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*.

PLDI'13, June 16–19, 2013, Seattle, WA, USA.
Copyright © 2013 ACM 978-1-4503-2014-6/13/06...$10.00

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.[1]

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

[1] Available at http://elm-lang.org/.

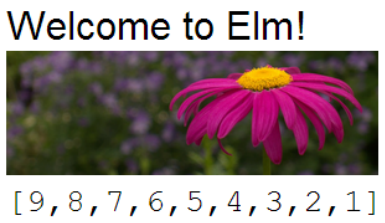


Figure 1. Purely functional graphics: basic layout

extends the core language with libraries and syntactic sugar to simplify the creation of expressive GUIs. In Section 4 we describe some of these libraries (including the embedding of discrete Arrowized FRP [25]). Elm has a fully functional compiler that compiles Elm programs to JavaScript and HTML, suitable for immediate inclusion in web applications. We describe the compiler in Section 5. We discuss related work in Section 6 and conclude in Section 7.

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 computation 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.

```
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 `Element`s, and produces an `Element`).

Values of type `Element` occupy a rectangular area of the screen when displayed, making `Element`s 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:

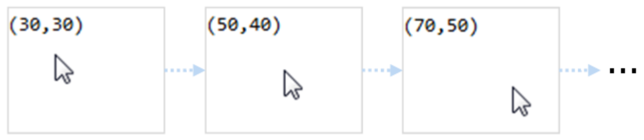


Figure 2. A basic FRP program: tracking mouse movement

```
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.[2]

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.

```
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 `Element`s. 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.

```
(inputField, tags) = Input.text "Enter a tag"

getImage tags =
  lift (fittedImage 300 200)
       (syncGet (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 `(inputField, tags)`, where `inputField` is a signal of graphical elements for the input field, and `tags` is a signal of strings. Each time the text in the input field changes (due to the user typing), both signals produces 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 result would be an unresponsive GUI that hangs (i.e., does not display the current mouse position) from when the user types in the input field until an image is fetched. By contrast, with 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.

$$
\begin{array}{rl}
\text{Numbers} & n \in \mathbb{Z} \\
\text{Variables} & x \in \mathsf{Var} \\
\text{Input signals} & i \in \mathsf{Input} \\
\\
\text{Expressions} & e ::= () \mid n \mid x \mid \lambda x\!:\!\eta.\ e \mid e_1\ e_2 \mid e_1 \oplus e_2 \\
 & \quad \mid \mathtt{if}\ e_1\ e_2\ e_3 \mid \mathtt{let}\ x = e_1\ \mathtt{in}\ e_2 \mid i \\
 & \quad \mid \mathtt{lift}_n\ e\ e_1\ ...\ e_n \mid \mathtt{foldp}\ e_1\ e_2\ e_3 \\
 & \quad \mid \mathtt{async}\ e \\
\\
\text{Simple types} & \tau ::= \mathsf{unit} \mid \mathsf{int} \mid \tau \to \tau' \\
\text{Signal types} & \sigma ::= \mathsf{signal}\ \tau \mid \tau \to \sigma \mid \sigma \to \sigma' \\
\text{Types} & \eta ::= \tau \mid \sigma
\end{array}
$$

Figure 3. Syntax of FElm

3.1 Syntax

The syntax of FElm is presented in Figure 3. Expressions include the unit value (), integers n, variables x, functions $\lambda x : \eta.\ e$, applications $e_1\ e_2$, let expressions `let` $x = e_1$ `in` e_2, and conditional expressions `if` $e_1\ e_2\ e_3$ which are all standard constructs for functional languages. (Conditional expression `if` $e_1\ e_2\ e_3$ evaluates to e_2 if e_1 evaluates to a non-zero value, e_3 otherwise.) Metavariable $\oplus$ 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 ($\mathtt{lift}_n\ e\ e_1\ ...\ e_n$, `foldp` $e_1\ e_2\ e_3$, and `async` e) are primitives for manipulating signals, and are described in more detail below.

Transforming and combining signals For each natural number $n \geq 0$, we have primitive $\mathtt{lift}_n\ e\ e_1\ ...\ e_n$ which allows a function e to transform and combine signals $e_1 \ldots e_n$.

