Functional Reactive Programming (Elm)

Mateusz Kołaczek

Seminarium: Zaawansowane programowanie funkcyjne

13.05.2015
Evan Czaplicki

*Elm: Concurrent FRP for functional GUIs*, 2012

Evan Czaplicki

*Controlling Time and Space: understanding the many formulations of FRP*, 2014.
It's not easy to find a definition of FRP. It's even harder to find a meaningful one.

Placeholder for FRP definition.
Placeholder for FRP definition.

It’s not easy to find a definition of FRP. It’s even harder to find a meaningful one.
What I consider as FRP

A way to:

- express time varying values in a declarative way
What I consider as FRP

A way to:

- express time varying values in a declarative way
- react to real world events in structured manner
GUI programming is not easy

```javascript
$('#target').click(function() {
    ...
});

$('#target').blur(function() {
    ...
});

$(' "#target"').mousemove(function(event) {
    ...
});
```
Signal is a value, that changes over time. That’s all.
Signal is a value, that changes over time. That’s all.

But...

- we don’t have to update it explicitly, it just always has the most recent value
Signal is a value, that changes over time. That’s all.

**But...**

- we don’t have to update it explicitly, it just always has the most recent value
- change in a signal’s value propagates automatically to dependent signals
Signal is a value, that changes over time. That’s all.

**But...**

- we don’t have to update it explicitly, it just always has the most recent value
- change in a signal’s value propagates automatically to dependent signals
- it represents a mutable value in a functional world
Signal is a value, that changes over time. That’s all.

But...

- we don’t have to update it explicitly, it just always has the most recent value
- change in a signal’s value propagates automatically to dependent signals
- it represents a mutable value in a functional world

When combined with pureness and immutability, it produces clean and simple reactive code. It’s an escape hatch from callback hell. Or event listener hell.
Simple signals

Elm's signals are discrete, not continuous. They are completely event-driven.

Examples

- `Mouse.position`
- `Windows.dimensions`
- `Time.every Time.second`
- `Time.fps 60`
Simple signals

Examples

- `Mouse.position`
- `Windows.dimensions`
- `Time.every Time.second`
- `Time.fps 60`

Signals in Elm are your program’s connection to the ‘real world’. Elm’s signals are discrete, not continuous. They are completely event-driven.
Signal graph

Annoys the programmer.

Mouse.position

Window.dimensions

Time.fps

Transformations: map, merge...

Combined inputs

Foldp, Update

view

Rendered state

Updated state

Mateusz Kołaczek

Functional Reactive Programming (Elm)
Goal: sketch an implementation of a simple snake game. It will:

- show how a typical Elm program looks like
- familiarize us with signals

We’ll visit all parts of the diagram from the previous slide in order of their execution.
keys = Mouse.arrows

timer = Time.fps 10
keys = Mouse.arrows

But...

keys : Signal { x:Int, y:Int }

* `{ x = 0, y = 0 }` no arrows.
* `{ x =-1, y = 0 }` left arrow.
* `{ x = 1, y = 1 }` up and right arrows.
* `{ x = 0, y =-1 }` down, left, and right arrows.
keys = Mouse.arrows
timer = Time.fps 10

But...

keys : Signal { x:Int, y:Int }
* ‘{ x = 0, y = 0 }‘ no arrows.
* ‘{ x =-1, y = 0 }‘ left arrow.
* ‘{ x = 1, y = 1 }‘ up and right arrows.
* ‘{ x = 0, y =-1 }‘ down, left, and right arrows.

What we want to get is:

type Direction = Up | Down | Left | Right
pressedKey : Signal Maybe Direction
Transforming record to single direction is straightforward:

direction \( \text{dir} = \)

\[
\begin{align*}
\text{if} & \quad (\text{dir}.x = 1) \&\& (\text{dir}.y = 0) \rightarrow \text{Just Right} \\
& \quad (\text{dir}.x = -1) \&\& (\text{dir}.y = 0) \rightarrow \text{Just Left} \\
& \quad (\text{dir}.x = 0) \&\& (\text{dir}.y = 1) \rightarrow \text{Just Up} \\
& \quad (\text{dir}.x = 0) \&\& (\text{dir}.y = -1) \rightarrow \text{Just Down} \\
\text{otherwise} & \rightarrow \text{Nothing}
\end{align*}
\]
Transforming record to single direction is straightforward:

direction dir =
  if  | (dir.x==1) && (dir.y==0) → Just Right
  | (dir.x==-1) && (dir.y==0) → Just Left
  | (dir.x==0) && (dir.y==1) → Just Up
  | (dir.x==0) && (dir.y==-1) → Just Down
  | otherwise → Nothing

