Elm: Concurrent FRP for Functional GUIs

Evan Czaplicki

30 March 2012

Abstract

Graphical user interfaces (GUIs) mediate almost all of our interactions with computers, whether it is through web pages, phone apps, or desktop applications. Functional Reactive Programming (FRP) is a promising approach to GUI design. This thesis presents Elm, a concurrent FRP language focused on easily creating responsive GUIs. Elm has two major features: (1) purely functional graphical layout and (2) support for Concurrent FRP. Purely functional graphical layout is a high level framework for working with complex visual components. It makes it quick and easy to create and combine text, images, and video into rich multimedia displays. Concurrent FRP solves some of FRP's long-standing efficiency problems: global delays and needless recomputation. Together, Elm's two major features simplify the complicated task of creating responsive and usable graphical user interfaces. This thesis also includes a fully functional compiler for Elm, available at elm-lang.org. This site includes an interactive code editor that allows you to write and compile Elm programs online with no download or install.

Acknowledgments

Thank you Stephen Chong, my thesis advisor. This thesis would not have been possible without your tireless guidance and encouragement. Thank you Greg Morrisett and Stuart Shieber, my two thesis readers. I am looking forward to your comments and impressions. I am grateful to all three of you, not only for reading and critiquing this thesis, but also for teaching wonderfully thought provoking and fun classes that have greatly influenced my study of Computer Science. Finally, thank you Natalie and Randi Czaplicki for your love and support.

Contents

1 Introduction

Functional Reactive Programming (FRP) is a declarative way to create reactive systems. FRP has already shown its potential in a diversity of domains: robotics [20, 35], music synthesis [16], animation [10], video games [7], and graphical user interfaces [6]. These domains are *reactive* because they require interaction with a wide range of outside inputs, from keyboards to accelerometers.

This work focuses on graphical user interfaces, the most important and widely-used of these domains. Graphical user interfaces (GUIs) mediate almost all of our interactions with computers, whether it is through web pages, phone apps, or desktop applications. Graphical user interfaces are also a particularly difficult domain. In addition to the normal challenges of programming – transforming and analyzing data – GUIs face the problems of data presentation and usability. A GUI also reacts to many outside input from users and sensors. A useful GUI must also be *responsive*, providing the user quick feedback in spite of the complex computations that occur behind the scenes.

Functional reactive programming is a *declarative* approach to GUI design. The term *declarative* makes a distinction between the "what" and the "how" of programming. A declarative language allows you to say *what* is displayed, without having to specify exactly *how* the computer should do it. With functional reactive programming, many of the irrelevant details are left to the compiler, freeing the programmer to think on a much higher level than with most existing GUI frameworks.

The term declarative is important because most current frameworks for graphical user interfaces are *not* declarative. They mire programmers in the many small, nonessential details of handling user input and modifying the display. The declarative approach of functional reactive programming makes it quick and easy to specify complex user interactions. FRP also encourages a greatly simplified approach to graphical layout. As a result, FRP makes GUI programming much more manageable than with traditional approaches.

As an example of declarative programming, we will consider a basic FRP program, shown in Figure 1. This example displays the current mouse position as text. Think of `Mouse.position` as a time-varying value that changes whenever the mouse moves. It is always equal to the current position of the mouse relative to the top left corner of the screen. The `asText` function converts a value – such as a pair of numbers – into text that can be displayed on screen. The `lift` function combines a function and a time-varying value. In this case, `lift` combines `asText` and `Mouse.position`, converting the

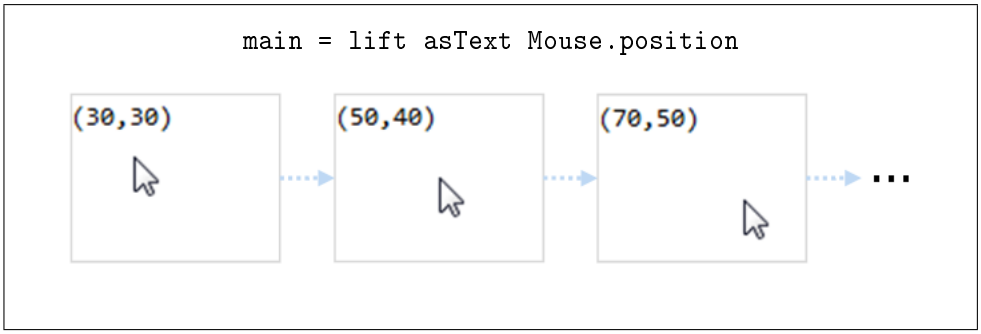


Figure 1: A Basic FRP Program: Tracking Mouse Movement

mouse position (which updates automatically) into displayable text (which also updates automatically). The value of `main` is displayed to the user, so the user sees the current mouse position. The corresponding diagram shows our example program as the mouse moves around the screen. Notice that the x and y coordinates increase as the mouse moves farther from the top left corner.

With one line of code, we have created a simple GUI that responds to user input, displays values on screen, and updates automatically. To perform these tasks in a traditional GUI framework – such as JavaScript – would require significantly more code: manually extracting the mouse position from an event and describing exactly how to update the displayed value. As a declarative framework, FRP makes this task considerably easier by taking care of the "how" of events, display, and updates.

This thesis presents Elm, a concurrent FRP language focused on easily creating responsive GUIs. We have already seen Elm in action with the example above, a working Elm program. Elm has two major features: (1) purely functional graphical layout and (2) support for *Concurrent* FRP. Purely functional graphical layout is a high level framework for working with complex visual components. It makes it quick and easy to create and combine text, images, and video into rich multimedia displays. Concurrent FRP solves some of FRP's long-standing efficiency problems: global delays and needless recomputation.

A *global delay* occurs when a GUI cannot immediately process incoming events, making the GUI less responsive. Past FRP formulations maintain a strict ordering of incoming events. They are processed one at a time, so if one event takes a long time, all of the subsequent events must wait. This strict ordering of events is not always necessary, especially in the case of events that are not directly related. Concurrent FRP allows the programmer to

specify when a strict ordering of events is unnecessary, resulting in more responsive GUIs. Concurrent FRP can safely include long computations without degrading user experience or requiring unnatural coding practices.

Needless recomputation occurs when a function is recomputed even though its input has not changed. In a purely functional language like Elm, every function – when given the same inputs – always produces the same output. Therefore, there is no need to recompute a function unless its inputs have changed. Many previous formulations of FRP do not take advantage of this fact. A program consists of many interdependent functions, but changing one input to the program is unlikely to affect *all* of them. Previously, when one of the many inputs to an FRP program changes, the *whole program* is recomputed even though most of the inputs have not changed. With this strategy, every incoming event can cause a significant number of needless recomputations. Elm's concurrent runtime system avoids this with memoization, an optimization technique that saves previously computed values to avoid future computations. Specifically, Elm saves the most recently computed values, so when one of the many inputs to a program changes, only functions that depend on that particular input must be recomputed.

These two efficiency improvements add to FRP's strengths as a GUI framework. In particular, Elm's mechanism for avoiding global delays is extremely useful, yet distressingly uncommon. Not only is it new for FRP, but it is also an improvement over a majority of popular modern GUI frameworks such as JavaScript or Java, which both enforce a strict order of events at the expense of either responsiveness or code clarity.

This thesis also presents a prototype implementation of Elm that compiles for the web. The Elm compiler produces HTML, CSS, and JavaScript. This has two major benefits: (1) Elm programs already run on any device that supports modern web standards and (2) Elm is much more accessible than many previous FRP frameworks. An overview of Elm, an interactive code editor, numerous examples, and documentation have all been posted online at elm-lang.org. The interactive editor allows you to write and compile Elm programs with no download or install. This ease of access permits developers to quickly dive into FRP and GUI programming even if they have little previous experience. The online implementation and learning resources are a significant portion of this thesis, making FRP research more accessible and raising awareness of Elm and FRP in general.

The structure of this thesis is as follows. Chapter 2 presents detailed background information on functional reactive programming and GUIs. It discusses the strengths and weaknesses of the related work on this topic, providing context and motivation for Elm's design choices. Chapters 3 and

4 contain a core language that captures the theoretical contributions of this thesis. Elm is compiled in two phases. The first phase – source language to well-behaved intermediate representation – is the topic of Chapter 3. This chapter also discusses Elm's relation to previous FRP frameworks. The second phase – intermediate representation to concurrent runtime system – is the topic of Chapter 4. Here we will discuss the details of the concurrent system and the resulting efficiency benefits. Chapter 5 is an introduction to the practical use of Elm. It describes how to create graphical user interfaces, using real examples to illustrate Elm's most important graphics libraries. Chapter 6 discusses Elm's prototype implementation. The Elm compiler produces HTML, CSS, and JavaScript. This means Elm programs can already run on any modern web browser. It also means that the Elm compiler must still overcome some of JavaScript's weaknesses, such as limited support for concurrency. Finally, Chapter 7 discusses how this work fits into past research and directions for future work.

2 Background and Related Works

A graphical user interface is an interactive multimedia display, but what is a *functional* graphical user interface? The term functional refers to a style of programming in which functions can be passed around like any other value, promoting code reuse. A functional GUI would be designed in the functional style.

Languages that are *purely* functional are very strict about how a program can interact with the rest of the computer. Such languages have two important properties: (1) all values are immutable, never changing after instantiation, and (2) functions are *pure*, always producing the same output when given the same input. These mutually dependent properties are great for program reliability and maintainability, yet they are quite rare in modern languages. This is because many important values are *mutable*, able to change. This includes the position of the mouse, whether a key is pressed, and any other input that changes over time. How can we reconcile purely functional programming and mutable values?

The restrictions of purely functional programming languages give us the opportunity to model mutable values on our own terms. If mutable values are introduced through "safe" abstractions, they need not undermine the pleasant properties of purely functional programs. This chapter addresses two important approaches to introducing mutablity: Functional Reactive Programming and Message-Passing Concurrency. Both approaches elegantly bridge the gap between purely functional languages and mutable values such as user input. By starting from scratch, these approaches reexamine the essense of mutable values and user input, resulting in new abstractions that simplify GUI programming.

This chapter first examines functional reactive programming (FRP). Elm uses this general approach to GUI programming. Next, we turn to message-passing concurrency for another perspective. Message-passing concurrency presents some important insights into GUI programming that can help make FRP more robust. In short, concurrency matters! Finally, we will examine the existing FRP GUI frameworks, trying to identify common pitfalls. This background information lays the groundwork for an efficient and useable implementation of FRP for GUIs.

2.1 Functional Reactive Programming

Functional Reactive Programming (FRP) is a declarative programming paradigm for working with mutable values. FRP recasts mutable values as *time-varying*

values, better capturing the temporal aspect of mutability. In FRP, time-varying values are called *signals*. Signals can represent any mutable value. For example, consider the position of the mouse. The mouse position signal – called `Mouse.position` in Elm – holds the current mouse position. When the mouse moves, the value of `Mouse.position` changes automatically.

Signals can also be transformed and combined. Imagine we have a function that takes a mouse position as input and determines whether the mouse is hovering over a certain button. We can apply this function to `Mouse.position`. This results in a signal of boolean values that updates automatically: true when the mouse is hovering over the button and false otherwise. In contrast with traditional approaches, this declarative style of FRP abstracts away many inconsequential details, allowing programmers to do more with less code.

The original formulation of FRP was extremely expressive, giving programmers many high-level abstractions. This expressiveness came at the cost of efficiency because there was not always a clear way to implement such high-level abstractions. Subsequent research aimed to resolve the tension between expressiveness and efficiency. We will focus on three major semantic families of Functional Reactive Programming: Classical FRP; Real-time FRP and Event-Driven FRP; and Arrowized FRP. We examine them chronologically to see how the semantics of FRP have evolved. As we move through FRP's three semantic families, we will better understand the remaining efficiency problems and how to resolve them.

2.1.1 Classical FRP

Functional Reactive Programming was originally formulated by Paul Hudak and Conal Elliott in their 1997 paper *Functional Reactive Animation* [10]. Their implementation is embedded in Haskell, a functional language that can easily be extended. Their initial formulation – which we will call Classical FRP – introduced two new types of values: Behaviors and Events.

Behaviors are continuous, time-varying values. This is represented as a function from a time to a value.

$$\texttt{Behavior}\ \alpha = \texttt{Time} \to \alpha$$

These time indexed functions are just like the equations of motion in Newtonian physics. At time t, the behavior has value v. Behaviors help with a very common task in animation: modeling physical phenomena. Position, velocity, acceleration, and analog signals can all

be represented quite naturally with behaviors. This abstraction helps produce concise and declarative code by leaving the details of updates to the compiler.