Intuitively, if e is a function of type $\tau_1 \to \tau_2 \to \cdots \to \tau_n \to \tau$, and expressions e_i are of type $\mathsf{signal}\ \tau_i$ respectively, then $\mathtt{lift}_n\ e\ e_1\ ...\ e_n$ applies function e to the values from signals $e_1 \ldots e_n$, and has type $\mathsf{signal}\ \tau$ (i.e., it is a signal that produces values of type τ). For example,

$$\mathtt{lift}_1\ (\lambda \mathtt{x}\!:\!\mathsf{int}.\ \mathtt{x+x})\ \mathtt{Window.width}$$

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

Signals can also be combined. For example, expression

```
lift₂ (λy:int. λz:int. y ÷ z) Mouse.x Window.width
```

takes two input signals of integers (`Mouse.x` and `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 `Mouse.x` and `Window.width`.

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

$$(\tau \to \tau' \to \tau') \to \tau' \to \mathsf{signal}\ \tau \to \mathsf{signal}\ \tau'$$

for any τ and τ'. Consider $\texttt{foldp}\ e_1\ e_2\ e_3$. Here, e_1 is a function of type $\tau \to \tau' \to \tau'$, e_2 is a value of type τ' and e_3 is a signal that produces values of type τ. Type τ' 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 $\texttt{foldp}\ e_1\ e_2\ e_3$ evaluates to the signal of accumulator values.

For example, suppose that `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.)

```
foldp (λk:int. λc:int. c+1) 0 Keyboard.lastPressed
```

Asynchronous composition The `async` primitive allows a programmer 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

FElm's type system ensures that FElm programs cannot produce signals of signals. There are two kinds of types: *simple types* τ and *signal types* σ. 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 σ include type $\mathsf{signal}\ \tau$ (for signals that produce values of type τ), 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 `foldp`) depends on the history of an input signal, say `Window.width`. To compute the current value of this signal, should we use the entire history of `Window.width`? But that would require saving all history of `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 `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 4.

$$\textsc{T-Unit}\ \frac{}{\Gamma \vdash () : \mathsf{unit}} \qquad \textsc{T-Number}\ \frac{}{\Gamma \vdash n : \mathsf{int}} \qquad \textsc{T-Var}\ \frac{\Gamma(x) = \eta}{\Gamma \vdash x : \eta} \qquad \textsc{T-Input}\ \frac{\Gamma(i) = \mathsf{signal}\ \tau}{\Gamma \vdash i : \mathsf{signal}\ \tau}$$

$$\textsc{T-Lam}\ \frac{\Gamma, x{:}\eta \vdash e : \eta'}{\Gamma \vdash \lambda x{:}\eta.\ e : \eta \to \eta'} \qquad \textsc{T-Async}\ \frac{\Gamma \vdash e : \mathsf{signal}\ \tau}{\Gamma \vdash \texttt{async}\ e : \mathsf{signal}\ \tau}$$

$$\textsc{T-Op}\ \frac{\Gamma \vdash e_1 : \mathsf{int} \quad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 \oplus e_2 : \mathsf{int}} \qquad \textsc{T-App}\ \frac{\Gamma \vdash e_1 : \eta \to \eta' \quad \Gamma \vdash e_2 : \eta}{\Gamma \vdash e_1\ e_2 : \eta'}$$

$$\textsc{T-Cond}\ \frac{\Gamma \vdash e_1 : \mathsf{int} \quad \Gamma \vdash e_2 : \eta \quad \Gamma \vdash e_3 : \eta}{\Gamma \vdash \texttt{if}\ e_1\ e_2\ e_3 : \eta} \qquad \textsc{T-Let}\ \frac{\Gamma \vdash e_1 : \eta \quad \Gamma, x : \eta \vdash e_2 : \eta'}{\Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \eta'}$$

$$\textsc{T-Lift}\ \frac{\Gamma \vdash e : \tau_1 \to \cdots \to \tau_n \to \tau \quad \Gamma \vdash e_i : \mathsf{signal}\ \tau_i\ \ \forall i \in 1..n}{\Gamma \vdash \texttt{lift}_n\ e\ e_1\ ...\ e_n : \mathsf{signal}\ \tau}$$

$$\textsc{T-Fold}\ \frac{\Gamma \vdash e_f : \tau \to \tau' \to \tau' \quad \Gamma \vdash e_b : \tau' \quad \Gamma \vdash e_s : \mathsf{signal}\ \tau}{\Gamma \vdash \texttt{foldp}\ e_f\ e_b\ e_s : \mathsf{signal}\ \tau'}$$