And
we’ll use a handy and well-known function:

map : (a → b) → Signal a → Signal b

pressedKey : Signal Maybe Direction
pressedKey = Signal.map direction Keyboard.arrows
So we have two sources of input, and want to update the state basing on them:

\[
\text{timer} = \text{Time.fds} \\
\text{pressedKey} = \text{Signal.map direction Keyboard.arrows}
\]

We need a signal carrying both those values at once.
Merging signals

So we have two sources of input, and want to update the state basing on them:

\[
\begin{align*}
timer &= \text{Time.fds} \\
\text{pressedKey} &= \text{Signal.map direction Keyboard.arrows}
\end{align*}
\]

We need a signal carrying both those values at once.

\[
\text{Signal.map2 : (a → b → c) → Signal a → Signal b → Signal c}
\]

\[
\text{Signal.map2 (,) pressedKey timer}
\]
So we have two sources of input, and want to update the state basing on them:

\[
\begin{align*}
\text{timer} &= \text{Time.fds} \\
\text{pressedKey} &= \text{Signal.map direction Keyboard.arrows}
\end{align*}
\]

We need a signal carrying both those values at once.

\[
\text{Signal.map2 : (a → b → c) → Signal a → Signal b → Signal c}
\]

\[
\text{Signal.map2 (,) pressedKey timer}
\]

But... We lose the way to distinguish what caused the update = TurboSnake.
Solution: union type.

type Update = Arrows (Maybe Direction) | Timer Float
Solution: union type.

```haskell
type Update = Arrows (Maybe Direction) | Timer Float
```

timer and pressedKey become:

```haskell
timer : Signal Update
timer = Signal.map Timer Time.fps
```

```haskell
pressedKey : Signal Update
pressedKey = Signal.map (Arrows <<< direction) Keyboard.arrows
```

(<<<) is just a function composition.
Still having two signals and no means to *merge* them? Merge to the rescue!

\[
\text{merge} : \text{Signal} \ a \rightarrow \text{Signal} \ a \rightarrow \text{Signal} \ a
\]

Merge interleaves two given signals producing merged one.
Still having two signals and no means to *merge* them? Merge to the rescue!

\[
\text{merge} : \text{Signal} \ a \rightarrow \text{Signal} \ a \rightarrow \text{Signal} \ a
\]

Merge interleaves two given signals producing merged one.

\[
\text{inputs} : \text{Signal} \ \text{Update}
\]

\[
\text{inputs} = \text{Signal}.\text{merge} \ \text{timer pressedKey}
\]

This signal carries the most recent update, *either* timer *or* keyboard update.
Folding signals

Goal: a signal that reflects the whole state of an application during execution.
Problem: this signal’s value not only depends on other signals, but also on it’s previous value

foldp : (a → state → state) → state → Signal a → Signal state

Simple example:
clickCount : Signal Int
clickCount = foldp (λ click total → total + 1) 0 Mouse.clicks

Real life (snake) example:
loop = Signal.foldp update initialState inputs
Goal: a signal that reflects the whole state of an application during execution.
Problem: this signal’s value not only depends on other signals, but also on its previous value.
We must fold from the past:

foldp : (a → state → state) → state → Signal a → Signal state
Folding signals

Goal: a signal that reflects the whole state of an application during execution.
Problem: this signal’s value not only depends on other signals, but also on its previous value.
We must fold from the past:

\[ \text{foldp} : (a \rightarrow \text{state} \rightarrow \text{state}) \rightarrow \text{state} \rightarrow \text{Signal} \ a \rightarrow \text{Signal} \ \text{state} \]

Simple example:

\[ \text{clickCount} : \text{Signal} \ \text{Int} \]
\[ \text{clickCount} = \text{foldp} \ (\lambda \text{click total} \rightarrow \text{total} + 1) \ 0 \ \text{Mouse.clicks} \]
Folding signals

Goal: a signal that reflects the whole state of an application during execution.
Problem: this signal’s value not only depends on other signals, but also on it’s previous value.
We must fold from the past:

\[
\text{foldp} : (a \rightarrow \text{state} \rightarrow \text{state}) \rightarrow \text{state} \rightarrow \text{Signal } a \rightarrow \text{Signal } \text{state}
\]

Simple example:

\[
\text{clickCount} : \text{Signal } \text{Int} \\
\text{clickCount} = \text{foldp } (\lambda \text{click } \text{total} \rightarrow \text{total} + 1) \ 0 \ \text{Mouse.clicks}
\]

Real life (snake) example:

\[
\text{loop} = \text{Signal.foldp} \ \text{update} \ \text{initialState} \ \text{inputs}
\]
What is a state?