Events represent a sequence of discrete events as a time-stamped list of values.

$$\texttt{Event}\ \alpha = [(\texttt{Time}, \alpha)]$$

The time values must increase monotonically. Events can model any sort of discrete events, from user input to HTTP communications. Originally intended for animations, events would commonly be used to model inputs such as mouse clicks and key presses. Events would be used the same way in GUIs.

This initial focus on animation strongly influenced the semantics and goals of future FRP work. In particular, *continuous* time-varying values have appeared in almost all subsequent work.

Although elegant, the original implementation of classical FRP – named Fran for "functional reactive animation" – is prone to difficult-to-spot space and time leaks. In other words, memory usage may grow unexpectedly (space leaks), resulting in unexpectedly long computations (time leaks).

Fran inherits a variety of space and time leaks from its host language, Haskell. Haskell delays computation until it is absolutely necessary, evaluating programs "lazily". This makes it easy to create infinite data structures, but it can also cause memory problems. In FRP, the value of a behavior may be inspected infrequently, and thus, an accumulated computation can become quite large over time, taking up more and more memory (space leak). When inspected, the entire accumulated computation must be evaluated all at once, potentially causing a significant delay or even a stack overflow (time leak). These leaks appear in the *implementation* of Fran. Subsequent Haskell FRP libraries were able to create a leak free implementation, but that does not fully address the problem. Even with a leak free implementation, space and time leaks can still appear in FRP programs. This problem is common to all Haskell FRP libraries.

Why embed in Haskell if it allows space and time leaks? Hudak argues that embedding a domain specific language in an established host language is often the best way to create a special purpose language [18]. Such a domain specific embedded language (DSEL) allows you to focus on your particular problem without designing a new language, writing an efficient compiler, and convincing everyone that your language is a good idea. Haskell's typeclasses provide a very general framework to embed sublanguages, making it a great

host language when your DSEL is amenable to laziness. In the case of FRP, laziness permits space and time leaks. Any Haskell-embedded FRP language can have space leaks. This problem is solved by embedding FRP in a strict language.

In addition to space and time leaks, Classical FRP allows the definition of behaviors that depend on past, present, or future values [21]. Programs that violate causality – relying on future values – are unimplementable. Programs that rely on past values can cause another kind of memory leak in which the program is forced to remember all past values of a `Behavior` or `Event`. This means that memory usage could grow in proportion to the time the program has been running.

2.1.2 Real-time FRP and Event-Driven FRP

Real-time FRP [34] was introduced by Paul Hudak and others at Yale in 2001. It aims to resolve the inefficiencies of Classical FRP. Real-Time FRP (RT-FRP) overcame both space and time leaks, but this came at the cost of expressiveness. To produce efficiency guarantees, RT-FRP introduced an isomorphism between Behaviors and Events:

$$\texttt{Event}\ \alpha \approx \texttt{Behavior}\ (\texttt{Maybe}\ \alpha)$$

where `Maybe` α is an abstract data type that can either be `Just` α (an event is occurring, here is the relevant value) or `Nothing` (an event is *not* occurring, nothing to see here). This simplifies the semantics of FRP. Both behaviors and events can be represented as a common type which has been called a *signal*.

$$\texttt{Signal}\ \alpha = \texttt{Time} \rightarrow \alpha$$

With this simpler mental model, RT-FRP then defines a language that ensures that signals cannot be used "unsafely", in ways that do not have a clear, efficient implementation. To do this, RT-FRP presents a two-tiered language: an unrestricted base language and a more limited reactive language for manipulating signals. The base language is a basic lambda calculus that supports recursion and higher order functions. This base language is embedded in a much more restrictive reactive language that carefully controls how signals can be accessed and created. The reactive language supports recursion but not higher-order functions.

The original paper on RT-FRP provides proofs about resource boundedness for both time and space [34]: For time, reactive updates will terminate as long as the embedded base language terms terminate as well. For space, memory will not grow unless it grows in the base language.

Although the separation between base language and reactive language made it much easier to prove properties about RT-FRP, it is somewhat less expressive than Classical FRP. Since the reactive language is not higher order, the connections between signals must all be explicitly defined in the source code. They cannot be specified with the full power of the embedded lambda calculus.

In 2002 – soon after introducing RT-FRP – Hudak et al. proposed *Event-Driven FRP* [35]. Event-Driven FRP (E-FRP) is a direct descendent of RT-FRP that introduces *discrete* signals, signals that only change on events. Thus, E-FRP programs are *event-driven* in that no changes need to be propagated unless an event has occurred.

This may seem needlessly restrictive, but as it turns out, many potential applications of FRP are highly event-oriented. The original E-FRP paper focused on controlling robots, but graphical user interfaces are also primarily event driven. We will explore discrete signals more in the next chapter.

Although Real-time FRP and Event-driven FRP solved many of the efficiency problems of Classical FRP, research focus has shifted away from this approach in hopes of recapturing the full expressiveness of Classical FRP. The last published paper on RT-FRP goes so far as to suggest a concurrent runtime as future work [36], but this was never pursued. Furthermore, neither RT-FRP nor E-FRP was ever adapted to create graphical user interfaces. These two lines of research appear to have fallen by the wayside, but the prospect of a concurrent, event-driven, and strict FRP language remains promising.

2.1.3 Arrowized FRP

Arrowized FRP [28] aims to maintain the full expressiveness of Classical FRP without the difficult-to-spot space and time leaks. Arrowized FRP was formulated at Yale in 2002 by Henrik Nilsson, Antony Courtney, and John Peterson. Borrowing from the recent results of RT-FRP and E-FRP, events are no longer used. Arrowized FRP (AFRP) instead uses *signal functions*. A signal function can be thought of as a function from signal to signal.

$$\texttt{SF}\ \alpha\ \beta = \texttt{Signal}\ \alpha \rightarrow \texttt{Signal}\ \beta$$

where signals are exactly the same as in Real-Time FRP.

$$\texttt{Signal}\ \alpha = \texttt{Time} \rightarrow \alpha$$

To avoid time and space leaks, signals are not directly available to the programmer [24]. Instead, one programs only with signal functions. These

signal functions are conceptually equivalent to regular functions, but they make it possible to rule out time and space leaks in the *implementation* of the AFRP system. Because signal functions are specified at the source level, it is possible to carefully control how recursive functions are evaluated, ensuring that intermediate computations are not kept in memory. This eliminates a major source of space and time leaks. Of course, as a Haskell-embedded framework, space and time leaks are still possible when signal functions accumulate a computation without inspecting its value. Thus, Arrowized FRP does not have space and time leaks in its implementation, but programs written with AFRP can still have them.

Signal functions also ensure causality. Signal functions explicitly model input and output, so any future dependent signal function can be ruled out statically [23, 32]. Signal functions belong to the category of arrows, an abstraction developed by John Hughes in 2000 [19]. This abstraction had not yet been revealed when Classical FRP was designed. As an added benefit, Ross Paterson's arrow notation [29] allows programmers to name the inputs and outputs of arrows in a controlled way, making the use of arrows much more natural.

Because there is no direct access to signals, Arrowized FRP may appear to be less expressive than Classical FRP. AFRP achieves the flexibility of Classical FRP with continuation-based switching and dynamic collections of signal functions. We will not discuss these two features in depth, but their general role is to allow signal functions to be *safely* moved around at runtime. Together, these two features provide comparable expressivity to Classical FRP without reintroducing space and time leaks.

Significant work has gone into making Arrowized FRP more efficient [27, 23], but there is a foundational problem that AFRP inherited from Classical FRP: *continuous* signals. While Classical FRP was still the dominant paradigm, Hudak and Wan proved that their semantics can be respected with one important condition: "as the sampling interval goes to zero, the implementation is faithful to the formal, continuous semantics" [33]. In other words, the continuous semantics cannot be respected unless updates are instantaneous, which is impossible on a real machine. We will call this *the instant-update assumption*. The instant-update assumption introduces at least two major sources of inefficiency: global delays and unnecessary updates.

Global delays are caused by long computations. All of the FRP systems we have seen so far enforce a strict order of events. Events are processed one at a time in the exact order of occurrence. The instant-update assumption has masked a fundamental issue with this approach. If an update takes zero

time, no event will ever cause a delay. But updates *do* take time, sometimes a significant amount of time. The instant-update assumption is simply not true. Long updates block all pending events. As a result, one long-running computation will slow down the whole program.

Unnecessary updates are caused by discrete inputs, such as mouse clicks. Continuous signals assume that signal values are *always* changing. Thus, it is necessary to recompute the whole program as frequently as possible. With continuous signals, there is always a new value waiting to be shown to the user, so every part of the program is recomputed.

The trouble is that not all time-varying values are continuous. Inputs like mouse clicks are discrete, changing relatively infrequently. In fact, all user input is discrete. And from a purely practical standpoint, continuous signals do not actually exist in a digital system. Even time is discrete.

When the whole program is recomputed, the many discrete inputs cause recomputations, even though their values may not have changed. This would be acceptable if the instant-update assumption was true, but in practice, it means that each update wastes time recomputing values which cannot have changed. Consider the case of mouse coordinates. Instead of waiting for new mouse movements, a continuous signal of mouse coordinates updates as frequently as possible. Updates occur even when the mouse is stationary, causing a cascade of needless recomputation.

In a graphical user interface, all of the major inputs are discrete. Clicks, key presses, window resizes, mouse movements. Any sort of user input is a discrete event. All of these values induce busy-waiting in a system built for *continuous* signals.

Global delays and unnecessary updates both result from the instant-update assumption and the use of continuous signals. They are not inherent to FRP or Arrowized FRP. In his 2009 paper *Push-Pull Functional Reactive Programming*, Conal Elliott suggests that a distinction between continuous and discrete signals is better than a distinction between behaviors and events [9]. Elliott does not provide a detailed description of a push-based, or event-driven, implementation. Such a description will be provided in this thesis. In an effort to avoid global delays and unnecessary updates, the rest of our discussion of FRP will focus on examining the repercussions of removing the instant-update assumption and dealing instead with discrete signals.

2.2 Message-Passing Concurrency

Message-passing concurrency is a functional approach to concurrent systems. John Reppy and Emden Gansner began advocating this approach as early

as 1987 [11]. Their subsequent work resulted in Concurrent ML and eXene, which together form the basis of a robust GUI framework. This work also demonstrates that GUIs are naturally concurrent. That is, the many independent components of a GUI can and should be handled concurrently.

Previous work in FRP has assumed that all events are strictly ordered and processed one at a time. Concurrent ML and eXene illustrate that this is an unnecessary restriction. The original order of events does not *always* need to be maintained, especially when some events are unrelated to others. Furthermore, many events should be processed at the same time. If one part of a program takes a long time, this should not block everything else. By relaxing the event ordering restrictions, message-passing concurrency provides solutions to Arrowized FRP's two efficiency problems. Conceptually, message-passing concurrency is a set of concurrent threads that communicate by sending messages. This paradigm is discrete and concurrent from the start, so it is robust to both discrete inputs and long running computations.

In this section, we will first discuss Concurrent ML (CML). CML is a well-designed and type-safe language that illustrates the strengths of message-passing concurrency. Second, we will discuss Concurrent ML's GUI framework: eXene. eXene will show us how concurrency is necessary to create robust and responsive GUIs.

2.2.1 Concurrent ML

Interactive systems, such as graphical user interfaces, are the primary motivation for much of the work on Concurrent ML [30]. User input is one of the most complicated aspects of an interactive system, with many input devices interacting with the overall system. Concurrent ML aims to make it easier to design interactive systems by recognizing their natural concurrency.

A Concurrent ML program is a collection of sequential threads that communicate via messages. Concurrent ML extends Standard ML which provides a well-designed, sequential base language.

CML threads are lightweight – very cheap to create and run. Threads communicate through *channels*, allowing information to flow between threads that have no shared state. This is the essence of message passing. A channel of type

```
type 'a chan
```

can carry values of type `'a` back-and-forth between threads. Values – or messages – are sent and received with two intuitively named functions:

```
val send :  ('a chan * 'a) -> unit
```

```
val recv :  'a chan -> 'a
```

Every `send` must be paired with a `recv`. Each function will block execution within its thread until a matching function is called. `send` blocks until another thread wants to `recv`. Therefore, send and receive are *synchronous*.

Concurrent ML's threads can be created dynamically with the `spawn` function:

```
val spawn :  (unit -> unit) -> thread_id
```

An unbounded number of threads can be created, but limited system resources may not allow all of them to run simultaneously. Therefore, threads must take turns. The CML runtime system periodically interrupts threads. Threads yield to each other automatically, ensuring that they all get to run. This is called preemptive scheduling.

