Jaakko Pallari

Multithread concurrency in a single thread environment

Master’s Thesis in Information Technology

May 15, 2015

University of Jyväskylä

Department of Mathematical Information Technology

Author: Jaakko Pallari

Contact information: `jaakko.j.pallari@jyu.fi`

Supervisors: Ville Tirronen, and Evan Czaplicki

Title: Multithread concurrency in a single thread environment

Työn nimi: Monisäikeinen yhdenaikaisuus yksisäikeisessä ympäristössä

Project: Master's Thesis

Study line: Software and telecommunication technology

Page count: 66+0

Abstract: There exists a growing need for software applications to be able to work concurrently. To help building concurrent applications, applications can be built in a reactive style. Elm programming language offers a way to build applications in a high-level reactive style, Functional Reactive Programming style. Elm's primary target platform is the web browser, which has limited support for concurrency constructs. Therefore, Elm's support for concurrency is limited, as well. In this thesis, we present a solution for enhancing Elm concurrency by extending the concurrency capabilities in the web browser.

Keywords: Concurrency, monad, Elm, Functional Reactive Programming, Concurrent FRP

Suomenkielinen tiivistelmä: On olemassa kasvava tarve saada sovellukset toimimaan yhdenaikaisesti. Sovellukset voidaan rakentaa noudattamaan reaktiivista tyyliä yhdenaikaisuuden avustamiseksi. Elm ohjelmointikieli tarjoaa keinon rakentaa sovelluksia korkeatasoisella reaktiivisella tyylillä, funktionaalisella reaktiivisella ohjelmointityylillä. Elmin pääasiallinen kohdeympäristö on WWW-selain, jossa on rajoittunut tuki yhdenaikaisille rakenteille. Tästä johtuen myös Elmin tuki yhdenaikaisuudelle on rajoittunut. Tässä tutkielmassa esitämme ratkaisun Elmin yhdenaikaisuuden tehostamiseksi laajentamalla WWW-selainten yhdenaikaisuuskeinoja.

Avainsanat: Yhdenaikaisuus, monadi, Elm, Functional Reactive Programming, Concurrent FRP

List of Figures

Contents

1 Introduction

There exists a growing need for software applications to be able to work concurrently both internally and with other applications. Many applications execute long-running computations, interact with external resources, and involve event driven systems. Instead of attempting to process all of the tasks in sequence, applications should have the ability to set the tasks to be executed when they have the resources they need. For example, an application should be able to have database queries and HTTP requests running in the background while the rest of the application processes UI events. This allows tasks to run concurrent to each other, which can increase application's efficiency at using the available resources to perform its purpose.

Applications can be built in reactive style to help interacting with concurrent tasks. Instead of waiting for results to become available, an application reacts to finished tasks. One way to achieve this is to set up a callback function that is to called when the task is finished.

An alternative way for building reactive applications is to use a higher-level abstraction such as Functional Reactive Programming (FRP). In FRP, interactions are modeled as time-varying values, signals, instead of manually wiring callbacks to tasks. Signals can be composed with pure functions to create new signals. Changes in one signal cascade to every signal that connects to it.

Elm is a programming language that embraces the FRP concept. Elm is an ML-like, pure functional, and type-safe language. The primary target of Elm's compiler is the web browser. The compiler compiles Elm source code to HTML, CSS, and JavaScript in order to make the programs executable in web browsers.

Because Elm's compiler targets the browser, the Elm programs inherit many of the challenges that JavaScript has. Even with the current advances in browser technology and standards, most browsers still lack the concurrency constructs that allow executing computations concurrently. Therefore, Elm currently has only partial support for concurrency.

In this thesis, we present a solution that can be used for enhancing Elm's concurrency capabilities. Our approach is to first determine the requirements for enhancing the concurrency, and then produce a solution that's compatible with the requirements and the browser environment. Instead of attempting to produce a full solution that could be immediately integrated to Elm, our goals is to produce a proof of concept example. The solution we provide in this thesis can either be expanded to a complete solution or used as a reference for future implementations.

The thesis is structured as follows. In the second chapter, we present the work related to this thesis. In the third chapter, we present some of the core concepts that are used as part of the solution presented in this thesis. In the fourth chapter, we explain the basics of FRP and the core ideas of Elm's FRP semantics, and discuss about the concurrency constructs that are useful to Elm's FRP system. In the fifth chapter, we present a concurrent, single-threaded FRP system based on the ideas presented in the third chapter. In the sixth chapter, we translate the core ideas from the fifth chapter for the browser environment. Finally, in the seventh chapter, we present the conclusions and discuss about how the research can be continued in future works.

2 Related work

This chapter introduces the background work that is related to this research. The found articles are grouped into research topics. Each section in this chapter covers one topic. The topics are not presented in any particular order. The topics covered in this chapter are:

- Functional Reactive Programming
- Continuations and continuation based concurrency
- Alternative methods for delivering concurrency in a single-thread
- Concurrency and FRP in the browser

Three different search methods were used for finding relevant literature. Keyword search is a method where literature is searched using particular keywords related to the research subject. Backward search is a method where new literature is pulled from references of previously known literature. Forward search is a method where new literature is found by locating articles that have references to previously known literature. Both backward and forward search methods include a step where the other works of found authors are searched for relevant literature. (Levy and Ellis 2006, p. 190–192) During this research, keyword search was used for finding the first articles, while forward and backward search were used for finding more related works out of the first found articles. The main sources for finding literature was Google Scholar. ACM Digital Library and IEEE Xplore were used as secondary sources of literature.

2.1 Functional Reactive Programming

Functional Reactive Programming (FRP) is one of the key topics behind the Elm programming language. There exists many variations and implementations of FRP, which are covered in this section.

