Asynchronous Functional Reactive Programming for GUIs

Evan Czaplicki
Harvard University
evan.czaplicki@post.harvard.edu

Stephen Chong
Harvard University
chong@seas.harvard.edu

Abstract
Graphical user interfaces (GUIs) mediate many of our interactions with computers. Functional Reactive Programming (FRP) is a promising approach to GUI design, providing high-level, declarative, compositional abstractions to describe user interactions and time-dependent computations. We present Elm, a practical FRP language focused on easy creation of responsive GUIs. Elm has two major features: simple declarative support for Asynchronous FRP; and purely functional graphical layout.

Asynchronous FRP allows the programmer to specify when the global ordering of event processing can be violated, and thus enables efficient concurrent execution of FRP programs; long-running computation can be executed asynchronously and not adversely affect the responsiveness of the user interface.

Layout in Elm is achieved using a purely functional declarative framework that makes it simple to create and combine text, images, and video into rich multimedia displays.

Together, Elm’s two major features simplify the complicated task of creating responsive and usable GUIs.

Categories and Subject Descriptors D.3.2 [Language Classifications]: Data-flow languages; Applicative (functional) languages

General Terms Languages, Design

Keywords Functional Reactive Programming, Graphical User Interfaces

1. Introduction

Elm is a functional reactive programming language that aims to simplify the creation of responsive graphical user interfaces (GUIs), and specifically targets GUIs for web applications. Functional reactive programming (FRP) applies pure functional programming paradigms to time-varying values, known as signals. FRP is a highly promising approach for implementing GUIs, where time-varying values can represent input and output (including user interaction, server requests and responses), and other information about the execution environment. By enforcing a purely functional programming style, programmers can explicitly model complex time-dependent relationships in a high-level declarative way.

However, previous FRP languages and implementations have suffered from at least two kinds of inefficiencies: unnecessary recomputation and global delays.

Semantics of most FRP languages assume that signals change continuously. Thus, their implementations sample input signals as quickly as possible, and continually recompute the program with the latest values for the signals. In practice, however, many signals change discretely and infrequently, and so constant sampling leads to unnecessary recomputation. By contrast, Elm assumes that all signals are discrete, and uses this assumption to detect when a signal is unchanged, and in that case, avoid unnecessary recomputation.

In Elm, signals change only when a discrete event occurs. An event occurs when a program input (such as the mouse position) changes. Events require recomputation of the program, as the result of the program may have changed. Previous FRP systems require that events are processed synchronously: one at a time in the exact order of occurrence. In general, synchronization is required to allow the programmer to reason about the behavior of the FRP system, and ensure correct functionality.

However, processing an event may take significant time, resulting in delays to the entire FRP system. Pipelining event processing can help to reduce latency, but because the global order of events must be respected, an event cannot finish processing until all previous events have finished. In GUIs, this is unacceptable behavior: the user interface should remain responsive, even if a previous user action triggered a long-running computation.

Elm provides programmers with a simple abstraction to specify when computation can occur asynchronously. Combined with pipelined execution of event processing, this allows long-running computation to execute concurrently with other event processing, avoids global delays, and allows GUIs to remain responsive.

The ability to specify asynchronous computation within an FRP paradigm is the key novelty of Elm. We formalize this language feature by presenting the semantics of a core Asynchronous FRP calculus. However, Elm is also a practical and expressive programming language, which we demonstrate through the implementation of an Elm compiler, and the use of the resulting compiler to develop rich, responsive GUIs that perform non-trivial computation. The Elm compiler produces JavaScript code that can be immediately and easily deployed in web applications. Elm is publicly available.

Like previous work on efficient FRP (e.g., [25, 30, 31]), Elm restricts use of signals to enable efficient implementation. Elm is strictly more expressive than these previous approaches, since (1) discrete signals generalize continuous signals [31], (2) we embed a discrete version of Arrowized FRP [25] in Elm, and (3) we additionally allow the programmer to specify when signal updates should be computed asynchronously.

The rest of the paper is structured as follows. In Section 2 we describe key features of Elm through several simple Elm programs. In Section 3 we present a formalization of the core language of Elm, including semantics and a type system. The full Elm language

Available at http://elm-lang.org/
2. Elm through Examples

We present key features of Elm through several simple Elm programs. The first example highlights Elm’s purely functional graphical layout. The second example demonstrates how Elm’s graphics primitives use FRP to easily produce rich interactive GUIs. The third example shows Elm’s async construct, which allows programmers to specify that potentially long-running computations should execute asynchronously, and thus ensures that the user interface remains responsive.

Example 1  The following Elm program lays out three graphical elements vertically, as shown in Figure 1.

```elm
content = flow down [ plainText "Welcome to Elm!" , image 150 50 "flower.jpg" , asText (reverse [1..9]) ]

main = container 180 100 middle content
```

Variable `main` is distinguished in Elm: the value of this variable will be displayed on the screen when the program is executed. In the program above, `main` is defined to be a 180 by 100 container with content defined by variable `content`. The content is placed in the middle of the container. (Positioning elements is notoriously difficult in HTML: Elm provides a simple abstraction, allowing the position of content within a container to be specified as `topLeft`, `midTop`, `topRight`, `midLeft`, `middle`, and so on.)

Variable `content` has type `Element`, indicating that it is a graphical element that can be displayed, and/or composed with other graphical elements. The value of `content` at runtime is a composite element, comprising three elements stacked vertically. The three elements are the text "Welcome to Elm!", a 150 by 50 image, and the text representation of a list containing numbers from 9 down to 1, which are combined together using function `flow : Direction -> [Element] -> Element` (i.e., `flow` is a function that takes a value of type `Direction`, a list of `Elements`, and produces an `Element`).

Values of type `Element` occupy a rectangular area of the screen when displayed, making `Elements` easy to compose. Elm provides primitives and library functions to create more complex `Element` values, and non-rectangular graphical elements. Elm and its libraries provide a simple declarative syntax and semantics, making layout easy to reason about.

