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 . . . 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

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 . . . 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)?

- 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
-

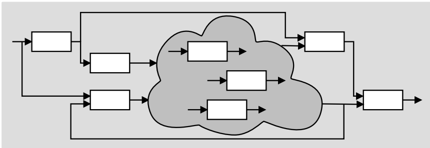
Key FRP Features

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

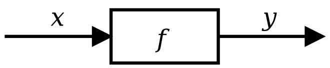
- FRP implemenattion embedded in Haskell
- Key concepts:
  - ***Signals***: time-varying values
  - ***Signal Functions***: functions on signals
  - ***Switching*** between signal functions
- Programming model: 

Yampa?

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

A good metaphor for hybrid systems!

Signal Functions



Intuition:

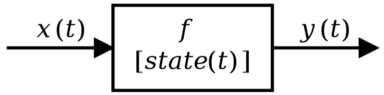
$$
\begin{aligned}
&\mathit{Time} \approx \mathbb{R}\\
&\mathit{Signal}\ a \approx \mathit{Time} \rightarrow a\\
&x :: \mathit{Signal}\ \mathit{T1}\\
&y :: \mathit{Signal}\ \mathit{T2}\\
&\mathit{SF}\ a\ b \approx \mathit{Signal}\ a \rightarrow \mathit{Signal}\ b\\
&f :: \mathit{SF}\ \mathit{T1}\ \mathit{T2}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

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

Signal Functions and State

Alternative view:

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



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

From this perspective, signal functions are:

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

Some Basic Signal Functions

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

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

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

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

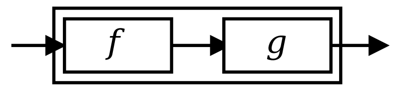
$$y(t) = \int_0^t x(\tau)\,\mathrm{d}\tau$$

Which are stateless and which are stateful?

Composition

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

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!

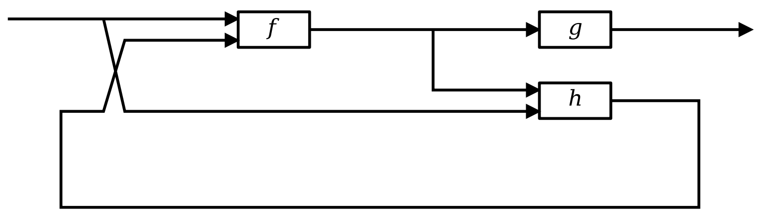
$$time :: SF\ a\ Time$$

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

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

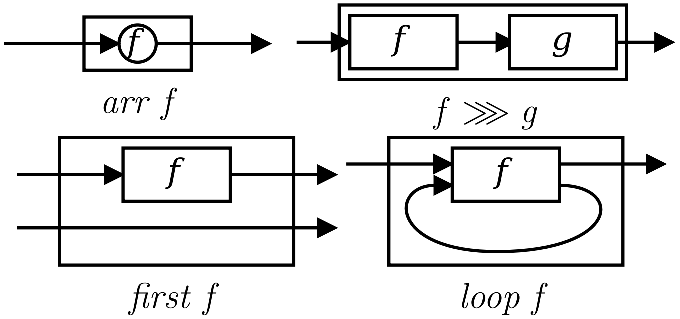
Systems

What about larger networks?
How many combinators are needed?



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

The Arrow framework (1)



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

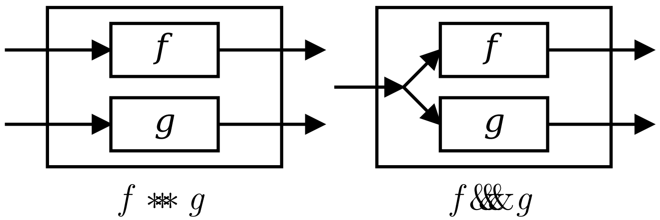
The Arrow framework (2)

Examples:

$$
\begin{aligned}
&identity :: SF\ a\ a \\
&identity = arr\ id \\
&constant :: b \to SF\ a\ b \\
&constant\ b = arr\ (const\ b) \\
&\hat{}\!\ll ::(b \to c) \to SF\ a\ b \to SF\ a\ c \\
&f\ \hat{}\!\ll sf = sf \ggg arr\ f
\end{aligned}
$$