Figure 4. Inference rules for typing judgment $\Gamma \vdash e : \eta$

The typing judgment for FElm has the form $\Gamma \vdash e : \eta$, indicating that expression e has type η under type context Γ, 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 $\texttt{if}\ e_1\ e_2\ e_3$ requires that the test expression e_1 has type int, and in particular, cannot be a signal. A program e is *well typed* if $\Gamma_{input} \vdash e : \eta$ holds for a type environment Γ_{input} that maps every input $i \in \mathsf{Input}$ 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 Application converts function applications to let expres-

$$
\begin{array}{rl}
\text{Values} & v ::= () \mid n \mid \lambda x\!:\!\tau.\ e \\
\text{Signal terms} & s ::= x \mid \texttt{let}\ x = s\ \texttt{in}\ u \mid i \mid \texttt{lift}_n\ v\ s_1\ ...\ s_n \\
& \quad \mid \texttt{foldp}\ v_1\ v_2\ s \mid \texttt{async}\ s \\
\text{Final terms} & u ::= v \mid s
\end{array}
$$

Figure 5. Syntax of intermediate language

$$
\begin{array}{l}
E ::= [\cdot] \mid E\ e \mid E \oplus e \mid v \oplus E \mid \\
\quad \texttt{if}\ E\ e_2\ e_3 \mid \texttt{let}\ x = E\ \texttt{in}\ e \mid \texttt{let}\ x = s\ \texttt{in}\ E \mid \\
\quad \texttt{lift}_n\ E\ e_1\ ...\ e_n \mid \texttt{lift}_n\ v\ s_1\ ...\ E\ ...\ e_n \mid \\
\quad \texttt{foldp}\ E\ e_2\ e_3 \mid \texttt{foldp}\ v_1\ E\ e_3 \mid \texttt{foldp}\ v_1\ v_2\ E \mid \\
\quad \texttt{async}\ E \\
F ::= [\cdot]\ e \mid [\cdot] \oplus e \mid v \oplus [\cdot] \mid \texttt{if}\ F\ e_2\ e_3 \mid \\
\quad \texttt{lift}_n\ [\cdot]\ e_1\ ...\ e_n \mid \texttt{foldp}\ [\cdot]\ e_2\ e_3 \mid \texttt{foldp}\ v_1\ [\cdot]\ e_3
\end{array}
$$

$$
\textsc{Context}\ \frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']}
\qquad
\textsc{Op}\ \frac{v = v_1 \oplus v_2}{v_1 \oplus v_2 \longrightarrow v}
\qquad
\textsc{Cond-True}\ \frac{v \neq 0}{\texttt{if}\ v\ e_2\ e_3 \longrightarrow e_2}
$$

$$
\textsc{Cond-False}\ \frac{}{\texttt{if}\ 0\ e_2\ e_3 \longrightarrow e_3}
\qquad
\textsc{Application}\ \frac{}{(\lambda x\!:\!\eta.\ e_1)\ e_2 \longrightarrow \texttt{let}\ x = e_2\ \texttt{in}\ e_1}
$$

$$
\textsc{Reduce}\ \frac{}{\texttt{let}\ x = v\ \texttt{in}\ e \longrightarrow e[v/x]}
$$

$$
\textsc{Expand}\ \frac{x \notin fv(F[\cdot])}{F[\texttt{let}\ x = s\ \texttt{in}\ u] \longrightarrow \texttt{let}\ x = s\ \texttt{in}\ F[u]}
$$

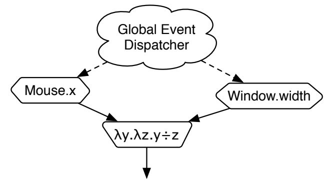
Figure 6. Semantics for functional evaluation

sions. Rule Reduce performs beta-reduction for let expressions `let` $x = v$ `in` e, but only when x is bound to a simple value. If x is bound to a signal term, then (as shown by context `let` $x = s$ `in` E) subexpression e is evaluated without substitution of x. This ensures that signal expressions are not duplicated, and reduces unnecessary computation in the second stage of evaluation, which is described in Section 3.3.2. (This is similar to a call-by-need calculus [5, 22], which avoids duplication of unevaluated expressions.)

Rule Expand expands the scope of a let expression to allow evaluation to continue. Contexts F describe where Rule Expand can be applied, and includes all contexts where a simple value v is required. For a context F, we write $fv(F[\cdot])$ for the set of free variables in F. We assume that expressions are equivalent up to renaming of bound variables, and an appropriate variable x can always be chosen such that $x \notin fv(F[\cdot])$.