Concurrent ML has some lower level concurrency primitives that make it possible to define a wide diversity of message passing constructs, even asynchronous ones. This includes two types of communication that will eventually be very important to us: mailboxes and multicast. (1) A mailbox allows messages to be sent asynchronously. Messages are sent to a mailbox without waiting for a receiver. Messages queue up until the receiver is ready to look at them. The receiver always takes the oldest message available. Furthermore, receiving a message only blocks when the mailbox is empty. A mailbox can also be called a FIFO channel or queued channel. This is useful for asynchronous communication. (2) Basic message passing is point-to-point: one sender and one receiver. Multicast messages have one sender and *many* receivers. This is useful when one thread must send the same message to many others.

Concurrent ML is powerful enough to implement a variety of higher level concurrency strategies, but the overall approach is not as declarative as FRP. The programmer must think more about the "how" of GUI programming. Instead of automatically updating and transforming signals, CML requires the programmer to explicitly shuttle updates between threads. Nonetheless, Concurrent ML is a powerful framework for GUIs.

2.2.2 eXene: Creating concurrent GUIs

To avoid the complexity of flexible event ordering, many GUI frameworks are sequential. This ensures that the original order of events is strictly maintained. JavaScript, for instance, has a central event loop where all events queue up to be dispatched one by one. This means that any long computation will block all other pending events, making the GUI unresponsive.

JavaScript programmers must *explicitly* break any expensive algorithm into manageable chunks. This explicit interleaving is a poor substitute for real concurrency.

eXene – Concurrent ML's GUI library – realizes the benefits of a concurrent GUI by giving the programmer full access to a concurrent system [13, 12, 15, 14, 8]. A graphical user interface is particularly well-suited to this model because events in one part of the system usually do not need to be synchronized with events in another. For instance, all key presses need to be handled in their original order, but it may be okay if timer updates jump ahead or fall behind relative to the key presses. This means that GUI components can be updated independently, not blocking each other.

Concurrent ML and eXene have the flexibility to take advantage of this common case, producing more efficient GUIs. This is great for responsiveness, but it pushes the task of maintaining the order of events onto the programmer. Just as with any low-level abstraction, speed gains are tied to a whole new way to write buggy programs. In the case of CML, this means programs that suffer from deadlock and livelock.

Even acknowledging the event ordering problem, the modular nature of graphical user interfaces make them well-suited to a concurrent implementation. Concurrency improves performance without requiring invasive code rewrites such as breaking expensive algorithms up into small pieces. Even this unpleasant solution would be difficult with FRP. Because of the instant-update assumption, previous FRP implementations update the whole program at once, so breaking an algorithm into smaller pieces would not result in faster updates. As we will see in Chapter 4, concurrency is the ideal way to mitigate the damage of expensive computations. We will try to reap the benefits of concurrency without the synchronization headache. In Elm, preservation of event ordering is handled primarily by the compiler, not the programmer.

2.3 Existing FRP GUI Frameworks

Both Classical FRP and Arrowized FRP have been used as the basis for GUI frameworks. These GUI frameworks suffer from the same efficiency problems of their base language, but the biggest problem for every implementation is accessibility. Accessibility includes ease of installation, quality of documentation and examples, and apparent conceptual difficulty.

Many Haskell-embedded frameworks have been created over the years. These include FranTk, the original GUI framework for Fran [31]; Fruit, AFRP's first GUI framework [6]; Yampa/Animas, AFRP's most recent GUI

frameworks [7, 16]; and Reactive Banana, based on classical FRP which avoids time leaks with carefully constructed API's. Every Haskell-embedded library suffers from three problems:

Space leaks: As we discussed before, this is just a part of Haskell. Laziness sometimes leads to unexpectedly slow updates and unnecessary memory usage.

Difficult installation: Haskell FRP libraries all rely on imperative GUI frameworks – Tcl/Tk, Gtk, or wxWidgets – which complicates the install process. Each of these frameworks comes with their own set of constraints, often not easily supporting new platforms such as Android or iOS.

Apparent conceptual difficulty: All Haskell-embedded libraries rely on Haskell's robust type system to achieve clarity and usability. Classical FRP uses monads to introduce behaviors and events. Arrowized FRP uses the arrow framework to manipulate signal functions. In both cases, new users must learn the syntax and basic theory of monads or arrows to even begin coding. Although these abstractions are clear to the experienced Haskell user, beginners can find them imposing and discouraging. Of course, the difficulty may be more psychological than conceptual, but it is a difficulty nonetheless. This perception of difficulty is compounded because much of the documentation for these GUI frameworks can be found only in academic papers.

Because Haskell-embedded FRP libraries map onto imperative backends, there is also a danger of incorporating imperative abstractions into the library. While not necessarily wrong, this results in GUI frameworks that feel imperative. FranTk included event listeners – an imperative abstraction – because they mapped more naturally onto the Tcl/Tk backend, but most other libraries have avoided imperative influence.

Of the frameworks discussed here, it appears that Reactive Banana is the only one that is actively maintained and improved as of this writing.

Not all efforts have been focused on Haskell. FRP has even appeared in some imperative languages. For instance, the Frappé framework [5] targets Java. This implementation was produced by Antony Courtney in 2001, right before co-authoring the initial Arrowized FRP paper. Another imperative embedding is Flapjax [26] which targets JavaScript. Flapjax was produced by a diverse collaboration in 2009. The portion of the Flapjax team based at Brown University has also produced an FRP system for Scheme [3].

Flapjax is particularly interesting in that it addresses all three of the problems faced by every Haskell implementation. JavaScript is a strict language, so time leaks must be created explicitly by manually delaying computation. Flapjax is a JavaScript library, so "installation" is easy: just include the file in your webpage. Flapjax also avoids the apparent conceptual difficulty of Haskell-embedded libraries because its documentation just does not talk about monads. Of course it *could* discuss behaviors and events in terms of monads, but the analogy is not particularly useful when developing in JavaScript. Because Flapjax is web based, many interactive examples are available online, further helping with accessibility.

Similar to Flapjax, Elm provides documentation and examples online at elm-lang.org in an attempt to avoid difficult installation and apparent conceptual difficulty.

3 The Core Language

The core language of Elm aims to provide flexibility and expressiveness, yet only commit to abstractions that can be implemented efficiently in a concurrent system.

Elm achieves expressiveness in two major ways. (1) The core language combines a basic functional language with a small set of reactive primitives. Programmers have direct access to signals, and signals can be transformed and combined with the full power of a functional language. By carefully restricting Elm's reactive primitives, we can avoid the problems of Classical FRP without giving up signals. (2) Elm's core language is compatible with Arrowized FRP. Elm's small set of reactive primitives do not provide the ability to dynamically change how signals are used at runtime. Classical FRP and Arrowized FRP both have this ability. By ensuring that Arrowized FRP can be embedded in Elm, we show that Elm can be just as expressive, even without directly embedding these features in the core language.

Elm's efficiency improvements are based on two design choices. (1) Elm does not make the instant-update assumption. This assumption actively ignores many significant barriers to efficiently implementing FRP. By avoiding the instant-update assumption, Elm can avoid global delays and needless recomputations. (2) The core language of Elm maps onto an intermediate language that cleanly separates reactive primitives from the rest of the language. Why bother with a two-tiered intermediate language? Real-Time FRP and Event-Driven FRP use a two-tiered language to great effect, allowing proofs about efficiency and memory usage and promoting efficient implementations. For RT-FRP and E-FRP, the two-tiered source language came at the cost of expressiveness. By only using the two-tiered representation *internally*, Elm can benefit from its useful properties without restricting source level expressiveness. Elm's intermediate representation also maps nicely onto a concurrent system.

3.1 The Syntax: Manipulating Discrete Signals

Global delays and needless recomputation are two major sources of inefficiency in modern FRP formulations. Both are rooted in *the instant-update assumption* which requires that recomputation takes zero time. Following in the footsteps of Event-Driven FRP [35], we will not make this assumption. Like E-FRP, Elm is entirely event-driven. These *discrete signals* contrast with continuous signals that are always changing. This means that all updates in Elm are event-driven, recomputing only when an event has occurred.

In Elm, no event occurs at exactly the same moment as another. This ensures that there is a strict ordering of events, and simplifies the mental model for events: events happen one at a time. The exact co-occurrence of events is also extremely uncommon in GUIs which deal primarily with user input. On the time-scale of computers, users are slow enough that user input events really are strictly ordered.

Elm's basic design choices point the way to a core language. We want a set of reactive primitives that work nicely with event-driven updates, discrete signals, and strictly ordered events. Elm's reactive primitives are embedded directly in a simple lambda calculus. This appears to be quite similar to the formulation of classical FRP, but our compilation strategy will disallow the major inefficiencies that we have come to expect of this syntax.

$$\begin{aligned} e ::= & () \mid n \mid \lambda x.\, e \mid e_1\ e_2 \mid x \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mid \\ & i \mid \texttt{lift}_n\ e\ e_1\ ...\ e_n \mid \texttt{foldp}\ e_1\ e_2\ e_3 \mid \texttt{async}\ e \end{aligned}$$

$$n \in \mathbb{Z} \qquad x \in \mathsf{Var} \qquad i \in \mathsf{Input}$$

For now, it is best to think of i as a unique identifier for input signals, such as mouse position or key presses. The Input set represents all possible identifiers for input signals. We will give a more detailed interpretation of Input in Chapter 4. These inputs i receive values from some external event source. The external source can be any system input, from mouse movements to memory access. Furthermore, Elm's input signals *must* have a default value; they cannot be undefined. As long as a signal is undefined, any program that depends on it will be undefined. It is best to rule this out altogether.

Basic signals are Elm's only way of accessing stateful system values; they are the inputs to an Elm program. But just having access is not enough. Elm also needs a way to *manipulate* signals. This is the role of Elm's other reactive primitives. With the $\texttt{lift}_n$ and `foldp` primitives, Elm accommodates three types of signal manipulation: transformation of a signal, combination of multiple signals, and past-dependent transformation of a signal.

Before discussing signal manipulation in depth, we must briefly introduce Elm's final primitive: `async`. Elm allows you to add the `async` annotation indicating when certain signal manipulations can be computed in parallel with the rest of the program. Such signals are *asynchronous*.

$$\texttt{async}\ ::\ \texttt{Signal}\ a \to \texttt{Signal}\ a$$

The `async` annotation takes in a signal and returns the exact same signal. The annotation just informs the Elm compiler that events flowing through

the signal do not need to be perfectly synchronized with the events in the rest of the system, allowing an important optimization. This primitive will be more fully described in Chapter 4. For now, it is enough to understand `async` simply as an annotation for the Elm compiler.

The following examples illustrate the usage and importance of Elm's reactive primitives. The syntax used in the examples is the actual surface syntax of Elm.

3.1.1 Transformation

Basic signals such as mouse position or window size are not interesting unless they can be transformed into some other value. This is accomplished with

$$\texttt{lift} :: (a \to b) \to \texttt{Signal}\ a \to \texttt{Signal}\ b$$

Think of `lift` as a directive to lift a normal function into a signal. This primitive can create time-indexed animations when given a function from times to pictures. One simple time-indexed function is a value that slides between negative and positive one, updating every fifth of a second:

```
cosWave = lift cos (Time.every 0.2)

sinWave = lift sin (Time.every 0.2)
```

The function `Time.every` takes a number n and produces a signal carrying the time since the program began, updated every n seconds. Lift can also be used on more traditional inputs such as `(Window.width :: Signal Int)` which carries the current width of the display. One basic use is fitting a component to the window width up to a certain value:

```
fitUpTo w = lift (min w) Window.width
```

The signal `(fitUpTo 800)` is equal to the window width up until window width 800 pixels, where it stops growing.

Elm also supports a basic signal called `(never :: Signal ())` which has the default value unit and never produces another value. This special input signal can be used to safely lift constant values into signals.

```
constant c = lift (λx → c) never
```

This transforms the default value of `never`, so the resulting signal carries the value `c`. This can be useful when you would like to pass a constant to a function that takes a signal.

3.1.2 Combination

Combining signals is also quite useful. For this task, we have

$$\texttt{lift2} :: (a \to b \to c) \to \texttt{Signal}\ a \to \texttt{Signal}\ b \to \texttt{Signal}\ c$$

This allows you to combine two signals. The given function acts on the most recent values available. With this we can combine some of our earlier signals:

```
point = lift2 (,) cosWave sinWave
```

The `(,)` function takes two values and pairs them together in a tuple, so `point` is a signal of number pairs. We can treat these number pairs as coordinates, much like the coordinates produced by the `Mouse.position` signal. From here you could calculate the current distance between different points:

```
lift2 distance point Mouse.position
```

The `distance` function takes two coordinates and returns the distance between them. The resulting signal dynamically updates as the values of `point` and `Mouse.position` change.

