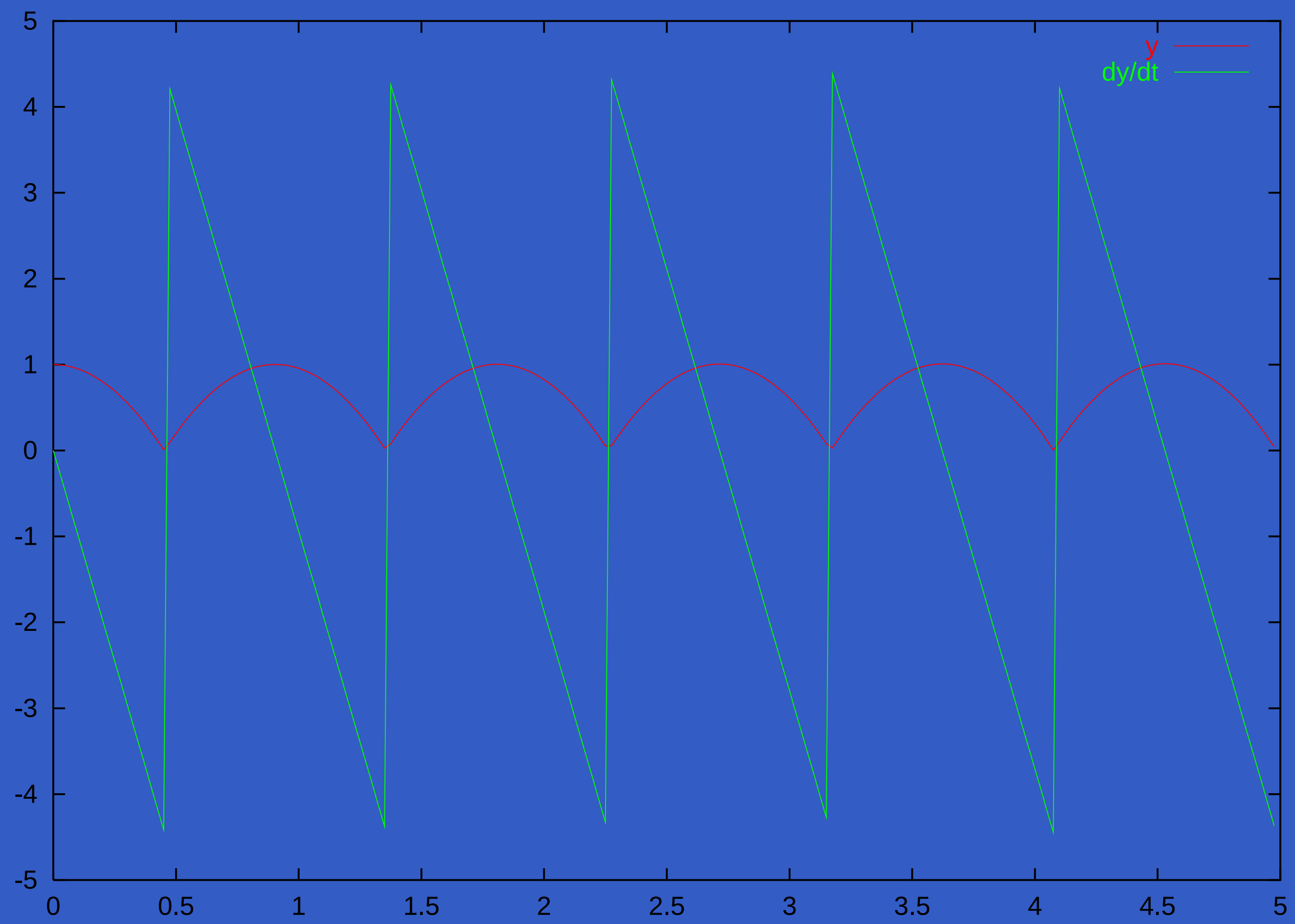
Function yielding SF to switch into

Modelling the Bouncing Ball: Part 3

Making the ball bounce:

$$
\begin{aligned}
&\mathit{bouncingBall} :: \mathit{Pos} \rightarrow \mathit{SF}\ ()\ (\mathit{Pos}, \mathit{Vel}) \\
&\mathit{bouncingBall}\ \mathit{y0} = \mathit{bbAux}\ \mathit{y0}\ 0.0 \\
&\quad \mathbf{where} \\
&\qquad \mathit{bbAux}\ \mathit{y0}\ \ \mathit{v0} = \\
&\qquad\quad \mathit{switch}\ (\mathit{fallingBall}'\ \mathit{y0}\ \ \mathit{v0})\ \$\ \lambda(y, v) \rightarrow \\
&\qquad\quad \mathit{bbAux}\ y\ (-v)
\end{aligned}
$$

