

Elm-style Functional Reactive Programming demystified

Sergei Winitzki

SF Types, Theorems, and Programming Languages

April 13, 2015

What is “functional reactive programming”

FRP has little to do with...

- multithreading, message-passing concurrency, “actors”
- distributed computing on massively parallel load-balanced clusters
- map/reduce, the “reactive manifesto”, (*insert latest fad here*)...

FRP is...

- **pure functions using temporal types as primitives**
 - ▶ (temporal type \approx lazy stream of events)

Transformational vs. reactive programs

Transformational programs	Reactive programs
example: <code>pdflatex elm_talk.tex</code>	example: any GUI program, OS
start, run, then stop	keep running indefinitely
read some input, write some output	wait for signals, send messages
execution: sequential, parallel	“main run loop” + concurrency
difficulty: algorithms	signal/response sequences
specification: classical logic?	classical temporal logic?
verification: proof of correctness?	model checking?
synthesis: extract code from proof?	temporal logic synthesis?
type theory: intuitionistic logic	intuitionistic <i>temporal</i> logic

Difficulties in reactive programming

- Input signals may come at unpredictable times
 - ▶ Imperative updates are difficult to keep in the correct order
 - ▶ Flow of events becomes difficult to understand
- Asynchronous (out-of-order) callback logic becomes opaque
- Inverted control (“the system will call you”) obscures the flow of data
- Some concurrency is usually required (e.g. background tasks)
 - ▶ Explicit multithreaded code is hard to write and debug

Motivation for FRP

- Reactive programs work on **infinite sequences** of input/output values
- Main idea: make infinite sequences implicit, as a new “temporal” type
 - ▶ (Elm) `Signal α` — an infinite sequence of values of type α
 - ▶ alternatively, a value of type α that “changes with time”
- Reactive programs are **pure functions**
 - ▶ a GUI is a pure function of type `Signal Inputs \rightarrow Signal View`
 - ▶ a Web server is a pure function `Signal Request \rightarrow Signal Response`
 - ▶ all mutation is **implicit** in `Signal α` ; our code is 100% immutable
 - ★ instead of updating an `x:Int`, we define a value of type `Signal Int`
 - ▶ asynchronous behavior is **implicit**: our code has no callbacks
 - ▶ concurrency / parallelism is **implicit**
 - ★ the runtime needs to provide the required scheduling of events

Elm in a nutshell

- Elm is a pure polymorphic λ -calculus with products and sums
- **Temporal type** $\Sigma\alpha$ — a time-dependent value of **ordinary** type α
- Temporal primitive terms:

`constant`: $\alpha \rightarrow \Sigma\alpha$

`map2`: $(\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow \Sigma\alpha \rightarrow \Sigma\beta \rightarrow \Sigma\gamma$

`foldp`: $(\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow \Sigma\alpha \rightarrow \Sigma\beta$

`async`: $\Sigma\alpha \rightarrow \Sigma\alpha$

- **No nested** temporal types: `constant (constant x)` is illegal!
- Domain-specific primitive types: `Bool`, `Int`, `Float`, `String`, `View`
- Standard library with data structures, HTML, HTTP, JSON, ...
 - ▶ ...and signals `Time.every`, `Mouse.position`, `Window.dimensions`, ...
- Try Elm online at <http://elm-lang.org/try>

*Elm type judgments

- Non-temporal expressions are evaluated **eagerly** in pure λ -calculus
- The runtime will cache all values to avoid recomputation

*Elm operational semantics

- Non-temporal expressions are evaluated **eagerly** in pure λ -calculus
- The runtime will cache all values to avoid recomputation

*Elm operational semantics

- Example
- *I work after the boss comes by and until the phone rings:*

```
let after_until w (b,r) = (w or b) and not r in  
  foldp after_until false (boss, phone)
```

“Hello, world” in Elm

- The value called `main` will be visualized by the runtime

```
import Graphics.Element (..)
import Text (..)
import Signal (..)

text : Element
text = plainText "Hello, World!"

main : Signal Element
main = constant text
```

Typical program structure in Elm

- A state machine:

`update: Command → State → State`

- A rendering function:

`draw: State → View`

- A manager that merges the required input signals into one:

- ▶ may use Mouse, Keyboard, Time, HTML stuff, etc.

