Optimizing Signal Graphs for Functional-Reactive Programs

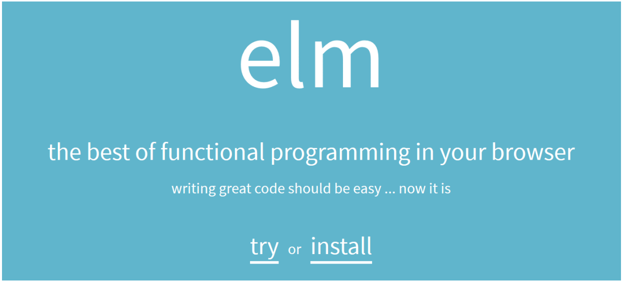
Janis Voigtländer

University of Bonn

July 28th, 2015

Elm – An FRP Language (`www.elm-lang.org`)

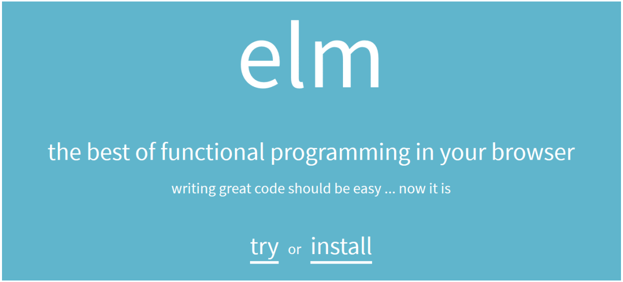
Elm – An FRP Language (`www.elm-lang.org`)



For me, primarily a teaching tool, using it for:

- beginning programmers at high school level

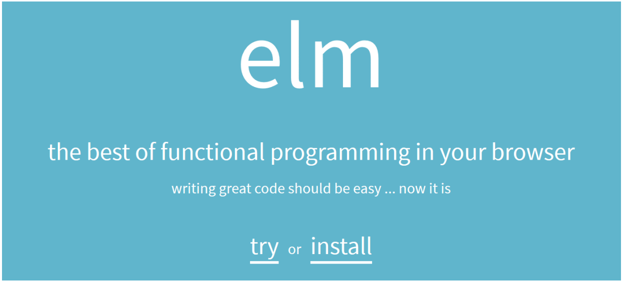
Elm – An FRP Language (`www.elm-lang.org`)



For me, primarily a teaching tool, using it for:

- beginning programmers at high school level
- a declarative programming course at university

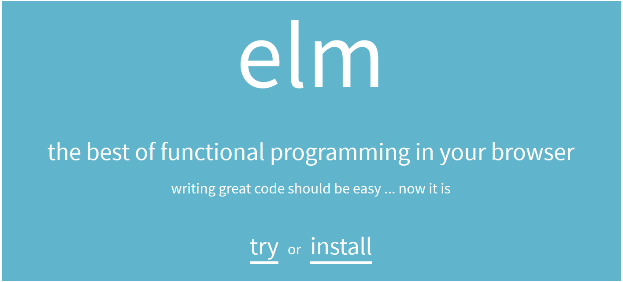
Elm – An FRP Language (`www.elm-lang.org`)



For me, primarily a teaching tool, using it for:

- beginning programmers at high school level
- a declarative programming course at university
- projects/theses on language implementation ?

Elm – An FRP Language (`www.elm-lang.org`)



For me, primarily a teaching tool, using it for:

- beginning programmers at high school level
- a declarative programming course at university
- projects/theses on language implementation ?