In Elm, any GUI that relies on multiple inputs requires the `lift2` primitive. Transformation with `lift` is not enough to combine signals. Furthermore, `lift2` can be used to define a lift on any number of signals:

```
lift3 f s1 s2 s3 = lift2 ($) (lift2 f s1 s2) s3
```

The `($)` function is function application, so `(f $ x)` is the same as `(f x)`. Although all lift functions can be derived from `lift2`, Elm uses a different primitive for stateless signal transformations: $\texttt{lift}_n$. The $\texttt{lift}_n$ primitive encompasses all stateless signal transformations, including `lift`, `lift2`, etc.

3.1.3 Past-dependent Transformation

So far we have no way to remember the past. The lift operations only act on the current value. Elm also has a stateful transformer:

$$\texttt{foldp} :: (a \to b \to b) \to b \to \texttt{Signal}\ a \to \texttt{Signal}\ b$$

The `foldp` function is meant to be read as "fold from the past", just as `foldl` for lists can be read "fold from the left". As `foldp` receives values from the input signal, it combines them with an accumulator. The output signal is the most recent value of the accumulator. This is needed for any

past-dependent user interface. For example, remembering the sequence of keys a user has typed:

```
foldp (λk word -> word ++ [k]) "" Keys.lastPressed
```

where (`Keys.lastPressed :: Signal Char`) carries the most recent key press. This expression produces a string signal that automatically updates upon keyboard events.

3.2 The Type System: Enforcing the Safe Use of Signals

Signals are generally quite flexible, but there are certain signals that we want to rule out altogether: signals of signals. Signals of signals would permit programs that dynamically change the structure of the Elm runtime system. In many circumstances this is fine. For instance, there would be no problem if two input signals were dynamically switched in and out. The real issue is stateful signals created with `foldp`. Imagine if a stateful signal – dependent on mouse inputs – was switched out of the Elm program. Should it still update on mouse inputs? Should it only update when it is in the program? When working directly with signals, neither answer is pleasant. Elm opts to avoid this behavior by ruling out switching.

Elm's type system rules out signals of signals by dividing types into two categories: primitive types and signal types. Primitive types are safe types that are allowed everywhere, including in signals. Signal types are dangerous types that are allowed everywhere *except* in signals.

$$\tau ::= \mathsf{unit} \mid \mathsf{number} \mid \tau \to \tau'$$
$$\sigma ::= \tau\ \mathsf{signal} \mid \tau \to \sigma \mid \sigma \to \sigma'$$
$$\eta ::= \tau \mid \sigma$$

Mnemonics for types τ, σ, and η are as follows: Tau is for type. Sigma is for signal type. And with a very strong accent, Eta is for either.

Our typing judgments for the lambda calculus primitives are extremely permissive. Higher-order functions and recursive functions *can* include signals. The important restrictions are on our reactive primitives. For instance, $\texttt{lift}_n$ can only lift functions that act on base types τ, not signal types σ. This makes it impossible to create signals of signals. Even if such signals were permitted, there is no primitive to turn a signal of signals into a useable signal.

Notice that the types of our input signals i must be looked up in the typing environment. Because these signals are Elm's direct access to the system, their types must be explicitly stated in the default typing environment.

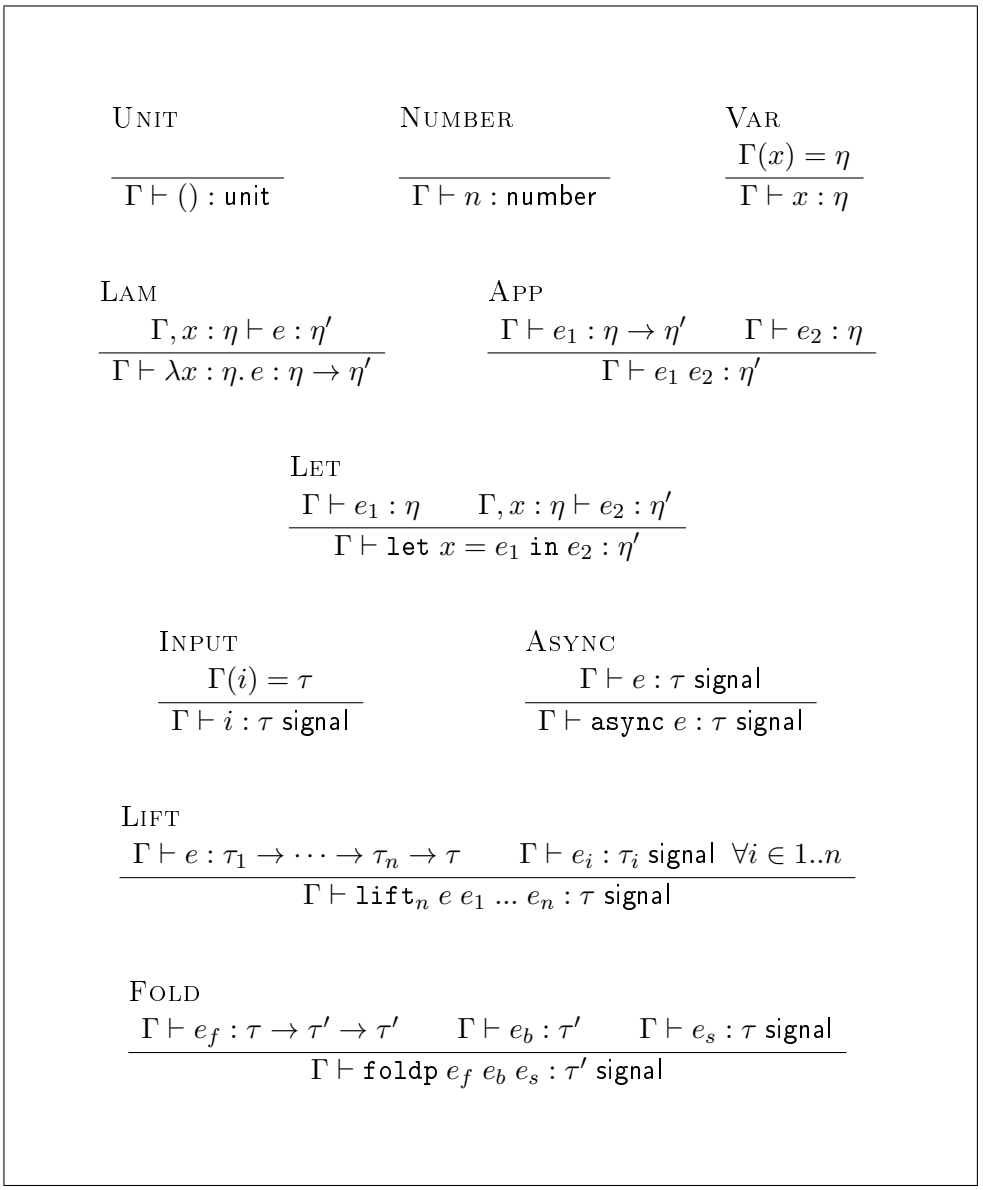


Figure 2: Typing Judgments

3.3 Embedding Arrowized FRP in Elm

By embedding Arrowized FRP in Elm, we show that Elm and AFRP can achieve equal levels of expressiveness. As we saw in the previous section, Elm's type system rules out dynamic signal switching to avoid ill-defined uses of stateful signals. Embedding Arrowized FRP in Elm will give us dynamic switching without the ambiguous behavior. We will begin this task with a more general question about theoretical expressiveness: how does Elm's core language relate to Classical FRP and Arrowized FRP? McBride and Paterson's work on applicative functors clarifies this question [25].

As we discussed in the previous section, Elm does not allow signals of signals, but otherwise signals can be combined in a fairly general way. These restrictions can be helpfully examined in terms of applicative functors.

```
class Functor f => Applicative f where
    pure  :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b

instance Applicative Signal where
    pure c = constant c
    f <*> x = lift2 ($) f x
```

So signals in Elm are applicative functors, but are they also monads? Classical FRP uses monads to represent its signals, but does Elm? A monad is simply an applicative functor with one extra function:

```
class Applicative m => Monad m where
    join :: m (m a) -> m a
```

Monads can be flattened. This would allow signals of signals to be flattened, precisely what we want to avoid. With `join`, it would be possible to dynamically change the structure of the signal graph, a problem for stateful signal transformers. Elm signals do not have this capability by design, and therefore, they are not monads.

So signals in Elm are not monads, but are they arrows? The work on Arrowized FRP has shown that the arrow abstraction is a powerful and safe framework for signals. According to McBride and Paterson, "By fixing the first argument of an arrow type, we obtain an applicative functor" [25]. We have already shown that Elm signals are applicative functors, so they must also be arrows which have their first argument fixed.

In our case, the first argument of the arrow type is "the world". Recall that in Arrowized FRP, the arrow type is called a signal function, so an

arrow type from a to b would be written as (`SF` a b). Thanks to McBride and Paterson, we have the following isomorphism.

```
Signal a = SF World a
```

Each signal in Elm represents a signal function that transforms the world into some value. So yes, Elm's signals are arrows too.

McBride and Paterson ultimately conclude that "We have identified Applicative functors, an abstract notion of effectful computation lying between Arrow and Monad in strength" [25]. Working through this with signals has shown us that Elm falls between Arrowized FRP and Classical FRP in terms of theoretical expressiveness. These connections suggest three directions for future work:

Nicer syntax: McBride and Paterson's notation for applicative functors may be a good fit for Elm. Furthermore, a modified version of Paterson's arrow notation may also be desirable [29].

Fully Embedding Arrowized FRP: Since Elm's signals are an applicative functor – or an arrow from "the world" to a value – Arrowized FRP could be fully embedded in Elm. Elm's signals could compose easily with AFRP's signal functions. This means that Elm could potentially use AFRP's dynamic collections of signal functions. With this addition, Elm would have all of the expressive power of AFRP with the benefits of a concurrent runtime.

Potential Framework for Testing and Code Reuse: Since a signal can be understood as a signal function from "the world" to a value, it may be possible to modify "the world" before it is given to an Elm signal. For instance, a signal of mouse clicks – which normally receives events from the mouse – could be modified to receive events from a timer, indicating clicks at predefined intervals. Imagine providing a scripted world in which the mouse coordinates follow a pre-determined path, the keyboard input is timed and scripted, etc. This would allow automated interface testing directly in Elm. The arrow isomorphism could also allow existing components to be reused in new ways. A component written to rely on mouse coordinates could be modified to accept coordinates from keyboard input, HTTP responses, a timed signal, etc. If used too liberally, this could be a very dangerous addition, potentially making it difficult to change the implementation of a widely used component.

3.4 Syntax-Directed Initialization Semantics

Elm's source language is quite expressive. Programmers have direct access to signals, and Arrowized FRP can be fully embedded in Elm, providing us with all the expressivenes of AFRP. This expressiveness makes it difficult to directly compile the source language to a concurrent runtime system. This section describes how to convert Elm's source language into a well-behaved intermediate representation that can produce a concurrent system more easily. The intermediate representation is two-tiered, separating reactive values – such as `lift` and `foldp` – from the rest of the language. Elm's intermediate representation closely resembles the source language of Real-Time FRP and Event-Driven FRP.

Remember Elm's base expression language:

$$\begin{aligned} e ::= & () \mid n \mid \lambda x.\, e \mid e_1\ e_2 \mid x \mid \texttt{let}\ x = e_1\ \texttt{in}\ e_2 \mid \\ & i \mid \texttt{lift}_n\ e\ e_1\ ...\ e_n \mid \texttt{foldp}\ e_1\ e_2\ e_3 \mid \texttt{async}\ e \end{aligned}$$

$$n \in \mathbb{Z} \qquad x \in \mathsf{Var} \qquad i \in \mathsf{Input}$$

We will now specify initialization semantics which turn these expressions into values. In addition to traditional values v, Elm also has signal values s. This distinction separates base values and reactive values, just as in Real-Time FRP.

$$v ::= () \mid n \mid \lambda x.\, e$$

$$s ::= x \mid \texttt{let}\ x = s_1\ \texttt{in}\ s_2 \mid i \mid \texttt{lift}_n\ v\ s_1\ ...\ s_n \mid \texttt{foldp}\ v_1\ v_2\ s \mid \texttt{async}\ s$$

These v and s values will be our intermediate representation. Every source level expression e can be mapped onto either a traditional value v or signal value s. The signal values s will eventually form the basis of our concurrent runtime system, each value performing repeated computations. For example, the s value `(lift (clamp 100 200) Mouse.x)` produces a value between 100 and 200 based on the mouse coordinates, clamping the range of `Mouse.x`. This s value performs computation every time the mouse moves. If this s value were to be copied to multiple locations in our final intermediate representation, it would be computed multiple times in our concurrent runtime system, which is based on s values. We want to ensure that no s value appears multiple times to avoid needlessly duplicated computations.