If a FElm program is well typed, then the first stage of evaluation does not get stuck, and always evaluates to a final term. (We write $\longrightarrow^*$ for the reflexive transitive closure of the relation $\longrightarrow$.)

Theorem 1 (Type Soundness and Normalization)**.** Let type environment Γ_{input} map every input $i \in \mathsf{Input}$ to a signal type. If $\Gamma_{input} \vdash e : \eta$ then $e \longrightarrow^* u$ and $\Gamma_{input} \vdash u : \eta$ for some final term u.



`lift`$_2$ (λ`y`:int. λ`z`:int. `y` $\div$ `z`) `Mouse.x Window.width`

Figure 7. Signal graph for relative x-position of mouse

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 ($i \in \mathsf{Input}$), `lift`$_n$ terms, and `foldp` terms. The definition and use of variables define edges between the nodes. (Asynchronous terms `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 `lift`$_n$ $v\ s_1\ ...\ s_n$ has n incoming edges, and a node representing term `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, `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 `lift`$_n$ and `foldp` terms perform computation on values produced by signals. Lift terms perform (pure) computation on the current values of signals: term `lift`$_n$ $v\ s_1\ ...\ s_n$ applies function v to the current values of the n signals to produce a new value. A node for term `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 accumulator, to produce the new value of the accumulator, which is also the new value for the signal `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 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 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 noChange v value), then the node performs computation (which, in the case of a `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 noChange v' values are a form of memoization—allowing nodes to avoid needless recomputation—but in the case of `foldp` nodes, are also critical to ensure correct execution. For example, a `foldp` term that counts the number of key presses (as in Section 3.1) 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 `async` construct. The node for a term `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 noChange v value is propagated from the node. When signal s produces a new value, then a new event for term `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 `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 trigged by user actions.

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:

```
wordPairs = lift2 (,) words (lift toFrench words)
```

For simplicity, we assume that `words` is an input signal (i.e., words are given by the external environment), and function `toFrench` takes an English word and produces a French word, and may take significant time to perform the translation. Function `(,)` takes two arguments and constructs a pair from them. The signal graph for `wordPairs` is shown in Figure 8(a). Although function `toFrench` may take significant time, it is important that the propagation of values from signal `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 `wordPairs` signal with the mouse position.

```
lift2 (,) wordPairs 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 8(b) 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 `wordPairs` signal: it is perfectly fine

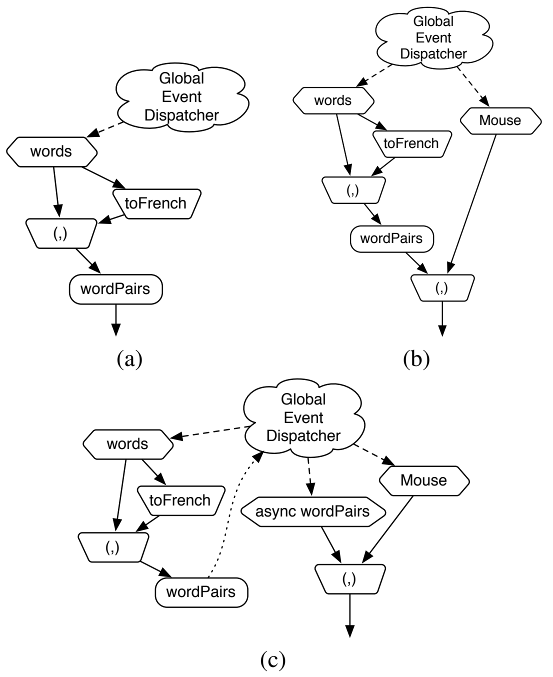


Figure 8. Example signal graphs

(and, in this case, preferable) to allow the mouse events to "jump ahead" of the translation computation.

The `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 `async` to indicate that the `wordPairs` subgraph should not block mouse updates.

