Reactive programming (1)

- Input arrives while system is running.
- Output is generated in response to input in an interleaved and timely fashion.

Contrast

The notions of time and time-varying values, or are inherent and central to reactive systems.

Reactive programming (2)

Reactive systems are
- generally concurrent
- often parallel
- often distributed

Thus, besides timeliness, difficulties related to development of concurrent, parallel, and distributed programming are also inherent.

The Synchronous Approach (1)

The “synchronous realisation” (France, 1980s):

If we heed the observation that time-varying values are central to reactive programming and
- express systems directly as of such entities
- adopt system-wide time, abstracting away processing delays (hence synchronous).

... then:

- systems can be described declaratively at a very high level of abstraction
- simple, deterministic semantics, facilitates reasoning
- many problems related to imperative idioms for concurrency and synchronisation simply vanishes.

Contrast programming with values at isolated points in time in a fundamentally temporally agnostic setting.

The Synchronous Approach (2)

The synchronous languages were invented in France in the 1980s. The first ones were:
- Esterel
- Lustre
- Signal

Have been very successful; e.g. lots of industrial applications.

Many new languages and variations since then.

Example: Robotics (1)

[PPDP'02, with Izzet Pembeci and Greg Hager, Johns Hopkins University]

Hardware setup:

Functional Reactive Programming

Functional Reactive Programming (FRP):
- Paradigm for reactive, concurrent programming in purely declarative (functional) setting.
- Originated from Functional Reactive Animation (Fran) (Elliott & Hudak).
- Has evolved in a number of directions and into different concrete implementations.
- (Usually) continuous notion of time and additional support for discrete events.

FRP applications

Some domains where FRP or FRP-like ideas have been used:
- Graphical Animation
- Robotics
- Vision
- GUIs
- Hybrid modeling
- Video games
- Sensor networks
- Audio processing and generation
- Financial, event-based systems
Related approaches

FRP related to:
- Synchronous languages, like Esterel, Lucid
Synchron.  
- Modeling languages, like Simulink, Modelica.

Distinguishing features of FRP:
- First class reactive components.
- Allows highly dynamic system structure.
- Supports hybrid (mixed continuous and discrete) systems.

Yampa

- An FRP system originating at Yale
  in Haskell (a Haskell library).
- Notionally continuous time.
- Discrete-time signals modelled by continuous-time signals and an option type, allowing for highly dynamic system structure.

Yampa?

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

A good metaphor for hybrid systems!

Signal functions

Key concept:

\[ x \xrightarrow{f} y \]

Intuition:

\[ \text{Signal } \alpha \cong \text{Time} \rightarrow \alpha \]
\[ x :: \text{Signal } T_1 \]
\[ y :: \text{Signal } T_2 \]
\[ f :: \text{Signal } T_1 \rightarrow \text{Signal } T_2 \]

Additionally: requirement.

are first class entities in Yampa:

\[ \text{SF } \alpha \beta \cong \text{Signal } \alpha \rightarrow \text{Signal } \beta \]

Signal functions and state

Alternative view:

Signal functions can encapsulate state.

\[ x(t) \xrightarrow{f} [\text{state}(t)] \rightarrow y(t) \]

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

Functions on signals are either:
- \( y(t) \) depends on \( x(t) \) and \( \text{state}(t) \)
- \( y(t) \) depends only on \( x(t) \)

Building systems (1)

How to build systems? Think of a signal function as a block. Blocks have inputs and outputs and can be combined into larger blocks. For example, serial composition:

\[ (\text{Serial Combinator}) : :: \text{SF } \alpha \beta \rightarrow \text{SF } \gamma \delta \rightarrow \text{SF } \alpha \delta \]

A combinator can be defined that captures this idea:

\[ (\text{Serial Combinator}) : :: \text{SF } \alpha \beta \rightarrow \text{SF } \gamma \delta \rightarrow \text{SF } \alpha \delta \]

Example: Video tracker

Video trackers are typically stateful signal functions:

Building systems (2)

But systems can be complex:
Arrows

Yampa uses John Hughes’ framework:
- Abstract data type interface for function-like types (or “blocks”, if you prefer).
- Particularly suitable for types representing process-like computations.
- Provides a minimal set of “wiring” combinators.

