Functional Reactive Programming

Lecture 1: Introduction to FRP, Yampa, and Arrows

Henrik Nilsson

School of Computer Science and Information Technology
University of Nottingham, UK
Outline

- Brief introduction to FRP and Yampa
- Signal functions
- Arrows
Reactive programming

Reactive systems:

Input arrives incrementally while the system is running. Output is generated in response to input in an interleaved and timely fashion. Contrast transformational systems. The notions of time-varying values, or signals are inherent and central for reactive systems.
Reactive programming

*Reactive systems:*

- Input arrives *incrementally* while system is running.
Reactive programming

Reactive systems:

- Input arrives *incrementally* while system is running.
- Output is generated in response to input in an interleaved and *timely* fashion.
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.*
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.
What is Functional Reactive Programming (FRP)?

- Paradigm for reactive programming in a functional setting.
What is Functional Reactive Programming (FRP)?

- Paradigm for reactive programming in a functional setting.
- Originated from Functional Reactive Animation (Fran) (Elliott & Hudak).
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.
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.
Key FRP features

- First class reactive components.
- Synchronous: all system parts operate in synchrony.
Key FRP features

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

FRP related to:

- Synchronous languages, like Esterel, Lucid Synchrone.
- Modeling languages, like Simulink, Modelica.
What is *Yampa*?

- The most recent Yale FRP implementation.

People:
- Antony Courtney
- Paul Hudak
- Henrik Nilsson
- John Peterson
What is **Yampa**?

- The most recent Yale FRP implementation.
  People:
  - Antony Courtney
  - Paul Hudak
  - Henrik Nilsson
  - John Peterson

Yampa

What is Yampa?

- Structured using arrows.
Yampa

What is Yampa?

- Structured using arrows.
- Continuous-time signals (conceptually)
What is *Yampa*?

- Structured using *arrows*.
- *Continuous-time* signals (conceptually)
- Option type *Event* to handle discrete-time signals.
**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.
Yampa?
Yampa?

Yet
Another
Mostly
Pointless
Acronym
Yampa?

Yet
Another
Mostly
Pointless
Acronym

???
Yampa?

Yet
Another
Mostly
Pointless
Acronym

???

No . . .
Yampa is a river . . .
Yampa?

...with long calmly flowing sections...
Yampa?

... and abrupt whitewater transitions in between.

A good metaphor for hybrid systems!
Signal functions (1)

Key concept: *functions on signals.*
Key concept: **functions on signals**.

Intuition:

\[
\text{Signal } \alpha \approx \text{Time} \rightarrow \alpha
\]

\[
x :: \text{Signal } T1
\]

\[
y :: \text{Signal } T2
\]

\[
f :: \text{Signal } T1 \rightarrow \text{Signal } T2
\]
Signal functions (2)

Additionally, *causality* required: output at time $t$ must be determined by input on interval $[0, t]$. 
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 $t$
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 $[0, t]$. 
Signal functions in Yampa

- **Signal functions** are *first class entities*.
  
  Intuition: $\text{SF } \alpha \beta \approx \text{Signal } \alpha \rightarrow \text{Signal } \beta$
Signal functions in Yampa

- **Signal functions** are *first class entities*. Intuition: \( \text{SF } \alpha \beta \approx \text{Signal } \alpha \rightarrow \text{Signal } \beta \)

- **Signals** are *not* first class entities: they only exist indirectly through signal functions.
Signal functions and state

Alternative view:
Signal functions and state

Alternative view:

Signal functions can encapsulate state.

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

Thus, really a kind of process.
Signal functions and state

Alternative view:

Signal functions can encapsulate state.

\[ \text{state}(t) \] 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:

![Diagram of video tracker]

- Video stream
- Tracker [prev. pos.]
- Tracked object position

(234,192)
Example: Robotics (1)

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

Hardware setup:
Example: Robotics (2)

Software architecture:

- Application
  - Frob
  - FRP (Yampa)
  - Pioneer drivers
- FVision
- XVision2

Haskell

C/C++
Example: Robotics (3)
In Yampa, systems are described by combining signal functions (forming new signal functions).
In Yampa, systems are described by combining signal functions (forming new signal functions).

For example, serial composition:

```
A combinator can be defined that captures this idea:
(<<<) :: SF a b -> SF b c -> SF a c
```
In Yampa, systems are described by combining signal functions (forming new signal functions).