Simulation of the Bouncing Ball



Modelling Using Impulses

Using a `switch` to capture the interaction between the ball and the floor may seem unnatural.

Modelling Using Impulses

Using a `switch` to capture the interaction between the ball and the floor may seem unnatural.

A more appropriate account is that an ***impulsive*** force is acting on the ball for a short time.

Modelling Using Impulses

Using a `switch` to capture the interaction between the ball and the floor may seem unnatural.

A more appropriate account is that an ***impulsive*** force is acting on the ball for a short time.

This can be abstracted into ***Dirac Impulses***: impulses that act instantaneously (Nilsson 2003).

Modelling Using Impulses

Using a `switch` to capture the interaction between the ball and the floor may seem unnatural.

A more appropriate account is that an ***impulsive*** force is acting on the ball for a short time.

This can be abstracted into ***Dirac Impulses***: impulses that act instantaneously (Nilsson 2003).

Yampa does provide a derived version of `integral` capturing the basic idea:

$$
\begin{aligned}
&\mathit{impulseIntegral} :: \\
&\quad \mathit{VectorSpace}\ a\ k \Rightarrow \\
&\qquad \mathit{SF}\ (a, \mathit{Event}\ a)\ a
\end{aligned}
$$

The Decoupled Switch

$$
\begin{aligned}
&\mathit{dSwitch} :: \\
&\quad \mathit{SF}\ a\ (b, \mathit{Event}\ c) \\
&\quad \rightarrow (c \rightarrow \mathit{SF}\ a\ b) \\
&\quad \rightarrow \mathit{SF}\ a\ b
\end{aligned}
$$

The Decoupled Switch

$$
\begin{aligned}
& dSwitch :: \\
& \quad SF\ a\ (b, Event\ c) \\
& \quad \rightarrow (c \rightarrow SF\ a\ b) \\
& \quad \rightarrow SF\ a\ b
\end{aligned}
$$

- Output at the point of switch is taken from the old subordinate signal function, ***not*** the new residual signal function.

The Decoupled Switch

$$
\begin{aligned}
&dSwitch :: \\
&\quad SF\ a\ (b, Event\ c) \\
&\quad \rightarrow (c \rightarrow SF\ a\ b) \\
&\quad \rightarrow SF\ a\ b
\end{aligned}
$$

- Output at the point of switch is taken from the old subordinate signal function, ***not*** the new residual signal function.
- ***Output*** at the current point in time thus ***independent*** of whether or not the ***switching event*** occurs at that point. Hence decoupled. Useful e.g. in some feedback scenarios.

Lots of Switches . . .

$$rSwitch, drSwitch ::$$
$$SF\ a\ b \rightarrow SF\ (a, Event\ (SF\ a\ b))\ b$$

Lots of Switches . . .

$$rSwitch, drSwitch ::$$
$$SF\ a\ b \rightarrow SF\ (a, Event\ (SF\ a\ b))\ b$$

$$kSwitch, dkSwitch ::$$
$$SF\ a\ b \rightarrow SF\ (a, b)\ (Event\ c)$$
$$\rightarrow (SF\ a\ b \rightarrow c \rightarrow SF\ a\ b) \rightarrow SF\ a\ b$$

Lots of Switches . . .

$$rSwitch, drSwitch ::$$
$$SF\ a\ b \rightarrow SF\ (a, Event\ (SF\ a\ b))\ b$$

$$kSwitch, dkSwitch ::$$
$$SF\ a\ b \rightarrow SF\ (a, b)\ (Event\ c)$$
$$\rightarrow (SF\ a\ b \rightarrow c \rightarrow SF\ a\ b) \rightarrow SF\ a\ b$$

$$pSwitch, dpSwitch, rpSwitch, drpSwitch :: \ldots$$

Lots of Switches . . .

