Declarative Game Programming

**PPDP 2014**
*Distilled Tutorial*

Henrik Nilsson and Ivan Perez

School of Computer Science
University of Nottingham, UK
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.
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.
What is Functional Reactive Programming (FRP)?

- Paradigm for reactive programming in a functional setting.
- Idea: programming with time-varying entities.
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).
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.
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
- GUls
- Virtual Reality Environments
- Games
FRP Applications

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

- Graphical Animation
- Robotics
- Vision
- Sound synthesis
- GUls
- 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 implementation embedded in Haskell
Yampa

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

- FRP implementation 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:

\[ Time \approx R \]
\[ Signal \ a \approx Time \to a \]
\[ x :: Signal \ T1 \]
\[ y :: Signal \ T2 \]
\[ SF \ a \ b \approx Signal \ a \to Signal \ b \]
\[ f :: SF \ T1 \ T2 \]
Signal Functions

Intuition:

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

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 \textit{state}.

\[
\text{state}(t) \text{ summarizes input history } x(t'), \ t' \in [0, t].
\]
Signal Functions and State

Alternative view:

Signal functions can encapsulate state.

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

From this perspective, signal functions are:

- **stateful** if \( y(t) \) depends on \( x(t) \) and \( \text{state}(t) \)
- **stateless** if \( y(t) \) depends only on \( x(t) \)
Some Basic Signal Functions

\[\text{identity} :: SF \ a \ a\]
Some Basic Signal Functions

\[ \textit{identity} :: SF \ a \ a \]

\[ \textit{constant} :: b \rightarrow SF \ a \ b \]
Some Basic Signal Functions

\textit{identity} :: SF \ a \ a

\textit{constant} :: b \rightarrow SF \ a \ b

\textit{iPre} :: a \rightarrow SF \ a \ a
Some Basic Signal Functions

identity :: SF a a

constant :: b → SF a b

iPre :: a → SF a a

integral :: VectorSpace a s ⇒ SF a a

\[ y(t) = \int_{0}^{t} x(\tau) \, d\tau \]
Some Basic Signal Functions

identity :: $SF \ a \ a$

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

$\text{iPre :: } a \rightarrow SF \ a \ a$

integral :: $\text{VectorSpace } a \ s \Rightarrow SF \ a \ a$

$$y(t) = \int_{0}^{t} x(\tau) \ d\tau$$

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:

\[
( \ggg ) :: SF \ a \ b \to SF \ b \ c \to SF \ a \ c
\]
Composition

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

For example, serial composition:

\[
\begin{array}{c}
\mbox{f} \\
\arrow{\rightarrow}
\end{array}
\begin{array}{c}
\mbox{g}
\end{array}
\]

A \textit{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!

\[ time :: SF \ a \ Time \]
Quick exercise: Define time!

\[
time :: SF \ a \ Time
\]
\[
time = constant 1.0 \gg integral
\]
Quick exercise: Define time!

\[
time :: SF \ a \ Time
\]

\[
time = constant \ 1.0 \ \gg \ integral
\]

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

\[ \text{arr } f \]
\[ (\ggg) : SF a b \rightarrow SF b c \rightarrow SF a c \]
\[ \text{first } : SF a b \rightarrow SF (a, c) (b, c) \]
\[ \text{loop } : SF (a, c) (b, c) \rightarrow SF a b \]
The Arrow framework (2)

Examples:

\[\text{identity} :: SF \ a \ a\]
\[\text{identity} = \text{arr} \ \text{id}\]

\[\text{constant} :: b \to SF \ a \ b\]
\[\text{constant} \ b = \text{arr} (\text{const} \ b)\]

\[\hat{\ll} :: (b \to c) \to SF \ a \ b \to SF \ a \ c\]
\[f \hat{\ll} s f = s f \gg \text{arr} \ f\]
Some derived combinators:

\[ f \boxtimes g \]

\[ f \&\& g \]

\((\boxtimes) :: SF \; a \; b \to SF \; c \; d \to SF \; (a, c) \; (b, d)\)

