An Introduction to Functional Reactive Programming

Neil Sculthorpe

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

Lawrence, Kansas
30th March 2012

Reactive Programming

- Reactive Program: continually interacts with its environment in a timely manner.
- Examples: video games, mp3 players, robot controllers, aeroplane control systems ...
- Contrast with:
  - Interactive Programs, e.g. accessing a database
  - Transformational Programs, e.g. a compiler

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

What is Time?

What is Time?

''What then is time? If no one asks me, I know: if I wish to explain it to one that asketh, I know not.''
— St. Augustine, Confessions, 398AD.

What is Time?

''What then is time? If no one asks me, I know: if I wish to explain it to one that asketh, I know not.''
— St. Augustine, Confessions, 398AD.

- Two choices:
  - continuous time: $Time \approx \{t \geqslant 0 \mid t \in \mathbb{R}\}$
  - discrete time: $Time \approx \mathbb{N}$

What is Time?

''What then is time? If no one asks me, I know: if I wish to explain it to one that asketh, I know not.''
— St. Augustine, Confessions, 398AD.

- Two choices:
  - continuous time: $Time \approx \{t \geqslant 0 \mid t \in \mathbb{R}\}$
  - discrete time: $Time \approx \mathbb{N}$
- The original idea of FRP was to provide a continuous-time abstraction to the FRP programmer...

What is Time?

''What then is time? If no one asks me, I know: if I wish to explain it to one that asketh, I know not.''
— St. Augustine, Confessions, 398AD.

- Two choices:
  - continuous time: $Time \approx \{t \geqslant 0 \mid t \in \mathbb{R}\}$
  - discrete time: $Time \approx \mathbb{N}$
- The original idea of FRP was to provide a continuous-time abstraction to the FRP programmer...
- ...while automating the discretisation necessary for implementation.

Signals and Events

- FRP is based around time-varying values called signals (or behaviours):

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

Signals and Events

- FRP is based around time-varying values called signals (or behaviours):

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

- There are also instantaneous occurrences called events.

Signals and Events

- FRP is based around time-varying values called signals (or behaviours):

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

- There are also instantaneous occurrences called events.
- In a discrete-time setting, events can be embedded within signals:

$$Event\ a = Signal\ (Maybe\ a)$$

Signals and Events

- FRP is based around time-varying values called signals (or behaviours):

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

- There are also instantaneous occurrences called events.
- In a discrete-time setting, events can be embedded within signals:

$$Event\ a = Signal\ (Maybe\ a)$$

- In a continuous-time setting, they require a separate abstraction:

$$Event\ a \approx [(Time, a)]$$

Signal Functions

- FRP languages keep signals abstract, providing several signals, and functions on signals, as primitives.
- Some go further and only provide functions on signals as a first-class abstraction.
- These are called signal functions:

$$SF\ a\ b \approx Signal\ a \rightarrow Signal\ b$$

- The abstraction prevents many "bad" signal functions from being defined.
- E.g. causality can be enforced (the present cannot depend on the future).

Yampa: An FRP Language

- A DSL embedded in Haskell
- No signals, only signal functions
- Pretends to have continuous time
- Has been used for a variety of applications: video games, sound synthesis, robot simulators, GUIs, virtual reality, visual tracking, animal monitoring. . .

Some Yampa Primitives

Example Primitives

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

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

$delay :: Time \rightarrow a \rightarrow SF\ a\ a$

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

$tag :: Event\ a \rightarrow b \rightarrow Event\ b$

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

The Arrow Framework

Arrow Classes

class $Category\ (a :: * \rightarrow * \rightarrow *)$ **where**
  $id\ \ :: a\ b\ b$
  $(\circ) :: a\ c\ d \rightarrow a\ b\ c \rightarrow a\ c\ d$

class $Category\ a \Rightarrow Arrow\ (a :: * \rightarrow * \rightarrow *)$ **where**
  $arr\ \ :: (b \rightarrow c) \rightarrow a\ b\ c$
  $first :: a\ b\ c \rightarrow a\ (b, d)\ (c, d)$

class $Arrow\ a \Rightarrow ArrowLoop\ (a :: * \rightarrow * \rightarrow *)$ **where**
  $loop :: a\ (b, d)\ (c, d) \rightarrow a\ b\ c$

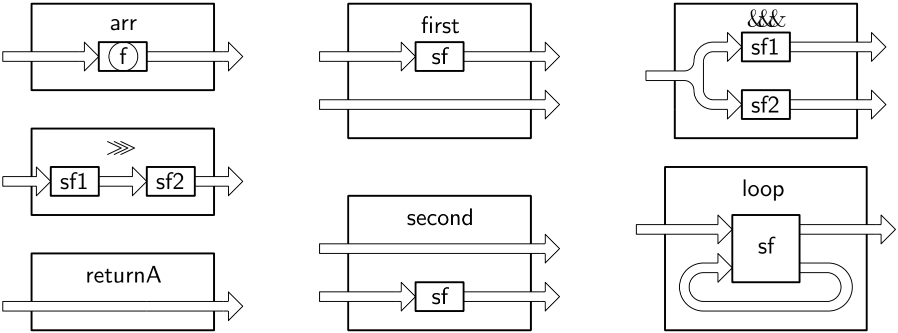
Yampa Signal Functions are Arrows

instance *Category SF* **where** ...

instance *Arrow SF* **where** ...

instance *ArrowLoop SF* **where** ...

Signal Functions Combinators



$$
\begin{aligned}
&\mathit{arr} &&:: (a \to b) \to SF\ a\ b \\
&(\ggg) &&:: SF\ a\ b \to SF\ b\ c \to SF\ a\ c \\
&\mathit{returnA} &&:: SF\ a\ a \\
&\mathit{first} &&:: SF\ a\ c \to SF\ (a,b)\ (c,b) \\
&\mathit{second} &&:: SF\ b\ c \to SF\ (a,b)\ (a,c) \\
&(\&\&\&) &&:: SF\ a\ b \to SF\ a\ c \to SF\ a\ (b,c) \\
&\mathit{loop} &&:: SF\ (a,c)\ (b,c) \to SF\ a\ b
\end{aligned}
$$

Examples

Example Signal Functions

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

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

Examples

Example Signal Functions

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

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

- See accompanying code for bouncing-ball example...

Yampa Implementation

The SF data type (simplified)

$$\textbf{data}\ SF\ a\ b \approx SF\ (DTime\ \rightarrow\ a\ \rightarrow\ (SF\ a\ b, b))$$

Yampa Implementation

The SF data type (simplified)

$$\textbf{data}\ SF\ a\ b \approx SF\ (DTime \rightarrow a \rightarrow (SF\ a\ b, b))$$

An alternative implementation

$$\textbf{data}\ SF\ a\ b \approx SF\ (DTime \rightarrow s \rightarrow a \rightarrow (s, b))$$

Push vs. Pull

- FRP languages usually take one of two implementation strategies:
  - Pull-driven implementations update at every time step (good for systems with continuously changing signals).
  - Push-driven implementations only update when an event occurs, and only the parts of the program that depend on that event (good for systems with signals that change at discrete points in time).
- In practice, FRP implementations contain a lot of optimisations to avoid unnecessary computation.
- But efficient implementation of FRP remains an open problem.

Summary

- FRP languages are domain-specific languages for reactive programming.
- Their key characteristic is an implicit notion of time.
- If you want to experiment with Yampa, I'd recommend Henrik Nilsson's recent mini-course: `http://www.cs.nott.ac.uk/~nhn/ITU-FRP2010/`