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
example: `pdflatex elm_talk.tex`	**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 *temporal* 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` 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` $\to$ `Signal View`
  - a Web server is a pure function `Signal Request` $\to$ `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 FRP runtime will provide the required scheduling of events

Czaplicki's original `Elm` 2012 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 combinators in core `Elm`:

  `constant:` $\alpha \to \Sigma\alpha$
  `map2:` $(\alpha \to \beta \to \gamma) \to \Sigma\alpha \to \Sigma\beta \to \Sigma\gamma$
  `foldp:` $(\alpha \to \beta \to \beta) \to \beta \to \Sigma\alpha \to \Sigma\beta$
  `async:` $\Sigma\alpha \to \Sigma\alpha$

- **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 λ-calculus (also with temporal types)

$$\frac{\Gamma, (x : \alpha) \vdash e : \beta}{\Gamma \vdash (\lambda x.e) : \alpha \to \beta}\ \textsc{Lambda} \qquad \frac{\Gamma \vdash e_1 : \alpha \to \beta \quad \Gamma \vdash e_2 : \alpha}{\Gamma \vdash (e_1\, e_2) : \beta}\ \textsc{Apply}$$

- Temporal types are denoted by $\Sigma\tau$
  - In these rules, type variables α, β,... **cannot** involve Σ:

$$\frac{\Gamma \vdash e : \alpha}{\Gamma \vdash (\text{constant}\ e) : \Sigma\alpha}\ \textsc{Const}$$

$$\frac{\Gamma \vdash m : \alpha \to \beta \to \gamma \quad \Gamma \vdash p : \Sigma\alpha \quad \Gamma \vdash q : \Sigma\beta}{\Gamma \vdash (\text{map2}\ m\ p\ q) : \Sigma\gamma}\ \textsc{Map2}$$

$$\frac{\Gamma \vdash u : \alpha \to \beta \to \beta \quad \Gamma \vdash e : \beta \quad \Gamma \vdash q : \Sigma\alpha}{\Gamma \vdash (\text{foldp}\ u\ e\ q) : \Sigma\beta}\ \textsc{FoldP}$$

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

$\texttt{Elm}$ operational semantics 1: Current values

- Non-temporal expressions are evaluated **eagerly** in pure λ-calculus
- All temporal expressions are built from input signals and combinators
- Every temporal expression has a **current value** denoted by $e^{[c]}$

$$\frac{\Gamma \vdash e : \Sigma\alpha \qquad \Gamma \vdash c : \alpha}{\Gamma \vdash e^{[c]} : \Sigma\alpha}\ \textsc{CurrentEval}$$

- Every predefined signal $i : \Sigma\alpha$ has an initial value: $i^{[a]}$
- **Initial** current values for combinators are evaluated:

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

$$\frac{\Gamma \vdash \text{map2 } m\ p^{[a]}\ q^{[b]} : \Sigma\gamma}{\Gamma \vdash (\text{map2 } m\ p\ q)^{[m\ a\ b]} : \Sigma\gamma}\ \textsc{Map2Init}$$

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

`Elm` operational semantics 2: Update steps

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

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

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

$$\mathbf{U}_{s\leftarrow b}\left\{s^{[a]}\right\}\Rightarrow s^{[b]}\qquad\mathbf{U}_{s\leftarrow b}\left\{s'^{[c]}\right\}\Rightarrow s'^{[c]}$$

- All computations during an update step are **synchronous**

`Elm` operational semantics 3: Updating combinators

- The whole program **remains** a static signal expression tree
- Only the current values are updated in all subexpressions

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

$$\mathbf{U}_{s\leftarrow a}\left\{\text{map2 } m\ p\ q\right\}$$

$$\Rightarrow \left(\text{map2 } m\ \mathbf{U}_{s\leftarrow a}\left\{p\right\}^{[b]}\ \mathbf{U}_{s\leftarrow a}\left\{q\right\}^{[c]}\right)^{[m\ b\ c]} \qquad \textsc{Map2Upd}$$