```
lift2 (,) (async wordPairs) Mouse.position
```

The signal graph for this program, shown in Figure 8(c), 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 `async` primitive breaks a fully synchronous graph into one primary subgraph and an arbitrary number of secondary subgraphs. Event order is maintained *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.

Translation to Concurrent ML We define the semantics of FElm by translation to Concurrent ML (CML) [27]. 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 9 defines a datatype and several CML helper functions that assist with the translation. The `event` data type wraps a value to indicate whether the value of the signal has changed. Function change takes an event value, and indicates whether it represents a changed value or not, and function bodyOf unwraps an `event` value to return the actual value. Function guid is used to obtain unique identifiers for source nodes.

Figure 10 presents the translation of signal terms to CML. Translation of signal s, denoted $[\![s]\!]$, produces a pair `(v,c)`, where

```
datatype 'a event = NoChange 'a | Change 'a
change e = case e of | NoChange _ => False
                     | Change _   => True
bodyOf e = case e of | NoChange v => v
                     | Change v   => v

counter = ref 0
guid () = counter := !counter + 1; !counter
```

Figure 9. Translation helper functions

```
$[\![i]\!]$ = $[\![\ \langle i,$ mc$_i$, v$_i \rangle\ ]\!]$

$[\![\ \langle$id, mc, v$\rangle\ ]\!]$ = spawn (fn _ => loop v) ; (v, c$_{out}$)
  where c$_{out}$ = mailbox ()
        loop prev =
          let msg = if (recv (port eventNotify)) == id
                       then Change (recv (port mc))
                       else NoChange prev
          in  send c$_{out}$ msg ; loop (bodyOf msg)

$[\![$lift$_n$ $f$ $s_1$ ... $s_n]\!]$ = spawn (fn _ => loop v) ; (v, c$_{out}$)
  where (v$_1$, c$_1$), ... , (v$_n$, c$_n$) = $[\![s_1]\!]$, ... , $[\![s_n]\!]$
     v, c$_{out}$ = $[\![f]\!]_{\mathcal{V}}$ v$_1$ ... v$_n$, mailbox ()
     loop prev =
       let (m$_1$, ... , m$_n$) = (recv c$_1$, ... , recv c$_n$)
         msg = if exists change [m$_1$, ... , m$_n$] then
                Change ($[\![f]\!]_{\mathcal{V}}$ (bodyOf x$_1$) ... (bodyOf x$_n$))
               else NoChange prev
       in  send c$_{out}$ msg ; loop (bodyOf msg)

$[\![$foldp $f$ $v$ $s]\!]$ = spawn (fn _ => loop $[\![v]\!]_{\mathcal{V}}$) ; ($[\![v]\!]_{\mathcal{V}}$, c$_{out}$)
  where (_, c$_{in}$), c$_{out}$ = $[\![s]\!]$, mailbox ()
        loop acc =
         let msg = case recv c$_{in}$ of
                    | NoChange _ -> NoChange acc
                    |   Change v -> Change ($[\![f]\!]_{\mathcal{V}}$ v acc)
             in  send c$_{out}$ msg ; loop (bodyOf msg)

$[\![$let $x$ $=$ $s_{in}$ in $s_{out}]\!]$ = spawn loop ;
                         (let $x_v$,$x_{ch}$ = v,mc$_{out}$ in $[\![s_{out}]\!]$)
  where (v, c$_{in}$), mc$_{out}$ = $[\![s_{in}]\!]$, mChannel ()
        loop () = send mc$_{out}$ (recv c$_{in}$) ; loop ()

$[\![x]\!]$ = ($x_v$, port $x_{ch}$)