What is an arrow? (1)
- A \( a \) of arity two.
- Three operators:
  - \( \text{arr} : (b \to c) \to a \ b \ c \)
  - \( \text{first} : a \ b \ c \to a \ (b,d) \ (c,d) \)
  - A set of algebraic laws that must hold.

Some arrow laws
\[
(f \ggg g) \ggg h = f \ggg (g \ggg h)
\]
\[
\text{arr} (f \ggg g) = \text{arr} f \ggg \text{arr} g
\]
\[
\text{arr} \text{id} \ggg f = f
\]
\[
f = f \ggg \text{arr} \text{id}
\]
\[
\text{first} (f \ggg g) = \text{arr} (\text{first} f)
\]
\[
\text{first} (f \ggg g) = \text{first} f \ggg \text{first} g
\]

Some more arrow combinators (1)
- \( \text{second} :: \text{Arrow} a \Rightarrow a \ b \ c \to a \ (d,b) \ (d,c) \)
- \( \text{third} :: \text{Arrow} a \Rightarrow a \ b \ c \to a \ (d,e) \ (e,f) \)
- \( \text{first} :: \text{Arrow} a \Rightarrow a \ b \ c \to a \ (b,d) \ (c,d) \)
- \( \text{third} :: \text{Arrow} a \Rightarrow a \ b \ c \to a \ (b,d) \ (e,f) \)
- \( \text{second} :: \text{Arrow} a \Rightarrow a \ b \ c \to a \ (d,b) \ (c,d) \)

What is an arrow? (2)
- These diagrams convey the general idea:

The \textit{loop} combinator
- Another important operator is \textit{loop}: a fixed-point operator used to express recursive arrows or

Some more arrow combinators (2)
- As diagrams:

Example: A Simple Network
- A simple network:

The arrow \textit{do} notation (1)
- Using the basic combinators directly can be cumbersome. Ross Paterson’s \textit{do}-notation for arrows provides a convenient alternative. Only!

\[
\text{proc pat} \Rightarrow \text{do} \{ \text{rec} \}
\]
\[
\text{pat}_1 \leftarrow \text{sfexp}_1 \leftarrow \text{exp}_1
\]
\[
\text{pat}_2 \leftarrow \text{sfexp}_2 \leftarrow \text{exp}_2
\]
\[
\ldots
\]
\[
\text{pat}_n \leftarrow \text{sfexp}_n \leftarrow \text{exp}_n
\]
\[
\text{returnA} \leftarrow \text{exp}
\]

Also: \( \text{let pat = exp} \equiv \text{pat} \leftarrow \text{arr id} \leftarrow \text{exp} \)
The arrow do notation (2)

Let us redo the example using this notation:

\[
\text{circuit}_v4 :: \text{A Double Double} \\
\text{circuit}_v4 = \text{proc } x \rightarrow \text{do} \\
y1 \leftarrow a1 \leftarrow x \\
y2 \leftarrow a2 \leftarrow y1 \\
y3 \leftarrow a3 \leftarrow x \\
\text{returnA} \leftarrow y2 + y3
\]

Yampa and Arrows

The Yampa signal function type is an arrow. Signal function instances of the core combinators:

- `arr :: (a -> b) -> SF a b`
- `<<< :: SF a b -> SF b c -> SF a c`
- `first :: SF a b -> SF (a,c) (b,c)`
- `loop :: SF (a,c) (b,c) -> SF a b`

Some further basic signal functions

- `identity :: SF a a`
- `constant :: b -> SF a b`
- `integral :: VectorSpace a s->SF a a`
- `time :: SF a Time`
- `(<=) :: (b->c) -> SF a b -> SF a c`
- `f (<=) sf = sf >>> arr f`