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

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

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 or b) and not r

boss : Signal Bool
phone: Signal Bool

i_work : Signal Bool
i_work = foldp after_unless false (boss, phone)
```

- Demo

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

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!
- Example: `map f s` where `f`:$\alpha \to \alpha$ takes a long time to compute
- Use `async` : $\Sigma\alpha \to \Sigma\alpha$
- Operational semantics: (i is a **new** input signal)

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

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

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

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

Example of using `async`

- UI shows results of long computations

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!
- No full concurrency (e.g., "dining philosophers")
- Incomplete semantics for `async` : $\Sigma\alpha \to \Sigma\alpha$
  - 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'` : $\alpha \to \Sigma\alpha \to \Sigma\alpha$ to specify initial value?

`Elm` cannot simulate "dining philosophers"

- A philosopher thinks for a random time, then eats for a random time
  - Can a signal value `p : Signal Unit` update itself at random times?
- No! There is no way to delay the update times of a signal **at runtime**
- `Time.delay: Int` $\rightarrow \Sigma\alpha \rightarrow \Sigma\alpha$ cannot use a time-varying delay value
- `Time.every: Int` $\rightarrow \Sigma$`Int` also requires a fixed delay value
- Cannot lift `Time.every` into Σ`Int` $\rightarrow \Sigma\Sigma$`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

*Possible extensions

- Complete semantics for async : $\alpha \to \Sigma\alpha \to \Sigma\alpha$
- 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:

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

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.

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

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:

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

$$a + b = [a_0 + b_0, a_1 + b_1, ...]\,;\; a' = \left[a'_0, a'_1, ...\right]\,;\; ab = [a_0 b_0, a_1 b_1, ...]$$

  Temporal operations:

$$\begin{aligned} \text{(Next)} \quad & \mathbf{N}a = [a_1, a_2, ...] \\ \text{(Sometimes)} \quad & \mathbf{F}a = [a_0 + a_1 + a_2 + ...,\; a_1 + a_2 + ...,\; ...] \\ \text{(Always)} \quad & \mathbf{G}a = [a_0 a_1 a_2 a_3 ...,\; a_1 a_2 a_3 ...,\; a_2 a_3 ...,\; ...] \end{aligned}$$

  Other notation (from modal logic):

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

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

$$p\mathbf{U}q = q + p\mathbf{N}\left(q + p\mathbf{N}\left(q + ...\right)\right)$$

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, $\mathbf{G}(\alpha \to \alpha\mathbf{U}\beta)$??
  - $x : \mathbf{N}\alpha$ 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:

$$\begin{aligned}
\text{(deferred apply)} \quad & \mathbf{N}\,(\alpha \to \beta) \to (\mathbf{N}\alpha \to \mathbf{N}\beta)\,; \\
\text{(streamed apply)} \quad & \mathbf{G}\,(\alpha \to \beta) \to (\mathbf{G}\alpha \to \mathbf{G}\beta)\,; \\
\text{(generate a stream)} \quad & \mathbf{G}\,(\alpha \to \mathbf{N}\alpha) \to (\alpha \to \mathbf{G}\alpha)\,; \\
\text{(read infinite stream)} \quad & \mathbf{G}\alpha \to \alpha\mathbf{N}(\mathbf{G}\alpha) \\
\text{(read finite stream)} \quad & \alpha\mathbf{U}\beta \to \beta + \alpha\mathbf{N}(\alpha\mathbf{U}\beta)
\end{aligned}$$

`Elm` as an FRP language

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

  $$\begin{aligned}
  \texttt{map2} &: (\alpha \to \beta \to \gamma) \to \mathbf{G}\alpha \to \mathbf{G}\beta \to \mathbf{G}\gamma \\
  \texttt{foldp} &: (\alpha \to \beta \to \beta) \to \beta \to \mathbf{G}\alpha \to \mathbf{G}\beta \\
  \texttt{async} &: \mathbf{G}\alpha \to \mathbf{G}\alpha
  \end{aligned}$$

- (`map2` makes **G** an applicative functor)
- `async` is a special *scheduling instruction*
- Limitations:
  - Cannot have a type $\mathbf{G}(\mathbf{G}\alpha)$, also not using **N** or **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 λ-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).

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.