Elliott and Hudak (1997) first introduced FRP in their article “Functional Reactive Animation”, where they explore the idea of using a declarative programming paradigm for composing interactive, multimedia animations. They described the formal semantics for the key components of the FRP system, behaviors and events, as well showed how they are implemented in the Haskell library Fran. Behaviors are first-class time-varying values that can be composed from other behaviors and events. The values of each behavior is updated automatically when the dependencies update. Events are representations of real world events as first-class values. Like behaviors, events can also be composed to create new events. Wan and Hudak (2000) continued refining the formal semantics of FRP in their article "Functional Reactive Programming from First Principles".

In the article, "Parallel Functional Reactive Programming", Peterson, Trifonov, and Serjantov (2000) demonstrated how FRP can be extended to encompass parallel systems. They extended FRP by expanding the FRP semantics with new functions and defining transformation rules for parallelism. Their implementation of parallel FRP is based on an existing Haskell parallel programming infrastructure, Linda.

Wan, Taha, and Hudak (2001) introduced the concept of Real-Time Functional Reactive Programming (RT-FRP), a variant of FRP where the space cost of each execution step is statically bound, in their article "Real-Time FRP". Using Real-Time FRP, they addressed the problems related to unpredictable resource consumption that usually occur in traditional FRP applications. Their solution is to have a well-defined notion of cost in the operational semantics in the RT-FRP system, and have the reactive language be separated from the normal programming language that is used for programming the application logic.

Nilsson, Courtney, and Peterson (2002) explored Arrowized Functional Reactive Programming (AFRP) in their article "Functional Reactive Programming, Continued". AFRP utilizes the arrow combinators for modeling reactive dataflows. Instead of modeling event sources as behaviors, AFRP focuses on the building flows of events as combinations of functions. Nilsson, Courtney, and Peterson (2002) explores the core components of AFRP, a continuation-based AFRP implementation, and examples of AFRP in a non-trivial application domain. They concluded that AFRP lacks the performance problems of the previous FRP implementations, while at the same time offers expressive and simple way to capture complex communication patterns.

In his article, "Monadic Functional Reactive Programming", Ploeg (2014) introduced a monadic way of building FRP applications. He covered how a Monadic FRP system can be built and how applications can be built with it. He concluded that the monadic interface is more natural way of building FRP applications than what the other FRP styles offer.

The key features and the semantics of Elm programming language are covered in the articles "Elm: Concurrent FRP for Functional GUIs" (Czaplicki 2012) and "Asynchronous Functional Reactive Programming for GUIs" (Czaplicki and Chong 2013). Besides presenting the core features of the language, it is pointed out in the articles how concurrency in Elm's runtime should work. The semantics of Elm are designed in a way where each top-level update is propagated to the whole signal graph in order to ensure that all signal values are synchronized. At the same time, signals avoid needless recomputations by tracking their dependencies update changes. The articles contain examples for implementing Elm's FRP constructs in Concurrency ML (Reppy 1993), which supports concurrency in the form of threads and channels.

2.2 Continuations and continuation-based concurrency

Elm's primary target platform, the the web browser, has a very limited support for concurrency. Thus in order to enhance the concurrency in Elm, it might be necessary to find a way to build concurrency on top of the browser platform without the support for additional concurrency constructs such as threads.

Continuation passing style (CPS) can be used for manipulating program flow in arbitrary ways, which makes it possible to implement different kinds of control flow mechanics such as gotos and exception handling. It can be also used for interleaving multiple computations, thus allowing one to build single-threaded concurrency without any concurrency constructs. This is achieved by structuring programs in a way where the next step of a computation is always reached with a function call while never returning a result back to the caller. In CPS, the next steps that a function call can continue to are always given as a parameter to the function.

Multitasking can be delivered using coroutines. The computations that coroutines hold can be interleaved from the points where the coroutines yield their execution to other coroutines, thus creating concurrency in a single-thread. Haynes, Friedman, and Wand (1984) demonstrated how coroutines can be built using continuations and functional abstraction in their article “Continuations and Coroutines”. They implemented coroutines as closures which holds the current state of the coroutine computation. When the coroutine computation yields its execution, the state is captured and stored in the closure’s inner state. Coroutines can then be continued later by calling the stored state. Haynes, Friedman, and Wand (1984) showed that it is possible to build coroutine-based concurrency without a direct support for coroutines in a programming language.

Engines are computations which can be executed a limited period of time before their execution is yielded for another engine. If the engine is able to complete the computation before the time runs out, the result is returned. Otherwise a new engine is returned, which will continue the previously interrupted computation. Haynes and Friedman (1984) The time component can be either be a counter that is counted down as the computation is advanced or an interrupt based timer. In the article “Engines from Continuations”, Dybvig and Hieb (1989) demonstrated how engine-based concurrency can be built with the help of continuations. The engine results are returned back to the executor by calling the engine’s return function which passes the return value back to the original caller through the caller’s continuation. Dybvig and Hieb (1989) also showed how engines can be nested inside eachother.

Hieb, Dybvig, and Anderson (1994) presented subcontinuations, a type of a continuation built for concurrency in mind, as an alternative for traditional continuations in their article “Subcontinuations”. Unlike traditional continuations, subcontinuations allows programs to request continuations back to any given point. The subcontinuation operator *spawn* can be used to establish a return point for the body that is passed to it. The body is given a continuation which the body can call to escape back to return point. The return point will be replaced with the value passed to the continuation. Using the combination of subcontinuations and Lisp’s *pcall*, a procedure for launching parallel computations, Hieb, Dybvig, and Anderson (1994) delivered tree-structured concurrency, where diverging computations return to their parent computation. They also demonstrated how engines can be implemented using subcontinuations. Kumar, Bruggeman, and Dybvig (1998) showed how subcontinuations can be implemented with the help of thread primitivies to utilize multithreaded environments.