Example 2  This example displays the position of the mouse pointer, as seen in Figure 2. Although extremely simple to describe, this is often head-scratchingly difficult to implement in today’s GUI frameworks, since the content is dynamically updated. In Elm, however, it is a one liner:

```elm
main = lift asText Mouse.position
```

This code relies on `signals`, which are the key abstraction of FRP. A signal is a value that changes over time. In Elm, a signal that represents a value of type `τ` changing over time has type `Signal τ`. For example, a graphical element that changes over time, such as an animation, is a `Signal Element`. Indeed, variable `main` typically has type `Signal Element`, since what to display on the screen changes over time.

In the code above, the position of the mouse is represented as a signal of a pair of integers, indicating the coordinates of the mouse.

```elm
Mouse.position : Signal (Int,Int)
```

Function `lift : (a -> b) -> Signal a -> Signal b` takes a function from values of type `a` to values of type `b`, and a signal of values of type `a`, and applies the function to each value, resulting in a signal of type `b`. In the example above, function `asText : a -> Element` (which converts Elm values into a textual representation) is applied to every value of the signal `Mouse.position`, thus converting a signal of coordinates into a signal of `Elements`. As the position of the mouse changes, so does the `Element` that is being displayed. This is shown in Figure 2.

In Elm, all signals are discrete. That means that instead of the system needing to frequently sample a continuously changing value (i.e., the system pulls values from the external environment), the system waits to be notified when the value has changed (i.e., values are pushed to the system only when they change). Elm is a push-based system [12], meaning that computation is performed only when values change. This reduces needless recomputation. In the example above, function `asText` is only applied to a mouse coordinate when the mouse position changes.

Example 3  This example highlights Elm’s ability to perform asynchronous computation over signals. It uses words entered by the user to find and fetch an image from a web service (such as Flickr), which may take significant time. The program simultaneously displays the current position of the mouse, with the position being updated in a timely manner, regardless of how long image fetching takes.

```elm
(getImage tags) = Input.text "Enter a tag"

getImage tags =
  lift (fittedImage 300 200)
  (asyncGet (lift requestTag tags))

scene input pos img =
  flow down [ input, asText pos, img ]

main = lift3 scene inputField
        Mouse.position
        (async (getImage tags))
```

[2] Variable `main` is also allowed to have type `Element`, as in Example 1.
Code `Input.text "Enter a tag"` creates a text input field, and returns a pair of `InputField`, `tag`), where `InputField` is a signal of graphical elements for the input field, and `tag` is a signal of strings. Each time the text in the input field changes (due to the user typing), both signals produce a new value: the updated element and the current text in the input field.

Function `getImage` takes a signal of strings, and returns a signal of images. For each string that the user enters, this function requests from the server a URL of an image with tags (i.e., keywords) that match the string. Function `requestTag` takes a string, and constructs an appropriate HTTP request to the server. (We elide the definition of `requestTag`, which simply performs string concatenation.) Expression `lift requestTag tags` is thus a signal of HTTP requests. Built-in function `syncGet` issues the requests, and evaluates to a signal of image URLs (actually, a signal of JSON objects returned by the server requests; the JSON objects contain image URLs). Function `fittedImage` (definition elided) takes dimensions and an image URL and constructs an image `Element`.

Function `scene` takes as its arguments `input` (an `Element`), `pos` (an arbitrary value) and `img` (an `Element`), and composes them vertically.

Value `main` puts the pieces together, applying function `scene` to three signals: the signal of elements representing the text field in which the user types, the mouse position (`Mouse.position`), and the signal of images derived from the user’s input (`getImage` tags). Primitive function `lift3` is similar to function `lift` described above, but takes a function `a -> b -> c -> d`, a signal of values of type `a`, a signal of values of type `b`, and a signal of values of type `c`, and applies the function to the current values of the three signals, each time any of the three signals produces a new value.

Keyword `async` indicates that the `getImage` computation should run asynchronously. Without the `async` keyword, the program must respect the global ordering of events: when the `tags` signal produces a new string, processing of new `Mouse.position` values must wait until an image is fetched from the server. The `async` keyword the program does not have to respect the order of values generated by `getImage` tags with respect to values generated by `Mouse.position`. The GUI remains responsive, regardless of how long it takes to fetch an image from the web service.

The `async` construct can be applied to any signal, and provides a simple, composable way for the programmer to specify when computation over a signal may occur asynchronously, and thus to ensure that long-running computation does not cause the GUI to become unresponsive.

The behavior of this example Elm program is difficult to implement in most practical programming frameworks. For example, implementation with JavaScript+AJAX requires nested call-backs. Also, such an example is impossible to implement in current FRP languages as no previous work permits explicit asynchrony and concurrent execution.

3. Core Language

The full Elm language contains libraries and syntactic sugar that simplify the creation of rich and responsive GUIs. In this section, we present a simple core language—FElm, for “Featherweight Elm”—that presents the semantics of Elm’s key abstractions.

FElm combines a simply-typed functional language with a small set of reactive primitives. Programmers have direct access to signals, which can be transformed and combined with the full power of a functional language. A type system restricts the use of reactive primitives to ensure that the program can be executed efficiently. FElm uses a two-stage semantics. In the first stage, functional constructs are evaluated, and signals are left uninterpreted. In the second stage, signals are evaluated: as the external environment generates new values for input signals, the new values are processed as appropriate.

3.1 Syntax

The syntax of FElm is presented in Figure 3. Expressions include the unit value (), integers `n`, variables `x`, functions `λx.η.e`, applications `e1 e2`, let expressions `let x = e1 in e2`, and conditional expressions `if e1 e2 e3` which are all standard constructs for functional languages. (Conditional expression `if e1 e2 e3` evaluates to `e2` if `e1` evaluates to a non-zero value, `e3` otherwise.) Metavariable `⊕` ranges over total binary operations on integers.

We assume a set of identifiers `Input` that denote input signals from the external environment, and use `i` to range over these identifiers. Signals can be thought of as streams of values. For example, an input signal representing the width of a window can be thought of as a stream of integer values such that every time the window width changes, a new value for the signal is generated. An input signal representing whether the left mouse button is currently pressed down may be represented as a stream of boolean values. Input signals may also include special signals, for example, to generate a new unit value at fixed time intervals, creating a timer.

An `event` occurs when an input signal generates a new value. Events may trigger computation. In FElm, every signal always has a “current” value: it is the most recent value generated by the signal, or, for an input signal that has not yet generated a new value, it is an appropriate default value associated with that input signal. Thus, every input signal is required to have a default value, which then induces default values for other signals.