To avoid copying signal values, Elm's intermediate representation includes let-expressions and variables. Signal values that are used many times are bound to a variable x, but not actually copied to the uses of x. This allows a signal value to appear only once, but to be used in many places.

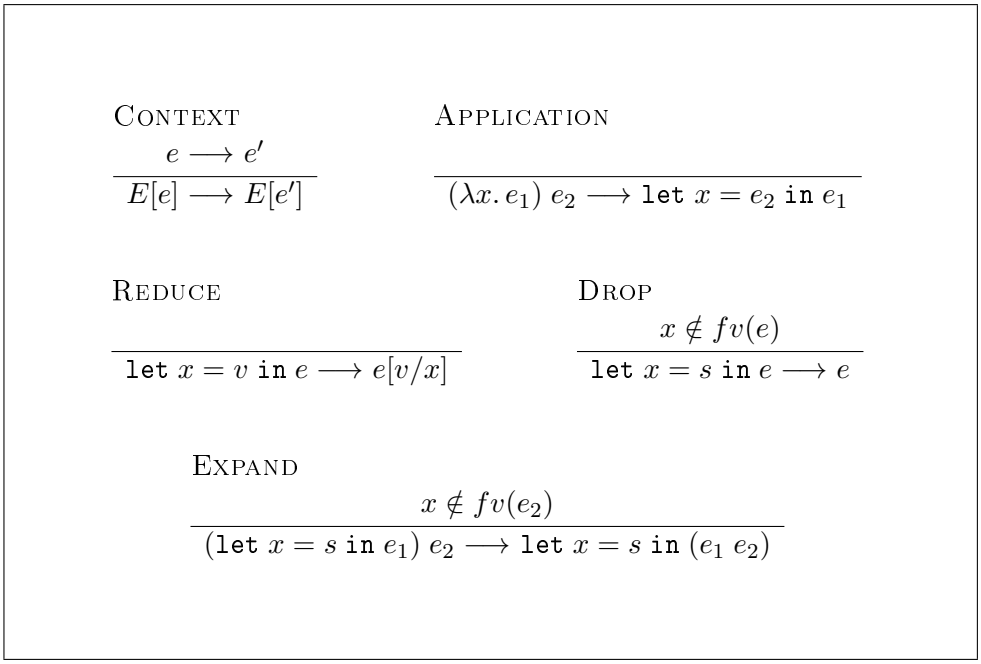


Figure 3: Syntax-directed Initialization Semantics

The mapping to s and v values is achieved with the evaluation context E and the initialization semantics shown in Figure 3. The evaluation context for initialization is fairly standard. This formulation is call-by-value to avoid copying signals, which would be possible with a call-by-name semantics.

$$
\begin{aligned}
E ::= {} & [\cdot] \mid E\; e \mid v\; E \mid \\
& \texttt{let}\; x = E\; \texttt{in}\; e \mid \texttt{let}\; x = s\; \texttt{in}\; E \mid \\
& \texttt{lift}_n\; E\; e_1\; ...\; e_n \mid \texttt{lift}_n\; v\; s_1\; ...\; E\; ...\; e_n \mid \\
& \texttt{foldp}\; E\; e_2\; e_3 \mid \texttt{foldp}\; v_1\; E\; e_3 \mid \texttt{foldp}\; v_1\; v_2\; E \mid \\
& \texttt{async}\; E
\end{aligned}
$$

The evaluation context E covers a majority of cases, but there are two kinds of reductions that remain unspecified: reduction of applications and of let-expressions. These cases require special care because they could allow multiple copies of s values, even in a call-by-value semantics. If we used standard β-reduction, values bound to a variable x would be copied to all uses of x. Our initialization semantics – shown in Figure 3 – are syntax-directed to avoid this.

Applications are converted into let-expressions with the "Application"

rule, shown in Figure 3. Thus, every bound value will be placed in a let-expression. In the case of traditional values v, we can just copy v to all of the places that use it. This is the role of the "Reduce" rule. By contrast, signal values s are *captured* by let-expressions. The signal value cannot escape from the let-expression; they are instead represented by a variable. This ensures that multiple *uses* of a signal do not require multiple *copies* of a signal.

Given an expression e, the $fv(e)$ function returns the set of free variables in that expression. It is used to ensure that the "Drop" and "Expand" rules are safe. The "Drop" rule throws out any bound signal value that is not actually used. The "Expand" rule allows function application in cases where a function uses a signal value multiple times. Any variable naming conflicts that occur because of the "Expand" rule can be resolved with α conversion – renaming variables.

Again, this initialization process results in either a traditional value v or signal value s. These two-tiered values closely resemble the surface syntax of RT-FRP and E-FRP, separating the reactive primitives from the more powerful primitives of the lambda calculus. The initialization process also avoids copying signal values. We now have an intermediate representation that is easier to map onto an efficient concurrent runtime system.

4 Concurrent FRP

Elm's source language allows declarative specifications of user interaction, but how can a specification be turned into an efficient runtime system? This chapter describes Concurrent FRP, an efficient variant of FRP that avoids the problems of space leaks, global delays, and needless recomputations.

By examining Concurrent ML and eXene in Chapter 2, we saw that GUIs are naturally concurrent. They have many distinct components that can safely run independently. Elm's intermediate representation can be easily mapped onto a concurrent system. This allows Elm to reap the benefits of concurrency without low-level concurrency primitives. Instead, concurrency is implicit in the structure of an Elm program. Each reactive primitive produces a thread, and the structure of an Elm program indicates how these threads should communicate.

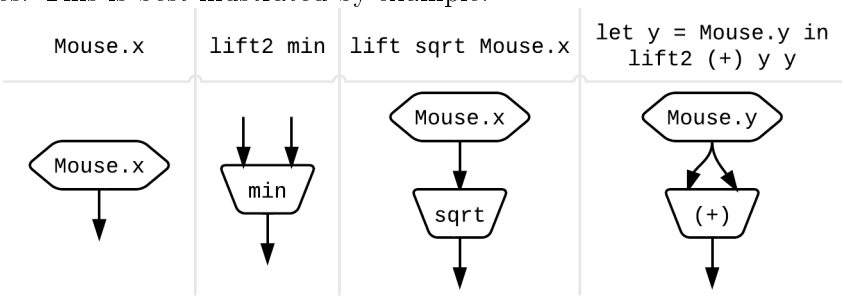
The order of events is not guaranteed in a concurrent system. Traditional FRP used sequential updates, so only one event could be processed at a time. This ensured that the original order of events was maintained. In Concurrent FRP, many updates can be processed at the same time. Because there are many paths through the concurrent system, it is possible for an event to "jump ahead" of others. For instance, key presses could get out of order, unexpectedly scrambling user input. As with many concurrent systems, Elm must provide a synchronization mechanism for each thread to maintain the original order of events. This mechanism also ensures that values are not recomputed unless absolutely necessary, avoiding the needless recomputation of previous FRP systems.

In Elm, synchronization is enabled by default, strictly maintaining the global order of events. But by its very nature, synchronization incurs a delay. It requires faster results to wait for slower results before moving on. In certain cases, it is acceptable to ignore the global order of events. Elm's `async` primitive allows programmers to explicitly annotate such cases. By indicating that synchronization is not necessary, Elm can avoid the delay incurred by synchronization, resulting in faster updates. The `async` primitive is the most important part of Concurrent FRP. It allows the Elm runtime system to run expensive computations entirely independently of the rest of the program. This finally solves the problem of global delays.

After addressing synchronization, we will discuss a direct mapping from Elm's intermediate representation to Concurrent ML. This will provide a more concrete specification of how the concurrent runtime system works. Finally, we will discuss the maximum theoretical benefits of our concurrent system and what that means for optimization.

4.1 An Intuition for Signal Graphs

How should Elm's intermediate representation be mapped onto a concurrent system? Every Elm program can be thought of as a *signal graph*: reactive primitives are nodes, connected by directed edges that indicate the flow of values. This is best illustrated by example:

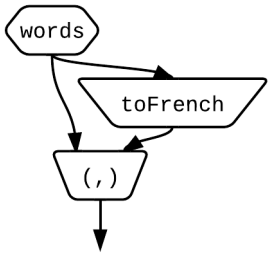


Input primitives such as `Mouse.x` and `Mouse.y` create hexagonal nodes with no incoming edges. The first example shows this very simple case. Lifted functions – as seen in the second example – create nodes that have one or more inputs, allowing us to transform signals. In the third example, lifting `sqrt` allows us to take the square root of the mouse's current x coordinate. Let-expressions make it possible to use one signal multiple times. In the third example, we use a let-expression to double the current y coordinate.

Nodes perform computation on the values received on their incoming edges. The result is then sent out of every outgoing edge. It is easy to imagine each node running in parallel, but problems arise quickly. How do we ensure that the order of events is maintained? 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)
```

We assume that `words` is a primitive input i. The signal graph for `wordPairs` would look like this:



Of the two inputs to `lift2`, the second input path is much more expensive. Say that `toFrench` references a English to French dictionary in memory,

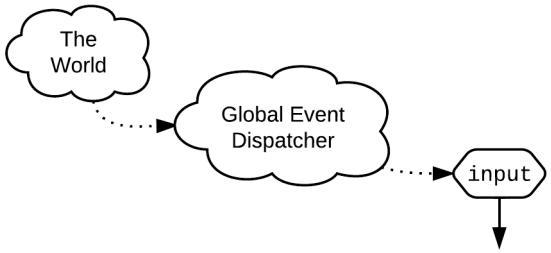
so it will always take longer than just showing the original word. Even if the word events are fed into the graph in order, there is no guarantee this order will be *preserved* as they flow through. Without event order guarantees, `wordPairs` may produce mismatched pairings.

Without a guarantee that event order is preserved, our concurrent system would create an unpredictable and inconsistent user experience in almost all cases. Imagine a GUI that sometimes processed keyboard input out of order! How can this be avoided?

4.2 Signal Graphs and the Global Event Dispatcher

Our event ordering problem is very common in distributed systems [22, 17]. In our case, the problem is simplified by the fact that Elm's signal graphs are always directed acyclic graphs (DAGs). Our reactive primitives created edges that pass information in only one direction (directed). Furthermore, It is impossible for an edge to depend on itself by looping back into the graph in an arbitrary way (acyclic). This means that a message will pass through each node only once, never more. Since we are working with DAGs, feedback loops are impossible. We just need to make sure that no event "jumps ahead" of any other as we saw in the `wordPairs` example.

Elm uses a *global event dispatcher* to ensure that events are properly distributed. This dispatcher manages all of the input to an Elm program. Whenever a new event comes in, the event dispatcher notifies all input nodes, even if the update does not change the node. Input nodes can be thought of as having an implied incoming edge from the global event dispatcher.



In this diagram, "The World" signifies any external data source. All interactions with the external world – mouse movements, key presses, file I/O, HTTP requests, etc. – pass from the world to the global event dispatcher.

Every node is reachable from an input node. By sending updates to every input node, we know that every internal node will eventually receive the updates as well. Our graph edges are queued – first-in-first-out – which

guarantees that no update "jumps ahead" of another.

More formally, the concurrent system and global event dispatcher works as follows:

- Each node is a concurrent thread, and each edge is a channel.
- Multi-input nodes must receive a value from each input edge before computing a result.
- All input nodes receive a notification when *any* event occurs. When the event is relevant to the input node, it passes along its new value. In the more common case, that the event is irrelevant, the input node passes along a message indicating that nothing has changed.
- Each edge in the graph is queued. This is useful for nodes with multiple inputs. If one input is updated more quickly than the others, it just waits on the queued edge. This accommodates many different flow rates within a signal graph.

Note that input nodes always propagate a message, even if it just says "no change". These "no change" messages are a form of memoization. A naive implementation would always propagate the actual value, causing every node in the graph to recompute its value. This is an easy way to ensure that every internal node receives messages from input nodes, but it is wasteful because an update indicates that *one* input value has changed, not *all* inputs. Therefore, much of resulting recomputation would be wasted work, producing exactly the same value as previously computed. The "no change" messages avoid these unnecessary recomputations.

Continuous formulations of FRP struggled with discrete inputs. Elm solves this problem with "no change" messages. These messages tell nodes to send along the latest computed value rather than computing it again. Thus, discrete inputs only cause computations when a relevant event occurs.

The synchronization mechanisms specified in this section ensure that the order of events is maintained without causing unnecessary recomputations. Looking back at the `wordPairs` example, we see that these mechanisms make it impossible to create mismatched translations. The node that combines English and French words must wait on a message from *both* inputs. Messages on the English path just queue up if they arrive too quickly.

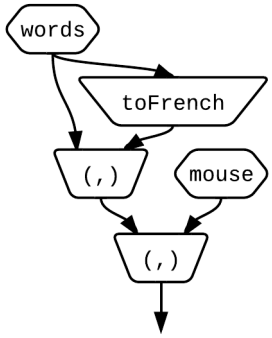
4.3 Asynchronous Updates

There is a tension between synchronization and fast updates. Elm's synchronization scheme is necessary, but it imposes an update delay. Lift nodes are forced to wait for an input on *all* incoming edges, even if one input takes much longer to compute than the others. This may cause an unacceptable delay.

Consider an expression that combines our `wordPairs` signal with the mouse position.

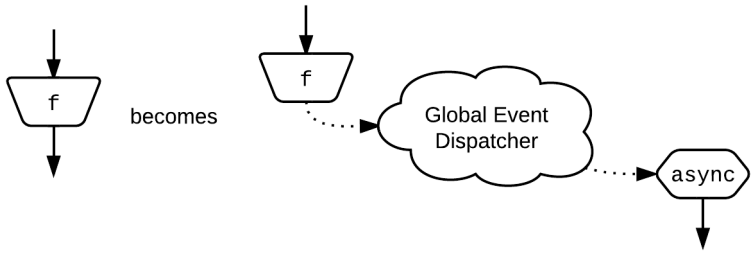
```
lift2 (,) wordPairs Mouse.position
```

For simplicity, this example uses `Mouse.position` directly, but the following arguments still apply when the mouse position is used in more exciting ways. The important aspect of this example is that one input updates quickly and frequently, whereas the other input updates less often and less quickly. This example gives us the following graph:



We know that the order of events must be maintained within the `wordPairs` section of the graph, but do we need to maintain the *global* order of events? Imagine that this graph receives one `words` event and then five `mouse` events. All mouse inputs would be delayed until the English word has been successfully translated. But why? These inputs are unrelated, so there is no reason for the mouse updates to wait. From the user's perspective, it is perfectly fine if the mouse events "jump ahead" of the translation events. The resulting system would be more responsive.

This is the role of the `async` primitive. Think of it as an annotation that says, "this subgraph does not need to respect the global order of events". The `async` primitive simply performs a local graph rewrite.

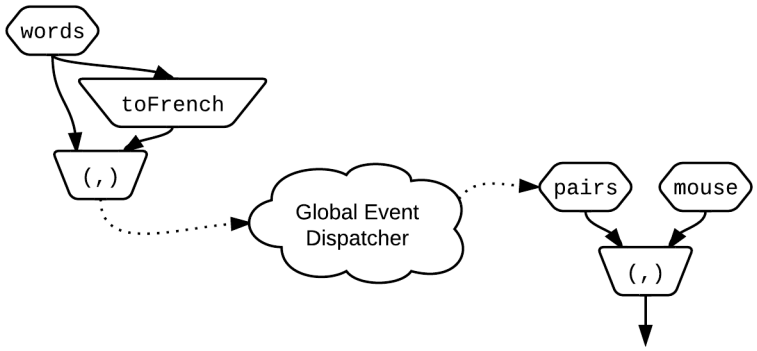


It replaces a normal graph node with an input node, allowing the computation `f` to take place while other events flow through the graph. This behavior is better explained by returning to our example.

In this case, we can use `async` to indicate that the `wordPairs` subgraph should not block mouse updates.