A Simple Elm Program

Signals ...

```
behavior : Signal (Time, (Int, Int))
behavior = timestamp (Signal.sampleOn (Time.every second)
                                      Mouse.position)
```

A Simple Elm Program

Signals ...

```
behavior : Signal (Time, (Int, Int))
behavior = timestamp (Signal.sampleOn (Time.every second)
                                      Mouse.position)
```

... and functions:

```
view (t, (x, y)) =
  let
    (cx, cy) = (100 * cos t, 100 * sin t)
    (px, py) = (toFloat x - 100, 100 - toFloat y)
  in
    collage 200 200 [move (cx, cy) (filled red (circle 10)),
                     traced defaultLine (path [(cx, cy),
                                               (px, py)])]
```

A Simple Elm Program

Signals ...

```
behavior : Signal (Time, (Int, Int))
behavior = timestamp (Signal.sampleOn (Time.every second)
                                      Mouse.position)
```

... and functions:

```
view (t, (x, y)) =
  let
    (cx, cy) = (100 * cos t, 100 * sin t)
    (px, py) = (toFloat x - 100, 100 - toFloat y)
  in
    collage 200 200 [move (cx, cy) (filled red (circle 10)),
                     traced defaultLine (path [(cx, cy),
                                               (px, py)])]

main = Signal.map view behavior
```

A Simple Elm Program ... and its Signal Graph

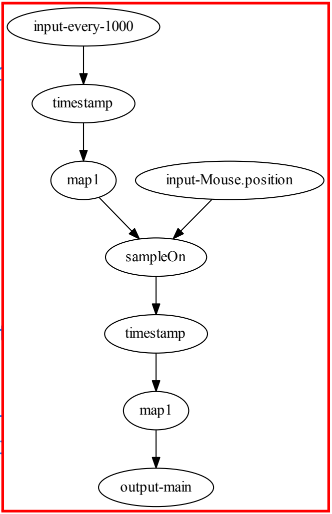
Signals ...

$\texttt{behavior}$: Signal (Time, (Int, Int))
$\texttt{behavior} = \texttt{timestamp}$ (Signal.$\texttt{sampleOn}$

... and functions:

$\texttt{view}\ (t, (x, y)) =$
let
$(cx, cy) = (100 * \texttt{cos}\ t, 100 * \texttt{sin}\ t)$
$(px, py) = (\texttt{toFloat}\ x - 100, 100 -$
in
$\texttt{collage}\ 200\ 200\ [\texttt{move}\ (cx, cy)\ (\texttt{fil}$
$\texttt{traced defaultL}$

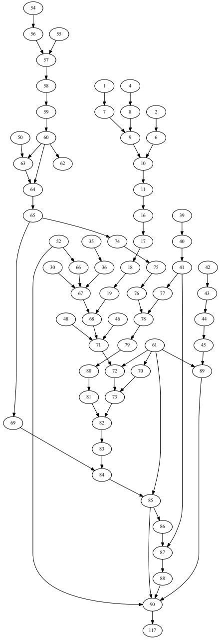
$\texttt{main} =$ Signal.$\texttt{map view behavior}$



Optimizing Signal Graphs

Why ?

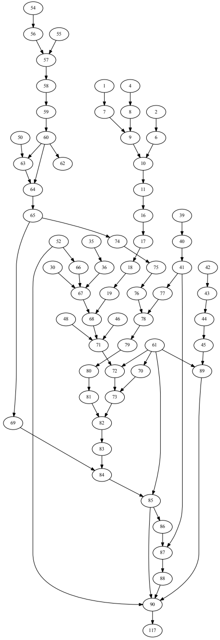
- communication flow
  structure / overhead



Optimizing Signal Graphs

Why ?

- communication flow
  structure / overhead
- avalanches of '`no-update`'s



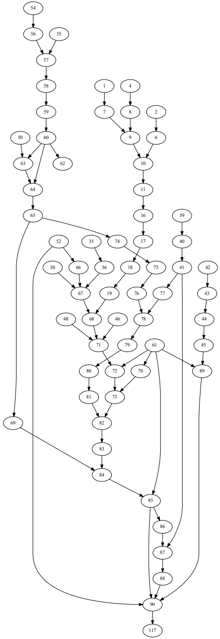
Optimizing Signal Graphs

Why ?

- communication flow structure / overhead
- avalanches of '`no-update`'s

How ?