For example, serial composition:

A *combinator* can be defined that captures this idea:

\[
(\gggg) :: \text{SF } a \ b \rightarrow \text{SF } b \ c \rightarrow \text{SF } a \ c
\]
But systems can be complex:
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.
John Hughes’ *arrow* framework:

- Abstract data type interface for function-like types.
- Particularly suitable for types representing process-like computations.
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.
Yampa and Arrows (3)

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.
What is an arrow? (1)

- A type constructor $\text{a}$ of arity two.
What is an arrow? (1)

- A *type constructor* `a` of arity two.
- Three operators:
What is an arrow? (1)

- A type constructor $a$ of arity two.
- Three operators:
  - lifting:
    
    \[
    \text{arr :: } (b \to c) \to a \backslash b \backslash c
    \]
What is an arrow? (1)

- A type constructor $a$ of arity two.
- Three operators:
  - **lifting**: $arr :: (b \rightarrow c) \rightarrow a \ b \ c$
  - **composition**: $(\\gg\\gg) :: a \ b \ c \rightarrow a \ c \ d \rightarrow a \ b \ d$
What is an arrow? (1)

- A **type constructor** `a` of arity two.
- Three operators:
  - **lifting**:
    \[
    \text{arr} :: (b \to c) \to a \ b \ c
    \]
  - **composition**:
    \[
    (\ggg) :: a \ b \ c \to a \ c \ d \to a \ b \ d
    \]
  - **widening**:
    \[
    \text{first} :: a \ b \ c \to a \ (b, d) \ (c, d)
    \]
What is an arrow? (1)

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

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

- A set of **algebraic laws** that must hold.
What is an arrow? (2)

These diagrams convey the general idea:

- **arr f**
- **f >>> g**
- **first f**
The **Arrow class**

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 \((\to)\) in that case.

**Exercise 1:** Suggest suitable definitions of

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

for this case!

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

- \texttt{arr} = \texttt{id}
Solution:

- \( \text{arr} = \text{id} \)

To see this, recall

\[
\text{id} :: \ t \rightarrow t
\]

\[
\text{arr} :: (b \rightarrow c) \rightarrow a \ b \ c
\]
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)

- $f >>> g = \lambda a \rightarrow g(fa)$
Functions are arrows (3)

- \( f >>> g = \lambda a \rightarrow g (f a) \) \text{ or } \( f >>> g = g \circ f \)
- \( f >>>> g = (b, d) \rightarrow (f b, d) \)
Functions are arrows (3)

- $f >>> g = \lambda a \rightarrow g(f a)$  \textit{or}
- $f >>> g = g \circ f$  \textit{or even}
- $(>>>) = \text{flip}(.)$
Functions are arrows (3)

- \( f >>> g = \lambda a \rightarrow g (f a) \quad \text{or} \)
- \( f >>> g = g \circ f \quad \text{or even} \)
- \( (>>>) = \text{flip} \ (. \) \)
- \( \text{first } f = \lambda (b, d) \rightarrow (f \ b, d) \)
Arrow instance declaration for functions:

instance Arrow (->) where
  arr = id
  (>>>) = flip (.)
  first f = \( (b,d) \rightarrow (f b,d) \)
Arrow laws

\[(f >>> g) >>> h = f >>> (g >>> h)\]

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

\[\text{arr} \text{id} >>> f = f >>> \text{arr} \text{id}\]

\[\text{first}(\text{arr} f) = \text{arr} (\text{first} f)\]

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

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

\[(f >>> g) >>> h = f >>> (g >>> h)\]
\[\text{arr} (f >>> g) = \text{arr} f >>> \text{arr} g\]
Arrow laws

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

\[(f >>> g) >>> h = f >>> (g >>> h)\]

\[arr (f >>> g) = arr f >>> arr g\]

\[arr id >>> f = f\]

\[f = f >>> arr id\]

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

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

\[(f \gggg g) \gggg h = f \gggg (g \gggg h)\]
\[\text{arr } (f \gggg g) = \text{arr } f \gggg \text{arr } g\]
\[\text{arr } \text{id} \gggg f = f\]
\[f = f \gggg \text{arr } \text{id}\]
\[\text{first } (\text{arr } f) = \text{arr } (\text{first } f)\]
\[\text{first } (f \gggg g) = \text{first } f \gggg \text{first } g\]
Arrow laws

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

**Exercise 2:** Draw diagrams illustrating the first and last law!
Another important operator is loop: a fixed-point operator used to express recursive arrows or feedback:

\[ \text{loop } f \]
The \texttt{loop} combinator (2)

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

\begin{verbatim}
class Arrow a => ArrowLoop a where
  loop :: a (b, d) (c, d) -> a b c
\end{verbatim}

Remarkably, the four combinators \texttt{arr}, \texttt{>>>}, \texttt{first}, and \texttt{loop} are sufficient to express any conceivable wiring!
Some more arrow combinators (1)

second :: Arrow a =>
          a b c -> a (d,b) (d,c)

(***) :: Arrow a =>
          a b c -> a d e -> a (b,d) (c,e)

(&&&) :: Arrow a =>
          a b c -> a b d -> a b (c,d)
Some more arrow combinators (2)

As diagrams:

second $f$

$f$ $\&\&\&$ $g$

$f$ $\&\&\&$ $g$
Some more arrow combinators (3)

Exercise 3: Describe the following circuit using arrow combinators:

```
  a1  a2  a3 :: A Double Double
```

Exercise 4: The combinators second, (***), and (&&&) are not primitive, but defined in terms of arr, (>>>), and first. Suggest suitable definitions!
Reading (1)


Reading (2)