COMP4075/G54RFP: Lecture 17

Arrows, FRP, and Games

Henrik Nilsson

University of Nottingham, UK

Program Games Declaratively?

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

Program Games Declaratively?

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

Perhaps not so surprising:

Program Games Declaratively?

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

Perhaps not so surprising:

- Many pragmatical reasons: performance, legacy issues, . . .

Program Games Declaratively?

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

Perhaps not so surprising:

- Many pragmatical reasons: performance, legacy issues, . . .
- State and effects are pervasive in video games: Is declarative programming even a conceptually good fit?

But Why **NOT**, Really?

Many eloquent and compelling cases for functional programming in general:

But Why NOT, Really?

Many eloquent and compelling cases for functional programming in general:

- John Backus, 1977 ACM Turing Award Lecture: Can Programming Be Liberated from the von Neumann Style?

But Why **NOT**, Really?

Many eloquent and compelling cases for functional programming in general:

- John Backus, 1977 ACM Turing Award Lecture: Can Programming Be Liberated from the von Neumann Style?
- John Hughes, recent retrospective: Why Functional Programming Matters (on YouTube, recommended)

But Why **NOT**, Really?

Many eloquent and compelling cases for functional programming in general:

- John Backus, 1977 ACM Turing Award Lecture: Can Programming Be Liberated from the von Neumann Style?
- John Hughes, recent retrospective: Why Functional Programming Matters (on YouTube, recommended)

One key point: Program with whole values, not a word-at-a-time. (Will come back to this.)

Possible Gains

High profile people in the games industry have pointed out potential benefits:

Possible Gains

High profile people in the games industry have pointed out potential benefits:

- John D. Carmack, id Software:
  Wolfenstein 3D, Doom, Quake

Possible Gains

High profile people in the games industry have pointed out potential benefits:

- John D. Carmack, id Software:
  Wolfenstein 3D, Doom, Quake
- Tim Sweeney, Epic Games:
  The Unreal Engine

Possible Gains

High profile people in the games industry have pointed out potential benefits:

- John D. Carmack, id Software:
  Wolfenstein 3D, Doom, Quake
- Tim Sweeney, Epic Games:
  The Unreal Engine

E.g. pure, declarative code:

- promotes parallelism
- eliminates many sources of errors

“Whole Values” for Games?

How should we go about writing video games “declaratively”?

“Whole Values” for Games?

How should we go about writing video games “declaratively”?

In particular, what should those “whole values” be?

“Whole Values” for Games?

How should we go about writing video games “declaratively”?

In particular, what should those “whole values” be?

- Could be conventional entities like vectors, arrays, lists and aggregates of such.

“Whole Values” for Games?

How should we go about writing video games “declaratively”?

In particular, what should those “whole values” be?

- Could be conventional entities like vectors, arrays, lists and aggregates of such.
- Could even be things like pictures.

“Whole Values” for Games?

How should we go about writing video games “declaratively”?

In particular, what should those “whole values” be?

- Could be conventional entities like vectors, arrays, lists and aggregates of such.
- Could even be things like pictures.

But we are going to go one step further and consider programming with ***time-varying entities***.

Take-home Message # 1

Take-home Message # 1

Video games can be programmed declaratively by describing ***what*** entities are ***over*** time.

Take-home Message # 1

Video games can be programmed declaratively by describing ***what*** entities are ***over*** time.

Our whole values are things like:

- The totality of input from the player
- The animated graphics output
- The entire life of a game object

Take-home Message # 1

Video games can be programmed declaratively by describing ***what*** entities are ***over*** time.

Our whole values are things like:

- The totality of input from the player
- The animated graphics output
- The entire life of a game object

We construct and work with ***pure*** functions on these:

- The game: function from input to animation
- In the game: fixed point of function on collection of game objects

Take-home Message # 1 (cont.)

- That said, 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.
- See Perez & Nilsson (2015) for one approach.

Take-home Message # 2

You too can program games declaratively . . .

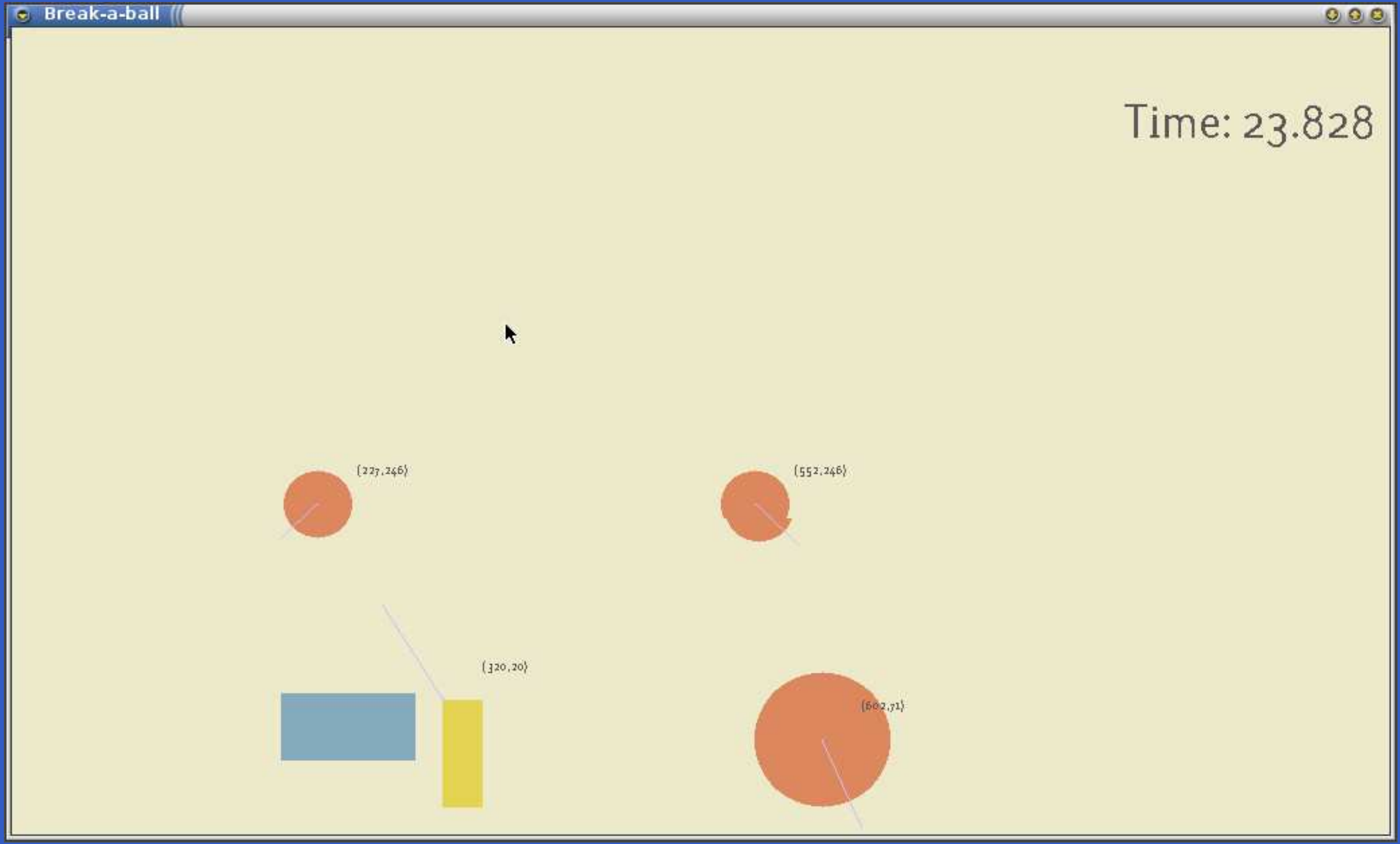
Take-home Message # 2

You too can program games declaratively . . . today!

Play Store: Keera Breakout (Keera Studios)

Take-home Game!

Or download one for free to your Android device!



Play Store: Pang-a-lambda (Keera Studios)

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

Functional Reactive Programming

- Key idea: Don’t program one-time-step-at-a-time, but describe an evolving entity as whole.

