Reactive programming

**Reactive systems:**
- Input arrives *incrementally* while system is running.
- Output is generated in response to input in an interleaved and *timely* fashion.

Contrast **transformational systems**.

The notions of
- time
- time-varying values, or *signals*

are inherent and central for reactive systems.

---

Functional Reactive Programming

What is Functional Reactive Programming (FRP)?
- Paradigm for reactive programming in a functional setting.
- Originated from Functional Reactive Animation (Fran) (Elliott & Hudak).
- Has evolved in a number of directions and into different concrete implementations.
FRP applications

Some domains where FRP has been used:
- Graphical Animation (Fran: Elliott, Hudak)
- Robotics (Frob: Peterson, Hager, Hudak, Elliott, Pembeci, Nilsson)
- Vision (FVision: Peterson, Hudak, Reid, Hager)
- GUIs (Fruit: Courtney)
- Hybrid modeling (Nilsson, Hudak, Peterson)

Key FRP features

- First class reactive components.
- Synchronous: all system parts operate in synchrony.
- Support for hybrid (mixed continuous and discrete time) systems.
- Allows dynamic system structure.

Related languages and paradigms

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

Yampa

What is Yampa?
- The most recent Yale FRP implementation.
  People:
  - Antony Courtney
  - Paul Hudak
  - Henrik Nilsson
  - John Peterson
- A Haskell combinator library, a.k.a. Domain-Specific Embedded Language (DSEL).
Yampa

What is Yampa?

- Structured using arrows.
- Continuous-time signals (conceptually)
- Option type Event to handle discrete-time signals.
- Advanced switching constructs to describe systems with dynamic structure.

Signal functions (1)

Key concept: functions on signals.

Intuition:

\[
\begin{align*}
\text{Signal } \alpha & \approx \text{Time} \to \alpha \\
x & :: \text{Signal T1} \\
y & :: \text{Signal T2} \\
f & :: \text{Signal T1} \to \text{Signal T2}
\end{align*}
\]

Signal functions (2)

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

Signal functions are said to be

- pure or stateless if output at time \( t \) only depends on input at time \( t \)
- impure or stateful if output at time \( t \) depends on input over the interval \([0, t]\).

A good metaphor for hybrid systems!
Signal functions in Yampa

- **Signal functions** are *first class entities.*
  Intuition: $SF \alpha \beta \approx \text{Signal } \alpha \rightarrow \text{Signal } \beta$
- **Signals** are *not* first class entities: they only exist indirectly through signal functions.

Signal functions and state

Alternative view:

Signal functions can encapsulate *state.*

$\begin{align*}
x(t) & \rightarrow f \\
\{\text{state}(t)\} & \rightarrow y(t)
\end{align*}$

$\text{state}(t)$ summarizes input history $x(t')$, $t' \in [0, t]$. Thus, really a kind of *process.*

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 $x(t)$

Example: Video tracker

Video trackers are typically stateful signal functions:

Video stream $\rightarrow$ Tracker $\{\text{prev. pos.}\}$ $\rightarrow$ (234,192) $\rightarrow$ Tracked object position

Example: Robotics (1)

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

Hardware setup:
Example: Robotics (2)

Software architecture:

<table>
<thead>
<tr>
<th>Application</th>
<th>FVision</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frob</td>
<td></td>
</tr>
<tr>
<td>FRP (Yampa)</td>
<td></td>
</tr>
<tr>
<td>Pioneer drivers</td>
<td></td>
</tr>
<tr>
<td>XVision2</td>
<td></td>
</tr>
</tbody>
</table>

Haskell

C/C++

Example: Robotics (3)

Yampa and Arrows (1)

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

For example, serial composition:

\[
(f \circ g)
\]

A combinator can be defined that captures this idea:

\[
(\gg \gg \gg) :: SF\ a\ b \to SF\ b\ c \to SF\ a\ c
\]

Yampa and Arrows (2)

But systems can be complex:

How many and what combinators do we need to be able to describe arbitrary systems?
John Hughes’ *arrow* framework:
- Abstract data type interface for function-like types.
- 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.

A *type constructor* \( a \) of arity two.

