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
19th April 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:
  - Transformational Programs, e.g. a compiler
  - Interactive Programs, e.g. accessing a database

What type of program?

Greeting

```
greeting = do putStrLn "What is your first name?"
              n1 ← getLine
              putStrLn "And what is your family name?"
              n2 ← getLine
              putStrLn ("Hello " ⧺ n1 ⧺ " " ⧺ n2)
```

Insertion Sort

```
isort :: Ord a ⇒ [a] → [a]
isort []       = []
isort (x : xs) = insert x (isort xs)

insert :: Ord a ⇒ a → [a] → [a]
insert x []                   = [x]
insert x (a : as) | x >  a = a : insert x as
                  | x ⩽ a = x : a : as
```

What type of program?

Greeting — Interactive

```
greeting = do putStrLn "What is your first name?"
              n1 ← getLine
              putStrLn "And what is your family name?"
              n2 ← getLine
              putStrLn ("Hello " ⧺ n1 ⧺ " " ⧺ n2)
```

Insertion Sort — Transformational

```
isort :: Ord a ⇒ [a] → [a]
isort []       = []
isort (x : xs) = insert x (isort xs)

insert :: Ord a ⇒ a → [a] → [a]
insert x []                  = [x]
insert x (a : as) | x >  a = a : insert x as
                  | x ⩽ a = x : a : as
```

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.

- 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.

- 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.
- One way to represent events is as *Maybe* types within signals:

$$Signal\ (Maybe\ a)$$

Signal Functions

- FRP languages keep signals abstract, providing several signals, and functions on signals, as primitives.

Signal Functions

- FRP languages keep signals abstract, providing several signals, and functions on signals, as primitives.
- This has several advantages, e.g.
  - enforcing causality
  - optimisation opportunities

Signal Functions

- FRP languages keep signals abstract, providing several signals, and functions on signals, as primitives.
- This has 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

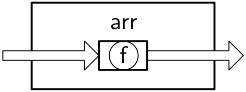
- FRP languages keep signals abstract, providing several signals, and functions on signals, as primitives.
- This has several advantages, e.g.
  - enforcing causality
  - optimisation opportunities
- Some languages go further and only provide functions on signals as a first-class abstraction.
- These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

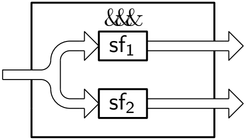
$$SF\ a\ b \approx Signal\ a \rightarrow Signal\ b$$

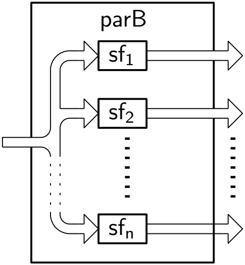
Yampa: An FRP Language

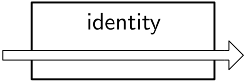
- A DSL embedded in Haskell
- No signals, only signal functions
- Pretends to have continuous time
- Has been used for a variety of applications: video games, sound synthesis, robot simulators, GUIs, virtual reality, visual tracking, animal monitoring. . .

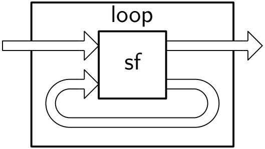
Yampa Routing Combinators

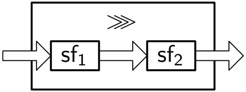












$$
\begin{array}{ll}
arr & :: (a \rightarrow b) \rightarrow SF\ a\ b \\
identity & :: SF\ a\ a \\
(\ggg) & :: SF\ a\ b \rightarrow SF\ b\ c \rightarrow SF\ a\ c \\
(\&\&\&) & :: SF\ a\ b \rightarrow SF\ a\ c \rightarrow SF\ a\ (b, c) \\
parB & :: [SF\ a\ b] \rightarrow SF\ a\ [b] \\
loop & :: SF\ (a,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

Some Yampa Primitives

Events

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

Some Yampa Primitives

Events

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

$$\mathit{tag} :: \mathit{Event}\ a \to b \to \mathit{Event}\ b$$

Some Yampa Primitives

Events

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

$$\mathit{tag} :: \mathit{Event}\ a \rightarrow b \rightarrow \mathit{Event}\ b$$

Time-Dependent Primitives

$$\mathit{integral} :: \mathit{Num}\ a \Rightarrow \mathit{SF}\ a\ a$$

$$\mathit{delay} :: \mathit{Time} \rightarrow a \rightarrow \mathit{SF}\ a\ a$$

$$\mathit{edge} :: \mathit{SF}\ \mathit{Bool}\ (\mathit{Event}\ ())$$

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

Examples

Example Yampa Code

```
localTime :: SF a Time
localTime  =  arr (const 1) ⋙ integral

after :: Time → SF a (Event ())
after t  =  localTime ⋙ arr (⩾ t) ⋙ edge

iIntegral :: Num x ⇒ x → SF x x
iIntegral x  =  integral ⋙ arr (+x)

switchWhen :: SF a b → SF b (Event e) → (e → SF a b) → SF a b
switchWhen sf sfe  =  switch (sf ⋙ (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 $(f :: a \rightarrow x)$

```
λ (a, b) →
  let x = f a
      y = g (b, x)
  in h (x, y, b)
```

Arrow Notation

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

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

```
λ (a, b) →
  let x = f a
      y = g (b, x)
  in h (x, y, b)
```

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

```
λ (a, b) → do
  x ← f a
  y ← g (b, x)
  h (x, y, b)
```

Arrow Notation

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

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

```
λ (a, b) →
  let x = f a
      y = g (b, x)
  in h (x, y, b)
```

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

```
proc (a, b) → do
  x ← f ≺ a
  y ← g ≺ (b, x)
  h     ≺ (x, y, b)
```

Bouncing Balls

See accompanying code. . .

Yampa Implementation

The SF data type (simplified)

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

($DTime$ is the amount of time passed since the previous sample.)

Summary

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