Functional Reactive Programming

- Key idea: Don't program one-time-step-at-a-time, but describe an evolving entity as whole.
- FRP originated in Conal Elliott and Paul Hudak's work on Functional Reactive Animation (Fran).

Functional Reactive Programming

- Key idea: Don't program one-time-step-at-a-time, but describe an evolving entity as whole.
- FRP originated in Conal Elliott and Paul Hudak's work on Functional Reactive Animation (Fran). Highly cited 1997 ICFP paper; ICFP award for most influential paper in 2007.

Functional Reactive Programming

- Key idea: Don't program one-time-step-at-a-time, but describe an evolving entity as whole.
- FRP originated in Conal Elliott and Paul Hudak's work on Functional Reactive Animation (Fran). Highly cited 1997 ICFP paper; ICFP award for most influential paper in 2007.
- FRP has evolved in a number of directions and into different concrete implementations.

Functional Reactive Programming

- Key idea: Don't program one-time-step-at-a-time, but describe an evolving entity as whole.
- FRP originated in Conal Elliott and Paul Hudak's work on Functional Reactive Animation (Fran). Highly cited 1997 ICFP paper; ICFP award for most influential paper in 2007.
- FRP has evolved in a number of directions and into different concrete implementations.
- We will use Yampa: an arrows-based FRP system embedded in Haskell.

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

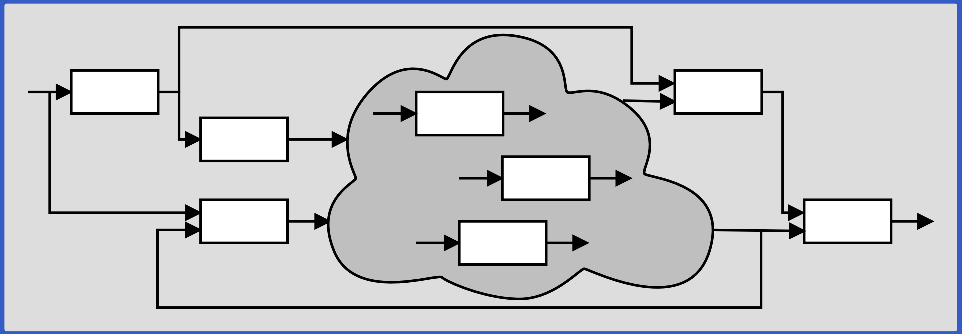
- FRP implemenattion embedded in Haskell

Yampa

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

Yampa

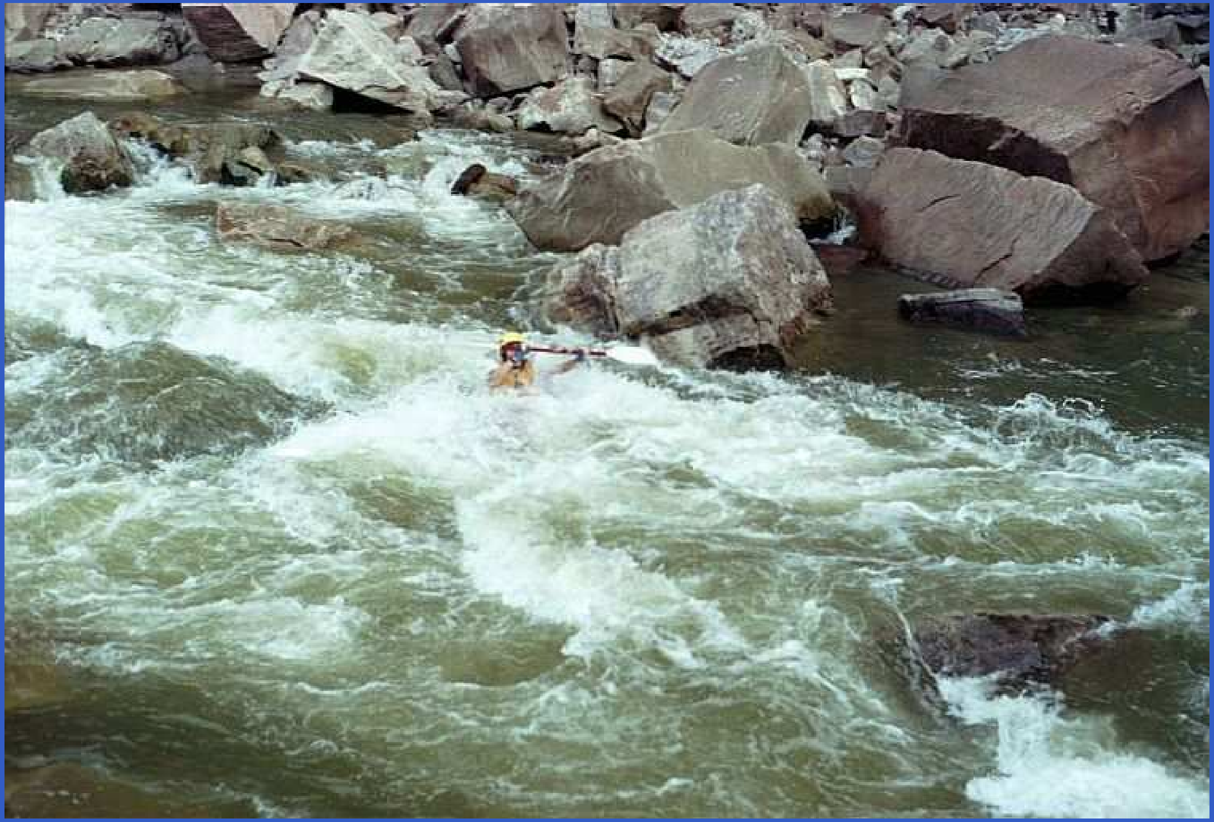
- FRP implemenattion embedded in Haskell
- Key concepts:
  - ***Signals***: time-varying values
  - ***Signal Functions***: functions on signals
  - ***Switching*** between signal functions
- Programming model:



Yampa?

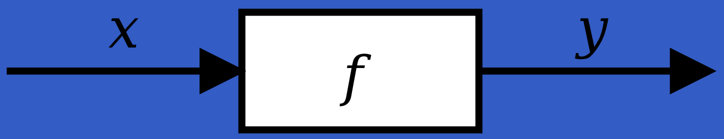
Yampa?

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

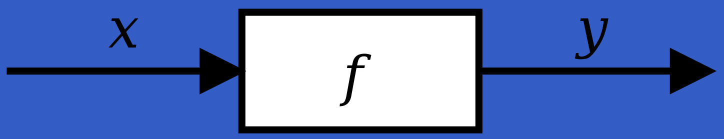


A good metaphor for hybrid systems!

Signal Functions

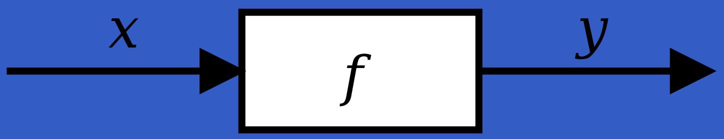


Signal Functions



Intuition:

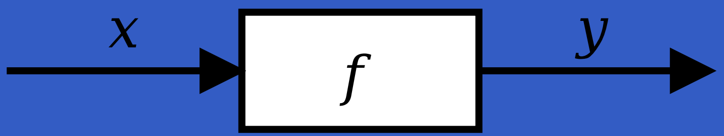
Signal Functions



Intuition:

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

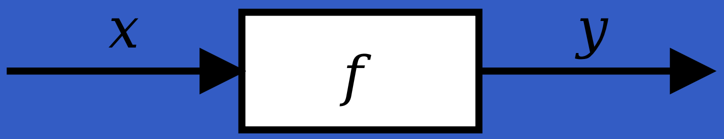
Signal Functions



Intuition:

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

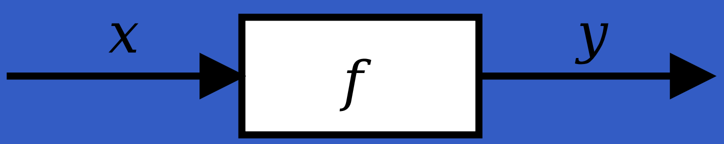
Signal Functions



Intuition:

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

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 $[0, t]$.