- as a start, collapse chains of nodes



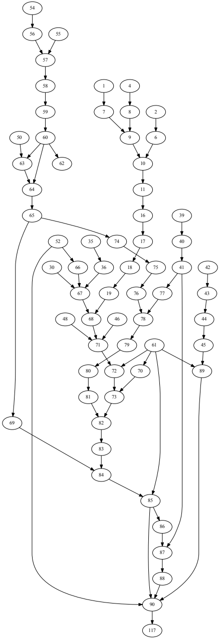
Optimizing Signal Graphs

Why ?

- communication flow structure / overhead
- avalanches of '`no-update`'s

How ?

- as a start, collapse chains of nodes
- by some kind of syntactic fusion ?



Fusion of Signal Primitives

A simple case:

$$\text{Signal.}\texttt{map}\ f\ (\text{Signal.}\texttt{map}\ g\ \textit{signal})$$
$$\rightsquigarrow$$
$$\text{Signal.}\texttt{map}\ (f \ll g)\ \textit{signal}$$

Fusion of Signal Primitives

A simple case:

$$\text{Signal.}\texttt{map}\ f\ (\text{Signal.}\texttt{map}\ g\ \mathit{signal})$$
$$\rightsquigarrow$$
$$\text{Signal.}\texttt{map}\ (f \ll g)\ \mathit{signal}$$

Further candidates:

- Time.`timestamp`
- Signal.`dropRepeats`
- Signal.`filter`
- Signal.`filterMap`
- Signal.`foldp`

Problems with Syntactic Fusion

A not so simple case:

$$\text{Signal.}\texttt{map}\ f\ (\text{Signal.}\texttt{foldp}\ g\ k\ \mathit{signal})$$

$$\rightsquigarrow$$

$$???$$

Problems with Syntactic Fusion

A not so simple case:

$$\text{Signal.}\texttt{map}\ f\ (\text{Signal.}\texttt{foldp}\ g\ k\ \mathit{signal})$$
$$\rightsquigarrow$$
$$???$$

Actually detecting fusable chains:

$\mathit{signal}_1 = \text{Signal.}\texttt{map}\ g\ \mathit{signal}$
$\mathit{signal}_2 = \text{Signal.}\texttt{map}\ f\ \mathit{signal}_1$ -- inline signal_1 ?

Problems with Syntactic Fusion

A not so simple case:

$$\text{Signal.}\texttt{map}\ f\ (\text{Signal.}\texttt{foldp}\ g\ k\ \mathit{signal})$$

$$\rightsquigarrow$$

$$???$$

Actually detecting fusable chains:

$\mathit{signal}_1 = \text{Signal.}\texttt{map}\ g\ \mathit{signal}$

$\mathit{signal}_2 = \text{Signal.}\texttt{map}\ f\ \mathit{signal}_1$ -- inline signal_1?

$\mathit{signal}_3 = \texttt{do-whatever-with}\ \mathit{signal}_1$ -- what now?

Phase Separation

Fact: Signal graphs in Elm are static (once created).

Phase Separation

Fact: Signal graphs in Elm are static (once created).

The types make it so!

Phase Separation

Fact: Signal graphs in Elm are static (once created).

The types make it so!

Conceptually, 3 phases in executing an Elm program:

1. compiling Elm to JavaScript;
2. running some JavaScript, setting up the signal graph of nodes, which embed further JavaScript;
3. sending events to the signal graph, running the JavaScript embedded in nodes.

Phase Separation

Signal graph construction: 'red' code.
Pure functions in nodes: 'green' code.

```
behavior = timestamp (Signal.sampleOn (Time.every second)
                                      Mouse.position)

view (t, (x, y)) =
  let
    (cx, cy) = (100 * cos t, 100 * sin t)
    (px, py) = (toFloat x - 100, 100 - toFloat y)
  in
    collage 200 200 [move (cx, cy) (filled red (circle 10)),
                     traced defaultLine (path [(cx, cy),
                                               (px, py)])]

main = Signal.map view behavior
```

Phase Separation

Signal graph construction: 'red' code.
Pure functions in nodes: 'green' code.

