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

**Signal Functions**

Intuition:

\[
\begin{align*}
\text{Time} & \approx \mathbb{R} \\
\text{Signal } a & \approx \text{Time} \to a \\
x & :: \text{Signal } T1 \\
y & :: \text{Signal } T2 \\
SF \ a \ b & \approx \text{Signal } a \to \text{Signal } b \\
f & :: SF \ T1 \ T2
\end{align*}
\]

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

**Yampa?**

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

A good metaphor for hybrid systems!

**Signal Functions and State**

Alternative view:

Signal functions can encapsulate **state**.

\[
\begin{align*}
\text{state}(t) & \text{ summarizes input history } x(t'), t' \in [0, t].
\end{align*}
\]

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

\[ \text{identity} :: SF \; a \; a \]
\[ \text{constant} :: b \rightarrow SF \; a \; b \]
\[ \text{iPre} :: a \rightarrow SF \; a \; a \]
\[ \text{integral} :: \text{VectorSpace} \; a \; s \Rightarrow SF \; a \; a \]
\[ y(t) = \int_{0}^{t} x(\tau) \, d\tau \]

Which are stateless and which are stateful?

Composition

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

For example, serial composition:

\[ f \circ g \]

A **combinator** that captures this idea:

\[ (\circ) :: SF \; a \; b \rightarrow SF \; b \; c \rightarrow SF \; a \; c \]

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

Systems

Quick exercise: Define time!

\[ \text{time} :: SF \; a \; \text{Time} \]
\[ \text{time} = \text{constant} \; 1.0 \circlearrowright \text{integral} \]

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

John Hughes’s **Arrow** framework provides a good answer!
The Arrow framework (1)

arr \ f \rightarrow \ f \Rightarrow g \rightarrow f

first f \rightarrow \ loop f

arr :: (a \rightarrow b) \rightarrow SF a b

(\Rightarrow) :: 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

The Arrow framework (2)

Examples:

identity :: SF a a

identity = arr id

constant :: b \rightarrow SF a b

constant b = arr (const b)

\Leftarrow :: (b \rightarrow c) \rightarrow SF a b \rightarrow SF a c

f \Leftarrow sf = sf \Rightarrow arr f

Some derived combinators:

f \Rightarrow g

g \Rightarrow f

(\Rightarrow) :: SF a b \rightarrow SF c d \rightarrow SF (a, c) (b, d)

(\Leftarrow) :: SF a b \rightarrow SF a c \rightarrow SF a (b, c)

Constructing a network

loop (arr (\lambda(x, y) \rightarrow ((x, y), x))

\Rightarrow (first f

\Rightarrow (arr (\lambda(x, y) \rightarrow (x, (x, y)))) \Rightarrow (g \Leftarrow h))))
Arrow notation

\[
\text{proc } x \rightarrow \text{do}
\quad \text{rec}
\quad \begin{align*}
    & u \leftarrow f \leftarrow (x, v) \\
    & y \leftarrow g \leftarrow u \\
    & v \leftarrow h \leftarrow (u, x) \\
    & \text{return } A \leftarrow y
\end{align*}
\]

A Bouncing Ball

\[
y = y_0 + \int v \, dt \\
v = v_0 + \int -9.81 \\
\text{On impact:} \\
v = -v(t-) \\
\text{(fully elastic collision)}
\]

Modelling the Bouncing Ball: Part 1

Free-falling ball:

type Pos = Double

type Vel = Double

fallingBall :: Pos -> Vel -> SF () (Pos, Vel)

fallingBall y0 v0 = proc () -> do
    v ← (v0+) ≪ integral ≫ -9.81
    y ← (y0+) ≪ integral ≪ v
    returnA ← (y, v)

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:

data Event a = NoEvent | Event a

Discrete-time signal = Signal (Event a).
Some Event Functions and Sources

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

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.