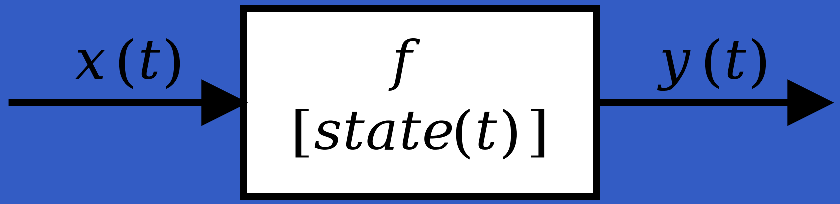
Signal Functions and State

Alternative view:

Signal Functions and State

Alternative view:

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

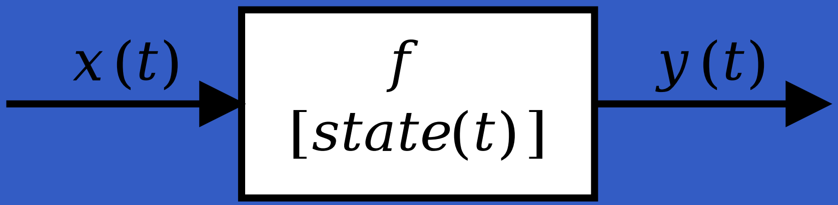


$state(t)$ summarizes input history $x(t')$, $t' \in [0, t]$.

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?

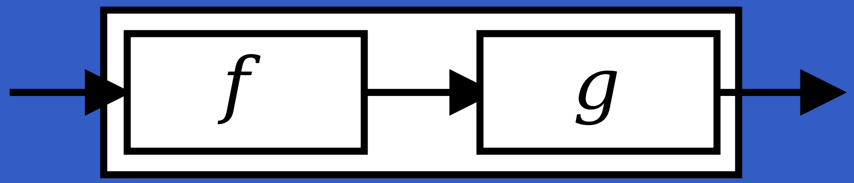
Composition

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).

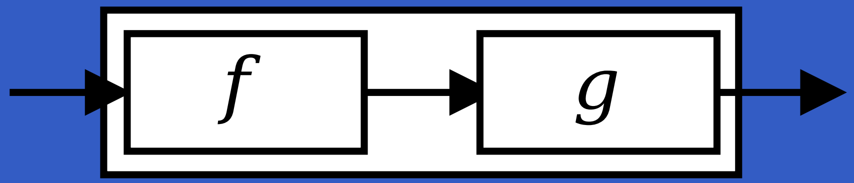
For example, serial composition:



Composition

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

For example, serial composition:



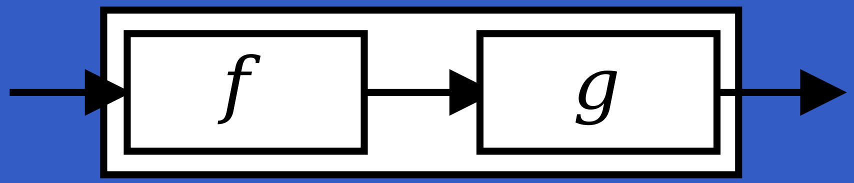
A *combinator* that captures this idea:

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

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!

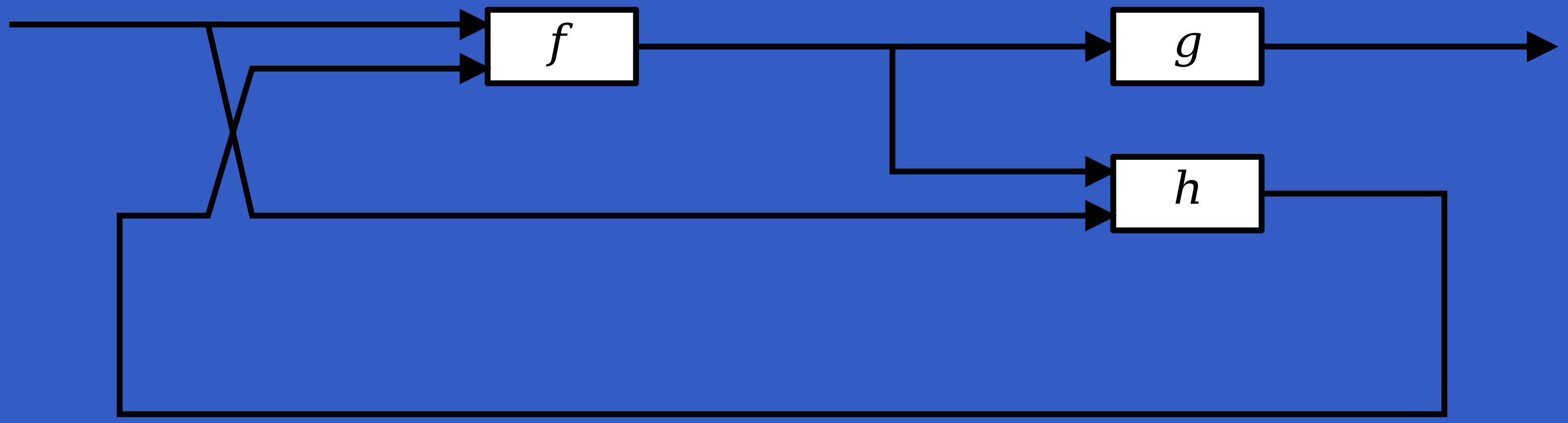
$$time :: SF \;\; a \;\; Time$$

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

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

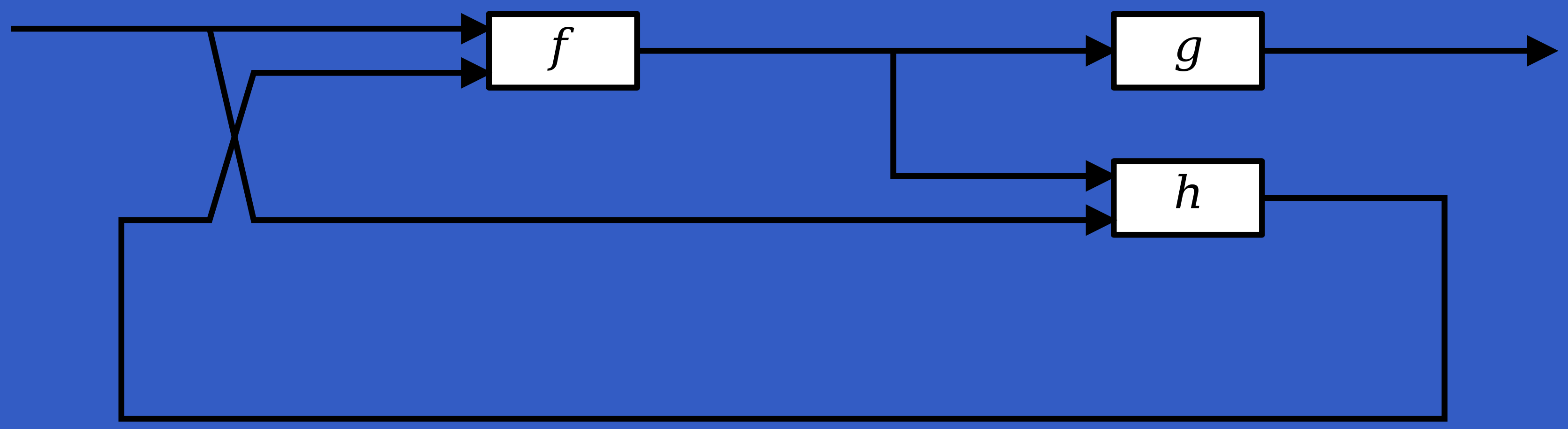
Systems

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)

John Hughes' ***arrow*** framework:

- Abstract data type interface for ***function-like objects*** (or "blocks") with ***effects***.

The Arrow framework (1)

John Hughes’ ***arrow*** framework:

- Abstract data type interface for ***function-like objects*** (or “blocks”) with ***effects***.
- Particularly suitable for types representing ***process-like computations***.