\((\&\& \& \& \& \&) :: SF \; a \; b \to SF \; a \; c \to SF \; a \; (b, c)\)
Constructing a network

Diagram:

\[ f \rightarrow g \rightarrow h \]
Constructing a network

loop

first

f

>>>   >>>   >>>

g

***

h

PPDP 2014: Declarative Game Programming – p.20/52
Constructing a network

\[ \text{loop} \left( \text{arr} \left( \lambda (x, y) \rightarrow ((x, y), x) \right) \right) \]

\[ \ggg \left( \text{first} \ f \right) \]

\[ \ggg \left( \text{arr} \left( \lambda (x, y) \rightarrow (x, (x, y)) \right) \ggg (g \bowtie h) \right) \]
Arrow notation

[Diagram showing arrow notation with nodes labeled f, g, and h]
Arrow notation

\[ x \rightarrow f \rightarrow u \rightarrow g \rightarrow y \]
\[ x \downarrow \rightarrow h \rightarrow v \]
Arrow notation

\[
\text{proc } x \rightarrow \text{ do }
\]

\[
\text{rec}
\]

\[
u \leftarrow f \leftarrow (x, v)
\]

\[
y \leftarrow g \leftarrow u
\]

\[
v \leftarrow h \leftarrow (u, x)
\]

\[
\text{return } A \leftarrow y
\]
A Bouncing Ball

\[ y = y_0 + \int v \, dt \]

\[ v = v_0 + \int -9.81 \]

On impact:

\[ v = -v(t^-) \]

(fully elastic collision)
Modelling the Bouncing Ball: Part 1

Free-falling ball:

```plaintext
type Pos = Double

type Vel = Double

fallingBall :: Pos → Vel → SF () (Pos, Vel)

fallingBall y0 v0 = proc () → do
    v ← (v0 +) \langle integral \rangle − 9.81
    y ← (y0 +) \langle integral \rangle v
    returnA (y, v)
```
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.
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:

\[
data Event a = NoEvent \mid Event a
\]
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:

\[
\text{data } \text{Event } a = \text{NoEvent} \mid \text{Event } a
\]

*Discrete-time signal* \( = \text{Signal} \left( \text{Event } a \right) \).
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)
Detecting when the ball goes through the floor:

```
fallingBall' ::
    Pos → Vel → SF () ((Pos, Vel), Event (Pos, Vel))
fallingBall' y0 v0 = proc () → do
    yv@(y, _) ← fallingBall y0 v0 ()
    hit ← edge → y ≤ 0
    returnA ← (yv, hit `tag` yv)
```
Q: How and when do signal functions “start”?
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 function 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.
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.

\[
\text{switch}::
\]

\[
SF\ a\ (b,\ Event\ c) \\
\rightarrow\ (c\ \rightarrow\ SF\ a\ b) \\
\rightarrow\ SF\ a\ b
\]
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.

\[
\text{switch} ::
\begin{align*}
SF \ a \ (b, \ Event \ c) \\
\rightarrow \ (c \rightarrow SF \ a \ b) \\
\rightarrow \ SF \ a \ b
\end{align*}
\]
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.

\[
\text{switch::} \\
SF\ a\ (b, \text{Event}\ c) \\
\rightarrow (c \rightarrow SF\ a\ b) \\
\rightarrow SF\ a\ b
\]
Making the ball bounce:

\[\text{bouncingBall} :: \text{Pos} \rightarrow \text{SF} () (\text{Pos}, \text{Vel})\]
\[\text{bouncingBall } y0 = \text{bbAux } y0 \ 0.0\]

where
\[\text{bbAux } y0 \ v0 =\]
\[\text{switch } (\text{fallingBall'} \ y0 \ v0) \ \& \ \text{\textlambda}(y, v) \rightarrow \]
\[\text{bbAux } y (-v)\]
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:

\[
\text{impulseIntegral} ::
\text{VectorSpace } a \to k \Rightarrow \text{SF} (a, \text{Event } a) a
\]
The Decoupled Switch

\[ dSwitch :: \]
\[ SF~a~(b,~Event~c) \]
\[ \rightarrow (c \rightarrow SF~a~b) \]
\[ \rightarrow SF~a~b \]
The Decoupled Switch

dSwitch ::

\[ SF \ a \ (b, \ Event \ c) \rightarrow (c \rightarrow SF \ a \ b) \rightarrow SF \ a \ b \]