In order to take advantage of continuation control flow mechanisms, the programs must be written in continuation-passing style. However, writing programs in continuation-passing style is more counterintuitive compared to traditional style of programming, the direct style (DS). This is because functions in CPS cannot return values, and must pass the return value to a callback function instead. In order to utilize the power of continuations while still keeping the traditional programming style, an automatic conversion from direct style to CPS could be attempted. Flanagan et al. (1993) explored the ideas behind transforming higher-order languages into CPS in their article "The Essence of Compiling with Continuations". Using a simplified version of the Scheme language, they showed the general rules for transforming programs written in direct style into continuation-passing style. The size of the program generated by the basic transformation strategy increases considerably, but it can be reduced with an additional simplification phase that removes the expressions unneccessary to the final program. Flanagan et al. (1993) introduced A-reductions, which replaces the continuations that only deliver context to inner computations with more efficient *let*-expressions. According to them, the resulting form, the A-Normal form, is a good intermediate representation for compilers.

2.3 Alternative methods for delivering concurrency in a single-thread

As mentioned earlier, there is a need for finding a way to provide concurrency in the browser platform in order to enhance concurrency in Elm. Continuation passing style offers a framework for building various kinds of concurrency constructs in a single-thread system. However, there exists other methods that don't rely on writing programs in CPS.

In the article "Trampolined style", Ganz, Friedman, and Wand (1999) introduced a programming style where computations are split into steps where one step produces the next steps of the rest of the computation. The difference between CPS and trampolined style is that in CPS the next step is always reached by calling a function, while in trampolined style the next computation is returned to the caller of the previous computation. These computations are executed step-by-step using a custom scheduler called the trampoline. Ganz, Friedman, and Wand (1999) showed how interleaving can achieved by alternating executions between multiple trampolined functions. They also demonstrated how programs written in tail form can be transformed into trampolined style.

Another way to encapsulate interleavable computations is to use monads. A data type is a monad if it follows the monadic interface for composing the instances of the data type to create new instances. Monad only defines the interface and laws on which the underlying data type has to operate, and it is up the data type to define the characteristics of the computation and behavior of the composition. One of the benefits of monads is that they offer a generic way of composing computations while hiding the details that go into composing computations, thus allowing the programmer to focus on developing programs at a higher level than what the computation composition occurs on.

In the article "A Poor Man's Concurrency Monad", Claessen (1999) demonstrated how concurrency can be provided using a coroutine monad that simulates multi-threading. The computations can be split into multiple execution threads in the context of the monad with a special fork command, which works similar to how Haskell's own native fork command works. At its core, the monad uses continuations to compose the computations and requires a special scheduler for executing and interleaving the built computations. The monad shown in the article also allows adding concurrency to existing monads. The monad augments the target monad with concurrency capabilities while still allowing the use of monadic interface for composing computations. Thus features such as data synchronization can be achieved by combining the coroutine monad with a monad that provides those capabilities.

In the article "Cheap (But Functional) Threads", Harrison and Procter (2005) demonstrated how concurrency can be built using a resumption monad, a monad that specializes in providing multitasking using resumptions. They show the ideas behind the basic and reactive formulations of the resumption monad. The basic resumption monad is used for building interleavable computations, while the reactive resumption monad also includes the notion of request-and-response interaction. They also showed ways resumption-based systems can be reasoned with, and demonstrate the use of resumption monads by building an example operating system kernel around them.

In his web article "From zero to cooperative threads in 33 lines of Haskell code", Gonzales (2013) demonstrated how free monads can be used for creating a coroutine monads. The free monad coroutine is similar to the CPS coroutine monad in that both allow simulating multithreading by composing interleavable computations using the monad interface, and both include similar semantics for launching new execution threads in the context of the monad. Instead of using continuations, the free monad coroutines are build out of free monad trees with a custom type used for directing the flow of the computation.

In the article "Beauty in the Beast", Swierstra and Altenkirch (2007) provided functional semantics for three common impure components: teletype IO, mutable state, and concurrency. Their approach for modeling concurrency is to use Haskell's existing concurrency library as an example, and create a similar API using custom data structures. Their concurrency API also simulates multithreading by interleaving parallel computations and supports starting new execution threads between computations.

2.4 Concurrency and FRP in the browser

The browser platform has a very scarce support for concurrency. Until the HTML5 specification, the browser platform has officially supported only event driven style of programming. HTML5 introduced Web Workers as a tool for multitasking. Like threads, Web Workers run code parallel to the the main execution thread. Unlike threads, Web Workers can only execute code from a document independent of the thread that starts the worker, and they can only communicate back and forth using string messages. (Consortium et al. 2010)

Maki and Iwasaki (2007) tackled the issue of multithreading in JavaScript in their article "JavaScript Multithread Framework for Asynchronous Processing". They proposed preemptive scheduled multithreading library for building concurrency into browser applications. The library allows applications to be programmed in traditional multithreading style instead of event driven style. The library is written in JavaScript, thus it preserves portability across different browsers. However, Maki and Iwasaki (2007) encountered significant overhead in using the library and hoped that the implementation could be optimized further.

Besides Elm, there exists other programming languages that compile to JavaScript. Many of them have their own solutions for building concurrency.

In their article “How to Run your Favorite Language in Web Browsers”, Canou, Chailloux, Vouillon, et al. (2012) showed how OCaml code can be run in the browser by compiling OCaml to JavaScript or interpreting OCaml bytecode using a separate virtual machine. They also presented approaches for implementing concurrency in the browser. The OCaml virtual machine, OBrowser, is able to simulate preemptive threads by distributing execution time between virtual machine threads, and letting each thread execute a part at a time. Another way to deliver concurrency is to use an existing OCaml concurrency library on top of the virtual machine.