The remaining expressions (`lift0 e1 ... en, foldp e1 e2 e3`, and `async e`) are primitives for manipulating signals, and are described in more detail below.

Transforming and combining signals For each natural number `n ≥ 0`, we have primitive `liftn e e1 ... en` which allows a function `e` to transform and combine signals `e1 ... en`.

Intuitively, if `e` is a function of type `τ1 → τ2 → ··· → τn → τ`, and expressions `e1` are of type signal `τi` respectively, then `liftn e e1 ... en` applies function `e` to the values from signals `e1 ... en`, and has type signal `τ` (i.e., it is a signal that produces values of type `τ`). For example,

```
  lift1 (λx:int. x+x) Window.width
```

is a signal of integers obtained by doubling the values from input signal `Window.width`.

Figure 3. Syntax of FElm
Signals can also be combined. For example, expression

\[
\text{lift}_2 (\text{Ax}: \text{int} \cdot \text{Ay}: \text{int} \cdot \text{y} \cdot \text{Ax} \cdot \text{Ay}) \cdot \text{Mouse.x Window.width}
\]
takes two input signals of integers (\text{Mouse.x} and \text{Window.width}), and produces a signal of integers by combining values from the two input signals. It computes the relative position of the mouse’s \(x\)-coordinate with respect to the window width. Note that the relative position is computed each time an event occurs for either input signal. Moreover, events are globally ordered, and signal computation is synchronous in that it respects the global order of events: the sequence of relative positions will reflect the order of events on \text{Mouse.x} and \text{Window.width}.

**Past-dependent transformation** The \text{lift}_n primitives operate only on the current values of signals. By contrast, the \text{foldp} primitive can compute on both the current and previous values of a signal. Primitive \text{foldp} performs a fold on a signal from the past, similar to the way in which the standard function \text{foldl} performs a fold on lists from the left of the list. The type of \text{foldp} is

\[
(\tau \to \tau' \to \tau') \to \tau' \to \text{signal } \tau \to \text{signal } \tau'
\]

for any \(\tau\) and \(\tau'\). Consider \text{foldp} \(e_1 e_2 e_3\). Here, \(e_1\) is a function of type \(\tau \to \tau' \to \tau'\), \(e_2\) is a value of type \(\tau'\) and \(e_3\) is a signal that produces values of type \(\tau\). Type \(\tau'\) is the type of the accumulator, and \(e_2\) is the initial value of the accumulator. As values are produced by signal \(e_3\), function \(e_1\) is applied to the new value and the current accumulator, to produce a new accumulator value. Expression \text{foldp} \(e_1 e_2 e_3\) evaluates to the signal of accumulator values.

For example, suppose that \text{Keyboard.lastPressed} is an input signal that indicates the latest key that has been pressed. Then the following signal counts the number of key presses. (We assume that which key is pressed is encoded as an integer.)

\[
\text{foldp} (\text{Ax: int} \cdot \text{Ax: int} \cdot \text{Ax: int} + 1) 0 \text{ Keyboard.lastPressed}
\]

**Asynchronous composition** The \text{async} primitive allows a program to specify when certain computations on signals can be computed asynchronously with respect to the rest of the program. This annotation allows the programmer to specify when it is permissible to ignore the global ordering of events. The use and benefit of this construct is described in more detail, including a thorough example, in Section 3.3.2.

### 3.2 Type System

\(\text{FElm}\)'s type system ensures that \(\text{FElm}\) programs cannot produce signals of signals. There are two kinds of types: \textit{simple types} \(\tau\) and \textit{signal types} \(\sigma\). Type syntax is presented in Figure 3. Simple types include base types (unit for the unit value and int for integers) and functions from simple types to simple types (\(\tau \to \tau'\)). Signal types \(\sigma\) include type \text{signal} \(\tau\) (for signals that produce values of type \(\tau\)), and functions that produce signal types (\(\tau \to \sigma \) and \(\sigma \to \sigma'\)).

The type system rules out programs that use signals of signals, for the following reason. Intuitively, if we have signals of signals, then after a program has executed for, say 10 minutes, we might create a signal that (through the use of \text{foldp}) depends on the history of an input signal, say \text{Window.width}. To compute the current value of this signal, should we use the entire history of \text{Window.width}? But that would require saving all history of \text{Window.width} from the beginning of execution, even though we do not know whether the history will be needed later. Alternatively, we could compute the current value of the signal just using the current and new values of \text{Window.width} (i.e., ignoring the history). But this would allow the possibility of having two identically defined signals that have different values, based on when they were created. We avoid these issues by ruling out signals of signals. This does not overly restrict the expressiveness of the language, which we discuss further in Section 3.3.

The typing judgment for FElm has the form \(\Gamma \vdash e : \eta\), indicating that expression \(e\) has type \(\eta\) under type context \(\Gamma\), which maps variables and input signal identifiers to types. Figure 4 presents inference rules for this judgment. The rules for the primitive operators that manipulate signals are as described above, and the remaining inference rules are mostly standard. Note that the typing rule for conditional expressions \(\text{if} e_1 e_2 e_3\) requires that the test expression \(e_1\) has type \(\text{int}\), and in particular, cannot be a signal. A program \(e\) is \textbf{well typed} if \(\Gamma_{\text{input}} \vdash e : \eta\) holds for a type environment \(\Gamma_{\text{input}}\) that maps every input \(i \in \text{Input}\) to a signal type.

### 3.3 Semantics

FElm programs are evaluated in two stages. In the first stage, all and only functional constructs are evaluated, resulting in a term in an intermediate language that clearly shows how signals are connected together. The intermediate language is similar to the source language of Real-Time FRP [30] and Event-Driven FRP [31]. In the second stage, signals are evaluated: new values arriving on input signals trigger computation in a push-based manner.

#### 3.3.1 Functional Evaluation

We define an intermediate language according to the grammar in Figure 5. The intermediate language separates values of simple types (unit, numbers, and functions over simple types), and signal terms (which will be the basis for the evaluation of signals).

