

# Elm-style Functional Reactive Programming demystified

Sergei Winitzki

SF Types, Theorems, and Programming Languages

April 13, 2015

# Part 1. Functional reactive programming in Elm

FRP has little to do with...

- multithreading, message-passing concurrency, “actors”
- distributed computing on massively parallel, load-balanced clusters
- map/reduce, the “reactive manifesto”

FRP means...

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

FRP is probably most useful for:

- GUI programming

Elm is...

- a viable implementation of FRP geared for Web apps

# Transformational vs. reactive programs

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

# Difficulties in reactive programming

Usually, reactive programs are written imperatively...

- 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
  - ▶ “callback hell”: deeply nested callbacks, all mutating data
- 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  $\alpha$`  — an infinite sequence of values of type  $\alpha$
  - ▶ alternatively, a value of type  $\alpha$  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  $\alpha$` ; 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 FRP runtime will provide the required scheduling of events

# Czaplicki's original Elm 2012 in a nutshell

- Elm is a pure polymorphic  $\lambda$ -calculus with products and sums
- **Temporal type**  $\Sigma\alpha$  — a lazy sequence of values of type  $\alpha$
- Temporal **combinators** in core Elm:

`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 ill-typed!
- 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`, ...
  - ▶ ...and some utility functions: `map`, `merge`, `drop`, ...

# Details: Elm type judgments [Czaplicki 2012]

- Polymorphically typed  $\lambda$ -calculus (also with temporal types)

$$\frac{\Gamma, (x : \alpha) \vdash e : \beta}{\Gamma \vdash (\lambda x. e) : \alpha \rightarrow \beta} \text{LAMBDA} \quad \frac{\Gamma \vdash e_1 : \alpha \rightarrow \beta \quad \Gamma \vdash e_2 : \alpha}{\Gamma \vdash (e_1 e_2) : \beta} \text{APPLY}$$

- Temporal types are denoted by  $\Sigma\tau$

- In these rules, type variables  $\alpha, \beta, \gamma$  **cannot** involve  $\Sigma$ :

$$\frac{\Gamma \vdash e : \alpha}{\Gamma \vdash (\text{constant } e) : \Sigma\alpha} \text{CONST}$$
$$\frac{\Gamma \vdash m : \alpha \rightarrow \beta \rightarrow \gamma \quad \Gamma \vdash p : \Sigma\alpha \quad \Gamma \vdash q : \Sigma\beta}{\Gamma \vdash (\text{map2 } m p q) : \Sigma\gamma} \text{MAP2}$$
$$\frac{\Gamma \vdash u : \alpha \rightarrow \beta \rightarrow \beta \quad \Gamma \vdash e : \beta \quad \Gamma \vdash q : \Sigma\alpha}{\Gamma \vdash (\text{foldp } u e q) : \Sigma\beta} \text{FOLDP}$$

- A value of type  $\Sigma\Sigma\alpha$  is impossible in a well-typed expression!

# Elm operational semantics 1: Current values

- Non-temporal expressions are evaluated **eagerly** in pure  $\lambda$ -calculus
- Temporal expressions are built from **input** signals and combinators
  - ▶ It is not possible to “consume” a signal  $(\Sigma\alpha \rightarrow \beta)$ !
- Every temporal expression has a **current value** denoted by  $e^{[c]}$

$$\frac{\Gamma \vdash e : \Sigma\alpha \quad \Gamma \vdash c : \alpha}{\Gamma \vdash e^{[c]} : \Sigma\alpha} \text{CURVAL}$$

- Every predefined **input** signal  $i : \Sigma\alpha, i \in \mathcal{I}$  has an initial value:  $i^{[a]}$
- **Initial** current values for all expressions are derived:

$$\frac{\Gamma \vdash (\text{constant } c) : \Sigma\alpha}{\Gamma \vdash (\text{constant } c)^{[c]} : \Sigma\alpha} \text{CONSTINIT}$$

$$\frac{\Gamma \vdash (\text{map2 } m \ p^{[a]} \ q^{[b]}) : \Sigma\gamma}{\Gamma \vdash (\text{map2 } m \ p \ q)^{[m \ a \ b]} : \Sigma\gamma} \text{MAP2INIT}$$

$$\frac{\Gamma \vdash (\text{foldp } u \ e \ q) : \Sigma\beta}{\Gamma \vdash (\text{foldp } u \ e \ q)^{[e]} : \Sigma\beta} \text{FOLDPINIT}$$



