Functional Reactive Programming and Elm

Sergei Winitzki

BayHac 2015

June 14, 2015

Part 1. Functional reactive programming and `Elm`

FRP has little to do with...

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

FRP means (in my definition)...

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

FRP is probably most useful for:

- GUI and event-driven programming

`Elm` is...

- a viable implementation of FRP geared for Web GUI apps

Difficulties in reactive programming

Imperative implementation is one problem among many...

- 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

FRP basics

- Reactive programs work on **infinite streams** of input/output values
- Main idea: make streams **implicit**, as a new "temporal" type
  - $\Sigma\alpha$ — an infinite stream of values of type α
  - alternatively, $\Sigma\alpha$ is a value of type α that "changes with time"
- Reactive programs are viewed as **pure functions**
  - a GUI is a pure function of type Σ `Inputs` $\rightarrow \Sigma$ `View`
  - a Web server is a pure function Σ `Request` $\rightarrow \Sigma$ `Response`
  - all mutation is **implicit** in the program
    - ★ instead of updating an `x:Int`, we define a value of type Σ `Int`
    - ★ our code is 100% immutable, no side effects, no `IO` monads
  - 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 (2012) core `Elm` in a nutshell

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

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

- **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`, `sampleOn`, ...

Details: `Elm` type judgments [Czaplicki 2012]

- Polymorphically typed λ-calculus (also with temporal types)

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

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

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

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

$$\frac{\Gamma \vdash u : \alpha \to \beta \to \beta \quad \Gamma \vdash e : \beta \quad \Gamma \vdash q : \Sigma\alpha}{\Gamma \vdash (\text{scan}\ u\ e\ q) : \Sigma\beta}\ \text{Scan}$$

`Elm` operational semantics 0: Current values

- An `Elm` program is a temporal expression of type ... $\to \Sigma$`View`
  - Temporal expressions are built from **input signals** and combinators
  - It is not possible to "consume" a signal ($\Sigma\alpha \to \beta$)
  - A value of type $\Sigma\Sigma\alpha$ is impossible in a well-typed expression
- Every temporal expression has a **current value** denoted by $e^{[c]}$

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

- Some current values are "freshly updated" while others are "stale"
  - Denote a fresh value by asterisk: $e^{[c]^*}$
  - In `Elm`, current values are implemented with an `Either` type

Elm operational semantics 1: Initial values

Every temporal expression has an initial (*stale*) current value

- 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:

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

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

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

`Elm` operational semantics 2: Updating signals

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

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

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

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

- `Elm` has an efficient implementation:
  - The instances of input signals within expressions are not duplicated
  - Stale current values are cached and not recomputed

$\mathtt{Elm}$ 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 subexpressions

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

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

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

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

- All computations during an update step are **synchronous**
  - The expression $\mathbf{U}_{s\leftarrow b}\left\{e^{[c]}\right\}$ is reduced **after** all subexpressions of e
  - Current values are non-temporal and are evaluated **eagerly**
  - `scan` does **not** update unless its subexpression is freshly updated

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 scan

- Specification:
  - *I work only after the boss comes by and unless the phone rings*
- Implementation:

```
after_unless : (Bool, Bool) -> Bool -> Bool
after_unless (b,p) w = (w || b) && not p

boss_and_phone : Signal (Bool,Bool)

i_work : Signal Bool
i_work = scan after_unless False (boss_and_phone)
```

- Demo (boss_phone_work.elm)

Typical GUI boilerplate in `Elm`

- A state machine with stepwise update:

```
update : Command → State → State
```

- A rendering function (`View` is either `Element` or `Html`):

```
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 (scan update init_state merge_inputs)
```

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”)

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

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:

$$0a = 0, \quad 1a = a, \quad a+0=a, \quad a+1=1,$$
$$a+a=a, \quad aa=a, \quad a+a'=1, \quad aa'=0,$$
$$(a+b)' = a'b', \quad (ab)' = a'+b', \quad (a')' = a$$
$$a(b+c) = ab+ac, \quad (a+b)(a+c) = a+bc$$

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

LTL: temporal specification

Whenever the boss comes by my office, I will start working.
Once I start working, I will keep working until the telephone rings.

$$\mathbf{G}\left((b \to \mathbf{F}w)(w \to w\mathbf{U}r)\right) = \mathbf{G}\left(\left(b' + \mathbf{F}w\right)\left(w' + w\mathbf{U}r\right)\right)$$

Whenever the button is pressed, the dialog will appear.
The dialog will disappear after 1 minute of user inactivity.

$$\mathbf{G}\left((b \to \mathbf{F}d)(d \to \mathbf{F}t)\left(d \to d\mathbf{U}td'\right)\right)$$

- The timer t is an external event and is *not specified* here
- Difficult to say "x stays true until further notice"

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, $\mathbf{G}(\alpha \to \alpha \mathbf{U} \beta)$??
  - $x : \mathbf{N}\alpha$ 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 `constant`, `map2`, `scan`

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

- (`map2` makes **G** an applicative functor)
- 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 advantages and the current limitations of `Elm`. 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.pdfHigher-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.