A small step operational semantics for evaluation of FElm programs to the intermediate language is given in Figure 6. A FElm program \(e\) will evaluate to a final term, which is either a simple value \(v\), or a signal term \(s\). The inference rules use evaluation contexts \(E\) to implement a left-to-right call-by-value semantics. Rule \textbf{APPLICATION} converts function applications to let expres-
3.3.2 Signal Evaluation

If a FElm program evaluates to a simple value \( v \), then the program does not use signals, and is not a reactive program. If, however, it evaluates to a signal term \( s \) then we perform the second stage of evaluation, performing computation as input signals produce new values. We first present the intuition behind signal evaluation, then present a working example to solidify the intuition. We then provide a semantics for signal terms by translation to Concurrent ML (CML) [27].

**Intuition** A signal term can be visualized as a directed acyclic graph, where nodes are input signals (\( e \in \text{Input} \)), \( \text{lift}_n \) terms, and \( \text{foldp} \) terms. The definition and use of variables defines edges between the nodes. (Asynchronous terms \( \text{async} s \) are described below.) We refer to these visualizations as *signal graphs*.

Nodes representing input signals have no incoming edges from other signal nodes, a node representing term \( \text{lift}_n \) has \( n \) incoming edges, and a node representing term \( \text{foldp} v_1 v_2 s \) has a single incoming edge. Signal graphs are acyclic since FElm has no way to construct recursive signal terms. Figure 7 shows an example of a signal graph for a simple FElm program.

We refer to nodes with no incoming edges as *source nodes*. Source nodes include input signals, and, as we will see below, \( \text{async} s \) nodes. Source nodes are the source of new values. An *event* occurs when the signal represented by a source node produces a new value. Input signal events are triggered by the external environment, for example, mouse clicks or key presses. A *global event dispatcher* is responsible for notifying source nodes when events occur. The global event dispatcher ensures that events are totally ordered and no two events occur at exactly the same time. In signal graphs, we draw dashed lines from the global event dispatcher to all source nodes.

Nodes for \( \text{lift}_n \) and \( \text{foldp} \) terms perform computation on values produced by signals. Lift terms perform (pure) computation on the current values of signals: \( \text{term} \text{lift}_n v_1 ... v_n s \) applies function \( v \) to the current values of the \( n \) signals to produce a new value. A node for term \( \text{foldp} v_1 v_2 s \) maintains the current value of the accumulator (initially \( v_2 \)), and when a new value is received from signal \( s \), applies function \( v_1 \) to the new value and the current value of the accumulator, to produce the new value of the accumulator, which is also the new value for the signal \( \text{foldp} v_1 v_2 s \).

Conceptually, signal computation is *synchronous*: when an event occurs (i.e., a source node produces a new value), then it is as if the new value propagates completely through the signal graph before the next event is processed. However, if the actual semantics used this approach, then global delays would occur, as processing of a new event would be blocked until processing of all previous events has finished.

We maintain the simple synchronous semantics, but allow more efficient implementations, by pipelining the execution of the signal graph: conceptually, each node in a signal graph has its own thread of control to perform computation, and each edge in a signal graph holds an unbounded FIFO queue of values. Whenever an event occurs, *all* source nodes are notified by the global event dispatcher.
the one source node to which the event is relevant produces the new value, and all other source nodes generate a special value \texttt{noChange} \(v\), where \(v\) is the current (unchanged) value of the signal. Thus, every source node produces one value for each event. When a node in the signal graph performs computation, it must take a value from the front of each incoming edge’s queue: computation at the node is blocked until values are available on all incoming edges. If all incoming values indicate no change, then the node does not perform any new computation, but instead propagates the value \texttt{noChange} \(v\), where \(v\) is the latest value of the signal represented by the node. If any value on the incoming edge is a new value (i.e., is not a \texttt{noChange} \(v\) value), then the node performs computation (which, in the case of a \texttt{foldp} node, updates the state) and propagates the newly computed value. This preserves the simple synchronous semantics, but allows computation at each node to occur in parallel.

The \texttt{noChange} \(v\) values are a form of memoization—allowing nodes to avoid needless recomputation—but in the case of \texttt{foldp} nodes, are also critical to ensure correct execution. For example, a \texttt{foldp} term that counts the number of key presses (as in Section \ref{sec:example2}), should increment the counter only when a key is actually pressed, not every time any event occurs.

The synchrony of execution can be bypassed with the \texttt{async} construct. The node for a term \texttt{async} \(s\) is a source node: it has no incoming edges from other signal nodes, and whenever an event at another source node occurs, a \texttt{noChange} \(v\) value is propagated from the node. When signal \(s\) produces a new value, then a new event for term \texttt{async} \(s\) is generated. That is, when \(s\) produces a value, it is treated like an event from the external environment, and will be processed in a pipelined manner by the signal graph. The \texttt{async} construct, combined with pipelined execution of the signal graph, allows the programmer to easily separate long-running computations, while maintaining a relatively straightforward semantics for the FRP program. This allows a programmer to ensure that a GUI remains responsive even in the presence of significant computation triggered by user actions.

\textbf{Example} Consider the following program which takes in a signal of English words, pairing both the original word and the French translation of the word in a 2-tuple:

\[
\text{wordPairs} = \text{lift2} (,) \text{words} (\text{lift toFrench} \text{words})
\]

For simplicity, we assume that \texttt{words} is an input signal (i.e., words are given by the external environment), and function \texttt{toFrench} takes an English word and produces a French word, and may take significant time to perform the translation. Function \texttt{(,)} takes two arguments and constructs a pair from them. The signal graph for \texttt{wordPairs} is shown in Figure \ref{fig:example8a}. Although function \texttt{toFrench} may take significant time, it is important that the propagation of values from signal \texttt{words} is synchronous, as each English word must be matched up with its translation. This example motivates the need for FRPs to be synchronous: the programmer must be able to ensure that values from signals are matched up appropriately.

However, there is a tension between synchronization and fast updates. Consider an expression that combines our \texttt{wordPairs} signal with the mouse position.

\[
\text{lift2} (,) \text{wordPairs} \text{Mouse.position}
\]

The intent of this program is to display the current position of the mouse, in addition to translating English words to French. Figure \ref{fig:example8b} shows the signal graph for this program. However, synchronous update means that while translation is occurring, the mouse position will not be updated. In this case, it is not important to maintain the synchronization between updates of the mouse position and updates of the \texttt{wordPairs} signal: it is perfectly fine (and, in this case, preferable) to allow the mouse events to “jump ahead” of the translation computation.