A bouncing ball

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

Modelling the bouncing ball: part 1

Free-falling ball:

\[
\text{type Pos = Double} \\
\text{type Vel = Double} \\
\text{fallingBall ::} \\
\text{Pos -> Vel -> SF () (Pos, Vel)} \\
\text{fallingBall y0 v0 = proc () -> do} \\
v y \leftarrow (v0 +) \langle< \text{integral} -< -9.81 \\
y v \leftarrow (y0 +) \langle< \text{integral} -< v \\
\text{returnA} \leftarrow (y, v)
\]

Events

Conceptually, signals are only defined at discrete points in time, often associated with the occurrence of some event. Yampa models discrete-time signals by lifting the of continuous-time signals:

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

Associating information with an event occurrence:

\[
\text{tag :: Event a -> b -> Event b}
\]

Modelling the bouncing ball: part 2

Detecting when the ball goes through the floor:

\[
\text{fallingBall'} :: \\
\text{Pos -> Vel -> SF () ((Pos,Vel), Event (Pos,Vel))} \\
\text{fallingBall'} y0 v0 = proc () -> do} \\
\text{yv@(y, _)} \leftarrow \text{fallingBall y0 v0} \langle< () \\
\text{hit} \leftarrow \text{edge} \langle< y \leftarrow 0 \\
\text{returnA} \leftarrow (yv, \text{hit 'tag' yv})
\]

Switching

Q: How and when do signal functions “start”? A: • “apply” a signal functions to its input signal at some point in time. • This creates a “running” signal function • The new signal function instance often replaces the previously running instance. Switchers thus allow systems with 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.

\[
\text{switch ::} \\
\text{SF a (b, Event c) -> (c -> SF a b) -> SF a b}
\]
Modelling the bouncing ball: part 3

Making the ball bounce:

```haskell
bouncingBall :: Pos -> SF () (Pos, Vel)
bouncingBall y0 = bbAux y0 0.0
where
  bbAux y0 v0 = switch (fallingBall' y0 v0) $ \(y,v) ->
  bbAux y (-v)
```

Dynamic signal function collections

Idea:

- Switch over of signal functions.
- On event, “freeze” running signal functions into collection of signal function, preserving encapsulated state.
- Modify collection as needed and switch back in.

Describing the alien behavior (1)

```haskell
type Object = SF ObjInput ObjOutput
alien :: RandomGen g =>
  g -> Position2 -> Velocity -> Object
alien g p0 vyd = proc oi -> do
  rec
    rx <- noiseR (xMin, xMax) g <- ()
    smpl <- occasionally g 5 () <- ()
    xd <- hold (point2X p0) <- smpl 'tag' rx ...
```

Describing the alien behavior (2)

```haskell...
let axd = 5 * (xd - point2X p)
    - 3 * (vector2X v)
ayd = 20 * (vyd - (vector2Y v))
ad = vector2 axd ayd
h = vector2Theta ad
```

Describing the alien behavior (3)

```haskell...
let a = vector2Polar
    (min alienAccMax
      (vector2Rho ad))
  h
  vp <- iPre v0 -- v
  ffi <- forceField << (p, vp)
v <- (v0 +) "<< impulseIntegral
    (gravity "+" a, ffi)
p <- (p0 +) "<< integral << v
```

Simulation of bouncing ball

Example: Space Invaders

Overall game structure

Highly dynamic system structure?

Basic switch allows one signal function to be replaced by another.

- What about more general structural changes?
- What about state?
Describing the alien behavior (4)

```plaintext
sl <- shield <- oHit o1
die <- edge <- sl <= 0

returnA <- ObjOutput {
  ooObsObjState = oosAlien p h v,
  ooKillReq   = die,
  ooSpawnReq  = noEvent
}
```

where

```plaintext
v0 = zeroVector
```

State in alien

Each of the following signal functions used in alien encapsulate state:

- noiseR
- impulseIntegral
- occasionally
- integral
- hold
- shield
- iPre
- edge
- forceField

Why not imperative, then?

If state is so important, why not stick to imperative/object-oriented programming where we have “state for free”? 

- Advantages of declarative programming retained:
  - High abstraction level.
  - Referential transparency, algebraic laws: formal reasoning is simpler.
- Synchronous approach avoids “event-call-back soup”, meaning robust, easy-to-understand semantics.