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

\( dSwitch :: \)
\[ SF\ a\ (b,\ Event\ c) \]
\[ \rightarrow\ (c\ \rightarrow\ SF\ a\ b) \]
\[ \rightarrow\ SF\ a\ b \]

- 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.
rSwitch, drSwitch ::

\[ SF \ a \ b \to SF \ (a, Event \ (SF \ a \ b)) \ b \]
Lots of Switches . . .

\[ r\text{Switch}, \; dr\text{Switch} :: \]
\[ SF \; a \; b \rightarrow SF \; (a, \; Event \; (SF \; a \; b)) \; b \]

\[ k\text{Switch}, \; dk\text{Switch} :: \]
\[ SF \; a \; b \rightarrow SF \; (a, \; b) \; (Event \; c) \]
\[ \rightarrow (SF \; a \; b \rightarrow c \rightarrow SF \; a \; b) \rightarrow SF \; a \; b \]
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 :: . . .
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 \textit{switch} or \textit{dSwitch} and a notion of \textit{ageing} signal functions:

\[ age :: SF \ a \ b \rightarrow SF \ a \ (b, SF \ a \ b) \]
Game Objects (1)

Observable aspects of game entities:

data Object = Object {
    objectName :: ObjectName,
    objectKind :: ObjectKind,
    objectPos :: Pos2D,
    objectVel :: Vel2D,
    objectAcc :: Acc2D,
    objectDead :: Bool,
    objectHit :: Bool,
    ...
}

...
data ObjectKind = Ball Double |
| Paddle Size2D |
| Block Energy Size2D |
| Side Side |
Game Objects (3)

```haskell
type ObjectSF = SF ObjectInput ObjectOutput

data ObjectInput = ObjectInput { 
  userInput :: Controller, 
  collisions :: [Collision], 
  knownObjects :: [Object] 
}

data ObjectOutput = ObjectOutput { 
  outputObject :: Object, 
  harakiri :: Event () 
} 
```
Observing the Game World

- Note that \[ Object \] appears in the input type.
Observing the Game World

- Note that \( \text{Object} \) appears in the input type.
- This allows each game object to observe \textit{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:

\[
\text{loopPre} :: c \rightarrow SF (a, c) (b, c) \rightarrow SF a b \\
\text{loopPre } c\_\text{init } sf = \\
\text{loop } (\text{second } (i\text{Pre } c\_\text{init}) \ggg sf)
\]
Paddle, Take 1

\[
\text{objPaddle :: ObjectSF} \\
\text{objPaddle = proc (ObjectInput ci cs os) \rightarrow do} \\
\text{let name = "paddle"} \\
\text{let isHit = inCollision name cs} \\
\text{let } p = \text{refPosPaddle ci} \\
\text{v \leftarrow derivative} \leftarrow p \\
\text{returnA} \leftarrow \text{livingObject} \$ \text{Object} \{ \\
\text{objectName = name,} \\
\text{objectPos = } p, \\
\text{objectVel = } v, \\
\text{\ldots}\}\]
objPaddle :: ObjectSF

objPaddle = proc (ObjectInput ci cs os) → do
let name = "paddle"
let isHit = inCollision name cs
rec
  let v = limitNorm (20.0 ^ (refPosPaddle ci ^ p))
    maxVNorm
  p ← (initPosPaddle ^+) ^ integral ← v
returnA ← livingObject $ Object { ... }
Ball, Take 1

\[ \text{objBall} :: \text{ObjectSF} \]

\[ \text{objBall} = \]

\[ \text{switch followPaddleDetectLaunch} \ \lambda p \rightarrow \text{objBall} \]

\[ \text{followPaddleDetectLaunch} = \text{proc} \ oi \rightarrow \text{do} \]

\[ o \leftarrow \text{followPaddle} \leftarrow oi \]

\[ \text{click} \leftarrow \text{edge} \leftarrow \text{controllerClick} \]