State is just a value. A record, that encompasses whole state of application.

type alias Model =
  { snake : Snake,
    direction : Direction,
    pressedKey : Maybe Direction,
    gameOver : Bool,
    fruit : Maybe Position,
    seed : Random.Seed
  }

type alias Snake =
  { body : Queue.Queue Position,
    head : Position
  }
Time for lamentation

But hey, one global state? Where's the modularity? Nothing prevents you from dividing the model into smaller submodels, dividing inputs into subinputs, and updating into subupdates. The upside is, whole knowledge about the program is concentrated in one place.
‘But hey, one global state? Where’s the modularity? ‘
‘But hey, one global state? Where’s the modularity? ‘
Nothing prevents you from dividing model into smaller submodels, dividing inputs into subinputs, update into subupdates.

The upside is, whole knowledge about the program is concentrated in one place.
loop : Signal Model
loop = Signal.foldp update initialState inputs

inputs : Signal Update

initialState : Model

foldp : (a → state → state) → state → Signal a → Signal state

The type of foldp requires, that:

update : Update → Model → Model
loop : Signal Model
loop = Signal.foldp update initialState inputs

inputs : Signal Update
initialState : Model

foldp : (a → state → state) →
  state → Signal a → Signal state

The type of foldp requires, that:
update : Update → Model → Model

Function? What about the signals? FRP? Anything?
Game logic doesn’t contain any signals! We just get the inputs and previous state, and feed it to pure function for a new state. Unfortunately, that implies, that it’s not really interesting for us.
Game logic doesn’t contain any signals! We just get the inputs and previous state, and feed it to pure function for a new state. Unfortunately, that implies, that it’s not really interesting for us. But... Here’s the code:

```haskell
update : Update → Model → Model
update u state =
  if state.gameOver
    then state
    else
      case u of
        Timer _ → state |> updateFruit >> updateDirection >>
          updateSnake >> updateGameOver
        Arrows a → updatePressedKey a state
```
main : Signal Element

Element is an Elm representation of HTML element to display. We close the loop - after reacting to user input (declaratively), we produce the output (also declaratively).
main : Signal Element

Element is an Elm representation of HTML element to display. We close the loop - after reacting to user input (declaratively), we produce the output (also declaratively).

main = Signal.map2 view Window.dimensions loop

view : (Int, Int) → Model → Element
view (w',h') state = magical elm view code
main : Signal Element

Element is an Elm representation of HTML element to display. We close the loop - after reacting to user input (declaratively), we produce the output (also declaratively).

\[
\text{main} = \text{Signal.map2 view Window.dimensions loop}
\]

\[
\text{view} : (\text{Int, Int}) \rightarrow \text{Model} \rightarrow \text{Element}
\]

\[
\text{view} (w', h') \text{ state} = \text{magical elm view code}
\]

View code is interesting on its own, as Elm has its own API for drawing things on HTML page, but it’s not the topic of this presentation.
A typical Elm program consists of:

- a model
- a view
- inputs - signals are mainly here
- state update logic

It’s not forced in any way, but it emerges naturally from the way Elm is structured.
Elm is not the end of FRP

There are many implementations of FRP available. They can be roughly categorized:

- first order FRP (Elm is here!)
- higher order FRP (Fran)
- asynchronous data flow (FRP libraries in imperative languages)
- arrowized FRP (Netwire, brrrr...)