```
let pairs = async wordPairs in
  lift2 (,) pairs Mouse.position
```

This produces a graph with two fairly distinct sections: the "primary subgraph" that combines translation pairs with the mouse position and the "secondary subgraph" that produces translation pairs.



The primary subgraph – on the right – is now much smaller than the original fully-synchronous graph. It just combines two input values, a fairly quick operation. The values produced by the secondary subgraph are sent to the global event dispatcher whenever they are ready. Thus, the primary subgraph does not block mouse events while a translation is in progress. Notice that the `wordPairs` subgraph still ensures that translations are always paired correctly.

Returning to our original event order – one event for the `words` input and then five events for the `mouse` input – we see the that the translation no longer blocks mouse updates. The first event does not cause any new computation in the primary subgraph. The five mouse events are propagated through, performing minor computations. When the translation is finally complete, the global event dispatcher will pass the translation pair to the primary subgraph. This means the `pairs` update will come *after* all of the mouse updates.

Effectively, the `async` primitive breaks a fully synchronous graph into one primary subgraph and an arbitrary number of secondary subgraphs running independently. 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.

In cases that permit asynchrony, `async` can significantly improve update speeds by removing a long computation from the primary subgraph. This powerful optimization is often possible in graphical user interfaces, which tend to be fairly modular. With the `async` primitive, we have finally found a way to allow expensive computations without slowing down the entire program.

4.4 Translation to Concurrent ML

Having developed an understanding of Concurrent FRP, we will now explore how to translate Elm's intermediate representation to a concurrent runtime. We will target Concurrent ML because it is type-safe and theoretically well understood. Nonetheless, the approach taken here is quite general, so many other concurrent backends could be targeted with the same approach.

To help us with the translation, we define a small set of useful functions in Figure 4on page 35. These helper functions include two important definitions: the event data type which includes a "change" or "no change" flag to avoid needless recomputation and a `guid` function to create globally unique identifiers.

Next we have the basics of our concurrent runtime system in Figure 5 on page 35. The basic runtime system includes two important loops: the global event dispatcher `eventDispatch` which propagates events and the display loop `displayLoop` which updates the user's view as new display values are produced. Together, these loops are the input and output for an Elm program, with `eventDispatch` feeding values in and `displayLoop` outputing values to the screen.

The global event dispatcher receives notifications of new events through

```
datatype 'a event = NoChange 'a | Change 'a
either f g e = case e of | NoChange v => f v
                         | Change v => g v
change = either (fn _ => False) (fn _ => True)
bodyOf = either id id

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

Figure 4: Helper Functions

```
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 5: Concurrent ML Runtime System

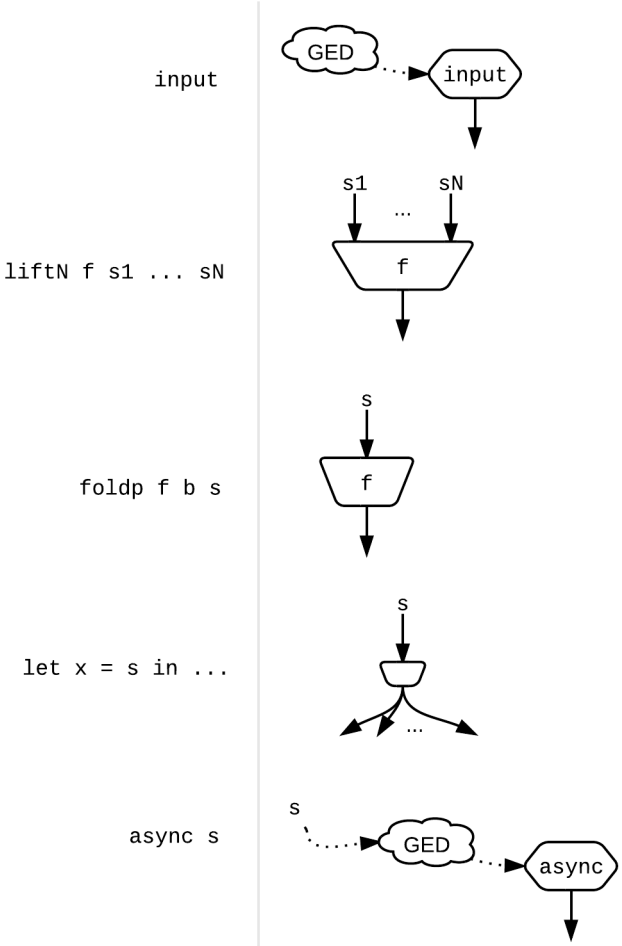
the `newEvent` mailbox. This mailbox is a queued channel, guaranteeing that the order of event notifications is preserved.

An event notification carries a unique identifier indicating which input has changed. Upon receiving an event notification, the global event dispatcher sends the event ID out through the `eventNotify` multi-cast channel. This multi-cast channel – or `mChannel` – sends the event ID to all input nodes.

The `displayLoop` depends on the translation of an Elm program in intermediate form 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.

We finally turn to the translation from s values to Concurrent ML as seen

is Figure 6 on page 38. This translation takes in an s value and produces a pair. The pair contains the initial value of the program and the signal graph that updates on events. The translations correspond to the graph representation we built up in previous sections, which are all displayed in the following diagram.



The most important part of each translation is the `loop` function which specifies the behavior of the resulting thread.

First we have the translation for input signals i. Until now we have just thought of an input as an identifier in the set `Input`, but now that we are providing a runtime system, we can give a better interpretation. When translating to CML, an input i is a 3-tuple containing (1) a unique identifier which allows the runtime system to differentiate between input signals, (2) a multi-channel event source that sends new values along from the runtime

system, and (3) a default value which ensures that no signal is undefined.

An input thread receives event IDs from the `eventNotify` multi-channel. If the ID matches the input thread's internal ID, it takes a value from its event source and passes it along. Otherwise, it just maintains its current value, passing a "no change" message.

Next we have the translation for lift nodes. A lift node has one function and n signals. Each of the n signals must be translated, producing n default inputs and n input channels. The default inputs are used to compute the default value of the current node. The `loop` for lift waits to receive a message from every input channel. It then checks to see if any of the inputs have changed. If so, the new output value is computed and a change is reported. Otherwise, the node sends a "no change" message with the most recent output value.

Fold nodes take one input, but rather than taking the default value from the input, fold nodes use their base value v as the default. Upon receiving a new input value, the fold node uses the accumulator function f to combine the new value and the existing node state. This new value is passed along to the next node and then saved as the current node state.

Let expressions and variables work together to ensure that Elm's run-time system does not have duplicate nodes. A let expression node serves as a multi-cast station, forwarding messages along to many different nodes. The let expression node computes a default value and creates a multi-cast channel. The variable node just returns the given default value and creates a port on the multi-cast channel.

Asynchronous nodes spawn a `loop` just like any other node, but instead of returning on the output channel of the loop, it returns a new input node. The `loop` ignores any "no change" messages. Since everything remains the same, there is no need to trigger a new global event. When the `loop` receives a new value, it passes it along to the newly created input node and informs the global event dispatcher that an event has occurred via the `newEvent` channel. This triggers an update, propagating the asynchronously computed value through the concurrent system.

```
⟦ ⟨id, mc_in, v⟩ ⟧ = spawn (fn _ => loop v) ; (v, c_out)
  where c_in, c_out = port mc_in, mailbox ()
        eid = port eventNotify
        loop prev =
          let msg = if recv eid == id then Change (recv c_in)
                                      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 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 (bodyOf x_1) ... (bodyOf x_n))
                    else NoChange prev
          in  send c_out msg ; loop (bodyOf msg)

⟦foldp f v s_in⟧ = spawn (fn _ => loop v) ; (v, c_out)
  where (_, c_in), c_out = ⟦s_in⟧, mailbox ()
        loop acc = let msg = case recv c_in of
                               | NoChange _ -> NoChange acc
                               |   Change v -> Change (f 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)

