# The essence and origins of FRP

*or*

## How you could have invented Functional Reactive Programming

Conal Elliott

LambdaJam 2015

# What is FRP?

# FRP's two fundamental properties

- Precise, simple denotation. (Elegant & rigorous.)
- *Continuous* time. (Natural & composable.)

# FRP's two fundamental properties

- Precise, simple denotation. (Elegant & rigorous.)
- *Continuous* time. (Natural & composable.)

Deterministic, continuous "concurrency".

# FRP's two fundamental properties

- Precise, simple denotation. (Elegant & rigorous.)
- *Continuous* time. (Natural & composable.)

Deterministic, continuous "concurrency".

Warning: most modern "FRP" systems have neither property. ☹

# FRP's two fundamental properties

- Precise, simple denotation. (Elegant & rigorous.)
- *Continuous* time. (Natural & composable.)

FRP *is not* about:

# FRP's two fundamental properties

- Precise, simple denotation. (Elegant & rigorous.)
- *Continuous* time. (Natural & composable.)

FRP *is not* about:

- graphs,
- updates and propagation,
- streams,
- doing

# Why (precise & simple) denotation?

# Why (precise & simple) denotation?

- Separates specification from implementation.
- *Simple* so that we *can* reach conclusions.
- *Precise* so that our conclusions will be *true*.
- Denotations have elegant, functional-friendly style.

# Why (precise & simple) denotation?

- Separates specification from implementation.
- *Simple* so that we *can* reach conclusions.
- *Precise* so that our conclusions will be *true*.
- Denotations have elegant, functional-friendly style.

An API is a language for communicating about a domain.

It helps to (really) understand what we're talking about.

# Why continuous & infinite (vs discrete/finite) time?

# Why continuous & infinite (vs discrete/finite) time?

Same benefits as for space (vector graphics):

# Why continuous & infinite (vs discrete/finite) time?

Same benefits as for space (vector graphics):

- Transformation flexibility with simple & precise semantics.
- Modularity/reusability/composability:
  - Fewer assumptions, more uses (resolution-independence).
  - More info available for extraction.
- Integration and differentiation: natural, accurate, efficient.

# Why continuous & infinite (vs discrete/finite) time?

Same benefits as for space (vector graphics):

- Transformation flexibility with simple & precise semantics.
- Modularity/reusability/composability:
  - Fewer assumptions, more uses (resolution-independence).
  - More info available for extraction.
- Integration and differentiation: natural, accurate, efficient.
- Quality/accuracy.
- Efficiency (adapative).
- Reconcile differing input sampling rates.

# Why continuous & infinite (vs discrete/finite) time?

Same benefits as for space (vector graphics):

- Transformation flexibility with simple & precise semantics.
- Modularity/reusability/composability:
  - Fewer assumptions, more uses (resolution-independence).
  - More info available for extraction.
- Integration and differentiation: natural, accurate, efficient.
- Quality/accuracy.
- Efficiency (adapative).
- Reconcile differing input sampling rates.

*Principle:* Approximations/prunings compose badly, so postpone.

See *Why Functional Programming Matters*.

# Semantics

Central abstract type: *Behavior a* — a “flow” of values.

# Semantics

Central abstract type: *Behavior a* — a "flow" of values.

Precise & simple semantics:

$$\mu :: \mathit{Behavior}\ a \rightarrow (T \rightarrow a)$$

where $T = \mathbb{R}$ (reals).

# Semantics

Central abstract type: *Behavior a* — a "flow" of values.

Precise & simple semantics:

$$\mu :: Behavior\ a \rightarrow (T \rightarrow a)$$

where $T = \mathbb{R}$ (reals).

Much of API and its specification can follow from this one choice.

# Original formulation

# API

$$
\begin{array}{ll}
\mathit{time} & :: \mathit{Behavior}\ T \\
\mathit{lift}_0 & :: a \to \mathit{Behavior}\ a \\
\mathit{lift}_1 & :: (a \to b) \to \mathit{Behavior}\ a \to \mathit{Behavior}\ b \\
\mathit{lift}_2 & :: (a \to b \to c) \to \mathit{Behavior}\ a \to \mathit{Behavior}\ b \to \mathit{Behavior}\ c \\
\mathit{timeTrans} & :: \mathit{Behavior}\ a \to \mathit{Behavior}\ T \to \mathit{Behavior}\ a \\
\mathit{integral} & :: \mathit{VS}\ a \Rightarrow \mathit{Behavior}\ a \to T \to \mathit{Behavior}\ a \\
\ldots &
\end{array}
$$

$$
\mathbf{instance}\ \mathit{Num}\ a \Rightarrow \mathit{Num}\ (\mathit{Behavior}\ a)\ \mathbf{where} \ldots
$$

$$\ldots$$

Reactivity later.