The Arrow framework (1)

John Hughes' ***arrow*** framework:

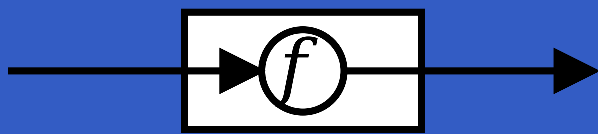
- Abstract data type interface for ***function-like objects*** (or "blocks") with ***effects***.
- Particularly suitable for types representing ***process-like computations***.
- Related to ***monads***, since arrows are computations, but more general.

The Arrow framework (1)

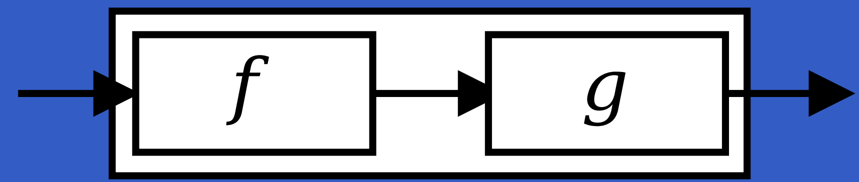
John Hughes' ***arrow*** framework:

- Abstract data type interface for ***function-like objects*** (or "blocks") with ***effects***.
- Particularly suitable for types representing ***process-like computations***.
- Related to ***monads***, since arrows are computations, but more general.
- Provides a minimal set of "wiring" combinators.

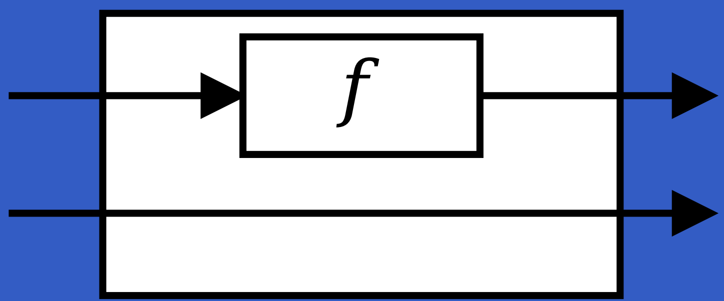
The Arrow framework (2)



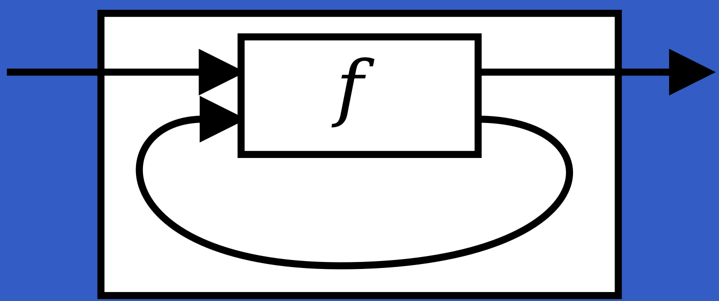
$arr\ f$



$f \ggg g$



$first\ f$



$loop\ f$

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

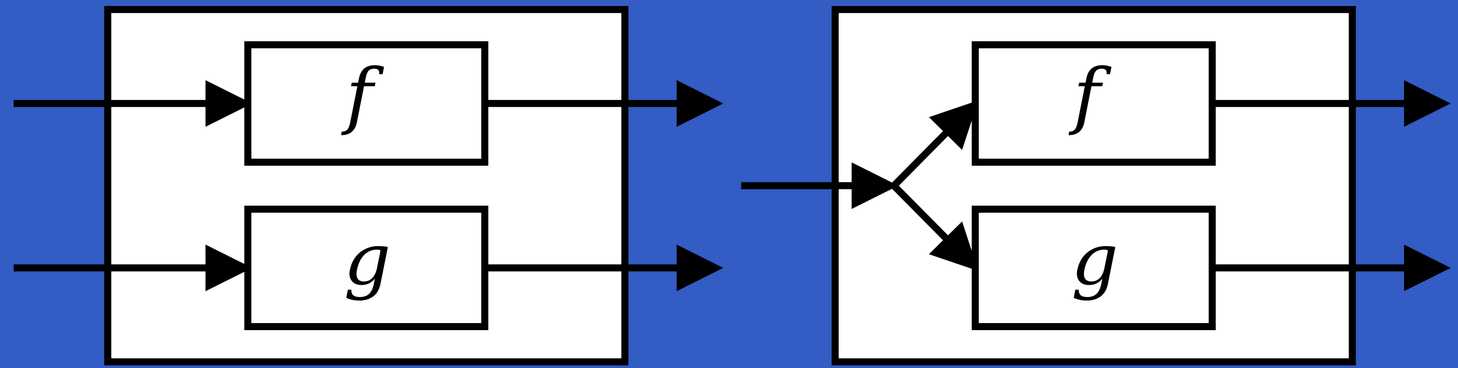
The Arrow framework (3)

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 (4)

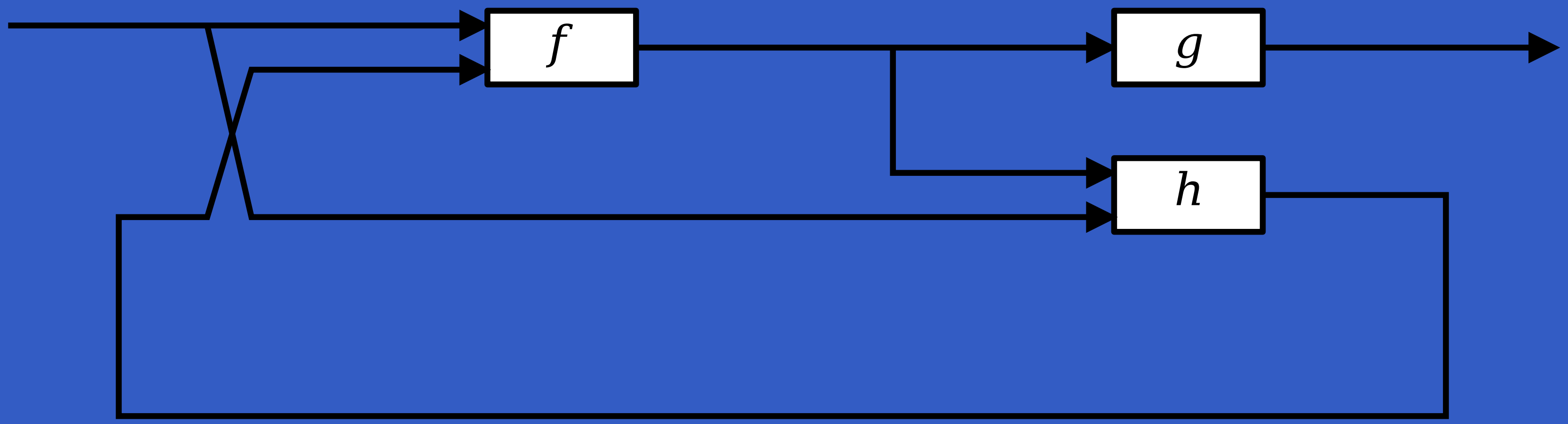
Some derived combinators:



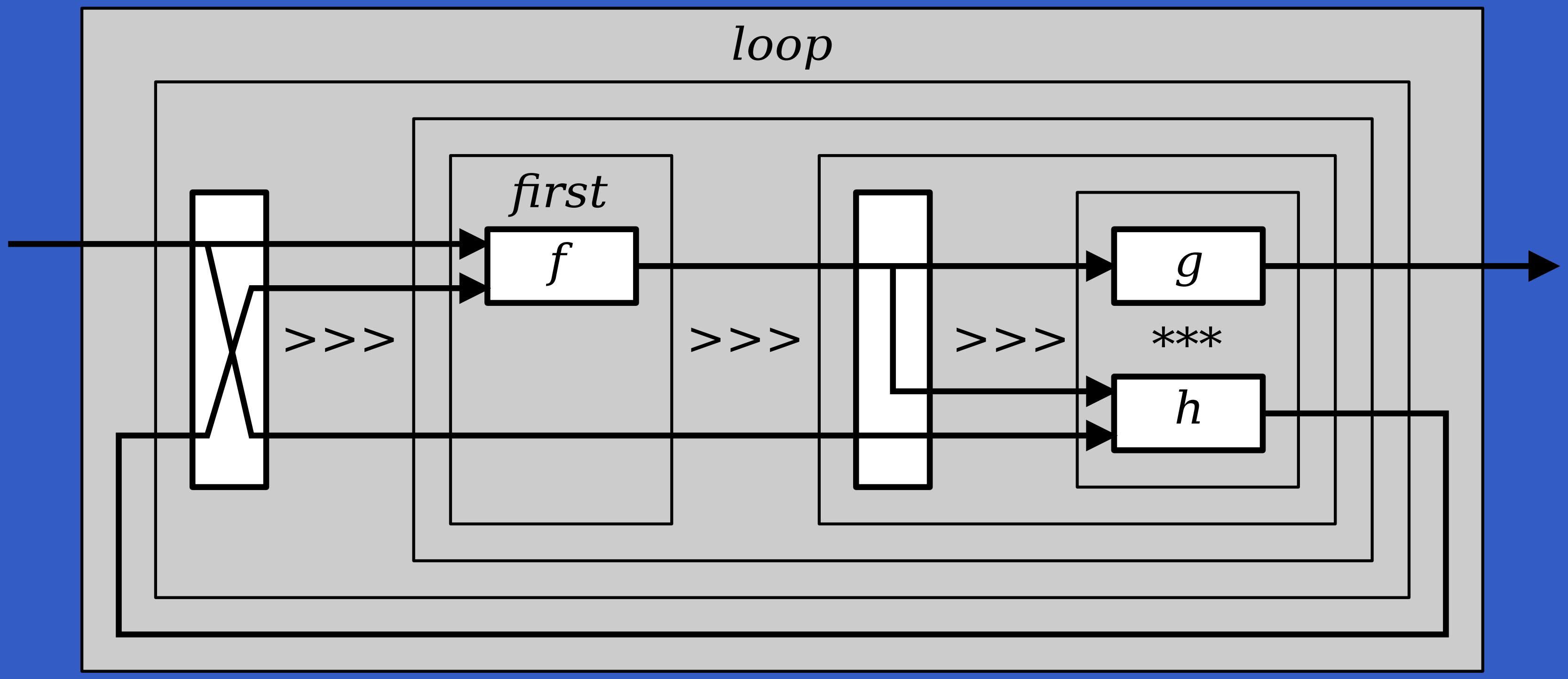
$f \ggg g$ $f \,\&\&\&\, g$

$$(***) \;::\; SF\ a\ b \to SF\ c\ d \to SF\ (a, c)\ (b, d)$$
$$(\&\&\&) \;::\; SF\ a\ b \to SF\ a\ c \to SF\ a\ (b, c)$$