$$\texttt{behavior} = \texttt{timestamp}\ (\mathsf{Signal.}\texttt{sampleOn}\ (\mathsf{Time.every}\ \mathit{second})\ \mathsf{Mouse.}\texttt{position})$$

$$\begin{aligned}
&\texttt{view}\ (t, (x, y)) = \\
&\quad \textbf{let} \\
&\qquad (cx, cy) = (100 * \texttt{cos}\ t, 100 * \texttt{sin}\ t) \\
&\qquad (px, py) = (\texttt{toFloat}\ x - 100, 100 - \texttt{toFloat}\ y) \\
&\quad \textbf{in} \\
&\qquad \texttt{collage}\ 200\ 200\ [\texttt{move}\ (cx, cy)\ (\texttt{filled red}\ (\texttt{circle}\ 10)), \\
&\qquad\qquad \texttt{traced defaultLine}\ (\texttt{path}\ [(cx, cy), (px, py)])]
\end{aligned}$$

$$\texttt{main} = \mathsf{Signal.}\texttt{map}\ \texttt{view}\ \texttt{behavior}$$

And there can be some 'yellow' code as well.

Thus Motivated Optimization Strategy

1. Wait until all ‘red’ (and maybe some ‘yellow’) JavaScript code has run.

Thus Motivated Optimization Strategy

1. Wait until all ‘red’ (and maybe some ‘yellow’) JavaScript code has run.
2. Traverse and shrink the signal graph, potentially moving around ‘green’ JavaScript function objects (which might reference ‘yellow’ ones).

Thus Motivated Optimization Strategy

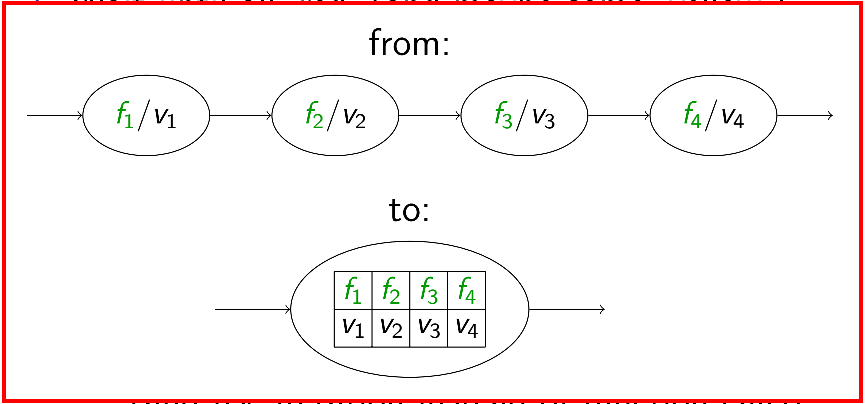
1. Wait until all ‘red’ (and maybe some ‘yellow’) JavaScript code has run.
2. Traverse and shrink the signal graph, potentially moving around ‘green’ JavaScript function objects (which might reference ‘yellow’ ones).
   - Create ‘fat nodes’ that do the work of a whole chain of nodes

Thus Motivated Optimization Strategy

1. Wait until all ‘red’ (and maybe some ‘yellow’) JavaScript code has run.
2. Traverse and shrink the signal graph, potentially moving around ‘green’ JavaScript function objects (which might reference ‘yellow’ ones).
   - Create ‘fat nodes’ that do the work of a whole chain of nodes,
   - but that do short-circuit when appropriate (and use iteration instead of function calls).

Thus Motivated Optimization Strategy

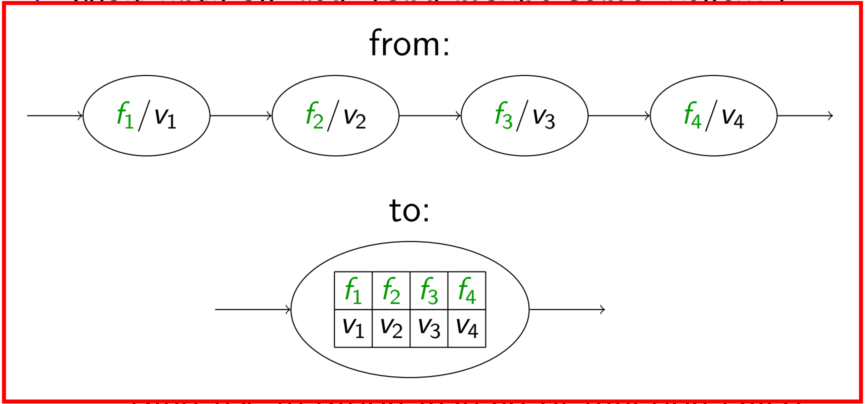
1. Wait until all ‘red’ (and maybe some ‘yellow’)



(and use iteration instead of function calls).