⟦async s_in⟧ = spawn loop ; ⟦ ⟨id, c_out, v⟩ ⟧
  where (v, c_in), c_out, id = ⟦s_in⟧, mChannel (), guid ()
        loop () = case recv c_in of
                    | NoChange _ -> loop ()
                    |   Change v -> send c_out v ;
                                    send newEvent id ; loop ()
```

Figure 6: Translation from Signal Values to CML

4.5 Benefits of Concurrency

Traditional FRP performed each update one-by-one. This is the worst case scenario in Concurrent FRP. Elm's concurrent runtime provides two major benefits – parallelism and pipelining – which can significantly improve update speeds.

Computations are said to be *parallel* when they are performed simultaneously in hardware. Given n computations that each take time t_i, running them sequentially would take $\sum t_i$ whereas running them in parallel would take $\max t_i$. Because an Elm program consists of many concurrent threads, the work required for one update can be parallelized. Computations can overlap each other in time, reducing the total time needed for a single update.

Parallelism improves *latency*: the time needed for a single update. Latency can be further improved with the `async` primitive. This primitive can remove long computations from an update entirely. When used on expensive computations, `async` also reduces the synchronization delays. Together, parallelism and `async` permit significant improvements to latency.

Traditional FRP computes each update one at a time. One update must end before the next update can begin. *Pipelining* is when multiple updates are performed simultaneously. Pipelining improves *throughput*: the rate at which multiple updates can be performed. Given a computation that takes time t, running it five times in sequence would take time $5t$. With pipelining, each of the five runs can overlap in time, making $5t$ the worst case scenario. Note that a single update still takes the same amount of time, but since they can occur simultaneously *multiple* updates can be processed in less time.

Pipelining is most effective when a computation can be broken up into many equally sized chunks. This maximizes the number of computations that can be running at the same time. If one chunk takes longer than the others it becomes a bottleneck. These bottlenecks can be removed with `async` primitive, further improving throughput.

Optimizations in Elm – such as graph rewrites or algebraic rewrites of the intermediate language – must consider parallelism and pipelining. Arrowized FRP was able to achieve a speed-up of two orders of magnitude by collapsing all of their signal functions – the equivalent of nodes – into a single monolithic node [23]. As we have seen, this does not allow any parallelism or pipelining, so this approach is not appropriate for Elm. Elm must strike a balance between reducing the number of nodes and taking advantage of parallelism and pipelining. Finding this balance is a topic for future work.

5 Functional GUIs with Elm

Elm is a declarative language for *graphical* user interfaces. Elm uses a declarative approach to graphical layout, allowing the programmer to say *what* they want to display, without getting bogged down in exactly *how* this should be done. Elm has two major categories of graphical primitives: elements and forms. These categories attempt to balance simplicity and flexibility, making it as easy as possible for a programmer to turn their *idea* for a display into an *actual* display.

Elm's basic visual building block is the `Element`, 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. Elm's elements are also designed to promote a clean separation between data and data presentation. Frameworks such as HTML conflate data and data presentation, forcing you to directly annotate data with how it should be displayed. By combining these two distinct aspects of a multimedia display, HTML becomes frail; it is difficult to add more data *and* to change how the data is displayed because each is dependent on the other.

A general framework for multimedia displays should not be limited to rectangles. Elm's second visual building block is the `Form`, which allows irregular shapes and text. Forms permit lines and irregular polygons to be displayed in a non-structured way. Forms can be moved, rotated, and scaled. They can overlap each other. Forms allow much greater freedom than Elm's rectangular elements, providing the ability to create richer and more complicated layouts when necessary.

Elements and forms are designed to work naturally with the functional reactive paradigm, cleanly interactive with Elm's reactive primitives. This chapter introduces all of the basics of GUI programming in Elm, using an example-driven approach. New functions will be accompanied by real examples of their use. You can follow along with the interactive code editor at elm-lang.org. There you will find a fully functional code editor and compiler, so you can get started with only a browser.

5.1 The Elements of Data Display

Elm's basic unit of presentation is the `Element`. These design elements also inspired the name of this language. For ease of composition, all Elements are rectangles. Primitive elements include text, images, and video.

```
text, image, video :: String -> Element
```

```
main = text "Hello, World!"
```

Hello, World!

Figure 7: Hello, World!

```
main = flow right [ video "bear.ogg", image "elm.jpg" ]
```

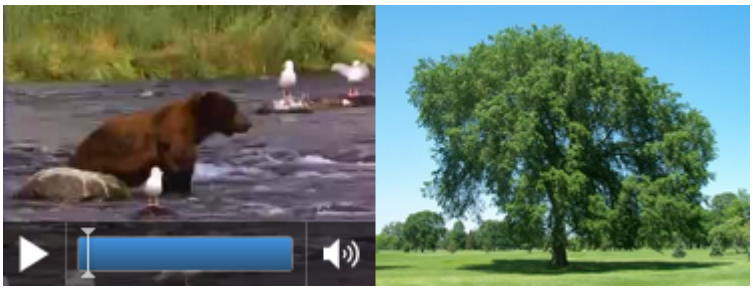


Figure 8: Element Layout

With the `text` function, we can already write our first real program in Elm, shown in Figure 7. The value of `main` is used as the screen output. In this case, we output the text "Hello, World!". Note that each of these primitives is rectangular, making them easy to combine. Elm has one major way to combine primitive elements: the `flow` function.

```
            flow :: Direction -> [Element] -> Element
       up, down :: Direction
    left, right :: Direction
inward, outward :: Direction
```

The `flow` function allows the creation of complicated layouts, as seen in Figure 8 which shows a video and an image. The `flow` function can define a number of useful composition functions.

```
above  e1 e2 = flow down  [ e1, e2 ]
below  e1 e2 = flow up    [ e1, e2 ]
beside e1 e2 = flow right [ e1, e2 ]
layer  elems = flow inward elems
```

Although not strictly necessary, these derived functions are sometimes more natural than `flow`. For instance, the layout code from Figure 8 – showing a

```
people = [ ( "Alonzo Church" , "church.jpg" )
         , ( "Kurt Godel"    , "godel.jpg"  )
         , ( "Alan Turing"   , "turing.jpg" )
         ]

display (name,pic) = image pic `above` text name
main = flow right (map display people)
```

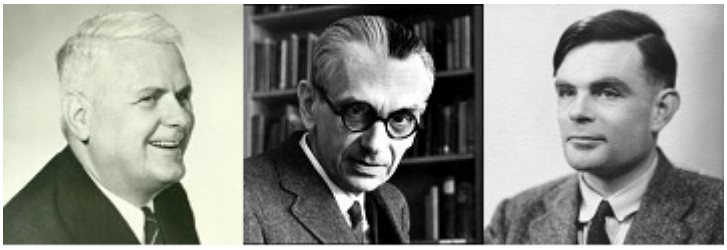


Figure 9: Combining Elements

video and image, side-by-side – can also be defined in terms of `beside`.

```
main = video "bear.ogg" `beside` image "elm.jpg"
```

A normal function quoted with grave accents can be used in the infix position, allowing a more natural reading of some functions.

In addition to creating and composing `Elements`, we need to be able to resize elements.

```
width, height :: Int -> Element -> Element
         size :: Int -> Int -> Element -> Element
```

This small set of primitives accounts for most common layouts. Let's look at these primitives in a more complicated example. Given a list of people, this example – shown in Figure 9 – displays their name and photo. In this example `people` is a variable bound to a list of pairs. The `display` variable is a function from pairs to elements, displaying a person's image above their name. The `map` function applies `display` to all of the pairs in `people`, resulting in a list of elements which can be displayed with `flow`.

The example in Figure 9 illustrates two of our goals. (1) It demonstrates the natural creation and composition of standard GUI elements. The `display` function is fairly straightforward, placing an image above a name. (2) More importantly, this example cleanly separates data from display. We

could add or remove tuples to `people` without touching `display`. Alternatively, we could completely change the display code without changing any data.

Elm's separation of data and display also make it easy to handle dynamic data. If the list of people was instead provided by a server call, our display code would not need to change drastically. We would just need to lift the code to act on a signal of people:

```
main = lift (flow right . map display) peopleSignal
```

This simple example would be frighteningly complicated with pure HTML and CSS. Note that the period signifies function composition, so `(f . g)` is the same as `(λx → f (g x))`.

5.2 Irregular Forms and Complex Composition

Rectangles are boring. What about triangles and pentagons? These shapes do not compose as naturally as rectangles, so they do not make good layout primitives. Nonetheless, they are important and useful.

To supplement the purely rectangular world of `Element`s, Elm has a more flexible set of layout primitives called *forms*. A form is an arbitrary 2D shape enhanced by texture and color. This includes lines, shapes, text, and images. There are currently two ways to create a form: with lines and with shapes. Forms can also be moved, rotated, and scaled.

A `collage` gives you the ability to combine a variety of 2D forms in an unstructured way. Every collage has a width, height, and list of forms.

```
collage :: Int -> Int -> [Form] -> Element
```

This defines both the available space and the forms that fill it. Furthermore, it ensures that an `Element` is still a well-behaved rectangle, even if its constituents are not.

5.2.1 Lines

Lines are the simplest sub-form, defined as a sequence of points. After creating a line, it can be turned into a form by adding color and visual style. As suggested by the function names, it is possible to create solid, dotted, and dashed lines.

```
                    line :: [(Int,Int)] -> Line
solid, dotted, dashed :: Color -> Line -> Form
```

```
zigzag = line [ (10,10), (20,20), (30,10), (40,20) ]
main = collage 50 30 [ solid red zigzag ]
```

Figure 10: Lines

These functions were named to allow very natural collage specifications. Consider the extremely small collage shown in Figure 10. The zigzag line is displayed in a declarative style that reads very naturally. The line is hard-coded in this example, but it need not be. Lines can be created with the full power of Elm, making it easier to create complicated lines and shapes.

5.2.2 Shapes

Shapes are a high-level way to think about forms in a collage. Rather than thinking on a pixel-by-pixel level, shapes give you an abstract entity that can be moved and manipulated. The existing shape primitives are as follows:

```
          polygon :: [(Int,Int)] -> Shape
rect, oval, ngon :: Int -> Int -> (Int, Int) -> Shape
filled, outlined :: Color -> Shape -> Form
   customOutline :: [Int] -> Color -> Shape -> Form
```

With `polygon`, shapes can be defined as a sequence of points. This differs from the `line` function in that a polygon is always closed, connecting the first and last point in the given sequence. This function allows the creation of any two-dimensional polygon, regular or irregular, concave or convex.

Although `shape` is extremely flexible, it can become tedious to define all shapes that way. Therefore, a number of common cases have special purpose functions. `rect` and `oval` create rectangles and ovals when given a width, height, and midpoint. The `ngon` function produces a regular polygon when given a number of edges, a radius, and a midpoint. Here radius represents the distance from midpoint to vertex.

Just as with a `Line`, a `Shape` becomes a form when given a color and visual style. The possible visual styles are exactly as their names suggest: `filled` for filled in shapes, `outlined` for a solid outline, and `customOutline` for arbitrary outlining patterns.

With this relatively small set of primitives, it is easy to create complex yet readable compositions. In the example in Figure 11, we first create a square, circle, and pentagon and then compose them together in a collage.

```
square   = rect 130 130 (70,70)
circle   = oval 110 110 (70,70)
pentagon = ngon  5   45 (70,70)

main = collage 140 140
         [ filled green pentagon
         , outlined black square
         , customOutline [8,4] blue circle
         ]
```

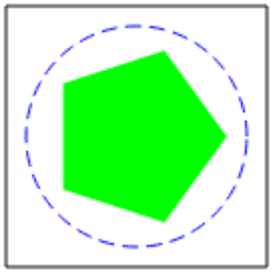


Figure 11: Creating and Combining Shapes

Just as with the `Line` example, creating complex irregular scenes is quite straightforward and natural.

5.2.3 Move, Rotate, and Scale

A set of standard transformations can be applied to any `Form`:

```
                move :: Int -> Int -> Form -> Form
       rotate, scale :: Int -> Form -> Form
```

With these transformations, forms can be arbitrarily moved, rotated, and scaled. This makes it possible to create abstract transformations and animations that act on any kind of form.

More advanced `Form` manipulation is possible, but not yet implemented. Elm's shape primitives open the door for complex shape creation with functions like `union` and `intersection` which could combine existing shapes or check for collisions. This would be quite useful, but calculating the union and intersection of arbitrary shapes is a notoriously difficult and costly problem in general. Although possible, it has not made it into Elm yet.

5.3 Reactive GUIs

So far we have only seen static values. These programs are extremely efficient but make for a rather uninteresting user interface. We'll now introduce some basic, event-driven signals:

Mouse Signals: For mouse coordinates we have `Mouse.x` and `Mouse.y`, both integer signals. We also have `Mouse.position`, a tuple of both mouse coordinates. To monitor the left button we have two boolean signals: `Mouse.isDown` which is true when the left button is pressed down and `Mouse.isClicked` which is true very briefly following each click.

Keyboard Signals: `Keyboard.keysDown` carries a list of currently pressed keys and `Keyboard.keyPress` carries a key option that briefly holds the value of each pressed key.

Window Signals: `Window.width` and `Window.height` provide the dimensions of window. `Window.dimensions` combines these two values into a single tuple.

Time Signals: `Time.before` and `Time.after` take a time and return a boolean signal indicating whether it is before or after that time. `Time.every` takes a time t. The resulting signal starts at zero and, every t seconds, it is updated to the time since the program began. This function permits time-indexed animations and allows update rates to be tuned by the programmer.

Input Signals: This includes standard input elements like text boxes, buttons, and sliders. All input signals are paired with an input element that can be displayed on the screen. Elm currently supports `Input.textField` and `Input.password`. Both take a string and produce a paired element and signal of strings. The string appears as greyed out text when the box is empty.

These signals can be quite useful in specifying GUIs. We already saw some simple reactive examples in Chapter 3, so we will now look at a more complex scenario. The example in Figure 12 displays a single image of a set. The user can cycle through all of the available images by pressing `'f'` and `'b'` which stand for forward and back. In the given diagram, the arrows indicate possible transitions between portraits, not actual graphical elements. The code naturally divides into a reactive section and a display section. If we