Yoo and Krishnamurthi (2013) presented Whalesong, a compiler that compiles Racket programs into JavaScript, in their article “Whalesong: Running Racket in the Browser”. They saw single-threading as a challenge for enabling pre-emptive and interruptible computations in the browser. To solve this, they implemented subcontinuations for the system, which allows implementing custom concurrency constructs in Racket. Their approach for compiling Racket to JavaScript was to reuse Racket compiler for compiling Racket to a more generic version of the Racket language and compile the generic language into executable JavaScript. The JavaScript implementation of the Racket compiler generates Racket bytecode which is then compiled into JavaScript in order to avoid interpretation costs in the custom virtual machine. The final code can then be executed using a custom trampoline.

Meyerovich et al. (2009) presented Flapjax, a programming language built on JavaScript, in their article “Flapjax: A Programming Language for Ajax Applications”. They demonstrated the structure of Flapjax programs through examples, and briefly talked about Flapjax’s implementation details. Flapjax embraces the FRP model in its framework for building reactive applications. In Flapjax, the browser events and AJAX calls are expressed as composable event streams.

3 Core concepts

In order to understand the ideas presented in this thesis, the concepts that the ideas leverage must be understood first. In this chapter, we present some of the core concepts are used in this thesis.

3.1 Monad and Monad transformer

A monad is a type of a computation that has two operations: a unit operation and a bind operation (Wadler 1992). Monads are represented as data types that have a type parameter for the value they produce. The unit operation can be used for creating new monad values from other values In Haskell, `return` function is the equivalent of the unit operation.

```
return :: Monad m => a -> m a
```

Bind operation can be used for composing monad values. The bind function has two parameters: a monad value and a function that produces a monad value. The bind function takes the produced value from the monad value and applies it to the given function. The result of the bind function is a monad value that produces the same value as what is received from the monad value that the given function produces. In Haskell, the operator $>>=$ is the equivalent of the bind operation.

```
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

One of the benefits of monads is that it allows abstracting complex computation composition behind a flexible, simple interface. Each different monad can encapsulate their own kind of computations, and each have their own implementations for the monad operations. The flexibility of the interface makes it possible to extend the interface with functionality that is available for all monads. The simplicity of the interface makes it easy for custom data types to get access to all of the functionality build around monads.

Monad transformers are monads that can be extended with other monads (Liang, Hudak, and Jones 1995). A monad transformer's type is parametrized with another monad type that the monad transformer encapsulates. The bind function for a monad transformer not only composes the monad transformer values but also the monad values it encapsulates.

3.2 Free monads and free monad transformers

Free is a structure that consists of two kinds of values: pure values and values that produce new free monad values (Swierstra 2008). The types for both the pure value and the value that produces the free monad value are parameters for the free monad. In Haskell, free can be represented with a data type.

```
data Free f a = Pure a | Free (f (Free f a))
```

In order to create a free monad out of the free structure, the type parameter `f` must be a functor (Swierstra 2008). A functor is a data type with a type parameter for the values it contains. Functors have an operation, map, that applies the value from the given functor with the given function, and produces a functor that contains the result from the given function. The implementation of the map operation depends on the functor. In Haskell, the function `fmap` is the equivalent of the map operation.

```
fmap :: Functor f => (a -> b) -> f a -> f b
```

Assuming that the type parameter `f` is a functor, the functor and monad functions for the free structure can be implemented as shown in listing 3.1. In both the functor's map and monad's bind functions, the given function is applied directly to the pure function. If the value is a functor, then the operation is repeated for the value that the functor contains with the help of the functor's map function.

Listing 3.1. Functor and monad instances for free monad.

```
instance Functor f => Functor (Free f) where
  fmap f (Pure x) = Pure (f x)
  fmap f (Free t) = Free (fmap (fmap f) t)

instance Functor f => Monad (Free f) where
  return x = Pure x
  (Pure x) >>= f = f x
  (Free t) >>= f = Free (fmap (>>= f) t)
```

In order to extend free monads with arbitrary monads, the transformer variant of the free structure can be used instead. Listing 3.2 contains a Haskell implementation of free monad transformers.

The benefit of free monads is that it allows extending any functor with monad operations. A developer is only required to implement the functor interface for a custom data type in order to gain monad capabilities for the type. If the developer decides to modify the type later on, only the functor implementation needs to be updated.

Listing 3.2. Free monad transformer implementation based on the `Control.Monad.Trans.Free` module.

```haskell
import Control.Monad

data FreeF f a b = Pure a | Free (f b)

newtype FreeT f m a = FreeT { runFreeT :: m (FreeF f a (FreeT f m a)) }

instance (Functor f, Monad m) => Functor (FreeT f m) where
  fmap f (FreeT m) = FreeT (liftM f' m) where
    f' (Pure a)  = Pure (f a)
    f' (Free as) = Free (fmap (fmap f) as)

instance (Functor f, Monad m) => Monad (FreeT f m) where
  return a = FreeT (return (Pure a))
  FreeT m >>= f = FreeT $ m >>= \v -> case v of
    Pure a -> runFreeT (f a)
    Free w -> return (Free (fmap (>>= f) w))

lift :: (Monad m) => m r -> FreeT f m r
lift = FreeT . liftM Pure

wrap :: (Monad m) => f (FreeT f m a) -> FreeT f m a
wrap = FreeT . return . Free

liftF :: (Functor f, Monad m) => f a -> FreeT f m a
liftF = wrap . fmap return
```

3.3 Continuation Passing Style

In the traditional style of programming, a function call's result is returned once the function call ends. The result of the function call is then accessible for the function caller. In continuation passing style, functions are given another function as an additional parameter. Instead of returning the function result, functions written in CPS call the given function with the result instead. The function given as the parameter is called a *continuation* and it represents the future of the computation.

Compared to the traditional programming style, programming in CPS might seem counterintuitive: functions written in CPS can't be composed by passing the return value as a parameter for the next function. Instead, the outer function becomes the part of the continuation passed to the inner function. However, CPS opens up possibilities for implementing different kinds of control flow mechanisms. For example, CPS can be used for implementing labeled jumps (goto), coroutines (Haynes, Friedman, and Wand 1984), and exception handling.

Functions written in CPS are required to call the given continuation with the result of the function. For example, the CPS versions of addition and division operations could be written in Haskell as shown in listing 3.3. Instead of passing the continuation to the functions, the functions return a computation which can be evaluated by calling it with a continuation. Because of currying in Haskell, writing the functions in either way produces the same result. Here the separated format is used for emphasising the ability suspend the CPS computation. A suspended computation has the type `(a -> r) -> r`.

Listing 3.3. CPS addition, division, and average