## Elm operational semantics 2: Update steps

- Update steps happen only to **input signals**  $s \in \mathcal{I}$  and **one at a time**
- Update steps  $\mathbf{U}_{s \leftarrow a} \{ \dots \}$  are applied to the **whole program** at once:

$$\frac{\Gamma \vdash s : \Sigma\alpha \quad s \in \mathcal{I} \quad \Gamma \vdash a : \alpha \quad \Gamma \vdash e^{[c]} : \Sigma\beta \quad \Gamma \vdash e'^{[c']} : \Sigma\beta}{\Gamma \vdash \mathbf{U}_{s \leftarrow a} \{ e^{[c]} \} \Rightarrow e'^{[c]}}$$

- An update step on  $s$  will leave all other **input** signals unchanged:

$$\forall s \neq s' \in \mathcal{I} : \quad \mathbf{U}_{s \leftarrow b} \{ s^{[a]} \} \Rightarrow s^{[b]} \quad \mathbf{U}_{s \leftarrow b} \{ s'^{[c]} \} \Rightarrow s'^{[c]}$$

- Efficient implementation:
  - ▶ The instances of input signals within expressions are not duplicated
  - ▶ Unchanged current values are cached and not recomputed

## Elm operational semantics 3: Updating combinators

- Operational semantics does not reduce temporal expressions
  - ▶ The whole program **remains** a static temporal expression tree
  - ▶ Only the current values are updated in all subexpressions

$$\mathbf{U}_{s \leftarrow a} \left\{ (\text{constant } c)^{[c]} \right\} \Rightarrow (\text{constant } c)^{[c]} \quad \text{CONSTUPD}$$

$$\begin{aligned} \mathbf{U}_{s \leftarrow a} \{ \text{map2 } m \ p \ q \} \\ \Rightarrow \left( \text{map2 } m \ \mathbf{U}_{s \leftarrow a} \{ p \}^{[b]} \ \mathbf{U}_{s \leftarrow a} \{ q \}^{[c]} \right)^{[m \ b \ c]} \quad \text{MAP2UPD} \end{aligned}$$

$$\mathbf{U}_{s \leftarrow a} \left\{ (\text{foldp } u \ e \ q)^{[b]} \right\} \Rightarrow \left( \text{foldp } u \ e \ \mathbf{U}_{s \leftarrow a} \{ q \}^{[c]} \right)^{[u \ c \ b]} \quad \text{FOLDPUPD}$$

- All computations during an update step are **synchronous**
  - ▶ The expression  $\mathbf{U}_{s \leftarrow b} \{ e^{[c]} \}$  is reduced only after all subexpressions of  $e$

# GUI building: “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
```

- Try Elm online at <http://elm-lang.org/try>

# Example of using foldp

- Specification:

- ▶ *I work only after the boss comes by and unless the phone rings*

- Implementation:

```
after_unless : (Bool, Bool) -> Bool -> Bool
```

```
after_unless (b,r) w = (w || b) && not r
```

```
boss : Signal Bool
```

```
phone: Signal Bool
```

```
i_work : Signal Bool
```

```
i_work = foldp after_unless false (boss, phone)
```

- Demo

# Typical GUI boilerplate in Elm

- A state machine with stepwise update:

`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`

- Main 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!
- Example:  $\text{map } f \text{ } s$  where  $f : \alpha \rightarrow \alpha$  takes a long time to compute
- Use  $\text{async} : \Sigma \alpha \rightarrow \Sigma \alpha$
- Operational semantics: ( $i$  is a **new** input signal for each  $\text{async}$ )

$$\frac{\Gamma \vdash e^{[c]} : \Sigma \alpha}{\Gamma, (i : \Sigma \alpha) \vdash (\text{async}_i e)^{[c]} : \Sigma \alpha} \text{ASYNCINIT}$$

$$\mathbf{U}_{s \leftarrow a} \left\{ (\text{async}_i e)^{[c]} \right\} \Rightarrow \mathbf{U}_{i \leftarrow c'}^\dagger \left( \text{async}_i \mathbf{U}_{s \leftarrow a}^\dagger \{e\}^{[c']} \right)^{[c]} \text{ASYNCSCHEd}$$

$$\mathbf{U}_{i \leftarrow c'} \left\{ (\text{async}_i e)^{[c]} \right\} \Rightarrow (\text{async}_i e)^{[c']} \text{ASYNCUPD}$$

- The update computation  $\mathbf{U}_{s \leftarrow a}^\dagger \{e\}$  runs on another thread...
  - ▶ ...while the current value  $c$  remains unchanged
  - ▶ Another update  $\mathbf{U}_{i \leftarrow c'}^\dagger$  is **scheduled** but not yet triggered
  - ▶ When  $c'$  is ready,  $\mathbf{U}_{i \leftarrow c'} \{...\}$  runs and sets the current value to  $c'$

# Example of using async

- UI that shows results of some long computations:

```
draw : Int -> Int -> View
draw x y = ...
```

```
s : Signal Int -- input values
```

```
fSlow : Int -> Int
res1 = async (map fSlow s)
fFast : Int -> Int
res2 = map fFast s
```