\[ \text{(userInput} \ oi) \]

\[ \text{returnA} \leftarrow (o, \text{click} \tag{\text{objectPos}} \text{(outputObject} \ o)) \]
Ball, Take 2

\[ \text{objBall} :: \text{ObjectSF} \]

\[ \text{objBall} = \]

\[ \begin{array}{l}
\text{switch followPaddleDetectLaunch} \quad $ \lambda p \rightarrow$
\text{switch (freeBall p initBallVel && never)} \quad $ \lambda _- \rightarrow$
\text{objBall}
\end{array} \]

\[ \text{freeBall p0 v0} = \text{proc} \quad (\text{ObjectInput ci cs os}) \rightarrow \text{do} \]

\[ \begin{array}{l}
p \leftarrow (p0 \hat{+} \hat{+}) \hat{\ll} \text{integral} \leftarrow v0' \\
\text{returnA} \leftarrow \text{livingObject} \quad $ \{ \ldots \}$
\end{array} \]

\text{where}

\[ v0' = \text{limitNorm v0 maxVNorm} \]
Ball, Take 3

\[ \text{objBall} :: \text{ObjectSF} \]

\[ \text{objBall} = \]

\[ \text{switch} \ \text{followPaddleDetectLaunch} \ \bigg( \lambda p \rightarrow \bigg) \]

\[ \text{switch} \ (\text{bounceAroundDetectMiss} \ p) \ \bigg( \lambda _\rightarrow \bigg) \]

\[ \text{objBall} \]

\[ \text{bounceAroundDetectMiss} \ p = \text{proc} \ \text{oi} \rightarrow \text{do} \]

\[ o \gets \text{bouncingBall} \ p \ \text{initBallVel} \gets \text{oi} \]

\[ \text{miss} \gets \text{collisionWithBottom} \]

\[ \text{returnA} \gets (o, \text{miss}) \]
Making the Ball Bounce

\[
bouncingBall\ p_0\ v_0 = \\text{switch}\ (\text{moveFreelyDetBounce}\ p_0\ v_0)\ \&\ \lambda(p',\ v') \rightarrow bouncingBall\ p'\ v'
\]

\[
\text{moveFreelyDetBounce}\ p_0\ v_0 = \\
\text{proc}\ \text{oi}\ (\text{ObjectInput } \_\ cs \_ ) \rightarrow \text{do} \\
\hspace{1em} o \leftarrow \text{freeBall}\ p_0\ v_0 \leftarrow o\ i \\
\hspace{1em} ev \leftarrow \text{edgeJust} \lll \text{initially Nothing} \\
\hspace{2em} \leftarrow \text{changedVelocity}\ "ball"\ cs \\
\hspace{1em} \leftarrow \text{returnA}\ (o,\ \text{fmap}\ (\lambda v \rightarrow (\text{objectPos}\ (\ldots o),\ v))\ ev)
\]
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

dpSwitch

route

alien

gun

alien

bullet

killOrSpawn

ObjInput

[ObjectOutput]

ObjOutput
Dynamic Signal Function Collections

Idea:
Dynamic Signal Function Collections

Idea:

• Switch over \textit{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.

\[
\text{dpSwitch} :: \text{Functor } \text{col} \Rightarrow \\
(\forall sf . (a \to \text{col } sf \to \text{col } (b,sf))) \\
\to \text{col } (\text{SF } b \ c) \\
\to \text{SF } (a, \text{col } c) \text{ (Event } d) \\
\to (\text{col } (\text{SF } b \ c) \to d \to \text{SF } a \ (\text{col } c)) \\
\to \text{SF } a \ (\text{col } c)
\]
dpSwitch

Need ability to express:

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

dpSwitch :: Functor col =>

\((\forall sf . (a \rightarrow \text{col} \sf \rightarrow \text{col} \ (b,\sf)))\)
\rightarrow \text{col} \ (\text{SF} \ b \ c)

\rightarrow \text{SF} \ (a, \text{col} \ c) \ (\text{Event} \ d)