$$rSwitch, drSwitch ::$$
$$SF\ a\ b \rightarrow SF\ (a, Event\ (SF\ a\ b))\ b$$

$$kSwitch, dkSwitch ::$$
$$SF\ a\ b \rightarrow SF\ (a, b)\ (Event\ c)$$
$$\rightarrow (SF\ a\ b \rightarrow c \rightarrow SF\ a\ b) \rightarrow SF\ a\ b$$

$$pSwitch, dpSwitch, rpSwitch, drpSwitch :: \ldots$$

However, they can ***all*** be defined in terms of $switch$ or $dSwitch$ and a notion of ***ageing*** signal functions:

$$age :: SF\ a\ b \rightarrow SF\ a\ (b, SF\ a\ b)$$

Game Objects (1)

Observable aspects of game entities:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Object} = \mathit{Object}\ \{ \\
\quad \mathit{objectName} :: \mathit{ObjectName}, \\
\quad \mathit{objectKind} \ \ :: \mathit{ObjectKind}, \\
\quad \mathit{objectPos} \quad :: \mathit{Pos2D}, \\
\quad \mathit{objectVel} \quad :: \mathit{Vel2D}, \\
\quad \mathit{objectAcc} \quad :: \mathit{Acc2D}, \\
\quad \mathit{objectDead} \ :: \mathit{Bool}, \\
\quad \mathit{objectHit} \quad :: \mathit{Bool}, \\
\qquad \ldots \\
\}
\end{array}
$$

Game Objects (2)

$$\begin{aligned}\textbf{data}\ \mathit{ObjectKind} = {} & \mathit{Ball}\ \ \mathit{Double} \\ \mid {} & \mathit{Paddle}\ \mathit{Size2D} \\ \mid {} & \mathit{Block}\ \ \mathit{Energy}\ \mathit{Size2D} \\ \mid {} & \mathit{Side}\ \ \mathit{Side}\end{aligned}$$

Game Objects (3)

$$
\begin{array}{l}
\textbf{type}\ \mathit{ObjectSF} = \mathit{SF}\ \mathit{ObjectInput}\ \mathit{ObjectOutput} \\
\textbf{data}\ \mathit{ObjectInput} = \mathit{ObjectInput}\ \{ \\
\quad \mathit{userInput} \quad\quad\ :: \mathit{Controller}, \\
\quad \mathit{collisions} \quad\quad :: [\mathit{Collision}], \\
\quad \mathit{knownObjects} :: [\mathit{Object}] \\
\} \\
\textbf{data}\ \mathit{ObjectOutput} = \mathit{ObjectOutput}\ \{ \\
\quad \mathit{outputObject} \ :: \mathit{Object}, \\
\quad \mathit{harakiri} \quad\quad\ \ :: \mathit{Event}\ () \\
\}
\end{array}
$$

Observing the Game World

- Note that $[Object]$ appears in the input type.

Observing the Game World

- Note that $[\,Object\,]$ appears in the input type.
- This allows each game object to observe ***all*** live game objects.

Observing the Game World

- Note that $[\mathit{Object}]$ appears in the input type.
- This allows each game object to observe ***all*** live game objects.
- Similarly, $[\mathit{Collision}]$ allows interactions ***between*** game objects to be observed.

Observing the Game World

- Note that $[Object]$ appears in the input type.
- This allows each game object to observe ***all*** live game objects.
- Similarly, $[Collision]$ allows interactions ***between*** game objects to be observed.
- Typically achieved through delayed feedback to ensure the feedback is well-defined:

$$
\begin{array}{l}
loopPre :: c \rightarrow SF\ (a, c)\ (b, c) \rightarrow SF\ a\ b \\
loopPre\ c\_init\ sf = \\
\quad loop\ (second\ (iPre\ c\_init) \ggg sf)
\end{array}
$$