# Semantics

$$\begin{aligned}
\mu\ time &= \lambda t \rightarrow t \\
\mu\ (lift_0\ a) &= \lambda t \rightarrow a \\
\mu\ (lift_1\ f\ xs) &= \lambda t \rightarrow f\ (\mu\ xs\ t) \\
\mu\ (lift_2\ f\ xs\ ys) &= \lambda t \rightarrow f\ (\mu\ xs\ t)\ (\mu\ ys\ t) \\
\mu\ (timeTrans\ xs\ tt) &= \lambda t \rightarrow \mu\ xs\ (\mu\ tt\ t)
\end{aligned}$$

**instance** $Num\ a \Rightarrow Num\ (Behavior\ a)$ **where**

$$\begin{aligned}
fromInteger &= lift_0 \circ fromInteger \\
(+) &= lift_2\ (+)
\end{aligned}$$

...

# Semantics

$$
\begin{aligned}
\mu\ \mathit{time} &= \mathit{id} \\
\mu\ (\mathit{lift}_0\ a) &= \mathit{const}\ a \\
\mu\ (\mathit{lift}_1\ f\ \mathit{xs}) &= f \circ \mu\ \mathit{xs} \\
\mu\ (\mathit{lift}_2\ f\ \mathit{xs}\ \mathit{ys}) &= \mathit{liftA}_2\ f\ (\mu\ \mathit{xs})\ (\mu\ \mathit{ys}) \\
\mu\ (\mathit{timeTrans}\ \mathit{xs}\ \mathit{tt}) &= \mu\ \mathit{xs} \circ \mu\ \mathit{tt}
\end{aligned}
$$

$$
\begin{aligned}
&\mathbf{instance}\ \mathit{Num}\ a \Rightarrow \mathit{Num}\ (\mathit{Behavior}\ a)\ \mathbf{where} \\
&\quad \mathit{fromInteger} = \mathit{lift}_0 \circ \mathit{fromInteger} \\
&\quad (+) \qquad\quad\;\; = \mathit{lift}_2\ (+) \\
&\quad \ldots
\end{aligned}
$$

# Events

*Secondary* type:

$$\mu :: Event\ a \rightarrow [(T, a)] \quad \text{-- non-decreasing times}$$

$$\begin{aligned}
&never &&:: Event\ a \\
&once &&:: T \rightarrow a \rightarrow Event\ a \\
&(.|.) &&:: Event\ a \rightarrow Event\ a \rightarrow Event\ a \\
&(\Longrightarrow) &&:: Event\ a \rightarrow (a \rightarrow b) \rightarrow Event\ b \\
&predicate &&:: Behavior\ Bool \rightarrow Event\ () \\
&snapshot &&:: Event\ a \rightarrow Behavior\ b \rightarrow Event\ (a, b)
\end{aligned}$$

*Exercise:* define semantics of these operations.

# Reactivity

*Reactive* behaviors are defined piecewise, via events.

# Reactivity

*Reactive* behaviors are defined piecewise, via events:

$$switcher :: Behavior\ a \rightarrow Event\ (Behavior\ a) \rightarrow Behavior\ a$$

# Reactivity

*Reactive* behaviors are defined piecewise, via events:

$$switcher :: Behavior\ a \to Event\ (Behavior\ a) \to Behavior\ a$$

Semantics:

$$\mu\ (b_0\ `switcher`\ e)\ t = \mu\ (last\ (b_0 : before\ t\ (\mu\ e)))\ t$$

$$before :: T \to [(T, a)] \to [a]$$
$$before\ t\ os = [a \mid (t_a, a) \leftarrow os, t_a < t]$$

Important: $t_a < t$, rather than $t_a \leqslant t$.

# A more elegant specification for FRP (teaser)

# API

Replace operations with standard abstractions where possible:

$$\textbf{instance}\ \mathit{Functor\ Behavior}\ \textbf{where}\ \ldots$$
$$\textbf{instance}\ \mathit{Applicative\ Behavior}\ \textbf{where}\ \ldots$$
$$\textbf{instance}\ \mathit{Monoid}\ a \Rightarrow \mathit{Monoid}\ (\mathit{Behavior}\ a)\ \textbf{where}\ \ldots$$

$$\textbf{instance}\ \mathit{Functor\ Event}\ \textbf{where}\ \ldots$$
$$\textbf{instance}\ \mathit{Monoid}\ (\mathit{Event}\ a)\ \textbf{where}\ \ldots$$

Why?

# API

Replace operations with standard abstractions where possible:

**instance** *Functor Behavior* **where** ...
**instance** *Applicative Behavior* **where** ...
**instance** $Monoid\ a \Rightarrow Monoid\ (Behavior\ a)$ **where** ...

**instance** *Functor Event* **where** ...
**instance** $Monoid\ (Event\ a)$ **where** ...

Why?

- Less learning, more leverage.
- Specifications and laws for free.