Mateusz Kołaczek

Functional Reactive Programming (Elm)
Signals are connected to the world
First order FRP

- Signals are connected to the world
- Signals are infinite
First order FRP

- Signals are connected to the world
- Signals are infinite
- Signal graphs are static
First order FRP

- Signals are connected to the world
- Signals are infinite
- Signal graphs are static
- Synchronous by default - events are processed in order they came, you can’t (by default) finish processing later event before the earlier

This gives a few nice properties:
- Simplicity and efficiency
- Good architecture emerges naturally
- Hot swapping
- Time travel debugging
First order FRP

- Signals are connected to the world
- Signals are infinite
- Signal graphs are static
- Synchronous by default - events are processed in order they came, you can’t (by default) finish processing later event before the earlier

This gives a few nice properties
- Simplicity and efficiency
Signals are connected to the world
Signals are infinite
Signal graphs are static
Synchronous by default - events are processed in order they came, you can’t (by default) finish processing later event before the earlier

This gives a few nice properties
- Simplicity and efficiency
- Good architecture emerges naturally

First order FRP
First order FRP

- Signals are connected to the world
- Signals are infinite
- Signal graphs are static
- Synchronous by default - events are processed in order they came, you can’t (by default) finish processing later event before the earlier

This gives a few nice properties
- Simplicity and efficiency
- Good architecture emerges naturally
- Hot swapping
First order FRP

- Signals are connected to the world
- Signals are infinite
- Signal graphs are static
- Synchronous by default - events are processed in order they came, you can’t (by default) finish processing later event before the earlier

This gives a few nice properties

- Simplicity and efficiency
- Good architecture emerges naturally
- Hot swapping
- Time travel debugging
• Signals are connected to the world
• Signals are infinite
• Signal graphs are *dynamic*
• Synchronous by default
Higher order FRP

- Signals are connected to the world
- Signals are infinite
- Signal graphs are *dynamic*
- Synchronous by default

We can create new signals, delete signals, reconnect them in different ways at runtime.

```
join : Signal (Signal a) → Signal a
```
Higher is better?

\[
\begin{align*}
\text{clickCount} &: \text{Signal Int} \\
\text{clickCount} &= \text{count Mouse.clicks}
\end{align*}
\]

Innocent, isn’t it?

Because \( \text{count Mouse.clicks} = \text{clickCount} \), the value must be 4.

Imagine a program running for a year without restarting, where suddenly such signal is switched on.
Higher is better?

\[
\text{clickCount} : \text{Signal} \ \text{Int} \\
\text{clickCount} = \text{count Mouse.clicks}
\]

Innocent, isn’t it?

\[
\text{clicksOrZero} : \text{Bool} \rightarrow \text{Signal} \ \text{Int} \\
\text{clicksOrZero} \ b = \text{if} \ b \ \text{then} \ \text{count Mouse.clicks} \ \text{else constant}
\]

Because \(\text{count Mouse.clicks} = \text{clickCount}\), the value must be 4.

Imagine a program running for a year without restarting, where suddenly such signal is switched on.
Higher is better?

\[
\text{clickCount : Signal \; Int} \\
\text{clickCount = count \; Mouse.clicks}
\]

Innocent, isn’t it?

\[
\text{clicksOrZero : Bool \rightarrow Signal \; Int} \\
\text{clicksOrZero \; b = if \; b \; then \; count \; Mouse.clicks \; else \; constant}
\]

True: Click, click. False: Click, Click. True: what is the value now?
higher is better?

clickCount : Signal Int
    clickCount = count Mouse.clicks

innocent, isn’t it?

    clicksOrZero : Bool → Signal Int
    clicksOrZero b = if b then count Mouse.clicks else constant

true: click, click. false: click, click. true: what is the value now?
because \text{count Mouse.clicks} = \text{clickCount}, the value must be 4.
Higher is better?

\[
\text{clickCount} : \text{Signal Int} \\
\text{clickCount} = \text{count Mouse.clicks}
\]

Innocent, isn’t it?

\[
\text{clicksOrZero} : \text{Bool} \rightarrow \text{Signal Int} \\
\text{clicksOrZero} \ b = \text{if } b \text{ then count Mouse.clicks else constant}
\]

True: Click, click. False: Click, Click. True: what is the value now? Because \( \text{count Mouse.clicks} = \text{clickCount} \), the value must be 4. Imagine a program running for a year without restarting, where suddenly such signal is switched on.
Switching the signal on (creating a new signal) may need looking back through whole history. That means, memory usage grows linearly over time.

Possible solution: restrict signals with complicated types (linear types) to allow only safe signals.

Pros:
- we can reconfigure the graph!