`merge_inputs: Signal Command`

- Program boilerplate:

`init_state : State`

`main : Signal View`

`main = map draw $ foldp update init_state merge_inputs`

*Asynchrony and concurrency in Elm

- Long-running computations will delay signal updates
- Solutions: 1. Caching of all results. 2. Using `async`

Some limitations of Elm-style FRP

- No recursion of any kind
- No higher-order signals: $\Sigma(\Sigma\alpha)$ is disallowed by the type system
- No distinction between continuous time and discrete time
- The signal processing logic is fully specified statically
- No constructors for signals
 - ▶ Impossible to implement the “dining philosophers”!

Elm cannot do “dining philosophers”

- “Dining philosophers”: need to simulate a philosopher who thinks for a random time and then eats for a random time
- Can a signal value $p : \text{Signal Unit}$ update at random times?
 - ▶ No! There is no way to delay the update times of a signal **at runtime**.
 - ▶ $\text{Time.delay} : \text{Int} \rightarrow \Sigma \alpha \rightarrow \Sigma \alpha$ cannot use a time-varying delay value
 - ▶ $\text{Time.every} : \text{Int} \rightarrow \Sigma \text{Int}$ also requires a fixed delay value
 - ▶ Cannot lift Time.every into $\Sigma \text{Int} \rightarrow \Sigma \Sigma \text{Int}$ to achieve variable delay

The JavaScript backend for Elm

Features

- Good support for HTML/CSS, HTTP requests, JSON
- Good performance of caching views
- Transparent, declarative modeling of data

Limitations

- No implementation for `async` and no concurrency
- Ordinary recursion may generate invalid JavaScript
- The lack of recursive signals is compensated by *ad hoc* primitives

*Possible extensions

- Recursive definitions for signals
- Monadic signal combinators
- Signal constructors

Part 2. Temporal logic and FRP

This part of the talk is optional.

- Reminder (Curry-Howard): temporal logic expressions will be our types
- We only need to control the **order** of events: no “hard real-time”
- How to understand temporal logic:
 - ▶ classical propositional logic \approx Boolean arithmetic
 - ▶ intuitionistic propositional logic \approx same but without **true** / **false** dichotomy
 - ▶ (linear-time) temporal logic \approx Boolean arithmetic for *infinite sequences*
 - ▶ intuitionistic temporal logic \approx same but without **true** / **false** dichotomy
- In other words:
 - ▶ a temporal type represents a **single infinite sequence** of values

Boolean arithmetic: notation

- Classical propositional (Boolean) logic: $T, F, a \vee b, a \wedge b, \neg a, a \rightarrow b$
- A notation better adapted to school-level arithmetic: $1, 0, a + b, ab, a'$
- The only “new rule” is $1 + 1 = 1$
- Define $a \rightarrow b = a' + b$
- Some identities:

$$\begin{aligned}0a = 0, \quad 1a = a, \quad a + 0 = a, \quad a + 1 = 1, \\a + a = a, \quad aa = a, \quad a + a' = 1, \quad aa' = 0, \\(a + b)' = a'b', \quad (ab)' = a' + b', \quad (a')' = a \\a(b + c) = ab + ac, \quad (a + b)(a + c) = a + bc\end{aligned}$$

Boolean arithmetic: example

Of the three suspects A, B, C, only one is guilty of a crime.

Suspect A says: “B did it”. Suspect B says: “C is innocent.”

The guilty one is lying, the innocent ones tell the truth.

$$\phi = (ab'c' + a'bc' + a'b'c) (a'b + ab') (b'c' + bc)$$

Simplify: expand the brackets, omit aa' , bb' , cc' , replace $aa = a$ etc.:

$$\phi = ab'c' + 0 + 0 = ab'c'$$

The guilty one is A.

Propositional linear-time temporal logic (LTL)

- We work with *infinite boolean sequences* (“linear time”)

Boolean operations:

$$a = [a_0, a_1, a_2, \dots]; \quad b = [b_0, b_1, b_2, \dots];$$

$$a + b = [a_0 + b_0, a_1 + b_1, \dots]; \quad a' = [a'_0, a'_1, \dots]; \quad ab = [a_0 b_0, a_1 b_1, \dots]$$

Temporal operations:

$$\text{(Next)} \quad \mathbf{N}a = [a_1, a_2, \dots]$$

$$\text{(Sometimes)} \quad \mathbf{F}a = [a_0 + a_1 + a_2 + \dots, a_1 + a_2 + \dots, \dots]$$

$$\text{(Always)} \quad \mathbf{G}a = [a_0 a_1 a_2 a_3 \dots, a_1 a_2 a_3 \dots, a_2 a_3 \dots, \dots]$$

Other notation (from modal logic):

$$\mathbf{N}a \equiv \bigcirc a; \quad \mathbf{F}a \equiv \Diamond a; \quad \mathbf{G}a \equiv \Box a$$

- Weak Until: $p\mathbf{U}q = “p \text{ holds from now on until } q \text{ first becomes true”}$

$$p\mathbf{U}q = q + p\mathbf{N}(q + p\mathbf{N}(q + \dots))$$

Temporal logic redux

- LTL as type theory: do we use $\mathbf{N}\alpha$, $\mathbf{F}\alpha$, $\mathbf{G}\alpha$ as new types?
- Are they to be functors, monads, ...?
- What is the operational semantics? (i.e., how to compile this?)