Three operators:
- **lifting**:
  \[ \text{arr} :: (b \to c) \to a \, b \, c \]
- **composition**:
  \[ (>>&>&) :: a \, b \, c \to a \, c \, d \to a \, b \, d \]
- **widening**:
  \[ \text{first} :: a \, b \, c \to a \, (b, d) \, (c, d) \]

A set of *algebraic laws* that must hold.

In Haskell, a *type class* is used to capture these ideas (except for the laws):

```haskell
class Arrow a where
  arr :: (b -> c) -> a b c
  (>>>) :: a b c -> a c d -> a b d
  first :: a b c -> a (b, d) (c, d)
```
Functions are arrows (1)

Functions are a simple example of arrows. The arrow type constructor is just \( \rightarrow \) in that case.

**Exercise 1:** Suggest suitable definitions of

- \( \text{arr} \)
- \( (\ggg) \)
- \( \text{first} \)

for this case!

(We have not looked at what the laws are yet, but they are “natural”.)

Functions are arrows (2)

Solution:

- \( \text{arr} = \text{id} \)
  To see this, recall
  
  \[
  \text{id} :: t \rightarrow t \\
  \text{arr} :: (b \rightarrow c) \rightarrow a \ b \ c
  \]

  Instantiate with

  \[
  a = (\rightarrow) \\
  t = b \rightarrow c = (\rightarrow) \ b \ c
  \]

Functions are arrows (3)

- \( f \ggg g = \lambda a \rightarrow g (f \ a) \)  \( \text{or} \)
- \( f \ggg g = g \ . \ f \)  \( \text{or even} \)
- \( (\ggg) = \text{flip} \ (. \) \\
- \( \text{first} \ f = \lambda (b,d) \rightarrow (f \ b,d) \)

Functions are arrows (4)

**Arrow instance declaration for functions:**

\[
\begin{align*}
\text{instance Arrow (\rightarrow)} & \text{ where} \\
\text{arr} & = \text{id} \\
(\ggg) & = \text{flip} \ (. \) \\
\text{first} \ f & = \lambda (b,d) \rightarrow (f \ b,d)
\end{align*}
\]
Arrow laws

\[(f >>> g) >>> h = f >>> (g >>> h)\]
\[\text{arr } (f >>> g) = \text{arr } f >>> \text{arr } g\]
\[\text{arr } \text{id} >>> f = f\]
\[f = f >>> \text{arr } \text{id}\]
\[\text{first } (\text{arr } f) = \text{arr } (\text{first } f)\]
\[\text{first } (f >>> g) = \text{first } f >>> \text{first } g\]

**Exercise 2:** Draw diagrams illustrating the first and last law!

The loop combinator (1)

Another important operator is \text{loop}: a fixed-point operator used to express recursive arrows or feedback:

\[\text{loop } f\]

The loop combinator (2)

Not all arrow instances support \text{loop}. It is thus a method of a separate class:

\[
\text{class } \text{Arrow } a \Rightarrow \text{ArrowLoop } a \text{ where }
\]
\[
\text{loop} :: a (b, d) (c, d) \rightarrow a b c
\]

Remarkably, the four combinators \text{arr}, >>>>, first, and \text{loop} are sufficient to express any conceivable wiring!

Some more arrow combinators (1)

\text{second} :: \text{Arrow } a => a b c \rightarrow a (d, b) (d, c)

\text{(***)} :: \text{Arrow } a => a b c \rightarrow a d e \rightarrow a (b, d) (c, e)

\text{(&&)} :: \text{Arrow } a => a b c \rightarrow a b d \rightarrow a b (c, d)
Some more arrow combinators (2)

As diagrams:

\[ \text{second } f \]

\[ f \text{ *** } g \]

\[ f \text{ &&& } g \]

Some more arrow combinators (3)

Exercise 3: Describe the following circuit using arrow combinators:

\[ \begin{array}{c}
\text{a1} \\
\text{a2} \\
\text{a3}
\end{array} \]

\[ a1, a2, a3 :: \text{A Double Double} \]

Exercise 4: The combinators \text{second}, (***), and (&&&) are not primitive, but defined in terms of \text{arr}, (>>, and \text{first}. Suggest suitable definitions!

Reading (1)


Reading (2)