Paddle, Take 1

$$
\begin{array}{l}
objPaddle :: ObjectSF \\
objPaddle = \mathbf{proc}\ (ObjectInput\ ci\ cs\ os) \rightarrow \mathbf{do} \\
\quad \mathbf{let}\ name = \texttt{"paddle"} \\
\quad \mathbf{let}\ isHit \ = inCollision\ name\ cs \\
\quad \mathbf{let}\ p \qquad = refPosPaddle\ ci \\
\quad v \leftarrow derivative \prec p \\
\quad returnA \prec livingObject\ \$\ Object\ \{ \\
\qquad objectName = name, \\
\qquad objectPos = p, \\
\qquad objectVel \ = v, \\
\qquad \ldots \}
\end{array}
$$

Paddle, Take 2

$$
\begin{array}{l}
objPaddle :: ObjectSF \\
objPaddle = \mathbf{proc}\ (ObjectInput\ ci\ cs\ os) \rightarrow \mathbf{do} \\
\quad \mathbf{let}\ name = \texttt{"paddle"} \\
\quad \mathbf{let}\ isHit\ = inCollision\ name\ cs \\
\quad \mathbf{rec} \\
\qquad \mathbf{let}\ v = limitNorm\ (20.0 *\hat{}\ (refPosPaddle\ ci \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \hat{}-\hat{}\ p)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad maxVNorm \\
\qquad p \leftarrow (initPosPaddle\ \hat{}+\hat{}\ )\hat{}\ll integral \prec v \\
\quad returnA \prec livingObject\ \$\ Object\ \{\ldots\}
\end{array}
$$

Ball, Take 1

$$objBall :: ObjectSF$$
$$objBall =$$
$$\quad switch\ followPaddleDetectLaunch\ \$\ \lambda p \rightarrow$$
$$\quad objBall$$

$$followPaddleDetectLaunch = \textbf{proc}\ oi \rightarrow \textbf{do}$$
$$\quad o \leftarrow followPaddle \prec oi$$
$$\quad click \leftarrow edge \prec controllerClick\ (userInput\ oi)$$
$$\quad returnA \prec (o, click\ `tag`\ (objectPos\ (outputObject\ o)))$$

Ball, Take 2

$$objBall :: ObjectSF$$
$$objBall =$$
$$\quad switch\ followPaddleDetectLaunch\ \$\ \lambda p \rightarrow$$
$$\quad switch\ (freeBall\ p\ initBallVel \&\&\& never)\ \$\ \lambda\_ \rightarrow$$
$$\quad objBall$$

$$freeBall\ p0\ v0 = \mathbf{proc}\ (ObjectInput\ ci\ cs\ os) \rightarrow \mathbf{do}$$
$$\quad p \leftarrow (p0\ \hat{}+\hat{}\ )\hat{}\ll integral \prec v0'$$
$$\quad returnA \prec livingObject\ \$\ \{\ldots\}$$
$$\quad \mathbf{where}$$
$$\qquad v0' = limitNorm\ v0\ maxVNorm$$

Ball, Take 3

$$
\begin{array}{l}
objBall :: ObjectSF \\
objBall = \\
\quad switch\ followPaddleDetectLaunch \quad \$\ \lambda p \rightarrow \\
\quad switch\ (bounceAroundDetectMiss\ p)\ \$\ \lambda\_ \rightarrow \\
\quad objBall
\end{array}
$$

$$
\begin{array}{l}
bounceAroundDetectMiss\ p = \mathbf{proc}\ oi \rightarrow \mathbf{do} \\
\quad o \quad\ \leftarrow bouncingBall\ p\ initBallVel \prec oi \\
\quad miss \leftarrow collisionWithBottom \qquad \prec collisions\ oi \\
\quad returnA \prec (o, miss)
\end{array}
$$

Making the Ball Bounce

$bouncingBall\ p0\ v0 =$
$\quad switch\ (moveFreelyDetBounce\ p0\ v0)\ \$\ \lambda(p', v') \rightarrow$
$\quad bouncingBall\ p'\ v'$

$moveFreelyDetBounce\ p0\ v0 =$
$\quad \mathbf{proc}\ oi@(ObjectInput\ \_\ cs\ \_) \rightarrow \mathbf{do}$
$\qquad o \leftarrow freeBall\ p0\ v0 \prec oi$
$\qquad ev \leftarrow edgeJust \lll initially\ Nothing$
$\qquad\qquad \prec changedVelocity\ \texttt{"ball"}\ cs$
$\quad returnA \prec (o, fmap\ (\lambda v \rightarrow (objectPos\ (\ldots o), v))$
$\qquad\qquad ev)$