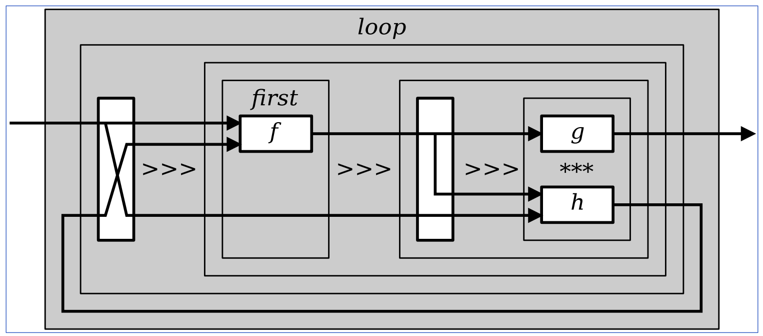
The Arrow framework (2)

Some derived combinators:



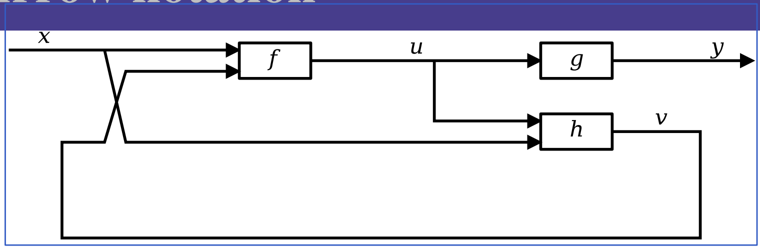
$$
\begin{aligned}
&(***) :: SF\ a\ b \to SF\ c\ d \to SF\ (a, c)\ (b, d) \\
&(\&\&\&) :: SF\ a\ b \to SF\ a\ c \to SF\ a\ (b, c)
\end{aligned}
$$

Constructing a network



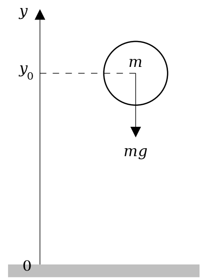
$$
\begin{aligned}
&loop\ (arr\ (\lambda(x, y) \to ((x, y), x)) \\
&\qquad \ggg (first\ f \\
&\qquad\qquad \ggg (arr\ (\lambda(x, y) \to (x, (x, y))) \ggg (g *** h))))
\end{aligned}
$$

Arrow notation



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

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:

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

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:

$$
\textbf{data}\ \mathit{Event}\ a = \mathit{NoEvent} \mid \mathit{Event}\ a
$$

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

Some Event Functions and Sources

$$tag :: Event\ a \to b \to Event\ b$$
$$never :: SF\ a\ (Event\ b)$$
$$now :: b \to SF\ a\ (Event\ b)$$
$$after :: Time \to b \to SF\ a\ (Event\ b)$$
$$repeatedly :: Time \to b \to SF\ a\ (Event\ b)$$
$$edge :: SF\ Bool\ (Event\ ())$$
$$notYet :: SF\ (Event\ a)\ (Event\ a)$$
$$once :: SF\ (Event\ a)\ (Event\ a)$$

Modelling the Bouncing Ball: Part 2

Detecting when the ball goes through the floor:

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

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.

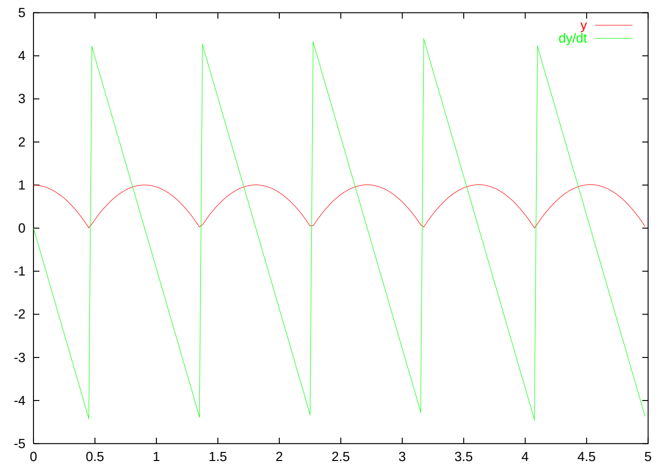
$$\begin{array}{l} switch :: \\ \quad SF\ a\ (b, Event\ c) \\ \quad \to (c \to SF\ a\ b) \\ \quad \to SF\ a\ b \end{array}$$

Modelling the Bouncing Ball: Part 3

Making the ball bounce:

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

Simulation of the Bouncing Ball



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:

$$
\begin{array}{l}
impulseIntegral :: \\
\quad VectorSpace\ a\ k \Rightarrow \\
\qquad SF\ (a, Event\ a)\ a
\end{array}
$$

The Decoupled Switch

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

- 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 ...