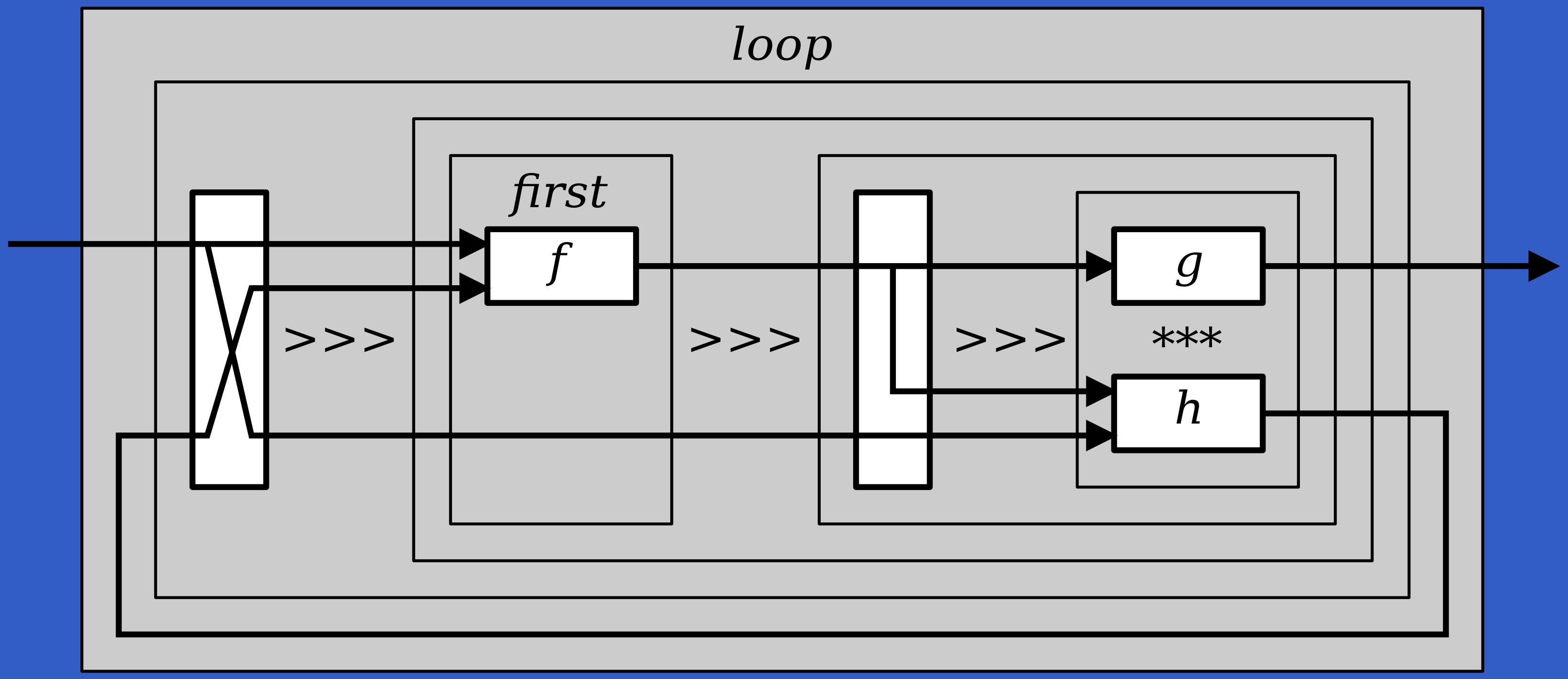
Constructing a network



Constructing a network

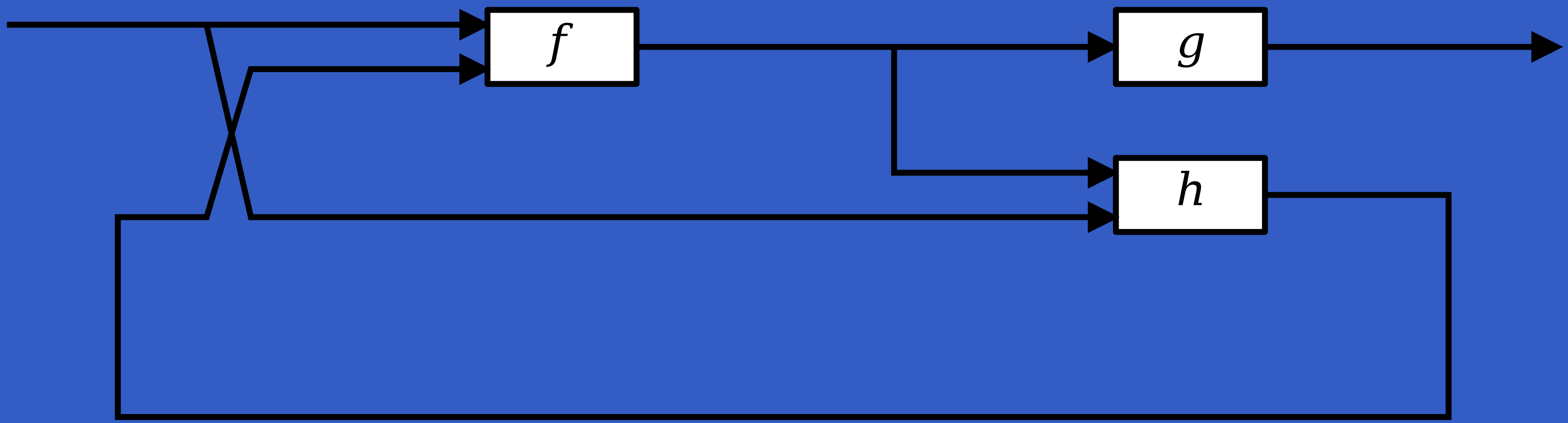


Constructing a network

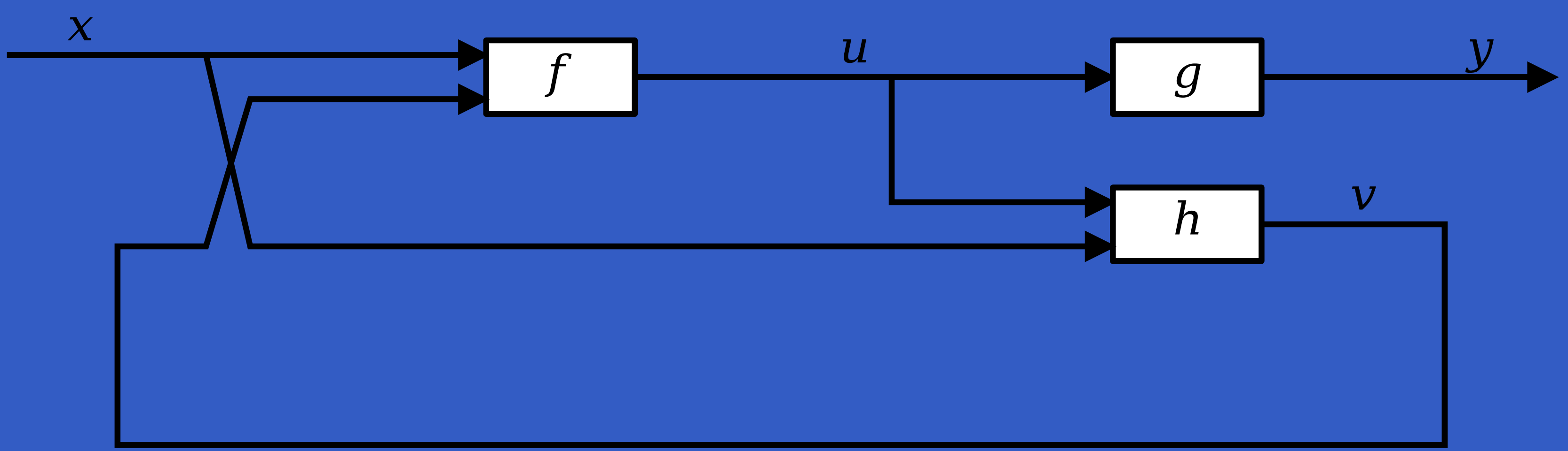


$$loop\ (arr\ (\lambda(x,y) \rightarrow ((x,y),x))$$
$$\ggg (first\ f$$
$$\ggg (arr\ (\lambda(x,y) \rightarrow (x,(x,y))) \ggg (g *** h))))$$

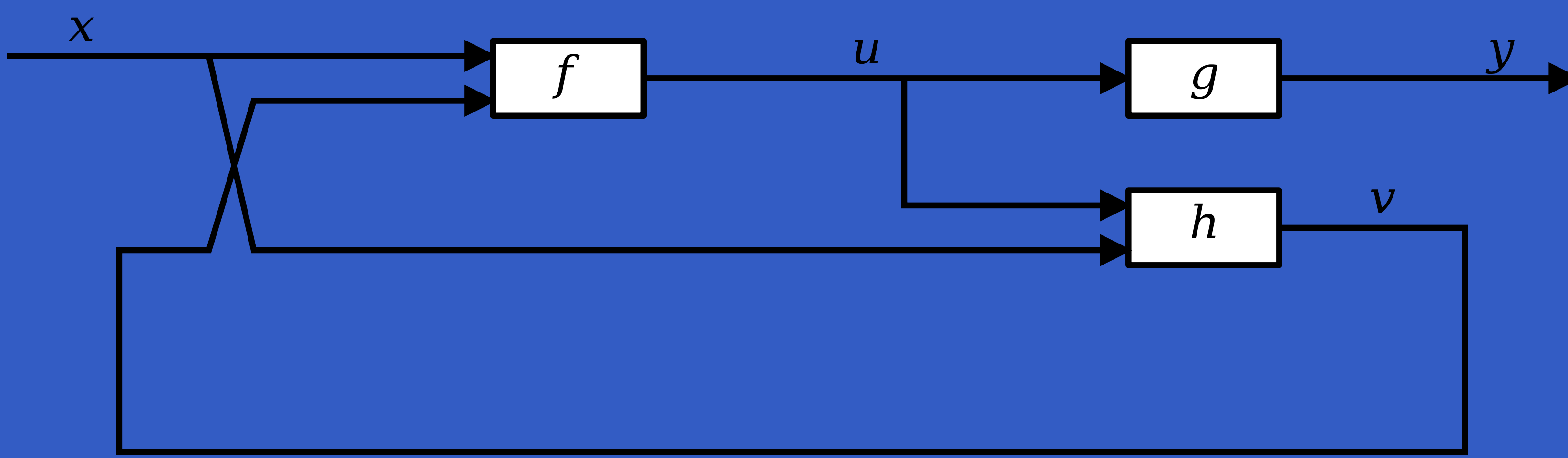
Arrow notation



Arrow notation

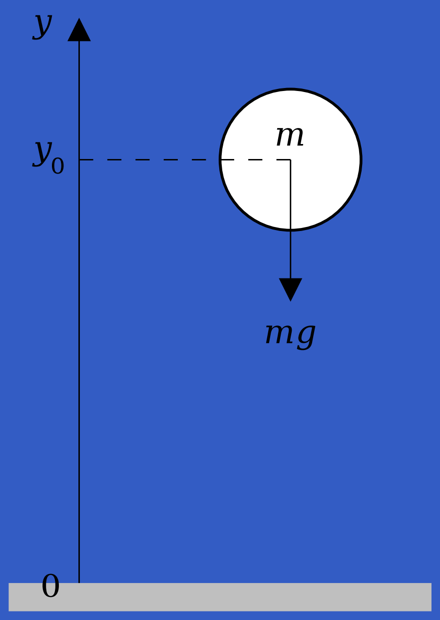


Arrow notation



$$
\begin{array}{l}
\textbf{proc } x \rightarrow \textbf{do} \\
\quad \textbf{rec} \\
\qquad u \leftarrow f \prec (x, v) \\
\qquad y \leftarrow g \prec u \\
\qquad v \leftarrow h \prec (u, x) \\
\quad returnA \prec y
\end{array}
$$

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:

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

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

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:

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

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.

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

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

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

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.