```haskell
addCps :: Num a => a -> a -> ((a -> r) -> r)
addCps a b = \k -> k (a + b)

divCps :: Fractional a => a -> a -> ((a -> r) -> r)
divCps a b = \k -> k (a / b)

avgCps :: Fractional a => a -> a -> ((a -> r) -> r)
avgCps a b = \k -> addCps a b (\s -> divCps s 2 k)
```

The listing 3.3 also contains a function for calculating the average of two numbers using the CPS version of addition and division. A pattern can be seen in the composition of the addition and division operation. A continuation is created for the computation (addition of two numbers). This continuation evaluates the second computation with the original continuation as the parameter. This pattern can be used for creating a function that joins a CPS computation with a function that can provide another CPS computation out of the result of the first computation. Listing 3.4 shows how the joining function can be implemented and how it can be used for implementing the average function.

Listing 3.4. Joining of suspended computations

```
cpsJoin :: ((a -> r) -> r)
        -> (a -> ((b -> r) -> r))
        -> ((b -> r) -> r)
cpsJoin s f = \k -> s (\x -> (f x) k)

avgCps :: Fractional a => a -> a -> ((a -> r) -> r)
avgCps a b = cpsJoin (addCps a b) (\s -> divCps s 2)
```

The way `cpsJoin` operates on CPS computations is similar to how `bind` function operates on monadic values. When the CPS computation is expressed as a monadic value, a continuation monad, the bind operation for the data type can be derived from `cpsJoin`. When the result type `r` is replaced with type `m r` where `m` is any monad, the continuation monad can be made into a monad tranformer. Listing 3.5 demonstrates how CPS computations can be implemented as monadic values and transfomers. The implementations for `return` and `bind` follow the same pattern in both the continuation monad and monad tranformer.

Listing 3.5. CPS computation as a monad and a monad transformer

```
newtype Cont  r a   = Cont  { runCont  :: (a ->   r) ->   r }
newtype ContT r m a = ContT { runContT :: (a -> m r) -> m r }

instance Monad (Cont r) where
    return x = Cont (\k -> k x)
    c >>= f = Cont (\k -> runCont c (\x -> runCont (f x) k))

instance (Monad m) => Monad (ContT r m) where
    return x = ContT (\k -> k x)
    c >>= f = ContT (\k -> runContT c (\x -> runContT (f x) k))