```
update key index =
    case key of { Just 'f' -> index + 1
                ; Just 'b' -> index - 1
                ; _        -> index }

index = foldp update 0 Keyboard.keyPress

pics = [ "church.jpg", "godel.jpg", "turing.jpg" ]
display i = image $ ith (i `mod` length pics) pics

main = lift display index
```

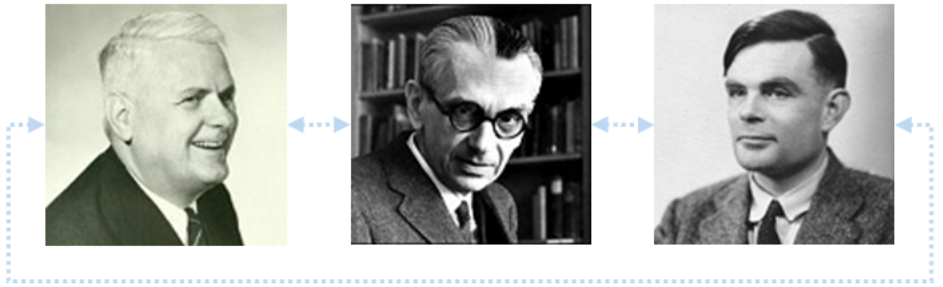


Figure 12: A Simple Slide-Show: Reacting to User Input

ever want to change how the user switches between images, we just need to change `index`. It could just as easily be replaced with a signal that counts mouse clicks or increments after a given time interval.

5.4 The Benefits of Functional GUIs

As our examples have illustrated, Elm can represent fairly complex interactions with a small amount of code. The signal abstraction encourages a separation between reactive code and display code. The display code itself lends itself to a clean separation of data and data presentation. GUI programmers strive for these separations, but imperative GUI frameworks test their resolve. In Elm, divisions between data code, display code, and user interaction code arise fairly naturally, helping programmers write robust GUI code.

6 Implementing Elm

Elm targets the most widespread GUI platform – the web. Targeting HTML, CSS, and JavaScript has many practical benefits:

- Elm can already run in any browser that supports modern web standards. This enables wide device support without an unwieldy backend.
- Browser-incompatibilities are no longer a problem for developers, only for compiler writers.
- Elm circumvents JavaScript's notoriously small standard library.
- The web's low-level API's can be encapsulated in more pleasant abstractions. For example, `collage` abstracts away the low-level imperative canvas interface. `Element`s abstract away many of the unpleasant aspects of composition and styling that come with HTML and CSS, such as the surprisingly difficult task of centering elements.

Elm's current implementation achieves all of these benefits. But targeting the web comes with a number of important downsides that have not yet been adequately addressed in this implementation. JavaScript has the following limitations:

- Tail-call optimization is generally unsupported today. This may change in the next revision of JavaScript (EcmaScript version 6, optimistically nicknamed Harmony).
- Concurrency support is limited. JavaScript supports "workers", which are independent threads of execution that can communicate only through message passing. This sounds extremely promising, but there are severe limitations on the kinds of values that can be passed. Currently, functions *cannot* be passed between workers; only primitive values such as numbers, strings, arrays, and records are allowed.

  Furthermore, workers are heavy-weight threads. They create OS-level threads with a large memory footprint, very different from threads in Concurrent ML. Therefore, it is not feasible to map nodes onto workers. Thread pooling would bring the cost of this strategy down, making it a practical option.

Neither of these limitations are fatal to Elm. In fact, the tail-call optimization problem is widely acknowledged by the JavaScript community, and

many source level solutions have been proposed. A compile-time optimization should be able to avoid this problem.

JavaScript's limited concurrency support is more troubling. Elm must work around the fact that JavaScript's "workers" use heavy-weight threads that cannot pass functions. If Elm avoids workers entirely, thread pooling would provide the benefits of `async`, but *not* parallelism. Elm could be more flexible though, conditionally allowing nodes to appear in a worker. If a node does not take functions as input or produce functions as output, it can be placed in a worker.

Alternatively, the concurrency problem can be "solved" with an extremely unpleasant hack. Elm could be compiled to a sublanguage that avoids variable capture [2]. From there, JavaScript's `toString` and `eval` functions could be used to transmit functions as strings. In addition to being quite infrastructure heavy, this would also provide an opportunity to compromise security. This option does not appear to be worthwhile.

Those are the future difficulties, but almost all of the code presented in this thesis already runs in the current implementation of Elm. The only exception is the keyboard signals. Fortunately, this feature does not pose any fundamental issues, so the addition of keyboard signals will be time-consuming but straightforward.

The Elm compiler itself is written entirely in Haskell. Haskell provides many useful libraries that make this a good choice. Many parsing libraries exist, making it *relatively* pleasant to create a lexer and parser. The current version of the compiler does some mild optimizations – such as constant propagation – before generating JavaScript. The runtime system for the generated JavaScript is a combination of HTML, CSS, and JavaScript that handles events and screen updates.

Elm's website is written primarily in Elm and served using Haskell. Excluding the code editor, the entire website is written in Elm. Haskell was chosen as the server framework because it was already the language used to write the compiler. This allows the server and compiler to interact very cleanly. Although this choice was initially made for convenience, it has become an important part of writing code in Elm. Rather than compile an Elm project and serve that code with some separate framework, the combination compiler/server can compile and serve Elm files automatically. When an Elm file is changed, the compiler/server automatically starts serving the new version, making the changes immediately viewable in a web browser.

7 Conclusion

Elm is a declarative approach to graphical user interfaces. Concurrent FRP allows programmers to specify user interactions on a very high level. Concurrent FRP also addresses some of FRP's long-standing efficiency problems: needless recomputation and global delays. An Elm programmer does not need to worry about including long computations or discrete inputs in their program. Furthermore, Elm's declarative graphics libraries make it simple to create complex multimedia displays. Together, Concurrent FRP and declarative graphics libraries simplify the creation of rich graphical user interfaces.

Elm also promotes an approach to functional GUIs that cleanly separates data from data presentation, and display code from reactive code. These natural separations make it easier to write modular and maintainable code.

Concurrent FRP creates many areas for future work:

- Exploring the connections between Elm and Arrowized FRP. A discrete version of AFRP would almost certainly map onto a concurrent system as described in this work. Since Elm compiles for the web, AFRP can also be adapted to run in browsers. Arrows could also be used to test and modify Elm signals.

- Creating events. It may be possible for events to trigger new events in a controlled way, allowing more robust signal creation. This may allow event-driven FRP to replicate the functionality of continuous signals.

- Filtering events. Elm notably leaves out a primitive that could filter out event updates. Because `foldp` updates on every incoming event, it might be nice to have a way to drop events entirely. This quickly requires tough design choices because, if introduced naively, such a primitive could lead to undefined signals. As presented, Elm signals are *always* defined, ruling out this problem. Is there a nice way to include this functionality without introducing undefined signals? Would `filter` for signals need a default value?

- An exploration of algebraic reductions of Elm's intermediate language. When does reduction help? Is it useful to create more nodes than specified by the programmer? How is concurrency best utilized?

- Elm's synchronization mechanisms currently require many "no change" messages. A more efficient solution might determine each node's update dependencies and pass the "no change" message directly to an

internal node, skipping over the many intermediate nodes that would have just forwarded the message along.

- An exploration of the connection between FRP and Umut Acar's self-adjusting code [1, 4]. These two paradigms appear to have much in common. Nodes that are updated as outlined by Acar are likely to be much more efficient.

I hope that my contributions and questions will help push FRP towards an efficient and expressive implementation. As a practical language, Elm creates the opportunity for many projects that focus on common day-to-day tasks such as a typed data-protocol between Elm and Haskell. From a research perspective, Elm provides a potential platform for future FRP work that is unencumbered by Haskell's design choices. I would welcome contributions or collaborations on either front.

References

[1] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Adaptive functional programming. *ACM Trans. Program. Lang. Syst.*, 28:990–1034, November 2006.

[2] Andrew Appel. *Compiling with continuations.* Cambridge University Press, 1992.

[3] Kimberley Burchett, Gregory H. Cooper, and Shriram Krishnamurthi. Lowering: A static optimization technique for transparent functional reactivity. In *In ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 71–80. ACM Press, 2007.

[4] Yan Chen, Joshua Dunfield, Matthew A. Hammer, and Umut A. Acar. Implicit self-adjusting computation for purely functional programs. In *Proceedings of the 16th ACM SIGPLAN international conference on Functional programming*, ICFP '11, pages 129–141, New York, NY, USA, 2011. ACM.

[5] Antony Courtney. Frappé: Functional reactive programming in java. In *Proceedings of the Third International Symposium on Practical Aspects of Declarative Languages*, PADL '01, pages 29–44, London, UK, 2001. Springer-Verlag.

[6] Antony Courtney and Conal Elliott. Genuinely functional user interfaces. In *In Proceedings of the 2001 Haskell Workshop*, pages 41–69, 2001.

[7] Antony Courtney, Henrik Nilsson, and John Peterson. The yampa arcade. In *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, Haskell '03, pages 7–18, New York, NY, USA, 2003. ACM.

[8] Dustin Deboer and Alley Stoughton. On the future of exene, 2005.

[9] Conal Elliott. Push-pull functional reactive programming. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, Haskell '09, pages 25–36, New York, NY, USA, 2009. ACM.

[10] Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming*, ICFP '97, pages 263–273, New York, NY, USA, 1997. ACM.

[11] Emden R. Gansner and John H. Reppy. A foundation for programming environments. In *Proceedings of the second ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments*, SDE 2, pages 218–227, New York, NY, USA, 1987. ACM.

[12] Emden R. Gansner and John H. Reppy. exene, 1991.

[13] Emden R. Gansner and John H. Reppy. A foundation for user interface construction. In Brad A. Myers, editor, *Languages for Developing User Interfaces*, pages 239–260. Bartlett, 1992.

[14] Emden R. Gansner and John H. Reppy. The exene library manual (version 0.4), 1993.

[15] Emden R. Gansner and John H. Reppy. *A multi-threaded higher-order user interface toolkit*, pages 61–80. John Wiley & Sons, Inc., New York, NY, USA, 1993.

[16] George Giorgidze and Henrik Nilsson. Switched-on yampa: declarative programming of modular synthesizers. In *Proceedings of the 10th international conference on Practical aspects of declarative languages*, PADL'08, pages 282–298, Berlin, Heidelberg, 2008. Springer-Verlag.

[17] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical report, Ithaca, NY, USA, 1994.

[18] Paul Hudak. Modular domain specific languages and tools. In *in Proceedings of Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society Press, 1998.

[19] John Hughes. Generalising monads to arrows. *Sci. Comput. Program.*, 37:67–111, May 2000.

[20] Gregory Hager Izzet Pembeci, Henrik Nilsson. System presentation – functional reactive robotics: An exercise in principled integration of domain-specific languages, 2002.

[21] Neelakantan R. Krishnaswami and Nick Benton. Ultrametric semantics of reactive programs, 2011.

[22] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21:558–565, July 1978.

[23] Hai Liu, Eric Cheng, and Paul Hudak. Causal commutative arrows and their optimization. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, pages 35–46, New York, NY, USA, 2009. ACM.

[24] Hai Liu and Paul Hudak. Plugging a space leak with an arrow. *Electron. Notes Theor. Comput. Sci.*, 193:29–45, November 2007.

[25] Conor Mcbride and Ross Paterson. Applicative programming with effects. *J. Funct. Program.*, 18:1–13, January 2008.

[26] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: a programming language for ajax applications. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 1–20, New York, NY, USA, 2009. ACM.

[27] Henrik Nilsson. Dynamic optimization for functional reactive programming using generalized algebraic data types. In *Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, ICFP '05, pages 54–65, New York, NY, USA, 2005. ACM.

[28] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, Haskell '02, pages 51–64, New York, NY, USA, 2002. ACM.

[29] Ross Paterson. A new notation for arrows. In *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, ICFP '01, pages 229–240, New York, NY, USA, 2001. ACM.

[30] John H. Reppy. *Concurrent programming in ML*. Cambridge University Press, 1999.

[31] Meurig Sage. Frantk - a declarative gui language for haskell. In *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, ICFP '00, pages 106–117, New York, NY, USA, 2000. ACM.

[32] Neil Sculthorpe and Henrik Nilsson. Safe functional reactive programming through dependent types. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, pages 23–34, New York, NY, USA, 2009. ACM.

[33] Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 242–252, New York, NY, USA, 2000. ACM.

[34] Zhanyong Wan, Walid Taha, and Paul Hudak. Real-time frp. In *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, ICFP '01, pages 146–156, New York, NY, USA, 2001. ACM.

[35] Zhanyong Wan, Walid Taha, and Paul Hudak. Event-driven frp. In *Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages*, PADL '02, pages 155–172, London, UK, UK, 2002. Springer-Verlag.

[36] Dana N. Xu and Siau-Cheng Khoo. Compiling real time functional reactive programming. In *Proceedings of the ASIAN symposium on Partial evaluation and semantics-based program manipulation*, ASIA-PEPM '02, pages 83–93, New York, NY, USA, 2002. ACM.