$[\![v]\!]$ = $[\![v]\!]_{\mathcal{V}}$

$[\![$async $s]\!]$ = spawn loop ; $[\![\ \langle$id, c$_{out}$, d$\rangle\ ]\!]$
  where (d, c$_{in}$), c$_{out}$, id = $[\![s]\!]$, mChannel (), guid ()
    loop () = case recv c$_{in}$ of
              | NoChange _ -> loop ()
              |   Change v -> send c$_{out}$ v ;
                              send newEvent id ; loop ()
```

Figure 10. Translation from signal terms to CML

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 $[\![\cdot]\!]_{\mathcal{V}}$ that translates values (i.e., unit (), integers n, and functions $\lambda x\!:\!\tau.\ e$) 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 appropriate 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 i is a CML integer value, that there is a suitable default value v$_i$ for the signal, and that there is a multicast channel mc$_i$ on which new values for the input signal will be sent. We abstract translation of this triple $\langle i, \texttt{mc}_i, \texttt{v}_i\rangle$, since it is re-used by the translation for `async`. Translation 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 contains 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 lift$_n$ f s_1 ... s_n creates a mailbox c$_{out}$ on which to send new values for the signal. It spawns a thread that waits to receive an `event` value from each of n signals. If any of the values indicate a change has occurred, then function f is applied to the current value of the signals to produce a value v and `Change` v is sent on the signal's channel. Otherwise, `NoChange` v' is sent on the channel, where v' is the last value of the input signal.

Translation of `foldp` f v s 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 ensure that FElm's runtime system does not have duplicate nodes. A let expression node serves as a multicast station, forwarding messages to possibly many different nodes. Translation of `let` x $=$ s_{in} `in` 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 $\langle \texttt{id}, \texttt{c}_{out}, \texttt{v}\rangle$, where `id` is a newly generated unique identifier for the source node representing the term, c$_{out}$ is a newly created multicast channel, and `d` 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 FElm CML runtime system, presented in Figure 11. There are two important loops: the global event dispatcher `eventDispatch` receives events from the `newEvent` mailbox, and notifies all source nodes of the new event, using the `eventNotify` multicast channel; the display loop `displayLoop` updates the user's view as new display values are produced by the translation of the program s. Together, these loops are the input and output for a FElm program, with `eventDispatch` feeding values in and `displayLoop` outputting values to the screen. Note that the `newEvent` mailbox is a FIFO queue, preserving the order of events sent to the mailbox.

The `displayLoop` depends on the translation of a signal term s. The translation produces both the `initialDisplay` which is the first screen to be displayed and the `nextDisplay` channel upon which display updates are sent. The display loop just funnels values from the `nextDisplay` channel to the screen.

```
newEvent = mailbox ()
eventNotify = mChannel ()
eventDispatch () =
    let id = recv newEvent in
           send eventNotify id ; eventDispatch ()

(initialDisplay, nextDisplay) = ⟦s⟧
send display initialDisplay
displayLoop () = let v = recv nextDisplay in
                 send display v ; displayLoop ()

spawn eventDispatch ; spawn displayLoop
```

Figure 11. FElm CML runtime system

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 `Element`s and `Form`s. 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

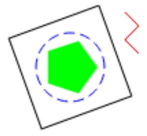
Elm has two major categories of graphical primitives: elements and forms. An *element* (a value of type `Element`) is a rectangle with a known width and height. Elements can contain text, images, or video. They can be easily created and composed, making it simple to quickly lay out a multimedia display. 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 `ngon` to construct regular polygons.

Several functions 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



```
square   = rect 70 70
pentagon = ngon  5 20
circle   = oval 50 50
zigzag   = path [ (0,0), (10,10), (0,30), (10,40) ]

main = collage 140 140
      [ filled green pentagon,
        outlined (dashed blue) circle,
        rotate (degrees 70)
               (outlined (solid black) square),
        move 40 40 (trace (solid red) zigzag) ]
```

Figure 12. Creating and combining shapes

Signal	Type and Description
`Mouse.position`	`Signal (Int, Int)` Current coordinates of the mouse.
`Mouse.clicks`	`Signal ()` Triggers on mouse clicks.
`Window.dimensions`	`Signal (Int, Int)` Current dimensions of window.
`Time.every`	`Time -> Signal Time` Update every t milliseconds.
`Time.fps`	`Float -> Signal Time` Time deltas, updating at the given FPS.
`Touch.touches`	`Signal [Touch]` List of ongoing touches. Useful for defining gestures.
`Touch.taps`	`Signal { x:Int, y:Int }` Position of the latest tap.
`Keyboard.keysDown`	`Signal [KeyCode]` List of keys that are currently pressed.
`Keyboard.arrows`	`Signal { x:Int, y:Int }` Arrow keys pressed. e.g., up+right is `{x=1,y=1}`
`Keyboard.shift`	`Signal Bool` Is the shift key down?
`Input.text`	`String -> (Signal Element, Signal String)` Create a text input field.

Figure 13. Some Elm input signals and signal constructors

width, height, and list of forms.

```
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

```
pics = [ "shells.jpg", "car.jpg", "book.jpg" ]
display i =
    image 475 315 (ith (i ‘mod‘ length pics) pics)

count s = foldp (\_ c -> c + 1) 0 s
index1 = count Mouse.clicks
index2 = count (Time.every (3 * second))
index3 = count Keyboard.lastPressed

main = lift display index1
```

Figure 14. A simple slide-show: reacting to user input

`Time.every` takes a floating point number t. The resulting signal is the current time, updated every t milliseconds. This signal enables time-indexed animations and allows update rates to be tuned by the programmer. Signal constructor `Time.fps` allows a programmer to specify a desired frame rate. The resulting signal is a sequence of time deltas (i.e., the time elapsed since the last event), making it easy to do time-stepped animations. The frame rate is managed by the Elm runtime system, accommodating performance constraints.