Modelling the Bouncing Ball: Part 2

Detecting when the ball goes through the floor:

fallingBall' ::

\[ Pos → Vel → SF () ((Pos, Vel), Event (Pos, Vel)) \]

\[
\text{fallingBall'} y0 v0 = \text{proc () → do}
\text{yv@(_,_) ← fallingBall y0 v0 → ()
hit ← edge → y ≤ 0
\text{returnA← (yv, hit 'tag' yv)}
\]

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.

\[
\text{switch ::}
\text{SF a (b, Event c)
→ (c → SF a b)
→ SF a b}
\]
Modelling the Bouncing Ball: Part 3

Making the ball bounce:

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

where

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

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:

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

The Decoupled Switch

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

- 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.
Lots of Switches ...

\begin{align*}
lSwitch, \ dSwitch :: & \quad SF \ a \ b \rightarrow SF \ (a, \ Event \ (SF \ a \ b)) \ b \\
kSwitch, \ dkSwitch :: & \quad 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
\end{align*}

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

\begin{align*}
age :: & \quad SF \ a \ b \rightarrow SF \ a \ (b, SF \ a \ b)
\end{align*}

Game Objects (1)

Observable aspects of game entities:

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

Game Objects (2)

\begin{verbatim}
data ObjectKind = Ball Double \\
  | Paddle Size2D \\
  | Block Energy Size2D \\
  | Side Side
\end{verbatim}

Game Objects (3)

\begin{verbatim}
type ObjectSF = SF ObjectInput ObjectOutput 

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

data ObjectOutput = ObjectOutput {
  outputObject :: Object,
  harakiri :: Event ()
}
\end{verbatim}
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:

\[
\text{loopPre} :: c \rightarrow SF (a, c) \rightarrow SF (b, c) \rightarrow SF a b
\]
\[
\text{loopPre } c_{\text{init}} \text{ sf } =
\text{loop } (\text{second } (\text{iPre } c_{\text{init}})) \gg sf
\]

Paddle, Take 1

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

Paddle, Take 2

\[
\text{objPaddle} :: \text{ObjectSF}
\]
\[
\text{objPaddle} = \text{proc } (\text{ObjectInput } ci \text{ cs } os) \rightarrow \text{do}
\]
\[
\text{let name } = "\text{paddle}"
\]
\[
\text{let isHit } = \text{inCollision } name \text{ cs}
\]
\[
\text{rec}
\]
\[
\text{let } v = \text{limitNorm } (20.0 \ast (\text{refPosPaddle } ci \leftarrow p))
\]
\[
\max V\text{Norm}
\]
\[
p \leftarrow (\text{initPosPaddle } \leftarrow) \preceq \text{integral } \rightarrow v
\]
\[
\text{returnA } \leftarrow \text{livingObject } \$$\text{Object }$
\]
\[
\ldots
\]

Ball, Take 1

\[
\text{objBall} :: \text{ObjectSF}
\]
\[
\text{objBall} =
\text{switch } \text{followPaddleDetectLaunch } \$$\lambda p \rightarrow
\text{objBall}
\text{followPaddleDetectLaunch } = \text{proc } oi \rightarrow \text{do}
\]
\[
o \leftarrow \text{followPaddle } \rightarrow oi
\]
\[
\text{click } \leftarrow \text{edge } \rightarrow \text{controllerClick } (\text{userInput } oi)
\]
\[
\text{returnA } \leftarrow (o, \text{click } '\text{tag}' (\text{objectPos}
\]
\[
\text{(outputObject } o)))
\]
Ball, Take 2

\[
\text{objBall} :: \text{ObjectSF} \\
\text{objBall} = \\
\quad \text{switch } \text{followPaddleDetectLaunch} \; \lambda p \to \\
\quad \text{switch } (\text{freeBall} \; p \; \text{initBallVel} \& \& \text{never}) \; \lambda \_ \to \\
\quad \text{objBall} \\
\text{freeBall} \; p0 \; v0 = \text{proc } (\text{ObjectInput} \; ci \; cs \; os) \to \text{do} \\
\quad p \leftarrow (p0 \wedge+ \wedge) \wedge \text{integral} \wedge v0' \\
\quad \text{returnA} \leftarrow \text{livingObject} \; \{ \ldots \} \\
\quad \text{where} \\
\quad v0' = \text{limitNorm} \; v0 \; \text{maxVNorm}
\]