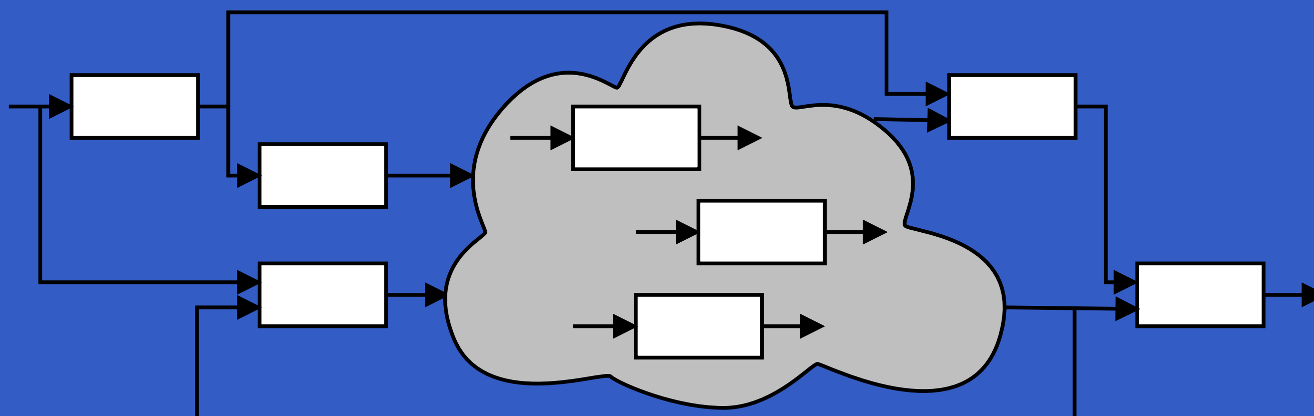
Highly dynamic system structure?

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?

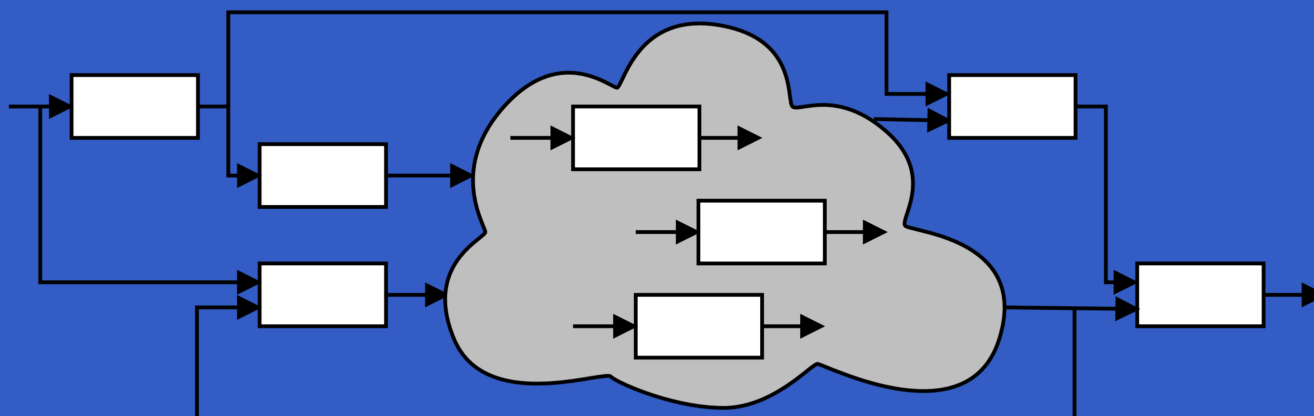


We want blocks to disappear!

Highly dynamic system structure?