Input components such as text boxes, buttons, and sliders are represented as a pair of signals: an element (for the graphical component) and a value (for the value input). For example, the library function

```
  Input.text :
         String -> (Signal Element, Signal String)
```

allows a programmer to construct a text input component. The code

```
(Input.text "Name")
```

returns a pair of a signal of `Elements` (for the graphical input field) and a signal of `Strings` representing the current user input. The graphical element uses the string `"Name"` as the greyed-out default text, indicating the desired user input. The element can be displayed on the screen, and the signals will be updated every time the text in the field changes.

Figure 14 shows how different user inputs can be used to implement a slide show. The code displays a single image from a list of images, `pics`. The user can cycle through the images by clicking the mouse. (In the accompanying diagram, the arrows indicate possible transitions between pictures, not actual graphical elements.) The program could easily be modified to change images using a timer as seen in `index2` which increments every three seconds. It could also change images when a key is pressed as in `index3`. The `Keyboard.lastPressed` signal updates to the latest character pressed.

4.3 Signal Functions

Elm embeds an implementation of discrete Arrowized FRP (AFRP), based on the naive continuation-based implementation described in the first AFRP paper [25]. To better capture the fact that our implementation is discrete (stepping forward only when provided with an input) this is called the `Automaton` library in Elm.

Arrowized FRP is a purely functional way to structure stateful programs. It introduced *signal functions* [25] which can encapsulate state and safely be switched in and out of a program. A signal function is conceptually equivalent to Elm's signal nodes: a set of external inputs, internal state, and an output. Although signal functions and signal nodes are equivalent, Elm's core language does not allow one to work with signal nodes directly. 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 `Automaton` is defined as a continuation that when given an input `a`, produces the next continuation and an output `b`.

```
data Automaton a b = Step (a -> (Automaton a b, b))
```

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:

```
pure : (a -> b) -> Automaton a b
init : (a -> b -> b) -> b -> Automaton a b
```

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

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

```
step : a -> Automaton a b -> (Automaton a b, b)
step input (Step f) = f input

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)
```

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

```
foldp : (a -> b -> b) -> b -> Signal a -> Signal b
foldp f base inputs = run (init f base) base inputs
```

We chose `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.

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 `f` can be made arbitrarily long-running. While the asynchronous code (signal `asyncEg`) will remain responsive (continuing to display `Mouse.x`), the synchronous case (signal `syncEg`) will be unresponsive until evaluation of `f` completes.

```
syncEg  = lift2 (,) Mouse.x (lift f Mouse.y)
asyncEg =
       lift2 (,) Mouse.x (async (lift f Mouse.y))
```

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 light-weight 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/) along with nearly 200 examples, including 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 (λ-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 3.3.2), 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 [1] (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

[1] U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive functional programming. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 247–259, New York, NY, USA, 2002. ACM.

[2] U. A. Acar, A. Ahmed, and M. Blume. Imperative self-adjusting computation. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 309–322, New York, NY, USA, 2008. ACM.

[3] U. A. Acar, G. E. Blelloch, K. Tangwongsan, and D. Türkoğlu. Robust kinetic convex hulls in 3D. In *Proceedings of the 16th Annual European Symposium on Algorithms*, pages 29–40, Berlin, Heidelberg, 2008. Springer-Verlag.

[4] H. Apfelmus. Reactive-banana. Haskell library available at http://www.haskell.org/haskellwiki/Reactive-banana, 2012.

[5] Z. M. Ariola and M. Felleisen. The call-by-need lambda calculus. *Journal of Functional Programming*, 7(3):265–301, May 1997.

[6] E. Chailloux, G. Cousineau, and A. Suàrez. MLgraph. http://www.pps.univ-paris-diderot.fr/~cousinea/MLgraph/mlgraph.html, 1992.

[7] G. H. Cooper. *Integrating Dataflow Evaluation into a Practical Higher-Order Call-by-Value Language*. PhD thesis, Brown University, May 2008.

[8] A. Courtney. Frappé: Functional reactive programming in Java. In *Proceedings of the Third International Symposium on Practical Aspects of Declarative Languages*, pages 29–44, London, UK, 2001. Springer-Verlag.