The \texttt{async} primitive allows the programmer to break this synchronization in a controlled way. Informally, the annotation says “this subgraph does not need to respect the global order of events”.

The following code uses \texttt{async} to indicate that the \texttt{wordPairs} subgraph should not block mouse updates.

\[
\text{lift2} (,) (\text{async wordPairs}) \text{Mouse.position}
\]

The signal graph for this program, shown in Figure \ref{fig:example8c}, has two distinct sections: the “primary subgraph” that combines translation pairs with the mouse position and the “secondary subgraph” that produces translation pairs. Effectively, the \texttt{async} primitive breaks a fully synchronous graph into one primary subgraph and an arbitrary number of secondary subgraphs. Event order is maintained \texttt{within} each subgraph, but not between them. The resulting graph is more responsive, but it does not respect global event ordering. In the example above, this is a desirable improvement.

\textbf{Translation to Concurrent ML} We define the semantics of FE\textit{lm} by translation to Concurrent ML (CML) \cite{Dowdy:2017}. We target CML because it has simple support for concurrency, it is type safe, and its theory is well understood.

The translation to CML is a direct instantiation of the intuition given above: each node in the signal graph has a thread that performs computation for the node, there is a channel for each edge in the signal graph (representing the FIFO queue for the edge), and the global event dispatcher is explicitly represented. Figure \ref{fig:translation} defines a datatype and several CML helper functions that assist with the translation. The \texttt{event} data type wraps a value to indicate whether the value of the signal has changed. Function \texttt{change} takes an event value, and indicates whether it represents a changed value or not, and function \texttt{bodyOf} unwraps an \texttt{event} value to return the actual value. Function \texttt{guid} is used to obtain unique identifiers for source nodes.

Figure \ref{fig:translation} presents the translation of signal terms to CML. Translation of signal \(s\), denoted \([s]\), produces a pair \((v, c)\), where

\[
\begin{align*}
\text{wordPairs} & = \text{lift2} (,) \text{words} (\text{lift toFrench} \text{words}) \\
\text{lift2} (,) \text{wordPairs} \text{Mouse.position} & = \text{translationTerms} \text{toCML} (\text{wordPairs Mouse.position})
\end{align*}
\]
datatype 'a event = NoChange 'a | Change 'a

counter = ref 0

guid () = counter := !counter + 1; !counter

change e = case e of | NoChange _ => False
| Change v => True

bodyOf e = case e of | NoChange v => v
| Change v => v

foldp v s = spawn (fn _ => loop v) ; ([v] v, c out) where c out = mailbox ()

let prev = loop acc = let msg = case recv c out of
| NoChange _ => NoChange acc
| Change v => Change (foldp v, v acc)
in send c out msg ; loop (bodyOf msg)

let x = s in s out] = spawn loop :
(1 + 1) x, v, mc out in [s out]

let x = s in s out] = spawn loop :
(1 + 1) x, v, mc out in [s out]

v is the default and initial value of the signal, and c is a mailbox, a
buffered non-blocking channel to which new values of the signal are sent. Recall that we assume that every input signal has an
appropriate default value, which then induces default values for all
other signals. We assume a translation function \([\cdot] : \mathbb{V}\) that translates values (i.e., unit (), integers n, and functions \(\forall x : \tau . e\)) to CML.

The translation of each signal term creates a new thread that
performs computation to produce values for the signal. For source
nodes, this involves listening on a multicast channel for messages
sent from the global dispatcher. For other nodes, the thread waits
for input on all incoming edges, then either performs the appropri-
ate computation for the node (if at least one of the incoming values
is an event of the form Change v), or propagates an event value
NoChange v where v is the latest computed value for the signal.
We now describe the translation of each signal term in more detail.

The translation for input signal \(i\) assumes that \(s\) is a CML in-
teger value, that there is a suitable default value \(v\), for the signal,
and that there is a multicast channel \(mc\), on which new values for
the input signal will be sent. We abstract translation of this triple
\((i, mc, v)\), since it is re-used by the translation for async. Trans-
lation of the triple creates a mailbox \(c_{out}\) to publish values for
the input signal. It creates a thread that receives messages from
the global event dispatcher on the eventNotify multicast channel
("recv (port eventNotify)"). For each event, the global event
dispatcher sends a message on the eventNotify channel that con-
tains the unique integer identifier of the source node for which the
event is relevant. If this is the id of the input signal, then the thread
takes a value \(v\) from the input signal channel ("recv (port mc)"
and sends Change \(v\) on \(c_{out}\). Otherwise, the thread sends value
NoChange \(v\)', where \(v'\) is the last value of the input signal.

Translation of \(\text{lift}_i \cdot f \Rightarrow v\) is similar to a lift term, except that
the fold has a single input signal, the initial value for the signal is
the explicitly specified value \(v\), and function \(f\) is given the previous
value of the fold signal in addition to values received on signal \(s\).

Translation of let expressions and variables work together to en-
sure that FEIm's runtime system does not have duplicate nodes. A
let expression node serves as a multicast stack, forwarding mes-
sages to possibly many different nodes. Translation of \(\text{let } x = s_{in} \Rightarrow s_{out}\) translates \(s_{in}\), and binds variables \(x_v\) and \(x_{ch}\) in
the translation of \(s_{out}\) to be, respectively, the initial value of signal \(s_{in}\) and a multicast channel on which values produced by \(s_{in}\) are
multicast. Thus, variables are translated to the pair \((x_v, port x_{ch})\),
where \(port x_{ch}\) returns a port that allows receipt of messages sent
on the multicast channel.

Translation of async \(s\) returns the translation of the triple
\((id, c_{out}, v)\), where \(id\) is a newly generated unique identifier for
the source node representing the term, \(c_{out}\) is a newly created
multicast channel, and \(v\) is the default value for signal \(s\). The
translation spawns a thread that listens for Change \(v\) events to be
produced by signal \(s\), upon receiving a new value, sends that value
on multicast channel \(c_{out}\), and notifies the global event dispatcher
that source node \(id\) has a new event ("send newEvent id").