instance (Monad m) => MonadTrans (ContT r m) where
    lift m = ContT (m >>=)
```

4 Functional Reactive Programming and Elm

In order to understand how the concurrency of Elm's JavaScript runtime can be improved, it is important to know the basics of FRP and how Elm's FRP system is structured. In this chapter, we first show the basic constructs and reasoning used in common FRP systems. Next, we examine the motivation to have an FRP system work concurrently. In the third and fourth sections, we examine the ideas for structuring Elm's FRP system so that it can execute parts of the system concurrently and asynchronously. In the final section, we briefly examine the current implementation of Elm's JavaScript runtime.

4.1 Basics of FRP

In FRP, time-varying values are expressed as components called **signals**. Each signal contains a value that changes over time, and these changes can be observed and reacted upon. New signals can be created by composing existing signals and pure functions together. The values from existing signals applied to pure functions form the value for the new signal. Changes in signals automatically propagate to observing signals, so the values of each signal are also automatically updated. (Czaplicki 2012) Not only does FRP offer nice abstractions for different ways to combine signals together, but also enhance the composability in the system. Unlike in event driven systems where callbacks are attached to event sources, it could be said that in FRP systems event sources are attached to event sources. Each signal can consume facts delivered by other signals, and also provide the same kind of interface for other signals.

Depending on the FRP system, there are many ways to compose signals, and some of them are shared among systems. Figure 1 shows an example of a FRP signal graph which is composed of common type of signals. In the graph, "mouse position" and "mouse click" are two signals from the environment that are available for the program to attach own signals to. The signal graph ends in "print to screen" signal which displays its latest value on each update.

Probably the most common way to compose signals is to map one signal's values into another signal's values using a given function. Whenever the source signal updates, the target signal's value becomes the source signal's value applied to the function. For example in figure 1 “X coordinate” signal transforms the mouse position value by extracting the X coordinate. An extension of this kind of composition is to map the values of multiple signals into one signal's values. Whenever one of the source signals updates, the target signal will be updated with the updated value and the old values of other signals. In figure 1, “Sum” signal combines the values of “X coordinate” and “count clicks” signals by adding them together.

Signal values can also be made dependent on signal's own previous values. One way to achieve this is to create signals that create their values by applying a given function with the value of a source signal and the existing value. This kind of signal resembles a higher order function that reduces a collection of values to a single value using a given function. Instead of reducing a collection, the signal reduces the values of a source signal. “Count clicks” in figure 1 is an example of a signal that depends on its previous state. Whenever “mouse click” updates, “count clicks” increments its previous value by one and makes the result its new value.

Some signals can also choose to filter updates from source signals on given conditions. For example, a signal can choose to update its own value only when a source signal's value matches with a given predicate. “Greater than 100” in figure 1 updates its value only when the value from “sum” is greater than 100.

4.2 Concurrent FRP

As seen in previous section, FRP applications form graphs of signals. Because changes propagate automatically from a signal to all the signals that are connected to it, one change causes all the signals in its travel path to update. Obviously when signal paths get longer or more signals are branched from existing signals, it takes more time to process one update. In applications with only non-frequently updating signals, the effect might not be that noticeable. However, when frequently updating signals (e.g. “mouse position” signal from figure 1) or slow updating signals (e.g. CPU or IO heavy signals) are introduced to the system, the program might start encountering high latency between updates which may eventually make the

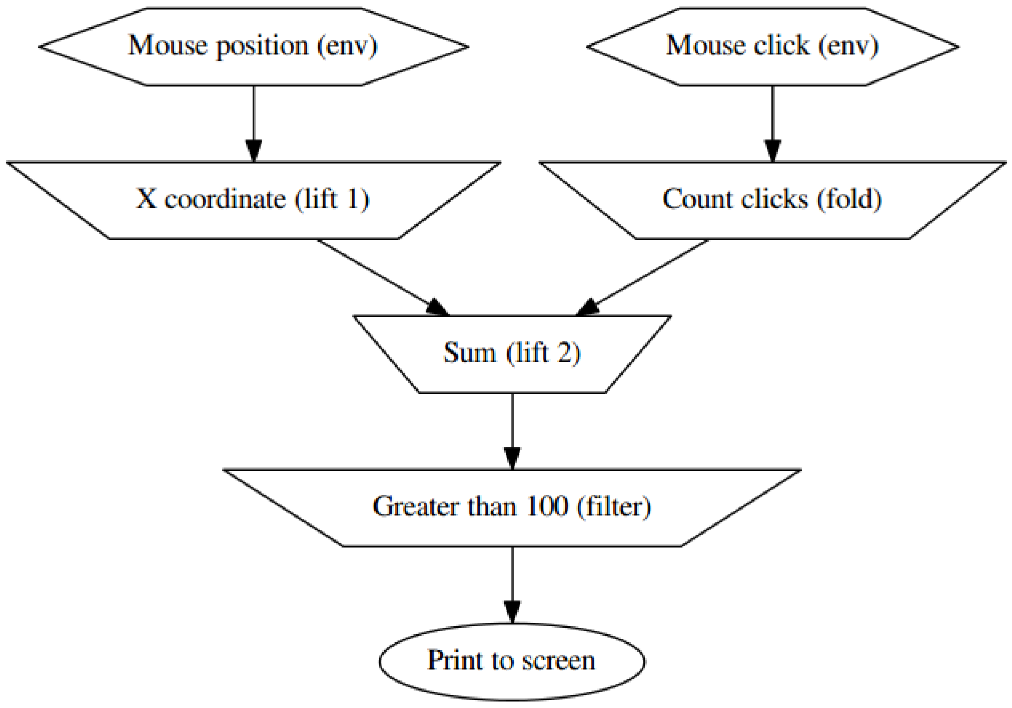

Figure 1. An example of how the signals in an FRP graph are ordered. Each node in the graph is one signal. The arrows indicate the flow of values from signal to signal.

program less responsive and less performant. Typical asynchronous tasks can easily congest a non-concurrent FRP system.

One way to improve responsiveness is to introduce concurrency to the system. There exists several ways to make FRP system concurrent. One way is to implement signals as a concurrent execution thread which reads incoming values from a queue, and updates the signal's own value accordingly (Czaplicki 2012). Another way could be to dispatch some of the value computations to a pool of execution threads where each thread keeps on producing given computations.

Whatever technique is used, making the system concurrent means that the order of events might not remain the same throughout the system. Therefore, concurrency might not work for programs that depend on synchronous event processing. Czaplicki and Chong (2013) used translation of words from English to French as an example of a signal graph where synchronization is required: when the translation of words is done separately from the combination of the original word and the translation, the word combining signals depends on both the original word and translation sources to be in sync. Therefore the translating signal can't be made asynchronous. Essentially this problem affects all the cases where a signal depends on the specific order of incoming values.

On the other hand, applications that need to execute asynchronous tasks also need a concurrent FRP system in order to draw the benefits from FRP. Without concurrency, the asynchronous parts of the application would have to be implemented outside of the FRP system so that the system wouldn't have to wait for asynchronous task to finish. For example if all of the signals are processed synchronously, creating a HTTP request in one signal would halt the whole system until the request is finished.

One of the key benefits of FRP is the the composition of asynchronous tasks and events with signals. Therefore, not having the concurrent capabilities in the system can defeat the purpose of having a FRP system in many cases.

4.3 Elm's FRP system

Elm has all four different kinds of signals described earlier. New signals are created by composing pure functions with existing signals using reactive primitives (Czaplicki and Chong 2013, p. 3). Reactive primitives are higher-order functions that take one or more signals and one or more pure functions as arguments and return a new signal as a result. Reactive primitives are essentially used for deriving new signals out of existing ones. For example, Elm's reactive primitive $lift_2$ can be used to combine the values of two signals into one signal using a 2-ary pure function. The type signature for $lift_2$ is

$$(a \rightarrow b \rightarrow c) \rightarrow Signal\ a \rightarrow Signal\ b \rightarrow Signal\ c$$

where the first parameter is the function used for combining the values of the second and third parameter signals resulting a signal with the type of $Signal\ c$.

In figure 1, the signal "Sum" is of type $Signal\ Int$, a signal that produces integer values. It is composed using $lift_2$ from signals "X coordinate" and "Count clicks" both of which are of type *Signal Int* as well. The combining operator, the sum of two integers, is of type $Int \rightarrow Int \rightarrow Int$. The definition for "Sum" is

$$sumSignal = lift_2\ sumOfTwo\ Xcoordinate\ clickCount$$

and it can be expressed in Elm source code as shown in code listing 4.1 where the first line defines the type for `sumSignal` and the second line defines the form.

Listing 4.1. sumSignal in Elm