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

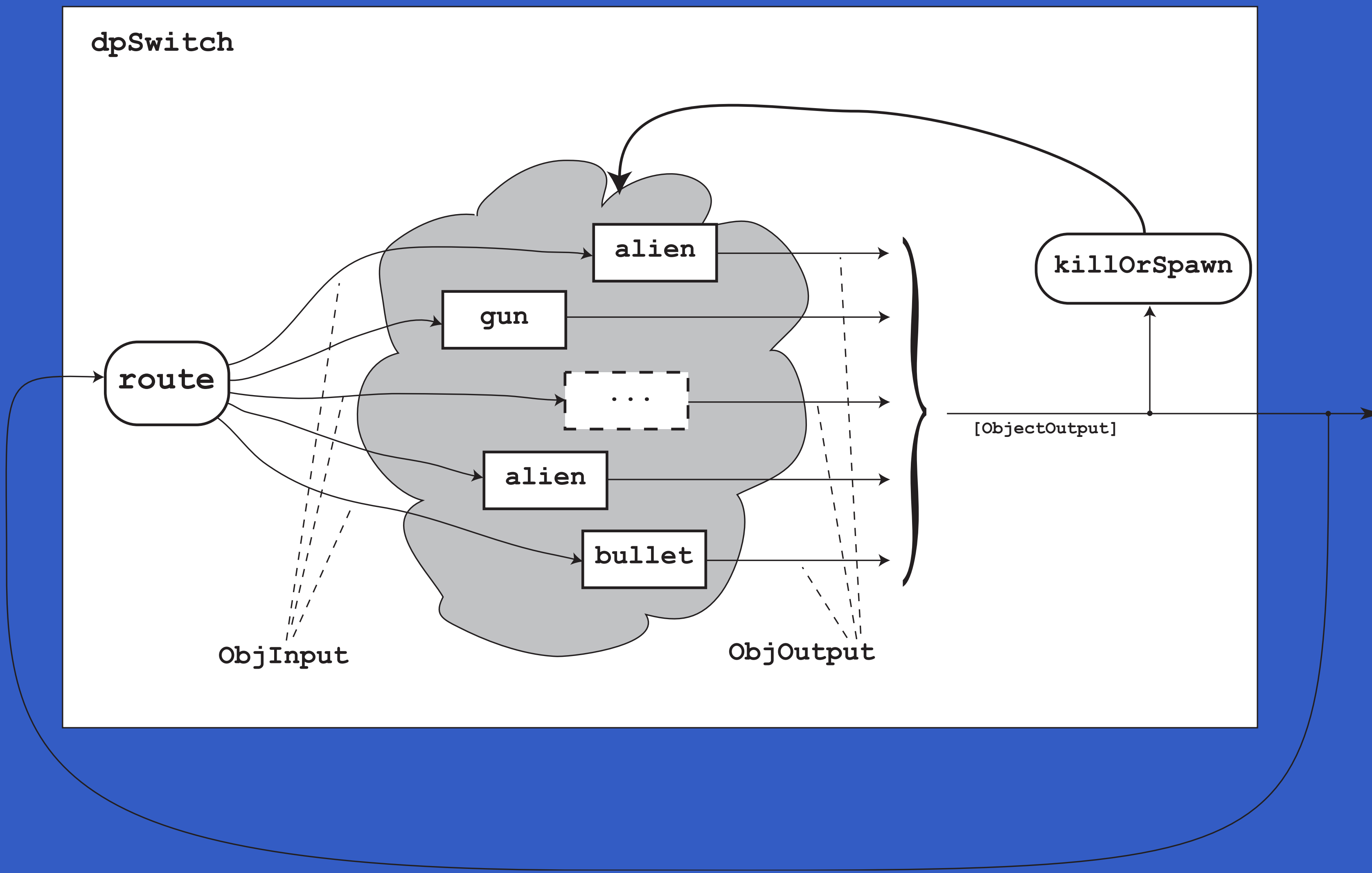
- What about more general structural changes?



We want blocks to disappear!

- What about state?