\rightarrow (\text{col} \ (\text{SF} \ b \ c) \rightarrow d \rightarrow \text{SF} \ a \ (\text{col} \ c))

\rightarrow \text{SF} \ a \ (\text{col} \ c)
需能够表达:
- 如何将输入路由到每个信号函数。
- 当集合的变化形状时。
- 如何变化集合变化形状。

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 changes shape.

\[
dpSwitch :: \text{Functor } \text{col} \Rightarrow
\]
\[
(\forall sf . (a \rightarrow \text{col } sf \rightarrow \text{col } (b,sf)))
\]
\[
\rightarrow \text{col } (\text{SF } b \ c)
\]
\[
\rightarrow \text{SF } (a, \text{col } c) \ (\text{Event } d)
\]
\[
\rightarrow (\text{col } (\text{SF } b \ c) \rightarrow d \rightarrow \text{SF } a \ (\text{col } c))
\]
\[
\rightarrow \text{SF } a \ (\text{col } c)
\]
**dpSwitch**

Need ability to express:

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

\[
\text{dpSwitch} :: \text{Functor} \ col \Rightarrow \\
(\forall \ sf . (a \to \ col \ sf \to \ col \ (b, sf))) \\
\to \ \col \ (\text{SF} \ b \ c) \\
\to \ \text{SF} \ (a, \col \ c) \ (\text{Event} \ d) \\
\to \ (\col \ (\text{SF} \ b \ c) \to \ d \to \ \text{SF} \ a \ (\col \ c)) \\
\to \ \text{SF} \ a \ (\col \ c)
\]

*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:

\[
Functor \ col \Rightarrow \\
(\forall sf \circ (a \rightarrow col \; sf \rightarrow col (b, sf)))
\]

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.
objBlockAt \((x, y) (w, h) = \)

\[
\text{proc } (\text{ObjectInput } ci \text{ cs } os) \rightarrow \text{ do} \\
\text{let } name = "blockat" \# show \((x, y)\) \\
\text{isHit } = \text{inCollision name cs} \\
\text{hit } \leftarrow \text{edge} \leftarrow \text{isHit} \\
\text{lives } \leftarrow \text{accumHoldBy } (+) 3 \leftarrow (\text{hit ‘tag‘ } (-1)) \\
\text{let } \text{isDead } = \text{lives } \leq 0 \\
\text{dead } \leftarrow \text{edge} \leftarrow \text{isDead} \\
\text{returnA} \leftarrow \text{ObjectOutput} \\
\text{(Object \{\ldots\})} \\
\text{dead}
\]
The Game Core

\[
\text{processMovement} ::
\]
\[
\left[ ObjectSF \right] \rightarrow SF \ ObjectInput \ (IL \ ObjectOutput)
\]
\[
\text{processMovement} \ \text{objs} = \]
\[
dpSwitchB \ \text{objs}
\]
\[
\quad (\text{noEvent} \rightarrow \text{arr suicidal Sect})
\]
\[
\quad (\lambda sfs' \ f \rightarrow \text{processMovement}' \ (f \ sfs'))
\]
\[
\text{loopPre} \ ([], [], 0) \$
\]
\[
\text{adaptInput}
\]
\[
\ggg \ \text{processMovement} \ \text{objs}
\]
\[
\ggg \ (\text{arr elementsIL} \& \& \text{detectCollisions})
\]
Recovering Blocks

\( \text{objBlockAt} (x, y) (w, h) = \)

\[ \text{proc} \ (\text{ObjectInput ci cs os}) \rightarrow \text{do} \]

\[ \text{let name} = \"\text{blockat}\" + \text{show} (x, y) \]

\[ \text{isHit} = \text{inCollision name cs} \]

\[ \text{hit} \leftarrow \text{edge} \leftarrow \text{isHit} \]

\[ \text{recover} \leftarrow \text{delayEvent} \ 5.0 \leftarrow \text{hit} \]

\[ \text{lives} \leftarrow \text{accumHoldBy} (+) 3 \]

\[ (\text{hit} \ 'tag' \ (-1)) \]

\[ 'lMerge' \ 'recover' \ 'tag' \ 1) \]

\[ \ldots \]