[9] A. Courtney and C. Elliott. Genuinely functional user interfaces. In *Proceedings of the 2001 ACM SIGPLAN Workshop on Haskell*, pages 41–69, 2001.

[10] A. Courtney, H. Nilsson, and J. Peterson. The Yampa arcade. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell*, pages 7–18, New York, NY, 2003. ACM.

[11] A. Demers, T. Reps, and T. Teitelbaum. Incremental evaluation for attribute grammars with application to syntax-directed editors. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 105–116, New York, NY, USA, 1981. ACM.

[12] C. Elliott. Push-pull functional reactive programming. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell*, pages 25–36, New York, NY, 2009. ACM.

[13] C. Elliott and P. Hudak. Functional reactive animation. In *Proceedings of the 2nd ACM SIGPLAN International Conference on Functional Programming*, pages 263–273, New York, NY, 1997. ACM.

[14] R. B. Findler and M. Flatt. Slideshow: functional presentations. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming*, pages 224–235, New York, NY, USA, 2004. ACM.

[15] S. Finne and S. Peyton Jones. Pictures: A simple structured graphics model. In *Proceedings of the Glasgow Functional Programming Workshop*, 1995.

[16] G. Giorgidze and H. Nilsson. Switched-on Yampa: declarative programming of modular synthesizers. In *Proceedings of the 10th International Conference on Practical Aspects of Declarative Languages*, pages 282–298, Berlin, Heidelberg, 2008. Springer-Verlag.

[17] P. Henderson. Functional geometry. In *Proceedings of the 1982 ACM Symposium on LISP and Functional Programming*, pages 179–187, New York, NY, USA, 1982. ACM.

[18] D. Ignatoff, G. H. Cooper, and S. Krishnamurthi. Crossing state lines: Adapting object-oriented frameworks to functional reactive languages. In *Proceedings of the International Symposium on Functional and Logic Programming*, 2006.

[19] S. N. Kamin and D. Hyatt. A special-purpose language for picture-drawing. In *Proceedings of the Conference on Domain-Specific Languages on Conference on Domain-Specific Languages (DSL), 1997*, pages 23–23, Berkeley, CA, USA, 1997. USENIX Association.

[20] H. Liu and P. Hudak. Plugging a space leak with an arrow. *Electron. Notes Theor. Comput. Sci.*, 193:29–45, Nov. 2007.

[21] H. Liu, E. Cheng, and P. Hudak. Causal commutative arrows and their optimization. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, pages 35–46, New York, NY, 2009. ACM.

[22] J. Maraist, M. Odersky, and P. Wadler. The call-by-need lambda calculus. *Journal of Functional Programming*, 8(3):275–317, May 1998.

[23] L. A. Meyerovich, A. Guha, J. Baskin, G. H. Cooper, M. Greenberg, A. Bromfield, and S. Krishnamurthi. Flapjax: a programming language for Ajax applications. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, pages 1–20, New York, NY, 2009. ACM.

[24] H. Nilsson. Dynamic optimization for functional reactive programming using generalized algebraic data types. In *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming*, pages 54–65, New York, NY, 2005. ACM.

[25] H. Nilsson, A. Courtney, and J. Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell*, pages 51–64, New York, NY, 2002. ACM.

[26] J. Peterson, V. Trifonov, and A. Serjantov. Parallel functional reactive programming. In *Proceedings of the Second International Workshop on Practical Aspects of Declarative Languages*, pages 16–31, London, UK, 2000. Springer-Verlag.

[27] J. H. Reppy. *Concurrent programming in ML*. Cambridge University Press, 1999. ISBN 9780521480895.

[28] M. Sage. FranTk - a declarative GUI language for Haskell. In *Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming*, pages 106–117, New York, NY, 2000.

[29] N. Sculthorpe and H. Nilsson. Safe functional reactive programming through dependent types. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, pages 23–34, New York, NY, 2009. ACM.

[30] Z. Wan, W. Taha, and P. Hudak. Real-time FRP. In *Proceedings of the 6th ACM SIGPLAN International Conference on Functional Programming*, pages 146–156, New York, NY, 2001. ACM.

[31] Z. Wan, W. Taha, and P. Hudak. Event-driven FRP. In *Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages*, pages 155–172, London, UK, UK, 2002. Springer-Verlag.

[32] D. N. Xu and S.-C. Khoo. Compiling real time functional reactive programming. In *Proceedings of the ASIAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 83–93, New York, NY, 2002. ACM.