An Introduction to Functional Reactive Programming Lecture 1 (of 2)

Neil Sculthorpe

Functional Programming Group
Information and Telecommunication Technology Center
University of Kansas
neil@ittc.ku.edu

EECS 776
Lawrence, Kansas
26th October 2012

Reactive Programming

- Reactive Program: continually interacts with its environment in a timely manner.
- Examples: video games, robot controllers, aeroplane systems . . .
- Contrast with:
  - Transformational Programs, e.g. a compiler
  - Interactive Programs, e.g. accessing a database

Functional Reactive Programming (FRP)

- FRP languages are domain-specific languages (the domain being reactive programming)
- Key characteristic: inherent notion of time
- Usually embedded in a host language (often Haskell)
- Also useful for modelling and simulation

Modelling Time

- The original idea of FRP was to provide a continuous-time abstraction to the FRP programmer...

Modelling Time

- The original idea of FRP was to provide a continuous-time abstraction to the FRP programmer...
- ...while automating the discretisation necessary for implementation.

Modelling Time

- The original idea of FRP was to provide a continuous-time abstraction to the FRP programmer...
- ...while automating the discretisation necessary for implementation.
- In practice:
  - FRP languages vary in how well they preserve this abstraction;
  - while some abandon it altogether.

Signals and Events

- FRP is based around time-varying values called signals:

$$\textbf{type}\ \mathit{Signal}\ a \approx \mathit{Time} \rightarrow a$$

Signals and Events

- FRP is based around time-varying values called signals:

  $\textbf{type}\ \mathit{Signal}\ a \approx \mathit{Time} \rightarrow a$

- There are also instantaneous occurrences called events.

Signals and Events

- FRP is based around time-varying values called signals:

  type $Signal\ a \approx Time \rightarrow a$

- There are also instantaneous occurrences called events.
- One (imperfect) way to represent events is as signals carrying *Maybe* types:

  type $EventSignal\ a \approx Signal\ (Maybe\ a)$

Signal Functions

- In FRP languages:
  - signals are abstract
  - signals, and functions on signals, are provided as primitives

Signal Functions

- In FRP languages:
  - signals are abstract
  - signals, and functions on signals, are provided as primitives
- Several advantages, e.g.
  - enforcing causality
  - optimisation opportunities

Signal Functions

- In FRP languages:
  - signals are abstract
  - signals, and functions on signals, are provided as primitives
- Several advantages, e.g.
  - enforcing causality
  - optimisation opportunities
- Some languages go further and only provide functions on signals as a first-class abstraction.

Signal Functions

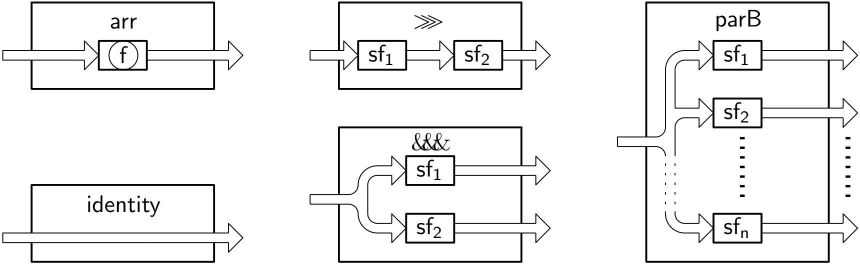
- In FRP languages:
  - signals are abstract
  - signals, and functions on signals, are provided as primitives
- Several advantages, e.g.
  - enforcing causality
  - optimisation opportunities
- Some languages go further and only provide functions on signals as a first-class abstraction.
- These are called signal functions:

$$\textbf{type}\ SF\ a\ b \approx Signal\ a \rightarrow Signal\ b$$

Yampa: An FRP Language

- A DSL embedded in Haskell
- No signals, only signal functions
- Pretends to have continuous time

Basic Yampa Routing Combinators



$$
\begin{aligned}
&arr && :: (a \rightarrow b) \rightarrow SF\ a\ b \\
&identity && :: SF\ a\ a \\
&(\ggg) && :: SF\ a\ b \rightarrow SF\ b\ c \rightarrow SF\ a\ c \\
&(\&\&\&) && :: SF\ a\ b \rightarrow SF\ a\ c \rightarrow SF\ a\ (b, c) \\
&parB && :: [SF\ a\ b] \rightarrow SF\ a\ [b]
\end{aligned}
$$

Some Yampa Primitives

Events

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

Some Yampa Primitives

Events

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

$$\textbf{instance}\ Functor\ Event\ \textbf{where}$$
$$\quad fmap\ =\ \ldots$$

$$
\begin{aligned}
&tag &&:: Event\ a \to b \to Event\ b \\
&rMerge &&:: Event\ a \to Event\ a \to Event\ a \\
&catEvents &&:: [Event\ a] \to Event\ [a]
\end{aligned}
$$

Some Yampa Primitives

Events

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

$\textbf{instance}\ Functor\ Event\ \textbf{where}$
$\quad fmap\ =\ \ldots$

$tag :: Event\ a \to b \to Event\ b$

$rMerge :: Event\ a \to Event\ a \to Event\ a$

$catEvents :: [Event\ a] \to Event\ [a]$

Time-Dependent Primitives

$integral :: Num\ a \Rightarrow SF\ a\ a$

$delay :: Time \to a \to SF\ a\ a$

$edge :: SF\ Bool\ (Event\ ())$

$switch :: SF\ a\ (b, Event\ e) \to (e \to SF\ a\ b) \to SF\ a\ b$

Examples

Example Yampa Code

$$\mathit{constant} :: b \rightarrow \mathit{SF}\ a\ b$$
$$\mathit{constant}\ b = \mathit{arr}\ (\lambda\ \_ \rightarrow b)$$