# Specifications for free

The *instance's meaning* follows the *meaning's instance*:

$$
\begin{aligned}
\mu\ (fmap\ f\ as) &\equiv fmap\ f\ (\mu\ as) \\
\\
\mu\ (pure\ a) &\equiv pure\ a \\
\mu\ (fs \circledast xs) &\equiv \mu\ fs \circledast \mu\ xs \\
\\
\mu\ \varepsilon &\equiv \varepsilon \\
\mu\ (top \diamond bot) &\equiv \mu\ top \diamond \mu\ bot
\end{aligned}
$$

# Specifications for free

The *instance's meaning* follows the *meaning's instance*:

$$
\begin{aligned}
\mu\ (fmap\ f\ as) &\equiv fmap\ f\ (\mu\ as) \\
\\
\mu\ (pure\ a) &\equiv pure\ a \\
\mu\ (fs \circledast xs) &\equiv \mu\ fs \circledast \mu\ xs \\
\\
\mu\ \varepsilon &\equiv \varepsilon \\
\mu\ (top \diamond bot) &\equiv \mu\ top \diamond \mu\ bot
\end{aligned}
$$

- Corresponds exactly to the original FRP denotation.
- Follows inevitably from a domain-independent principle.
- Laws hold for free.

# History

# 1983–1989 at CMU

- I went for graphics.
- Did program transformation, FP, type theory.
- Class in denotational semantics.

# 1989 at CMU

- Kavi Arya's visit:
  - *Functional animation*
  - Streams of pictures
  - Elegant

# 1989 at CMU

- Kavi Arya's visit:
  - *Functional animation*
  - Streams of pictures
  - Elegant, mostly
- John Reynolds's insightful remark:

  "You can think of streams as functions from the natural numbers.

# 1989 at CMU

- Kavi Arya's visit:
  - *Functional animation*
  - Streams of pictures
  - Elegant, mostly
- John Reynolds's insightful remark:

  "You can think of streams as functions from the natural numbers.
  Have you thought about functions from the *reals* instead?
  Doing so might help with the awkwardness of interpolation."

*Continuous time!*

# 1989 at CMU

- Kavi Arya's visit:
  - *Functional animation*
  - Streams of pictures
  - Elegant, mostly
- John Reynolds's insightful remark:

  "You can think of streams as functions from the natural numbers.
  Have you thought about functions from the *reals* instead?
  Doing so might help with the awkwardness of interpolation."

  *Continuous time!*
- I finished my dissertation anyway.

# 1990–93 at Sun: TBAG

- 3D geometry etc as first-class immutable values.
- Animation as immutable functions of continuous time.

# 1990–93 at Sun: TBAG

- 3D geometry etc as first-class immutable values.
- Animation as immutable functions of continuous time.
- Multi-way constraints on time-functions.
  Off-the-shelf constraint solvers (DeltaBlue & SkyBlue from UW).
- Differentiation, integration and ODEs specified via *derivative*.
  Adaptive Runge-Kutta-5 solver (fast & accurate).
- Reactivity via `assert`/`retract` (high-level but imperative).

# 1990–93 at Sun: TBAG

- 3D geometry etc as first-class immutable values.
- Animation as immutable functions of continuous time.
- Multi-way constraints on time-functions.
  Off-the-shelf constraint solvers (DeltaBlue & SkyBlue from UW).
- Differentiation, integration and ODEs specified via *derivative*.
  Adaptive Runge-Kutta-5 solver (fast & accurate).
- Reactivity via `assert/retract` (high-level but imperative).
- Optimizing compiler via partial evaluation.
- In Common Lisp, C++, Scheme.
- Efficient multi-user distributed execution for free.

# 1994–1996 at Microsoft Research: RBML/ActiveVRML

- Programming model & fast implementation for new 3D hardware.
- TBAG + denotative/functional reactivity.

# 1994–1996 at Microsoft Research: RBML/ActiveVRML

- Programming model & fast implementation for new 3D hardware.
- TBAG + denotative/functional reactivity.
- Add event algebra to behavior algebra.
- Reactivity via behavior-valued events.
- Drop multi-way constraints "at first".

# 1994–1996 at Microsoft Research: RBML/ActiveVRML

- Programming model & fast implementation for new 3D hardware.
- TBAG + denotative/functional reactivity.
- Add event algebra to behavior algebra.
- Reactivity via behavior-valued events.
- Drop multi-way constraints “at first”.
- Started in ML as “RBML”.
- Rebranded to “ActiveVRML”, then “DirectAnimation”.

# 1995–1999 at MSR: RBMH/Fran

- Found Haskell: reborn as “RBMH” (research vehicle).
- Very fast implementation via sprite engine.
- John Hughes suggested using *Arrow*.

# 1999 at MSR: first try at push-based implementation

- Algebra of imperative event listeners.
- Challenges:
  - Garbage collection & dependency reversal.
  - Determinacy of timing & simultaneity.
  - I doubt anyone has gotten correct.

# 2009: Push-pull FRP

- Minimal computation, low latency, *provably correct*.
- Push for reactivity and pull for continuous phases.
- "Push" is really blocked pull.
- More elegant API:
  - Standard abstractions.
  - Semantics as homomorphisms.
  - Laws for free.
- Reactive normal form, via equational properties (denotation!).
- Uses *lub* (basis of PL semantics).
- Implementation subtleties & GHC RTS bugs. Didn't quite work.

# 1996–2014: Paul Hudak / Yale

- Paul Hudak visited MSR in 1996 or so and saw RBMH.
- Encouraged publishing, and suggested collaboration.
- Proposed names "Fran" & "FRP".
- *Many* FRP-based papers and theses.

July 15, 1952 – April 29, 2015

# Questions

"But computers are discrete, ..."