The translation of program \(s\) is executed within the context of
the FEIm CML runtime system, presented in Figure 11. There are
two important loops: the global event dispatcher eventDispatch
catches all incoming edges, then either performs the appropri-
ate computation for the node (if at least one of the incoming values
is an event of the form Change v), or propagates an event value
NoChange v where v is the latest computed value for the signal.
We now describe the translation of each signal term in more detail.

The translation for input signal \(i\) assumes that \(v\) is a CML in-
teger value, that there is a suitable default value \(v\), for the signal,
and that there is a multicast channel \(mc\), on which new values for
the input signal will be sent. We abstract translation of this triple
\((i, mc, v)\), since it is re-used by the translation for async. Trans-
lation of the triple creates a mailbox \(c_{out}\) to publish values for
the input signal. It creates a thread that receives messages from
the global event dispatcher on the eventNotify multicast channel
("recv (port eventNotify)"). For each event, the global event
dispatcher sends a message on the eventNotify channel that con-
tains the unique integer identifier of the source node for which the
event is relevant. If this is the id of the input signal, then the thread
takes a value \(v\) from the input signal channel ("recv (port mc)"
and sends Change \(v\) on \(c_{out}\). Otherwise, the thread sends value
NoChange \(v\)', where \(v'\) is the last value of the input signal.

Translation of \(\text{lift}_i \cdot f \Rightarrow v\) is similar to a lift term, except that
the fold has a single input signal, the initial value for the signal is
the explicitly specified value \(v\), and function \(f\) is given the previous
value of the fold signal in addition to values received on signal \(s\).

Translation of let expressions and variables work together to en-
sure that FEIm's runtime system does not have duplicate nodes. A
let expression node serves as a multicast stack, forwarding mes-
sages to possibly many different nodes. Translation of \(\text{let } x = s_{in} \Rightarrow s_{out}\) translates \(s_{in}\), and binds variables \(x_v\) and \(x_{ch}\) in
the translation of \(s_{out}\) to be, respectively, the initial value of signal \(s_{in}\) and a multicast channel on which values produced by \(s_{in}\) are
multicast. Thus, variables are translated to the pair \((x_v, port x_{ch})\),
where \(port x_{ch}\) returns a port that allows receipt of messages sent
on the multicast channel.

Translation of async \(s\) returns the translation of the triple
\((id, c_{out}, v)\), where \(id\) is a newly generated unique identifier for
the source node representing the term, \(c_{out}\) is a newly created
multicast channel, and \(v\) is the default value for signal \(s\). The
translation spawns a thread that listens for Change \(v\) events to be
produced by signal \(s\), upon receiving a new value, sends that value
on multicast channel \(c_{out}\), and notifies the global event dispatcher
that source node \(id\) has a new event ("send newEvent id").

The translation of program \(s\) is executed within the context of
the FEIm CML runtime system, presented in Figure 11. There are
two important loops: the global event dispatcher eventDispatch
catches all incoming edges, then either performs the appropri-
ate computation for the node (if at least one of the incoming values
is an event of the form Change v), or propagates an event value
NoChange v where v is the latest computed value for the signal.
We now describe the translation of each signal term in more detail.
4. Building GUIs with Elm

Elm extends the FElm core calculus to a full language for building rich interactive GUIs, including libraries for constructing and composing GUI components. Elm can represent quite complex interactions with a small amount of code. The signal abstraction encourages a separation between reactive code and code focused on the layout of GUI components. This separation is due in part to Elm’s purely functional and declarative approach to graphical layout, which allows a programmer to say what they want to display, without specifying how this should be done.

Elm has additional base values, including strings, floating point numbers, time, tuples, and graphical values such as Elements and Forms. Elm libraries provide data structures such as options, lists, sets, and dictionaries. Elm provides support for receiving input from mouse, keyboard, and touch devices, for accessing window attributes, and for network communication via HTTP. Elm supports JSON data structures and Markdown (making text creation easier). Elm’s type system allows let-polymorphism and recursive simple types (but not recursive signal types). Elm supports type inference, has extensible records, and has a module system.

In this section we describe some key concepts and libraries for building GUIs with Elm, focusing on features unique to Elm.

4.1 Elements and Forms

Elm has two major categories of graphical primitives: elements and forms. An element (a value of type Element) is a rectangle with a known width and height. Elements can contain text, images, or video. They can be easily created and composed, making it simple to quickly lay out a multimedia display. Elements can be resized using functions width and height, both of which have type Int -> Element -> Element. Examples in the Introduction demonstrated the composition of elements.

A form (a value of type Form) allows non-rectangular shapes and text to be defined and composed. A form is an arbitrary 2D shape (including lines, shapes, text, and images) and a form can be enhanced by specifying texture and color. Forms can be moved, rotated, and scaled.

There are several subtypes of Form, including Line and Shape. Primitive functions can be used to construct forms, including line (which takes a list of points, and returns a line that goes through each point), polygon (which takes a list of points and returns an irregular polygon that goes through each of the points), and library functions such as rect to create rectangles, oval to construct ovals, and ngn to construct regular polygons.

Several forms allow lines and shapes to be given different colors, fills, and rendering. Forms can be moved, scaled, and rotated using primitive functions move, scale, and rotate.

Function collage provides the ability to combine forms in an unstructured way to produce an element. It takes as arguments width, height, and list of forms.

\[
\text{collage : Int -> Int -> [Form] -> Element}
\]

Figure 12 shows an example of creating several forms, changing their attributes, transforming them, and using collage to combine them into an element.

4.2 Reactive GUIs

To allow programmers to write GUIs that interact with the user, and react to user input and events in the external environment, Elm provides several primitive signals that can be used in Elm programs. These are the identifiers Input used in FElm to denote input signals from the external environment. Figure 13 describes some of Elm’s signals and signal constructors. Signal constructor
To better capture the fact that our implementation is discrete (stepping forward only when provided with an input) this is called the \textit{discrete Arrowized FRP} (AFRP). Arrowized FRP allows this, making it possible to dynamically reconfigure a single node. This direct embedding of AFRP gives Elm the flexibility of signal functions without resorting to the use of signals-of-signals.