Thus Motivated Optimization Strategy

1. Wait until all ‘red’ (and maybe some ‘yellow’)

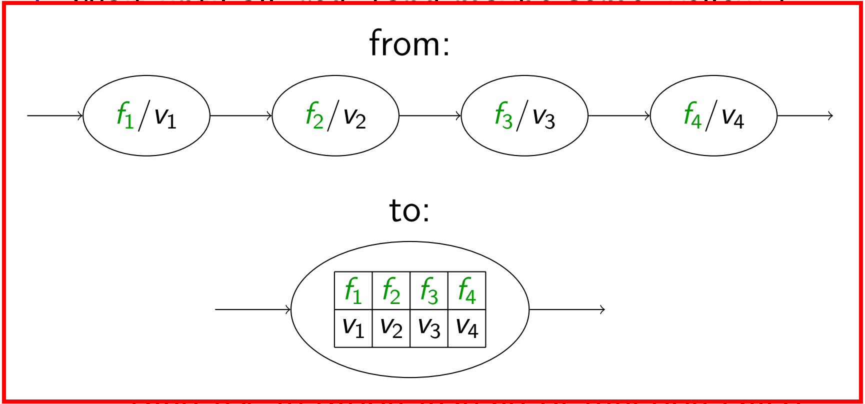


(and use iteration instead of function calls).

Sounds easy.

Thus Motivated Optimization Strategy

1. Wait until all 'red' (and maybe some 'yellow')

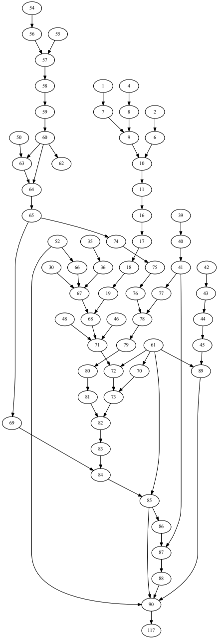


(and use iteration instead of function calls).

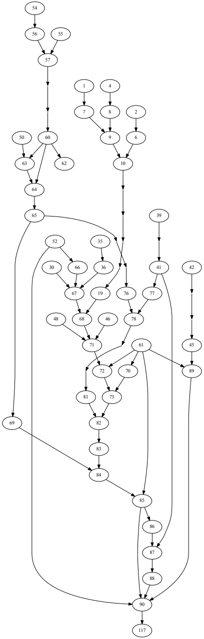
Sounds easy. Well, yes, but as always the devil is in the details. For example, it turns out JavaScript is an imperative language with mutable state...

So, Does it Work?

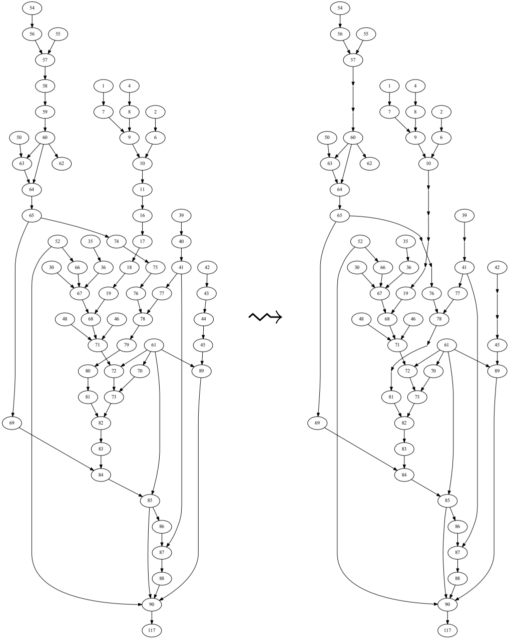
So, Does it Work? – Yes!