Making the Ball Bounce

\[
\text{bouncingBall} \; p0 \; v0 = \\
\quad \text{switch } (\text{moveFreelyDetBounce} \; p0 \; v0) \; \lambda (p', v') \to \\
\quad \text{bouncingBall} \; p' \; v' \\
\text{moveFreelyDetBounce} \; p0 \; v0 = \\
\quad \text{proc } oi@(\text{ObjectInput} \; _- \; cs \; _-) \to \text{do} \\
\quad o \leftarrow \text{freeBall} \; p0 \; v0 \leftarrow oi \\
\quad ev \leftarrow \text{edgeJust} \leftarrow \text{initially} \; \text{Nothing} \\
\quad \leftarrow \text{changedVelocity} \; "\text{ball}" \; cs \\
\quad \text{returnA} \leftarrow (o, \text{fmap} \; (\lambda v \to (\text{objectPos} \; \ldots \; o, v)) \; ev)
\]

Ball, Take 3

\[
\text{objBall} :: \text{ObjectSF} \\
\text{objBall} = \\
\quad \text{switch } \text{followPaddleDetectLaunch} \; \lambda p \to \\
\quad \text{switch } (\text{bounceAroundDetectMiss} \; p) \; \lambda \_ \to \\
\quad \text{objBall} \\
\text{bounceAroundDetectMiss} \; p = \text{proc } oi \to \text{do} \\
\quad o \leftarrow \text{bouncingBall} \; p \; \text{initBallVel} \leftarrow oi \\
\quad \text{miss} \leftarrow \text{collisionWithBottom} \leftarrow \text{collisions} \; oi \\
\quad \text{returnA} \leftarrow (o, \text{miss})
\]

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?
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.

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:

```
col sf 1 2 3 4
  a 1 2 3 4
  col sf 1 2 3 4
  f 1 2 3 4
```
The Routing Function Type

Universal quantification over the collection members:

\[
\text{Functor col} \Rightarrow (\forall sf \circ (a \rightarrow \text{col sf} \rightarrow \text{col} (b, sf)))
\]

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.

The Game Core

\[
\text{processMovement} :: [\text{ObjectSF}] \rightarrow \text{SF ObjectInput (IL ObjectOutput)}
\]

\[
\text{processMovement obj} = \\
\text{dpSwitchB obj} \\
(\text{noEvent } \rightarrow \text{arr suicidalSect}) \\
(\lambda sf' f \rightarrow \text{processMovement'} (f, sf'))
\]

\[
\text{loopPre} ([], [], 0) \&
\text{adaptInput} \\
\Rightarrow \text{processMovement obj} \\
\Rightarrow (\text{arr elemsIL} \& \text{detectCollisions})
\]

Blocks

\[
\text{objBlockAt} (x, y) (w, h) = \\
\text{proc (ObjectInput ci cs os) } \rightarrow \text{do} \\
\text{let name = "blockat" + show (x, y) } \\
\text{isHit = inCollision name cs} \\
\text{hit } \leftarrow \text{edge } \Rightarrow \text{isHit} \\
\text{lives } \leftarrow \text{accumHoldBy (+) 3 } \Rightarrow \text{hit 'tag' (−1)} \\
\text{let isDead = lives } \leq 0 \\
\text{dead } \leftarrow \text{edge } \Rightarrow \text{isDead} \\
\text{returnA } \Rightarrow \text{ObjectOutput} \\
(\text{Object \{\ldots\}}) \\
\text{dead}
\]

Recovering Blocks

\[
\text{objBlockAt} (x, y) (w, h) = \\
\text{proc (ObjectInput ci cs os) } \rightarrow \text{do} \\
\text{let name = "blockat" + show (x, y) } \\
\text{isHit = inCollision name cs} \\
\text{hit } \leftarrow \text{edge } \Rightarrow \text{isHit} \\
\text{recover } \leftarrow \text{delayEvent 5.0 } \Rightarrow \text{hit} \\
\text{lives } \leftarrow \text{accumHoldBy (+) 3 } \Rightarrow \text{(hit 'tag' (−1))} \\
\text{'lMerge' recover 'tag' 1} \\
\text{...}
\]