An \textit{automaton} is defined as a continuation that when given an input \textit{a}, produces the next continuation and an output \textit{b}.

\begin{verbatim}
data Automaton a b = Step (a -> (Automaton a b, b))
\end{verbatim}

This allows state to be kept around by continually placing it in the next continuation (the next step). Dynamic collections and dynamic switching are possible because an automaton is a pure data structure with no innate dependencies on inputs (whereas signals depend on an input by definition). Elm provides the following functions to make it easier to create pure and stateful automata:

\begin{verbatim}
pure : (a -> b) -> Automaton a b
init : (a -> b -> b) -> Automaton a b
\end{verbatim}

This lets a programmer create stateful components that can be switched in and out of a program. Notice the similarity between the types of \textit{init} and \textit{foldp}.

To use an \textit{Automaton}, Elm provides the \textit{step} function which steps an automaton forward once, and the \textit{run} function which feeds a signal into an automaton, stepping it forward on each change.

\begin{verbatim}
step : a -> Automaton a b -> (Automaton a b, b) 
step input (Step f) = f input
\end{verbatim}

\begin{verbatim}
run : Automaton a b -> b -> Signal a -> Signal b
run automaton base inputs = 
  let step' input (Step f, _) = f input in 
    lift snd (foldp step' (automaton,base) inputs)
\end{verbatim}

Functions \textit{run} and \textit{foldp} are equivalent in expressive power. We defined \textit{run} using \textit{foldp}, but we could have taken \textit{Automatons} as primitives and defined \textit{foldp} using \textit{run}, as follows.

\begin{verbatim}
foldp : (a -> b -> b) -> b -> Signal a -> Signal b 
foldp f base inputs = run (init f base) base inputs
\end{verbatim}

We chose \textit{foldp} as our underlying abstraction partly because it makes Elm more accessible to users unfamiliar with arrows, arrow notation, type classes, and category theory. It also allows us to improve on our naive embedding of discrete AFRP, or add alternate embeddings, without changing the core language.

\section{5. Implementation}

We have implemented an Elm-to-JavaScript compiler, and have a prototype implementation of Elm as an embedded domain-specific language (DSL) in Haskell.

The use of Asynchronous FRP has the potential to ensure that GUIs remain responsive despite long-running computation. We do not provide direct performance evaluations of Elm, although in our experience GUIs implemented with Elm have not required any performance tuning other than identifying signals to mark as asynchronous. We note that it is easy to write programs such that Elm provides arbitrarily better responsiveness over synchronous FRP. For example, in the following code, function \textit{f} can be made arbitrarily long-running. While the asynchronous code (signal \texttt{asyncEg}) will remain responsive (continuing to display \texttt{Mouse.x}), the synchronous case (signal \texttt{syncEg}) will be unresponsive until evaluation of \textit{f} completes.

\begin{verbatim}
asyncEg = lift2 (,) Mouse.x (lift f Mouse.y)
asyncEg = 
  lift2 (,) Mouse.x (async (lift f Mouse.y))
\end{verbatim}

Similarly, it is possible to write programs such that the pipelined evaluation of signals has arbitrarily better performance than non-pipelined execution by ensuring that the signal graph of the program is sufficiently deep.
**Elm-to-JavaScript compiler** The Elm-to-JavaScript compiler is implemented in about 2,700 lines of Haskell code. The output of compiling an Elm program is an HTML file. The compiler can also output a JavaScript file for embedding an Elm program into an existing project. The compiler output works on any modern browser. We provide a Foreign Function Interface to JavaScript to allow Elm to integrate with existing JavaScript libraries.

We have targeted JavaScript as it is extremely expressive for graphics and has unmatched cross-platform support. It is perhaps the most widely used and supported GUI platform, albeit notoriously difficult to reason about its graphical layout model and to create readable asynchronous code.

JavaScript has poor support for concurrency, and as such the Elm-to-JavaScript compiler supports concurrent execution only for asynchronous requests, such as HTTP requests and non-blocking file I/O. JavaScript does provide heavy-weight threads, called Web Workers, but communication and interaction between threads is limited. We investigated using Web Workers to implement async, but found their overhead to be too high compared with simpler approaches. We anticipate that if and when JavaScript supports lightweight threads, we will be able to more fully support asynchrony and concurrency in the Elm-to-JavaScript compiler.

Elm programs typically take longer to execute than an equivalent hand-written JavaScript program, in a similar way that hand-tuned assembly code is often more efficient than compiled high-level language code. Indeed, a key benefit of Elm is providing better abstractions to programmers for building GUIs. However, the current prototype Elm-to-JavaScript compiler is efficient enough to execute all existing Elm programs while providing a responsive user experience on commodity web browsers on standard laptop and desktop machines.

The Elm-to-JavaScript compiler has been used to build the Elm website [http://elm-lang.org/](http://elm-lang.org/) along with nearly 200 examples. Elm includes a nine-line analog clock, forms with client-side error checking, a graphing library that handles cartesian and radial coordinates, a slide-show API, and a Flickr image search interface. Elm has also been used to make Pong and other games, which require highly interactive GUIs.

**Elm-in-Haskell library** We also have a prototype Elm-in-Haskell library (144 lines of Haskell code) that supports the key features of Elm. While it does not have as extensive a suite of libraries and syntactic sugar, it does fully support the async construct and implement concurrent (pipelined) evaluation of signals. We use the Control.Concurrent.Chan.Strict module and Generalized Abstract Data Types to provide a type-safe implementation of Elm. By using strict channels we ensure that unevaluated expressions do not escape and cause unintended sequentialization.

6. Related Work

Elm is an asynchronous FRP programming language for creating rich and interactive GUIs. We focus on work related to the efficient FRP implementation, and to FRP frameworks for creating GUIs.

6.1 Efficient FRP Implementation

“Traditional” Functional Reactive Programming (FRP) was introduced in the Fran system [13] which suffered severe efficiency problems as a result of its expressive semantics. It permitted signals that depended on any past, present, or future value. As a result, its implementation had to save all past values of a signal just in case, growing memory linearly with time.

Real-time FRP (RT-FRP) [30] [32] ensures that signals cannot be used in ways that do not have an efficient implementation. Like Elm, RT-FRP uses a two-tiered language: an unrestricted base language (\(\lambda\)-calculus with recursion) and a limited reactive language for manipulating signals, which supports recursion but not higher-order functions (making it less expressive than traditional FRP). RT-FRP ensures that reactive updates will terminate as long as the base language terms terminate, and memory will not grow unless it grows in the base language. Although not strong guarantees, they are a significant improvement over traditional FRP, which might (needlessly) use space linear in a program’s running time.