$\rightsquigarrow$



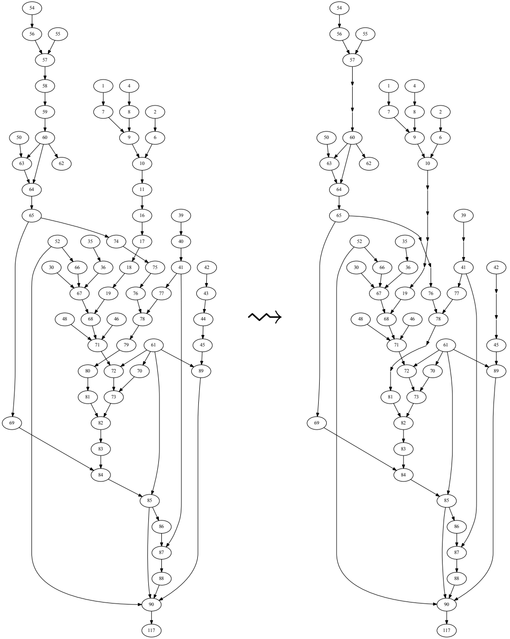
So, Does it Work? – Yes!



Open:

- evaluate impact on performance (beyond anecdotal)

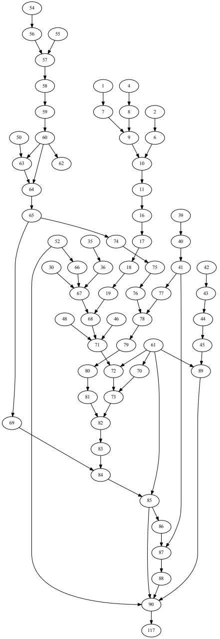
So, Does it Work? – Yes!



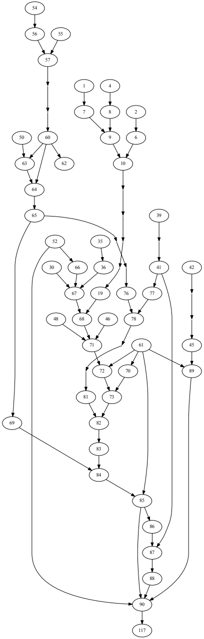
Open:

- evaluate impact on performance (beyond anecdotal)
- impact on debugging, hot-swapping ?

So, Does it Work? – Yes!



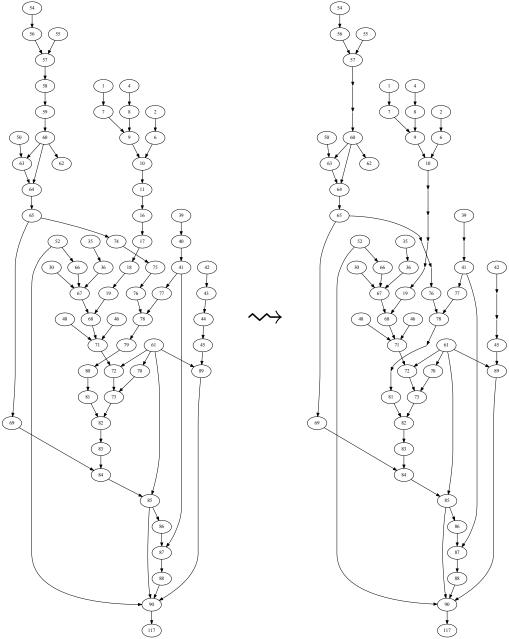
$\rightsquigarrow$



Open:

- evaluate impact on performance (beyond anecdotal)
- impact on debugging, hot-swapping ?
- deeper fusion (of ‘green’ functions) ?

So, Does it Work? – Yes!



Open:

- evaluate impact on performance (beyond anecdotal)
- impact on debugging, hot-swapping ?
- deeper fusion (of 'green' functions) ?
- other optimizations, . . .