A Brief Introduction to Functional Reactive Programming and Yampa

*FoP Away Day 17 January 2007*

Henrik Nilsson

School of Computer Science and Information Technology

University of Nottingham, UK
What is Functional Reactive Programming (FRP)?

- Umbrella-term for functional approach to programming reactive systems.
What is Functional Reactive Programming (FRP)?

- Umbrella-term for functional approach to programming reactive systems.
- Originated from Functional Reactive Animation (Fran) (Elliott & Hudak).
What is Functional Reactive Programming (FRP)?

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

- Umbrella-term for functional approach to programming reactive systems.
- Originated from Functional Reactive Animation (Fran) (Elliott & Hudak).
- Has evolved in a number of directions and into different concrete implementations.
- **Yampa**: An FRP implementation in the form of a Haskell combinator library, a.k.a. Domain-Specific Embedded Language (DSEL).
Signal functions

Key concept: *functions on signals.*
Signal functions

Key concept: *functions on signals*.

Intuition:

Signal $\alpha \approx \text{Time} \rightarrow \alpha$

$x :: \text{Signal T1}$

$y :: \text{Signal T2}$

$\text{SF } \alpha \beta \approx \text{Signal } \alpha \rightarrow \text{Signal } \beta$

$f :: \text{SF T1 T2}$
Key concept: *functions on signals.*

Intuition:

\[ \text{Signal } \alpha \approx \text{Time} \rightarrow \alpha \]
\[ x :: \text{Signal T1} \]
\[ y :: \text{Signal T2} \]
\[ \text{SF } \alpha \beta \approx \text{Signal } \alpha \rightarrow \text{Signal } \beta \]
\[ f :: \text{SF T1 T2} \]

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

Alternative view:
Signal functions and state

Alternative view:

Signal functions can encapsulate state.

\[ state(t) \] summarizes input history \( x(t'), t' \in [0, t] \).
Signal functions and state

Alternative view:

Signal functions can encapsulate \textit{state}.

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

From this perspective, signal functions are:

- \textbf{stateful} if \[ y(t) \] depends on \[ x(t) \] and \[ state(t) \]
- \textbf{stateless} if \[ y(t) \] depends only on \[ x(t) \]
Programming with signal functions

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:
Programming with 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:

\[(\ggg) :: SF\ a\ b \rightarrow SF\ b\ c \rightarrow SF\ a\ c\]
What about larger networks?
How many combinators are needed?
What about larger networks? How many combinators are needed?

John Hughes’s *Arrow* framework provides a good answer!
The Arrow framework (1)

These diagrams convey the general idea:

\[ \text{arr } f \]

\[ \text{first } f \]

\[ \text{loop } f \]

\[ \text{first } :: \ SF \ a \ b \rightarrow SF \ (a, c) \ (b, c) \]

\[ \text{loop } :: \ SF \ (a, c) \ (b, c) \rightarrow SF \ a \ b \]
The Arrow framework (2)

Some derived combinators:

\( \text{second } f \)  
\( f \&\& g \)  
\( f \& g \)
Example: Constructing a network
Example: Constructing a network

\( \text{loop} \)

\( \text{fst} \)

\( f \)

\( g \)

\( h \)

\( \cdots \)

\( \cdots \)

\( \cdots \)
Example: Constructing a network

Let's consider the following function:

\[ \text{loop} \left( \text{arr} \left( \lambda(x, y) \rightarrow ((x, y), x) \right) \right) \]

\[ \ggg (\text{fst} \ f) \]

\[ \ggg \left( \text{arr} \left( \lambda(x, y) \rightarrow (x, (x, y)) \right) \ggg (g \ circ h) \right) \]
The Arrow notation

\[ f \rightarrow g \rightarrow h \]

FoPAD: Brief Introduction to FRP & Yampa – p.10/14
The Arrow notation

\[ x \xrightarrow{f} u \xrightarrow{g} y \]

\[ x \xrightarrow{h} v \]
The Arrow notation

proc $x \rightarrow$ do

rec

$u \leftarrow f \leftarrow (x, v)$

$y \leftarrow g \leftarrow u$

$v \leftarrow h \leftarrow (u, x)$

$return A \leftarrow y$
How does it work?

- Essentially:

```haskell
newtype SF a b = SF (DeltaTime → a → (SF a b, b))
```
How does it work?

- Essentially:

  \[
  \text{newtype } SF\ a\ b = \ SF\ (\text{DeltaTime} \rightarrow \ a \rightarrow (SF\ a\ b, b))
  \]

- A top-level loop, *reactimate*, drives the computation.
How does it work?

• Essentially:

\[
\text{newtype } SF \ a \ b = \\
SF \ (\text{DeltaTime} \to a \to (SF \ a \ b, b))
\]

• A top-level loop, \textit{reactimate}, drives the computation.

Note that the system representation in principle is reconstructed at every time step.
Related languages and paradigms

FRP/Yampa related to:

- Synchronous dataflow languages, like Esterel, Lucid Synchrone.
Related languages and paradigms

FRP/Yampa related to:

- Synchronous dataflow languages, like Esterel, Lucid Synchrone.
- Modeling languages, like Simulink, Modelica.
What makes Yampa interesting?

- First class reactive components (signal functions).
What makes Yampa interesting?

- First class reactive components (signal functions).
- Supports hybrid (mixed continuous and discrete time) systems: option type `Event` represents discrete-time signals.
What makes Yampa interesting?

- First class reactive components (signal functions).
- Supports hybrid (mixed continuous and discrete time) systems: option type `Event` represents discrete-time signals.
- Supports dynamic system structure through `switching combinators`: 

![Diagram of switching combinators]
Example: Space Invaders