Event-Driven FRP (E-FRP) [31] builds on the efficiency gains of RT-FRP by introducing discrete signals. E-FRP programs are event-driven in that no changes need to be propagated unless an event has occurred (i.e., the value of a signal has changed) which improves efficiency of the computation.

Xu and Khoo [32] suggest a concurrent runtime for RT-FRP as future work but to the best of our knowledge, this has not been pursued. Like RT-FRP, Elm uses a two-tiered language to restrict the inefficient use of signals. However, whereas RT-FRP uses different syntactic constructs to separate functional and signal terms, Elm has a unified syntax but a stratified type system. This allows Elm to avoid non-standard syntactic constructs and accept more expressions, while restricting inefficient use of signals. Like E-FRP, Elm uses discrete signals to ensure efficiency and does not support recursively defined signals. The original paper on E-FRP suggests that discrete signals may be compatible with RT-FRP style recursion [31], but we have found that such an extension is not particularly important for creating GUIs. Improving upon these systems, we provide a concurrent runtime, and introduce a simple mechanism—the async annotation—to allow programmers to easily take advantage of concurrency.

Arrowized FRP (AFRP) [20] [21] [24] [25] [29] aims to regain much of the expressiveness of traditional FRP and keep the efficiency gains of RT-FRP. This is achieved by requiring programmers to use signal functions instead of having direct access to signals. Signal functions are conceptually equivalent to functions that take a signal of type \(a\) as an input, and return a signal of type \(b\).

Signal functions solve the semantic problems associated with dynamically-created stateful signals by keeping state in signal functions instead of in signals. Unlike traditional FRP, a signal function cannot depend on arbitrary past values. Thus, AFRP allows programs to dynamically change how signals are processed without space leaks. Although there is no direct access to signals, AFRP achieves the expressiveness of traditional FRP via continuation-based switching and dynamic collections of signal functions.

Signal functions can be embedded in Elm (see Section [4.3]), which brings the benefits of discrete AFRP to Elm programmers, allowing programs to dynamically create graphical components, without requiring signals of signals. Elm’s discrete semantics make it clear how to allow asynchrony and concurrency, which appears to be semantically problematic in a continuous AFRP.

Parallel FRP [26], like Elm, enables concurrent signal processing. Parallel FRP relaxes the order of events within a single signal, allowing events to be processed out of order. In a server setting, this means that requests do not need to be processed in the order received, so responses can be computed in parallel and returned immediately. In a GUI setting, such intra-signal asynchrony is typically inappropriate, as it would allow, for example, key presses to be processed out of order. Instead, Elm permits inter-signal asynchrony: relaxing the order of events between multiple signals. We believe that intra- and inter-signal asynchrony are compatible, but inter-signal asynchrony is more useful in GUI programming.

Self-adjusting computation (e.g., [1] [2] [11]) is motivated by the desire to reduce redundant re-computation. As seen in FElm’s pipelined signal evaluation (Section [5.3.2]), avoiding needless recomputation helps performance and is required for correctness (due to the stateful nature of the foldp construct). Although FElm
avoids some needless recomputation, it currently propagates many noChange v messages through the signal graph. Insights from self-adjusting computation could be used to reduce these messages and improve performance. Indeed, it appears likely that modifiable references (used in self-adjusting computation to represent values that may change and thus trigger recomputation) can be used to encode signals, and can also express asynchronous signals [3].

6.2 FRP GUI Frameworks

Both traditional FRP and Arrowized FRP have been used as the basis of several Haskell-embedded GUI frameworks, including: FranTk [28] (the original GUI framework for Fran [13]); Fruit [9] and Yampa/Animas [10–16] (based on Arrowized FRP); and Reactive Banana [4] (based on traditional FRP but avoids time leaks with carefully restricted APIs, disallowing signals-of-signals). Each of these Haskell-embedded libraries suffer from two problems. First, due to its lazy semantics, embedding the language in Haskell can lead to unexpectedly slow updates and unnecessary memory usage, which can be an issue for GUIs. Second, these Haskell-embedded libraries are a wrapper around a Haskell wrapper around an imperative GUI framework. This complicates both the design and the installation of the frameworks. Indeed, FranTk [28] includes event listeners—an imperative abstraction—because they map more naturally onto the Tcl/Tk backend.

Several frameworks embed FRP in imperative languages. Flapjax [23] extends JavaScript with traditional FRP. Flapjax interoperates easily with existing JavaScript code, as it is simply a JavaScript library. Frappé [8] adds FRP functionality to Java. Ignatoff et al. [18] adapt an object-oriented GUI library for use with the FRP language FrTime [7]. By necessity, these projects—including Elm—all rely on imperative GUI frameworks. This creates incentive to “work well” with existing code using these frameworks, or to be “close to the metal” of the framework. However, these incentives can be destructive, with imperative constructs muddying the clarity and simplicity of FRP. To maintain the declarative nature of FRP, Elm introduces a purely functional graphics library on a platform that is widely supported and easy to install. Interaction between Elm and JavaScript is handled by a simple foreign function interface (FFI) that maintains a clean separation between functional and imperative code.

Elm provides a functional abstraction for graphical layout. Elm’s collage API is similar to previous work on free-form functional graphics [6,14,15,17,19]. Unlike this previous work, Elm supports user interaction and designs for animation, enabling creation of GUIs as opposed to purely static pictures and graphics.

7. Conclusion

We have developed the Elm programming language, a practical asynchronous FRP language focused on easy creation of responsive GUIs. We introduced Asynchronous FRP to allow programmers to easily specify when a potentially long-running signal computation should be computed asynchronously, and thus enable efficient concurrent execution of FRP programs. We captured the key concepts of asynchronous FRP in a core calculus.

By combining asynchronous FRP with purely functional layout of graphical elements, Elm simplifies the complicated task of creating responsive and usable graphical user interfaces.

Acknowledgments

We thank Greg Morrisett, Stuart Shieber, the anonymous reviewers, Umut Acar, Stefan Muller, David Darais, Jan Midtgaard, and the Aarhus Programming Languages group for their helpful comments and feedback. We thank the Elm user community. This research is supported in part by the National Science Foundation under Grant No. 1054172.

References