Function yielding SF to switch into

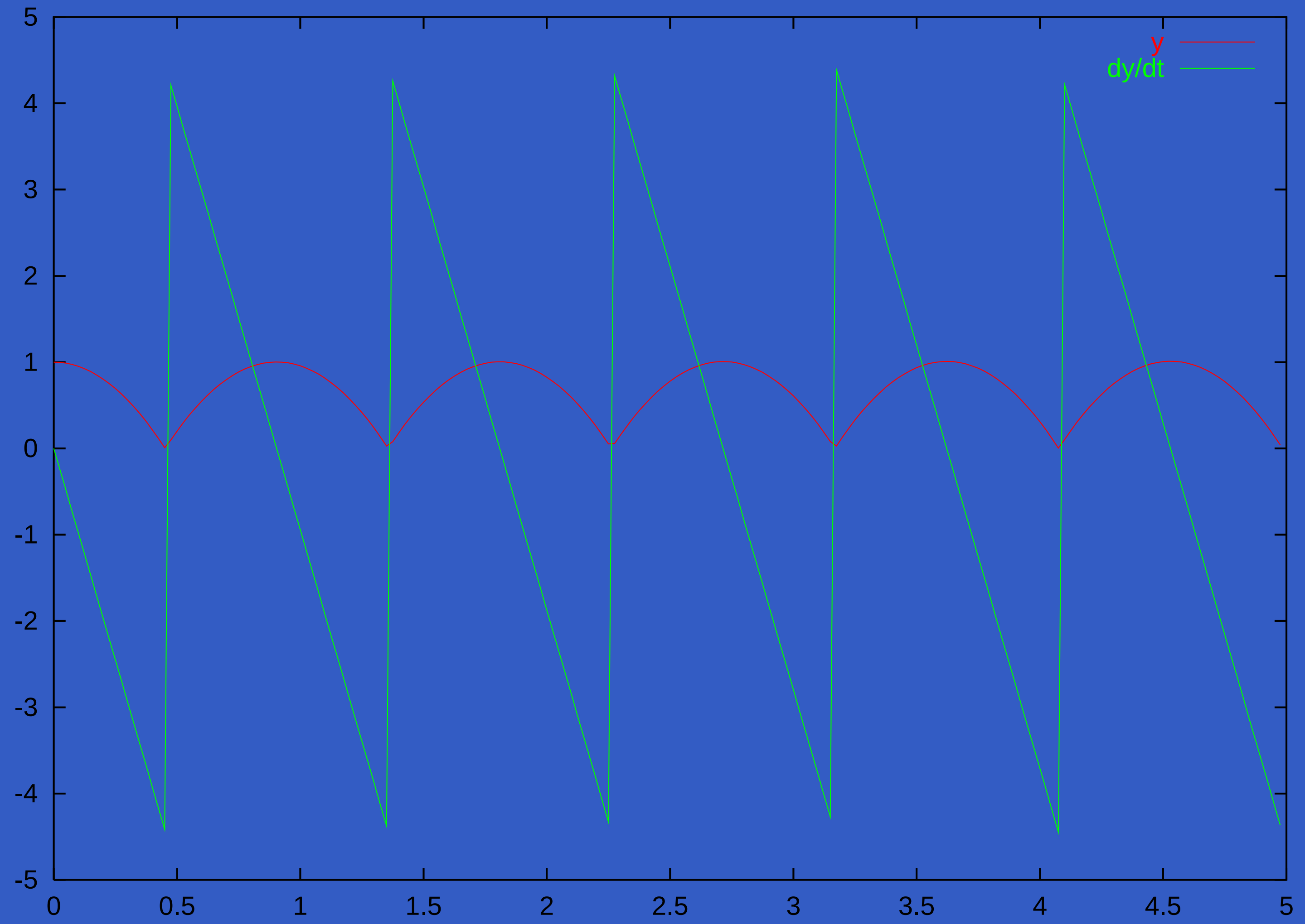
$$
\begin{aligned}
&switch :: \\
&\quad SF\ a\ (b, Event\ c) \\
&\quad \rightarrow \boxed{(c \rightarrow SF\ a\ b)} \\
&\quad \rightarrow SF\ a\ b
\end{aligned}
$$

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



The Decoupled Switch

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

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.

Game Objects (1)

Observable aspects of game entities:

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

Game Objects (2)

$$
\begin{array}{llll}
\mathbf{data}\ \mathit{ObjectKind} = & \mathit{Ball} & \mathit{Double} & \\
\mid & \mathit{Paddle} & \mathit{Size2D} & \\
\mid & \mathit{Block} & \mathit{Energy} & \mathit{Size2D} \\
\mid & \mathit{Side} & \mathit{Side} &
\end{array}
$$

Game Objects (3)

$$\textbf{type}\ \mathit{ObjectSF} = \mathit{SF}\ \mathit{ObjectInput}\ \mathit{ObjectOutput}$$

$$\begin{array}{l}
\textbf{data}\ \mathit{ObjectInput} = \mathit{ObjectInput}\ \{\\
\quad \begin{array}{ll}
\mathit{userInput} & :: \mathit{Controller},\\
\mathit{collisions} & :: [\mathit{Collision}],\\
\mathit{knownObjects} & :: [\mathit{Object}]
\end{array}\\
\}
\end{array}$$

$$\begin{array}{l}
\textbf{data}\ \mathit{ObjectOutput} = \mathit{ObjectOutput}\ \{\\
\quad \begin{array}{ll}
\mathit{outputObject} & :: \mathit{Object},\\
\mathit{harakiri} & :: \mathit{Event}\ ()
\end{array}\\
\}
\end{array}$$

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 $[\,Object\,]$ 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.

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:

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

Paddle, Take 1

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

Paddle, Take 2

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

Ball, Take 1

$$
\begin{aligned}
&\mathit{objBall} :: \mathit{ObjectSF}\\
&\mathit{objBall} =\\
&\quad \mathit{switch}\ \mathit{followPaddleDetectLaunch}\ \$\ \lambda p \rightarrow\\
&\quad \mathit{objBall}\\
&\mathit{followPaddleDetectLaunch} = \mathbf{proc}\ \mathit{oi} \rightarrow \mathbf{do}\\
&\quad o \quad\quad \leftarrow \mathit{followPaddle} \prec \mathit{oi}\\
&\quad \mathit{click} \leftarrow \mathit{edge} \quad\quad\quad \prec \mathit{controllerClick}\\
&\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (\mathit{userInput}\ \mathit{oi})\\
&\quad \mathit{returnA} \prec (o, \mathit{click}\ \text{`}\mathit{tag}\text{`}\ (\mathit{objectPos}\\
&\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (\mathit{outputObject}\ o)))
\end{aligned}
$$

Ball, Take 2

$$
\begin{array}{l}
objBall :: ObjectSF \\
objBall = \\
\quad switch\ followPaddleDetectLaunch \,\$\, \lambda p \rightarrow \\
\quad switch\ (freeBall\ p\ initBallVel \,\&\&\&\, never) \,\$\, \lambda\_ \rightarrow \\
\quad objBall
\end{array}
$$

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

Ball, Take 3

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

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

Making the Ball Bounce

$$
\begin{array}{l}
\mathit{bouncingBall}\ \mathit{p0}\ \mathit{v0} = \\
\quad \mathit{switch}\ (\mathit{moveFreelyDetBounce}\ \mathit{p0}\ \mathit{v0})\ \$\ \lambda(p', v') \rightarrow \\
\quad \mathit{bouncingBall}\ p'\ v'
\end{array}
$$

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

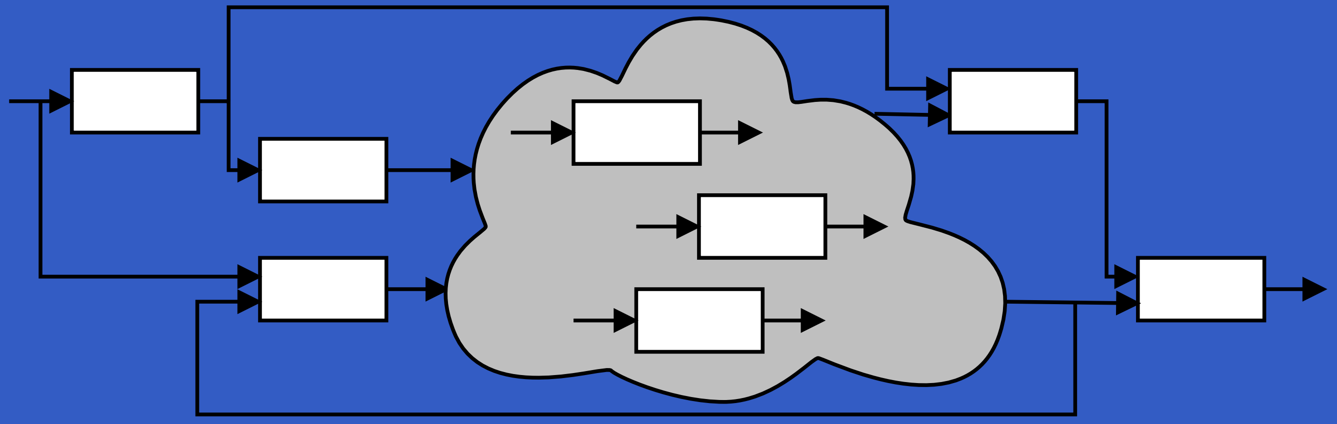
Highly dynamic system structure?

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?