```
sumSignal : Signal Int
sumSignal = lift2 (+) Xcoordinate clickCount
```

Besides the common reactive primitives, Elm also has a reactive primitive for deriving asynchronous signals from other signals. By default, all of the signal values are computed synchronously. The values for asynchronous signals are computed without blocking the rest of the system. (Czaplicki and Chong 2013, p. 4–5) Creating an asynchronous signals out of signals that are slow at producing values (for example an IO or CPU dependent signal) allows parts of the FRP system to continue computing new values without waiting for the slow signal to complete computing each of its values. Mixing asynchronous and synchronous signals allows creating responsive FRP applications where long-running computations are seamlessly integrated to the FRP system without losing the order of events in places where it is needed.

Synchronous updates in signals means that each signal has to wait for new values to arrive from all of the signal's source signals before it can compute a new value. An incoming value will always be paired with all of the other latest values from signal's source signals, therefore preventing the signal from "jumping ahead" by computing values from partial set of incoming values. (Czaplicki and Chong 2013, p. 5–6) For example in the signal graph in the figure 2, the signal E can't proceed computing a new value until it has received values from both signal A and C. On the other hand, since signals C and E depend on signal A, and E depends on C, E is guaranteed to receive the input and output values of C exactly at the same time.

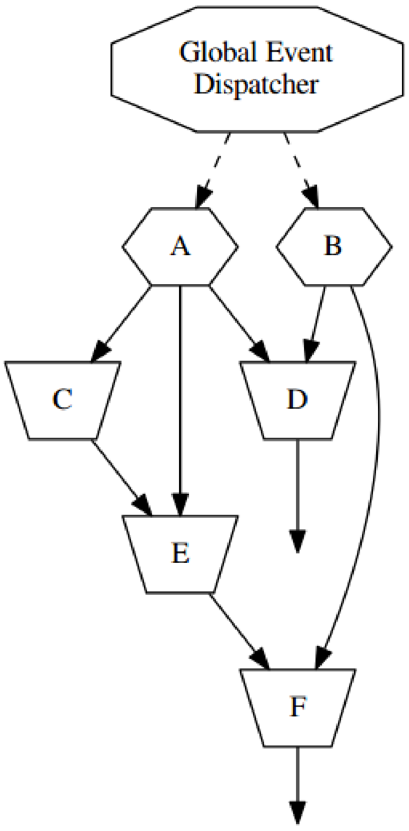

Figure 2. Example of a Elm signal graph where each source node is attached to a global event dispatcher.

Even thou signals are required to wait for all of their source signals to finish producing a new value on each event, parts of the FRP system can still compute new values simultaneously. If signal nodes compute values in concurrent execution threads, signals independent of each other may compute values parallel to each other (Czaplicki and Chong 2013, p. 5). For example in figure 2 if signal C has received a value from signal A, and signal D has received values from both signal A and B, signals C and D may compute their values at the same time. In essence, the ability to compute values in parallel allows having parts of the FRP system working concurrently instead of blocking the whole system while computing values for independent signal branches. It is an important feature to have because it can prevent global delay by allowing parts of the system to work concurrently while having some of the signals work synchronous to each other.

Because a signal can only update once all of its source signal values are gathered, all of the signals have to produce a value on each event in order to guarantee that each signal has the chance to update its value. For example in figure 2 if signal A updates its value but signal B doesn't, the value of signal D would not be updated until signal B is updated. Therefore when a new event occurs in Elm, all of the source signals in the environment will have to produce a value. Whenever a signal value does not need to update, it can pass a special value, no-change value, indicating that the value was not changed during the event. This allows avoiding unnecessary recomputations when processing new signal values. (Czaplicki and Chong 2013, p. 6) For example if a $lift_2$ based signal receives no-change values from both of its source signals, it can safely skip computing a new value and produce a no-change value instead.

In order to make sure that each source signal receives new events, each source signal is attached to a global event dispatcher. The dispatcher broadcasts new events to all the source signals. The signals receiving the events can determine whether to produce a new value or pass a no-change value based on whether the event was relevant to them or not. (Czaplicki and Chong 2013, p. 5–6) The global event dispatcher can be thought of as the one central source of events originating from outside the FRP system. The source signals can be thought of as the FRP signal representations of certain types of events originating from the dispatcher. In a GUI system, different types of events such as key presses and mouse clicks would be broadcast using the global event dispatcher, while each individual source signal would provide only one type of events and skipping the others.

The global event dispatcher can also be used for broadcasting values of signals as events. Specifically, the dispatcher works as the mediator for asynchronous signal values. When an asynchronous signal is derived from another signal, all the values of the originating signal are dispatched to the global event dispatcher. Furthermore, the actual asynchronous signal is set to listen for dispatcher events. The asynchronous signal produces a value for each original signal event, and produces a no-change value for all the other events. (Czaplicki and Chong 2013, p. 8) The layout for asynchronous signals in the signal graph is demonstrated in the figure 3, which is a variation of the graph shown in figure 2 with signal F set to depend on asynchronous values of signal E. The path marked red shows the flow of signal E values traveling from E to F through the global event dispatcher and the asynchronous signal derived from signal E, async E. When E produces a value, B produces a no-change value and async E produces a value from the event. When an event related to signal B is fired, B produces value while async-E produces a no-change value. Therefore, signal F is able to produce a new value when either signal E or B updates.

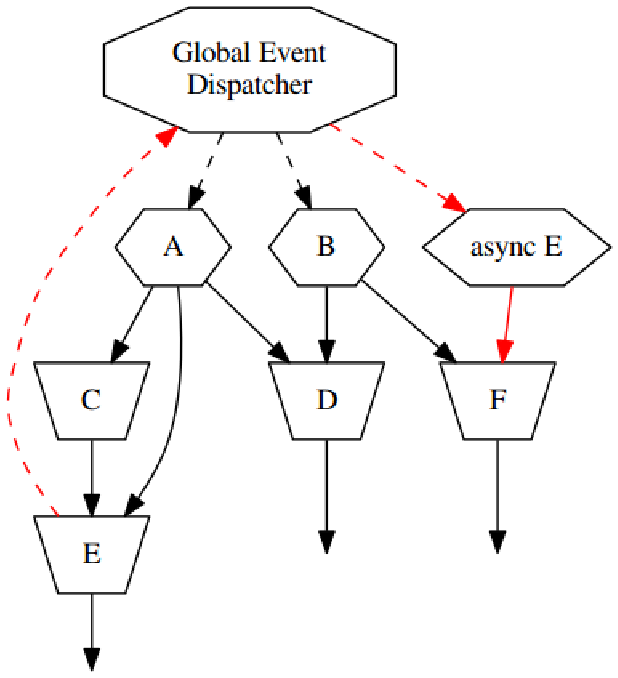

Figure 3. A variation of the graph shown in figure 2 with signal F set to depend on the asynchronous updates of signal E.

4.4 Concurrency constructs for Elm's FRP system

