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?

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 (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?

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 (adaptive).
- Reconcile differing input sampling rates.

Principle:
Approximations/prunings compose badly, so postpone.

See Why Functional Programming Matters.
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 (adaptive).
- Reconcile differing input sampling rates.

Principle:
Approximations/prunings compose badly, so postpone.

See Why Functional Programming Matters.
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: \textit{Behavior} \(a\) — a “flow” of values.
Central abstract type: *Behavior* \( a \) — a “flow” of values.

Precise & simple semantics:

\[
\mu :: \text{Behavior } a \rightarrow (T \rightarrow a)
\]

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

Central abstract type: $\text{Behavior } a$ — a “flow” of values.

Precise & simple semantics:

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

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

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

\[
\begin{align*}
time & \quad :: \text{Behavior } T \\
lift_0 & \quad :: a \to \text{Behavior } a \\
lift_1 & \quad :: (a \to b) \to \text{Behavior } a \to \text{Behavior } b \\
lift_2 & \quad :: (a \to b \to c) \to \text{Behavior } a \to \text{Behavior } b \to \text{Behavior } c \\
timeTrans & \quad :: \text{Behavior } a \to \text{Behavior } T \to \text{Behavior } a \\
integral & \quad :: \langle S a \rangle \Rightarrow \text{Behavior } a \to T \to \text{Behavior } a \\
\ldots
\end{align*}
\]

\textbf{instance } Num a \Rightarrow Num (\text{Behavior } a) \textbf{ where } ... \\
\ldots

Reactivity later.
Semantics

\[ \mu \text{ time} = \lambda t \to t \]
\[ \mu (\text{lift}_0 \ a) = \lambda t \to a \]
\[ \mu (\text{lift}_1 \ f \ xs) = \lambda t \to f (\mu xs \ t) \]
\[ \mu (\text{lift}_2 \ f \ xs \ ys) = \lambda t \to f (\mu xs \ t)(\mu ys \ t) \]
\[ \mu (\text{timeTrans} \ xs \ tt) = \lambda t \to \mu xs (\mu tt \ t) \]

\textbf{instance} Num \ a \Rightarrow Num (Behavior \ a) \ \textbf{where}

\textbf{fromInteger} = \text{lift}_0 \circ \text{fromInteger}

(+) = \text{lift}_2 (+)

...
Semantics

\[
\begin{align*}
\mu \text{time} & = \text{id} \\
\mu (\text{lift}_0 a) & = \text{const} \ a \\
\mu (\text{lift}_1 f \ xs) & = f \circ \mu \ xs \\
\mu (\text{lift}_2 f \ xs \ ys) & = \text{liftA}_2 \ f \ (\mu \ xs) \ (\mu \ ys) \\
\mu (\text{timeTrans} \ xs \ tt) & = \mu \ xs \circ \mu \ tt
\end{align*}
\]

\textbf{instance} \ Num \ a \ \Rightarrow \ Num \ (\text{Behavior} \ a) \ \textbf{where}

\text{fromInteger} = \text{lift}_0 \circ \text{fromInteger} \\
(+ \ ) = \text{lift}_2 \ (+) \\
\ldots
Events

Secondary type:

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

\[
\begin{align*}
\text{never} & :: Event a \\
\text{once} & :: T \rightarrow a \rightarrow Event a \\
(\cdot|\cdot) & :: Event a \rightarrow Event a \rightarrow Event a \\
(\Rightarrow\Rightarrow) & :: Event a \rightarrow (a \rightarrow b) \rightarrow Event b \\
\text{predicate} & :: \text{Behavior Bool} \rightarrow Event () \\
\text{snapshot} & :: Event a \rightarrow \text{Behavior b} \rightarrow Event (a, b)
\end{align*}
\]

Exercise: define semantics of these operations.
Reactivity

*Reactive* behaviors are defined piecewise, via events.
Reactivity

*Reactive* behaviors are defined piecewise, via events:

\[
\text{switcher} :: \text{Behavior } a \rightarrow \text{Event } (\text{Behavior } a) \rightarrow \text{Behavior } a
\]
Reactivity

Reactive behaviors are defined piecewise, via events:

\[ \text{switcher :: Behavior } a \rightarrow \text{Event (Behavior } a) \rightarrow \text{Behavior } a \]

Semantics:

\[ \mu (b_0 \text{"switcher" } e) t = \mu (\text{last } (b_0 : \text{before } t (\mu e))) t \]

\[ \text{before :: } T \rightarrow [(T, a)] \rightarrow [a] \]
\[ \text{before } t \hspace{0.5em} os = [a \mid (t_a, a) \leftarrow os, t_a < t] \]

Important: \( t_a < t \), rather than \( t_a \leq t \).
A more elegant specification for FRP (teaser)
Replace operations with standard abstractions where possible:

```haskell
instance Functor Behavior where ...
instance Applicative Behavior where ...
instance Monoid a ⇒ Monoid (Behavior a) where ...

instance Functor Event where ...
instance Monoid (Event a) where ...
```

Why?
Replace operations with standard abstractions where possible:

```haskell
instance Functor Behavior where ...
instance Applicative Behavior where ...
instance Monoid a ⇒ 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:

\[
\begin{align*}
\mu \ (fmap \ f \ as) & \equiv fmap \ f \ (\mu \ as) \\
\mu \ (\text{pure} \ a) & \equiv \text{pure} \ a \\
\mu \ (fs \ <*> \ xs) & \equiv \mu \ fs \ <*> \ \mu \ xs \\
\mu \ \varepsilon & \equiv \varepsilon \\
\mu \ (\text{top} \Diamond \ \text{bot}) & \equiv \mu \ \text{top} \Diamond \ \mu \ \text{bot}
\end{align*}
\]
Specifications for free

The instance’s meaning follows the meaning’s instance:

\[
\begin{align*}
\mu (fmap f \ as) & \equiv fmap f (\mu as) \\
\mu (\text{pure } a) & \equiv \text{pure } a \\
\mu (fs <\otimes> xs) & \equiv \mu fs <\otimes> \mu xs \\
\mu \varepsilon & \equiv \varepsilon \\
\mu (\text{top} \diamond \text{bot}) & \equiv \mu \text{top} \diamond \mu \text{bot}
\end{align*}
\]

- 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

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.
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 \textit{reals} instead? Doing so might help with the awkwardness of interpolation.”

  \textit{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 \textit{derivative}.
  Adaptive Runge-Kutta-5 solver (fast & accurate).
- Reactivity via \texttt{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.
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.
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, ...”