Examples

Example Yampa Code

$$constant :: b \rightarrow SF\ a\ b$$
$$constant\ b = arr\ (\lambda\ \_ \rightarrow b)$$

$$localTime :: SF\ a\ Time$$
$$localTime = constant\ 1 \ggg integral$$

Examples

Example Yampa Code

$$constant :: b \rightarrow SF\ a\ b$$
$$constant\ b = arr\ (\lambda\ \_ \rightarrow b)$$

$$localTime :: SF\ a\ Time$$
$$localTime = constant\ 1 \ggg integral$$

$$after :: Time \rightarrow SF\ a\ (Event\ ())$$
$$after\ t = localTime \ggg arr\ (\geqslant t) \ggg edge$$

Examples

Example Yampa Code

$$constant :: b \rightarrow SF\ a\ b$$
$$constant\ b = arr\ (\lambda\ \_ \rightarrow b)$$

$$localTime :: SF\ a\ Time$$
$$localTime = constant\ 1 \ggg integral$$

$$after :: Time \rightarrow SF\ a\ (Event\ ())$$
$$after\ t = localTime \ggg arr\ (\geqslant t) \ggg edge$$

$$iIntegral :: Num\ x \Rightarrow x \rightarrow SF\ x\ x$$
$$iIntegral\ x = integral \ggg arr\ (+x)$$

Examples

Example Yampa Code

$$constant :: b \rightarrow SF\ a\ b$$
$$constant\ b = arr\ (\lambda\ \_ \rightarrow b)$$

$$localTime :: SF\ a\ Time$$
$$localTime = constant\ 1 \ggg integral$$

$$after :: Time \rightarrow SF\ a\ (Event\ ())$$
$$after\ t = localTime \ggg arr\ (\geqslant t) \ggg edge$$

$$iIntegral :: Num\ x \Rightarrow x \rightarrow SF\ x\ x$$
$$iIntegral\ x = integral \ggg arr\ (+x)$$

$$switchWhen :: SF\ a\ b \rightarrow SF\ b\ (Event\ e) \rightarrow (e \rightarrow SF\ a\ b) \rightarrow SF\ a\ b$$
$$switchWhen\ sf\ sfe = switch\ (sf \ggg (identity\ \&\&\&\ sfe))$$

Arrow Notation

- Yampa uses a special **do** notation (from the Arrow framework)

Arrow Notation

- Yampa uses a special **do** notation (from the Arrow framework)

Pure Code $(f :: a \rightarrow x)$

$$
\begin{array}{l}
\lambda\,(a, b) \rightarrow \\
\quad \textbf{let}\ x \;=\; f\ a \\
\quad \phantom{\textbf{let}}\ y \;=\; g\ (b, x) \\
\quad \textbf{in}\ (x, y, b)
\end{array}
$$

Arrow Notation

- Yampa uses a special **do** notation (from the Arrow framework)

Pure Code $(f :: a \rightarrow x)$

$$
\begin{array}{l}
\lambda\,(a,b) \rightarrow \\
\quad \mathbf{let}\ x = f\ a \\
\quad\quad\ \ y = g\ (b,x) \\
\quad \mathbf{in}\ (x,y,b)
\end{array}
$$

Monadic Code $(f :: a \rightarrow m\ x)$

$$
\begin{array}{l}
\lambda\,(a,b) \rightarrow \mathbf{do} \\
\quad x \leftarrow f\ a \\
\quad y \leftarrow g\ (b,x) \\
\quad \mathit{return}\ (x,y,b)
\end{array}
$$

Arrow Notation

- Yampa uses a special **do** notation (from the Arrow framework)

Pure Code $(f :: a \rightarrow x)$

$$
\begin{array}{l}
\lambda\,(a, b) \rightarrow \\
\quad \mathbf{let}\ x = f\ a \\
\quad \phantom{\mathbf{let}}\ y = g\ (b, x) \\
\quad \mathbf{in}\ (x, y, b)
\end{array}
$$

Arrow Code $(f :: SF\ a\ x)$

$$
\begin{array}{l}
\mathbf{proc}\ (a, b) \rightarrow \mathbf{do} \\
\quad x \leftarrow f \prec a \\
\quad y \leftarrow g \prec (b, x) \\
\quad returnA \prec (x, y, b)
\end{array}
$$

Arrow Notation

- Yampa uses a special **do** notation (from the Arrow framework)

Pure Code $(f :: a \rightarrow x)$

$$
\begin{array}{l}
\lambda\,(a,b) \rightarrow \\
\quad \textbf{let}\; x = f\; a \\
\quad \phantom{\textbf{let}}\; y = g\;(b,x) \\
\quad \textbf{in}\;(x,y,b)
\end{array}
$$

Arrow Code $(f :: SF\; a\; x)$

$$
\begin{array}{l}
\textbf{proc}\;(a,b) \rightarrow \textbf{do} \\
\quad x \leftarrow f \;\prec\; a \\
\quad y \leftarrow g \;\prec\; (b,x) \\
\quad returnA \;\prec\; (x,y,b)
\end{array}
$$

- Note: *returnA* and *identity* are semantically equivalent

Example: Bouncing Ball

See accompanying code. . .

Summary

- FRP languages are domain-specific languages for reactive programming.
- Their key characteristic is an implicit notion of time.
- Yampa is one specific implementation of FRP.
- Exercise: Add additional balls to the Bouncing Ball example.
  - Code available at `http://www.ittc.ku.edu/~neil/talks.html`
  - Email scripts to me by Friday 2nd November.