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

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

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

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:

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

Game Objects (2)

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

Game Objects (3)

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

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:

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

Paddle, Take 1

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

Paddle, Take 2

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

Ball, Take 1

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

Ball, Take 2

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

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

Ball, Take 3

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

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

Making the Ball Bounce

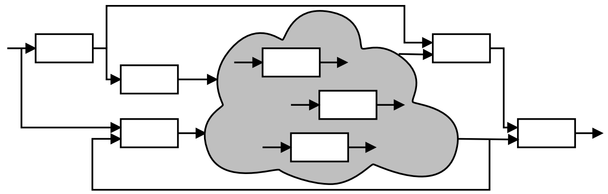
$$
\begin{array}{l}
\mathit{bouncingBall}\ p0\ v0 = \\
\quad \mathit{switch}\ (\mathit{moveFreelyDetBounce}\ p0\ v0)\ \$\ \lambda(p', v') \rightarrow \\
\quad \mathit{bouncingBall}\ p'\ v'
\end{array}
$$

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

Highly dynamic system structure?

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

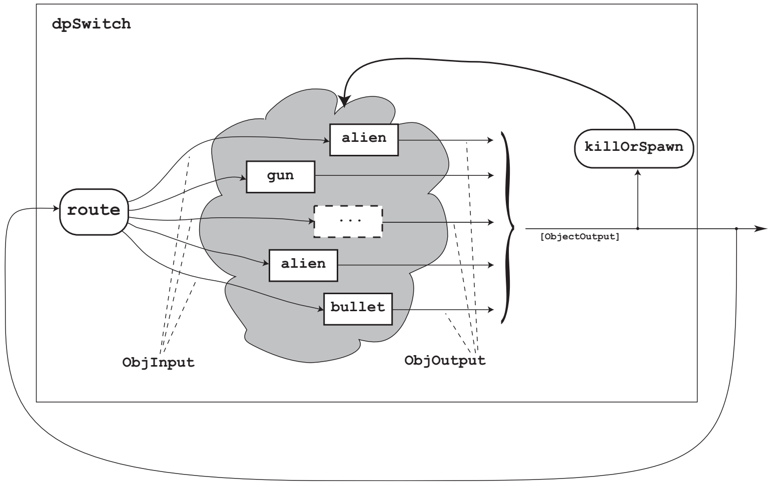
- What about more general structural changes?



We want blocks to disappear!

- What about state?

Typical Overall Game Structure



Dynamic Signal Function Collections

Idea:

- 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:

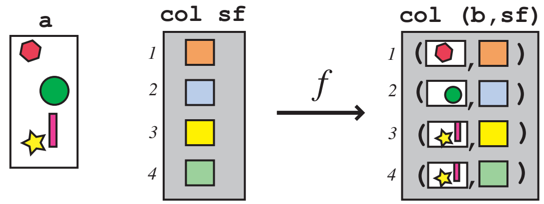
- How input routed to each signal function.
- When collection changes shape.
- How collection changes shape.

```
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)
```


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:

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

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

```
objBlockAt (x, y) (w, h) =
  proc (ObjectInput ci cs os) → do
    let name = "blockat" ++ show (x, y)
        isHit    = inCollision name cs
    hit ← edge                    ≺ isHit
    lives ← accumHoldBy (+) 3≺ (hit `tag` (−1))
    let isDead = lives ⩽ 0
    dead ← edge≺ isDead
    returnA≺ ObjectOutput
        (Object {. . .})
        dead
```


The Game Core

```
processMovement ::
  [ObjectSF] → SF ObjectInput (IL ObjectOutput)
processMovement objs =
  dpSwitchB objs
           (noEvent ⟶ arr suicidalSect)
           (λsfs′ f → processMovement′ (f sfs′))
loopPre ([ ], [ ], 0) $
  adaptInput
   ⋙ processMovement objs
   ⋙ (arr elemsIL &&& detectCollisions)
```


Recovering Blocks

```
objBlockAt (x, y) (w, h) =
  proc (ObjectInput ci cs os) → do
    let name = "blockat" ++ show (x, y)
        isHit    = inCollision name cs
    hit       ← edge                 ≺ isHit
    recover ← delayEvent 5.0≺ hit
    lives     ← accumHoldBy (+) 3
                   ≺ (hit `tag` (−1)
                        `lMerge` recover `tag` 1)
    . . .
```