Interpreting values typed by LTL

- What does it mean to have a value x of type, say, $\mathbf{G}(\alpha \rightarrow \alpha \mathbf{U} \beta)$??
 - ▶ $x : \mathbf{N}\alpha$ means that $x : \alpha$ will be available *only* at the *next* time tick (x is a **deferred value** of type α)
 - ▶ $x : \mathbf{F}\alpha$ means that $x : \alpha$ will be available at *some* future tick(s) (x is an **event** of type α)
 - ▶ $x : \mathbf{G}\alpha$ means that a (different) value $x : \alpha$ is available at *every* tick (x is an **infinite stream** of type α)
 - ▶ $x : \alpha \mathbf{U} \beta$ means a **finite stream** of α that may end with a β
- Some *temporal axioms* of intuitionistic LTL:

(deferred apply) $\mathbf{N}(\alpha \rightarrow \beta) \rightarrow (\mathbf{N}\alpha \rightarrow \mathbf{N}\beta)$;

(streamed apply) $\mathbf{G}(\alpha \rightarrow \beta) \rightarrow (\mathbf{G}\alpha \rightarrow \mathbf{G}\beta)$;

(generate a stream) $\mathbf{G}(\alpha \rightarrow \mathbf{N}\alpha) \rightarrow (\alpha \rightarrow \mathbf{G}\alpha)$;

(read infinite stream) $\mathbf{G}\alpha \rightarrow \alpha \mathbf{N}(\mathbf{G}\alpha)$

(read finite stream) $\alpha \mathbf{U} \beta \rightarrow \beta + \alpha \mathbf{N}(\alpha \mathbf{U} \beta)$

Elm as an FRP language

- λ -calculus with type $\mathbf{G}\alpha$, primitives `map2`, `foldp`, `async`

`map2` : $(\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow \mathbf{G}\alpha \rightarrow \mathbf{G}\beta \rightarrow \mathbf{G}\gamma$

`foldp` : $(\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow \mathbf{G}\alpha \rightarrow \mathbf{G}\beta$

`async` : $\mathbf{G}\alpha \rightarrow \mathbf{G}\alpha$

- (`map2` makes \mathbf{G} an applicative functor)
- `async` is a special *scheduling instruction*
- Limitations:
 - ▶ Cannot have a type $\mathbf{G}(\mathbf{G}\alpha)$, also not using \mathbf{N} or \mathbf{F}
 - ▶ Cannot construct temporal values by hand
 - ▶ This language is an *incomplete* Curry-Howard image of LTL!

Conclusions and outlook

- There are some languages that implement FRP in various *ad hoc* ways
- The ideal is not (yet) reached

*Conclusions and outlook

- The ideal is not (yet) reached

Abstract

In my day job, most bugs come from imperatively implemented reactive programs. FRP is a declarative approach that promises to solve my problems.

FRP can be defined as a λ -calculus with types given by a propositional intuitionistic linear-time temporal logic (LTL). Although the Elm language uses only a subset of LTL, it achieves high expressivity for GUI programming. I discuss the current limitations of Elm and outline some possible extensions. I will also briefly review the motivations behind and the connections between temporal logic, FRP, and Elm.

My talk will be understandable to anyone familiar with Curry-Howard and functional programming. (The first part of the talk does not rely on temporal logic or Curry-Howard.)

Suggested reading

- E. Czaplicki, S. Chong. [Asynchronous FRP for GUIs](#). (2013)
- E. Czaplicki. [Concurrent FRP for functional GUI](#) (2012).
- M. F. Dam. Lectures on temporal logic. Slides: [Syntax and semantics of LTL](#), [A Hilbert-style proof system for LTL](#)
- E. Bainomugisha, et al. [A survey of reactive programming](#) (2013).
- W. Jeltsch. [Temporal logic with Until, Functional Reactive Programming with processes, and concrete process categories](#). (2013).
- A. Jeffrey. [LTL types FRP](#). (2012).
- D. Marchignoli. [Natural deduction systems for temporal logic](#). (2002). – See Chapter 2 for a natural deduction system for modal and temporal logics.