Drawbacks:
- not simple at all
- possibly no hot-swapping and time travel debugger
- program architecture might get messier
Switching the signal on (creating a new signal) may need looking back through whole history. That means, memory usage grows linearly over time.
Possible solution: restrict signals with complicated types (linear types) to allow only safe signals.
Switching the signal on (creating a new signal) may need looking back through whole history. That means, memory usage grows linearly over time.
Possible solution: restrict signals with complicated types (linear types) to allow only safe signals.
Pros:
- we can reconfigure the graph!
Switching the signal on (creating a new signal) may need looking back through whole history. That means, memory usage grows linearly over time.

Possible solution: restrict signals with complicated types (linear types) to allow only safe signals.

Pros:
- we can reconfigure the graph!

Drawbacks:
- not simple at all
Switching the signal on (creating a new signal) may need looking back through whole history. That means, memory usage grows linearly over time.

Possible solution: restrict signals with complicated types (linear types) to allow only safe signals.

Pros:

- we can reconfigure the graph!

Drawbacks:

- not simple at all
- possibly no hot-swapping and time travel debugger
Switching the signal on (creating a new signal) may need looking back through whole history. That means, memory usage grows linearly over time.

Possible solution: restrict signals with complicated types (linear types) to allow only safe signals.

Pros:
- we can reconfigure the graph!

Drawbacks:
- not simple at all
- possibly no hot-swapping and time travel debugger
- program architecture *might* get messier
Asynchronous data flow

Examples: ReactiveCocoa, ReactiveExtensions, bacon.js
Examples: ReactiveCocoa, ReactiveExtensions, bacon.js

- Signals are connected to the world
- Signals are *finite*
- Signal graphs are *dynamic*
- Asynchronous by default
Examples: ReactiveCocoa, ReactiveExtensions, bacon.js

- Signals are connected to the world
- Signals are $finite$
- Signal graphs are $dynamic$
- Asynchronous by default

If your FRP is in imperative language, it probably falls into this category.
Asynchronous data flow is FRP for imperative languages. There is no requirement, that two same expressions yield same values. When we create new signal, we just start counting from zero.
Asynchronous data flow is FRP for imperative languages. There is no requirement, that two same expressions yield same values. When we create new signal, we just start counting from zero. Another problem is what to do with signals, which is no one listening to. Some libraries (for example ReactiveExtensions) provide a distinction to hot and cold signals. The first always update, the second just stop.
• Signals are *not* connected to the world
• Signals are infinite
• Signal graphs are *dynamic*
• Synchronous by default
Arrowized FRP

- Signals are *not* connected to the world
- Signals are infinite
- Signal graphs are *dynamic*
- Synchronous by default

AFRP can be embedded in first order FRP as a library. In Elm it’s Automaton.
pure : (a → b) → Automaton a b
Elm’s automaton API

\[
\text{pure} : (a \rightarrow b) \rightarrow \text{Automaton } a \ b \\
\text{plus}1 = \text{pure } (\lambda n \rightarrow n + 1)
\]
Elm’s automaton API

pure : (a → b) → Automaton a b

plus1 = pure (λn → n + 1)

(>>>) : Automaton a b → Automaton b c → Automaton a c
Elm's automaton API

pure : (a → b) → Automaton a b

plus1 = pure (λn → n + 1)

(>>>): Automaton a b → Automaton b c → Automaton a c

plus2 = plus1 >>> plus1
Elm’s automaton API

pure : (a → b) → Automaton a b

plus1 = pure (λn → n + 1)

(>>>) : Automaton a b → Automaton b c → Automaton a c

plus2 = plus1 >>> plus1

state : s → (a → s → s) → Automaton a s
Elm’s automaton API

```haskell
pure : (a → b) → Automaton a b

plus1 = pure (λn → n + 1)

(>>>): Automaton a b → Automaton b c → Automaton a c

plus2 = plus1 >>> plus1

state : s → (a → s → s) → Automaton a s

count : Automaton a Int
count = state 0 (λa total → total + 1)
```
An automaton can have state, that gets updated every time it receives input. When you switch the automaton of the signal graph, it doesn’t receive any input, so its state doesn’t change. That eliminates the lookback problem.
An automaton can have state, that gets updated every time it receives input. When you switch the automaton of the signal graph, it doesn’t receive any input, so its state doesn’t change. That eliminates the lookback problem.

In general, when you build program in ‘standard‘ Elm, the main building block are still functions, signal are somewhere at the top. When using Arrowized FRP, whole logic is expressed in terms of signals.
The tour is over

Questions?