Typical 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)
   -> 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

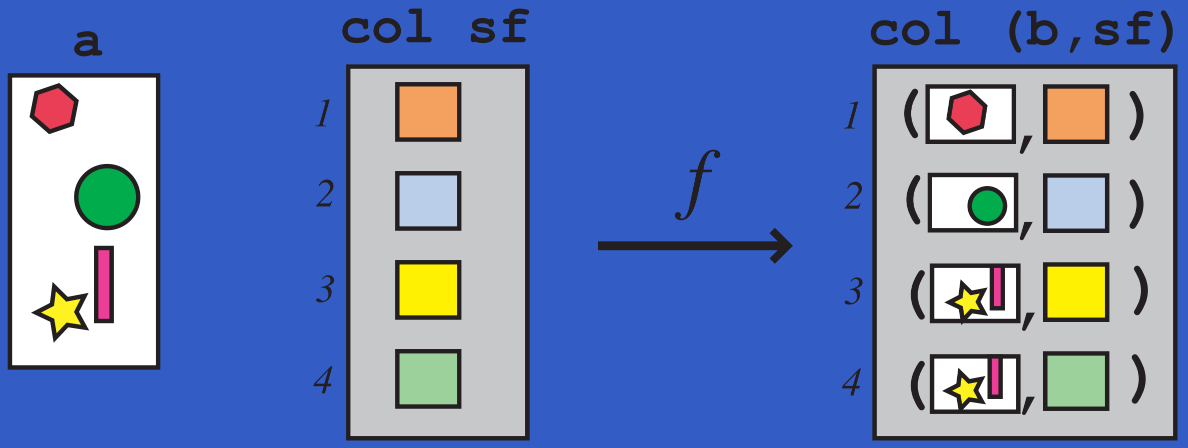
Idea:

- 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:

$$\mathit{Functor}\ \mathit{col} \Rightarrow$$
$$(\mathit{forall}\ \mathit{sf} \circ (a \rightarrow \mathit{col}\ \mathit{sf} \rightarrow \mathit{col}\ (b, \mathit{sf})))$$

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.

Blocks

$$
\begin{array}{l}
objBlockAt\ (x, y)\ (w, h) = \\
\quad \mathbf{proc}\ (ObjectInput\ ci\ cs\ os) \rightarrow \mathbf{do} \\
\quad\quad \mathbf{let}\ name = \texttt{"blockat"} \mathbin{+\!\!+} show\ (x, y) \\
\quad\quad\quad isHit \quad = inCollision\ name\ cs \\
\quad\quad hit \leftarrow edge \qquad\qquad \prec isHit \\
\quad\quad lives \leftarrow accumHoldBy\ (+)\ 3 \prec (hit\ `tag`\ (-1)) \\
\quad\quad \mathbf{let}\ isDead = lives \leqslant 0 \\
\quad\quad dead \leftarrow edge \prec isDead \\
\quad\quad returnA \prec ObjectOutput \\
\quad\quad\quad (Object\ \{\ldots\}) \\
\quad\quad\quad dead
\end{array}
$$

The Game Core

$$
\begin{array}{l}
processMovement :: \\
\quad [\,ObjectSF\,] \to SF\ ObjectInput\ (IL\ ObjectOutput) \\
processMovement\ objs = \\
\quad dpSwitchB\ objs \\
\qquad\qquad (noEvent \longrightarrow arr\ suicidalSect) \\
\qquad\qquad (\lambda sfs'\ f \to processMovement'\ (f\ sfs'))
\end{array}
$$

$$
\begin{array}{l}
loopPre\ ([\,],[\,],0)\ \$ \\
\quad adaptInput \\
\quad \ggg processMovement\ objs \\
\quad \ggg (arr\ elemsIL \&\&\& detectCollisions)
\end{array}
$$

Recovering Blocks

$$
\begin{array}{l}
objBlockAt\ (x, y)\ (w, h) = \\
\quad \mathbf{proc}\ (ObjectInput\ ci\ cs\ os) \rightarrow \mathbf{do} \\
\quad\quad \mathbf{let}\ name = \texttt{"blockat"} +\!\!+\ show\ (x, y) \\
\quad\quad\quad isHit \quad = inCollision\ name\ cs \\
\quad\quad hit \quad\quad \leftarrow edge \quad\quad\quad\quad \prec isHit \\
\quad\quad recover \leftarrow delayEvent\ 5.0 \prec hit \\
\quad\quad lives \quad\ \leftarrow accumHoldBy\ (+)\ 3 \\
\quad\quad\quad\quad\quad\quad \prec (hit\ `tag`\ (-1) \\
\quad\quad\quad\quad\quad\quad\quad\quad `lMerge`\ recover\ `tag`\ 1) \\
\quad\quad \ldots
\end{array}
$$