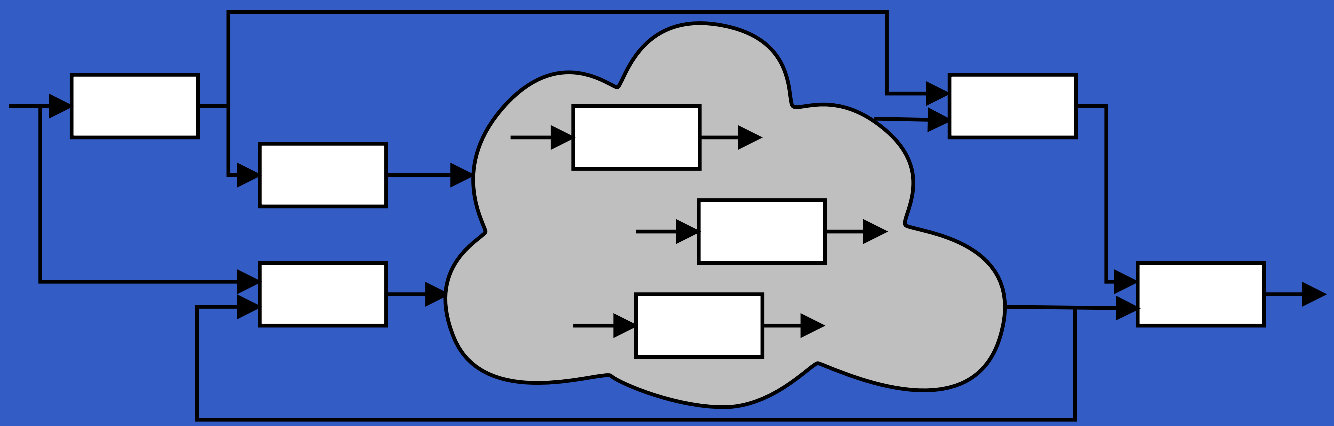


We want blocks to disappear!

Highly dynamic system structure?

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

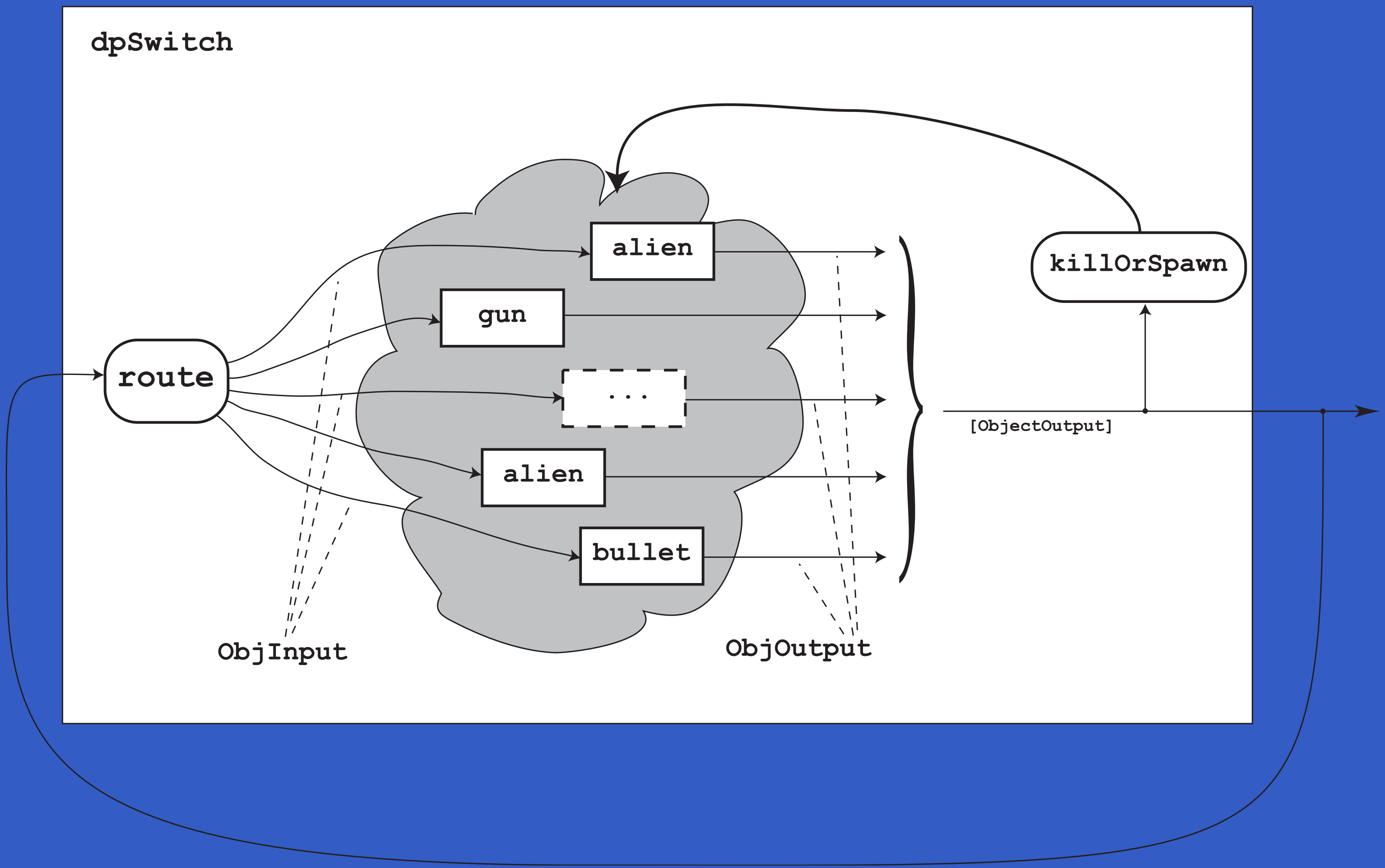
- What about more general structural changes?



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)                              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

Routing

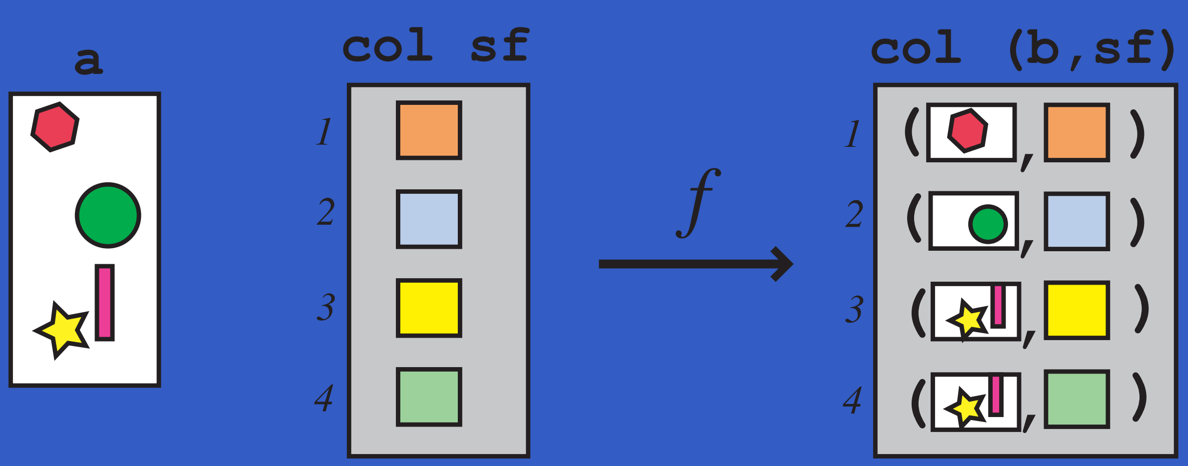
Idea:

- 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:

$$
\begin{aligned}
&\mathit{Functor}\ \mathit{col} \Rightarrow \\
&\quad (\mathit{forall}\ \mathit{sf} \circ (a \rightarrow \mathit{col}\ \mathit{sf} \rightarrow \mathit{col}\ (b, \mathit{sf})))
\end{aligned}
$$

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

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

The Game Core

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

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

Recovering Blocks

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

Reading (1)

- 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)

- Antony Courtney and Henrik Nilsson and John Peterson. The Yampa Arcade. In *Proceedings of the 2003 Haskell Workshop*, pp. 7–18, August 2003.
- Ivan Perez and Henrik Nilsson. Bridging the GUI gap with reactive values and relations. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell (Haskell'15)*, pages 47–58, 2015.