```
main : Signal View = map2 draw res1 res2
```

# Some limitations of Elm-style FRP

- 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 user-defined signals
- No recursion possible in signal definition!
- Incomplete semantics for `async` :  $\Sigma\alpha \rightarrow \Sigma\alpha$ 
  - ▶ Example: `async (map f s)` where `f` takes a long time
  - ▶ The initial value of this signal will not be available at initial time!
  - ▶ Need `async' :  $\alpha \rightarrow \Sigma\alpha \rightarrow \Sigma\alpha$`  to specify initial value?
- No full concurrency (e.g., “dining philosophers”)



# Elm cannot simulate “dining philosophers”

- A philosopher thinks for a random time, then eats for a random time
  - ▶ Can a signal value  $p : \text{Signal Unit}$  update itself 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  $\text{Time.every}$  into  $\Sigma \text{Int} \rightarrow \Sigma \Sigma \text{Int}$  to achieve variable delay

# The JavaScript backend for Elm (2015)

## Features:

- Good support for HTML/CSS, HTTP requests, JSON
- Good performance of caching HTML views
- Support for Canvas and HTML-free UI building

## Limitations:

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

# Elm-style FRP: the good parts

- Transparent, declarative modeling of data through ADTs
- Immutable and safe data structures (Array, Dict, ...)
- No runtime errors or exceptions!
- Space/time leaks are impossible!
- Language is Haskell-like but simpler for beginners
- Full type inference
- Easy deployment and interop in Web applications

# Some conservative extensions of Elm

- Fix initial value semantics for `async'` :  $\alpha \rightarrow \Sigma\alpha \rightarrow \Sigma\alpha$
- Allow recursive definitions for signals
  - ▶ generate updates as `s0`, `f(s0)`, `f(f(s0))`, ... are being computed:  
`s = async' s0 (map f s)`
- Add monadic signal combinator, `bind` :  $(\alpha \rightarrow \Sigma\beta) \rightarrow \Sigma\alpha \rightarrow \Sigma\beta$ 
  - ▶ use input signals from dynamically created UI:  
`viewS = map draw states`  
`states = foldp update_on_click (bind get_clicks viewS)`
- Allow user-defined signals constructed from asynchronous APIs
  - ▶ Generate signal updates whenever callback is called:  
`type Cαβ = α → (β → ⊥) → ⊥`  
`chain : Cαβ → Σα → Σβ`  
`some_async_api : Cαβ`  
`values_of_b = chain some_async_api values_of_a`

## Part 2. Temporal logic and FRP

- Reminder (Curry-Howard): logical expressions will be types
  - ▶ ...and the axioms will be primitive terms
- 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 LTL  $\approx$  Boolean arithmetic for *infinite sequences*
  - ▶ intuitionistic temporal logic ITL  $\approx$  same but without **true** / **false** dichotomy
- In other words:
  - ▶ an ITL 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:

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

# 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

Designers of FRP languages must face some choices:

- 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, ...?
- Which temporal axioms to use as language primitives?
- What is the operational semantics? (I.e., how to compile this?)

A sophisticated example: [Krishnaswamy 2013]

- uses full LTL with higher-order temporal types and fixpoints
- uses linear types to control space/time leaks

# 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  $\alpha$ )
  - ▶  $x : \mathbf{F}\alpha$  means that  $x : \alpha$  will be available at *some* future tick(s) ( $x$  is an **event** of type  $\alpha$ )
  - ▶  $x : \mathbf{G}\alpha$  means that a (different) value  $x : \alpha$  is available at *every* tick ( $x$  is an **infinite stream** of type  $\alpha$ )
  - ▶  $x : \alpha \mathbf{U} \beta$  means a **finite stream** of  $\alpha$  that may end with a  $\beta$
- 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

- $\lambda$ -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

- There are some languages that implement FRP in various *ad hoc* ways
- The ideal is not (yet) reached
- Elm-style FRP is a promising step in the right direction

# Abstract

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

FRP can be defined as a  $\lambda$ -calculus that admits temporal types, i.e. 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 will formally define the operational semantics of Elm. I discuss the current limitations of Elm and outline possible extensions. I also review 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 is a self-contained presentation of Elm that does not rely on temporal logic or Curry-Howard. The second part of the talk will explain the basic intuitions behind temporal logic and its connection with FRP.

# Suggested reading

E. Czaplicki, S. Chong. [Asynchronous FRP for GUIs](#). (2013)

E. Czaplicki. [Concurrent FRP for functional GUI](#) (2012).

N. R. Krishnaswamy.

<https://www.mpi-sws.org/~neelk/simple-frp.pdf> Higher-order functional reactive programming without spacetime leaks (2013).

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.