In order to make signals work concurrently, each signal node can be implemented using unbounded FIFO queues and execution threads. Each signal has a channel where new values are enqueued. In order to receive new values from other signals, signals can create queues that are subscribed to other signals' channels. When a new value is enqueued to a channel, the value gets enqueued to all of the queues subscribed to the channel. When a signal is created, an execution thread is started for the signal. The threads reads incoming values from the queues, computes a new value using the received values and a given function, and enqueues the new value to the signal's own channel. (Czaplicki and Chong 2013, p. 5–7) Each channel fans out its values to all the subscribed queues so that each connected signal will receive the new values. Because signals have only one output channel per signal, signals don't need to keep track of the signals depending on them.

The global event dispatcher also has its own combination of threads and queues. The dispatcher has one queue for incoming events and a channel for broadcasting events to source signals. The dispatcher thread reads new events from the queue, and enqueues the events to the channel. Source signals receive new events by subscribing to the dispatcher's channel. (Czaplicki and Chong 2013, p. 7) The global event dispatcher can also be replaced with a single channel. The channel acts as both the receiver and broadcaster of incoming events: enqueued events are immediately fanned out to the subscribing queues. This way the dispatcher doesn't need to have its a dedicated thread.

By circulating asynchronous values through global event dispatcher, the Elm's FRP system can be extended with asynchronous features without modifying its semantics for synchronous signals. Instead of changing the way incoming and outgoing values are handled, asynchronous communication is achieved by arranging the signals in a different way. When an asynchronous signal is created, a new thread starts enqueueing original signal emitted values to the global event queue (Czaplicki and Chong 2013, p. 7). This way there only needs to be one implementation for each reactive primitive: $lift_2$ doesn't need a separate implementation for creating asynchronous signals.

An alternative way for computing new values concurrently is to dispatch new computations to a pool of threads. This would mean that a group of threads is set up in the beginning of the program. Each thread waits for new computations to perform. A new computation is assigned to which thread is free at the moment. If all of the threads are busy, the computation is put into a queue instead. Because computations in the thread pool are not guaranteed to finish in the same order as they are dispatched, signals require additional constructs for maintaining synchronization in and between signals. Therefore creating a dedicated thread for each signal can result in a simpler solution. Thread pools allow keeping the amount of threads in a fixed size as the amount of signals increases, which is why thread pools can become more suitable for environments with limited amount of threads and a large amount of signals.

Regardless of how the concurrency is implemented, there exists a few requirements for what parts the system should be able to run concurrently. Specifically, the system should be able to compute values concurrently for signals that are parallel to each to other. If the execution interleaving happens only between each value computation, the values for parallel signals get computed in sequence when they could be computed parallel to each other. When values are computed in sequence, each computation delays the other computations as long as the computation takes time to finish. Long-running computations will delay all the shorter computations making the system unresponsive. Therefore, the execution interleaving should happen during value computation: long-running computations can be paused in order to let other computations have the chance to finish.

The environments which use native pre-emptive or scheduled threads as signal threads already have the ability to interleave multiple computations. In the environments that have only co-operative threads or don't have any built-in concurrency constructs, computation interleaving must be done manually: a task execution must be manually yielded in order to pass the turn to another task. The granularity of computation interleaving depends on how frequently the tasks have a chance to yield. However, Elm programs shouldn't contain any manual yields in order to retain the semantics described in the earlier section. Instead, yielding should be made part of the runtime implementation, so that the yielding occurs autonomously.

There are several ways to automate execution yielding. One obvious yielding point is queue access: when a signal thread attempts to read an incoming value from one of its queues, the thread will yield the execution if there is now value available. Yielding on queue access alone is not enough, because then yielding will happen only between value computations. One way to achieve more granular yielding is to use compiler techniques where function calls are split into parts. This way a separate task can interleave functions by calling different functions one part at a time. These compilation processes are presented in more detail in chapter 5.

4.5 JavaScript as a runtime language

The main target platform for Elm is the web browser. Elm programs are compiled to JavaScript and HTML. Because there are limited amount of concurrency constructs available in JavaScript, Elm's JavaScript runtime doesn't currently support arbitrary asynchronous signals (Czaplicki and Chong 2013, p. 10). Only native asynchronous operations, such as HTTP requests, have signal implementations that operate asynchronously.

Currently Elm's JavaScript runtime implements signal nodes using JavaScript objects. Each node object contains the signal's unique ID, the current value, an array of child nodes, and a method for notifying the signal of updates. When a new signal is created, the node object constructor appends the node to all of its parents' array of child nodes. When the signal produces a new value (or a no-change value), the signal changes its current value if needed, and calls each of its child node's notify method. All the source signal node objects are stored in a global array. When a new source signal is created, its node object is appended to the global array. The array acts as the global event dispatcher: new events get dispatched to source signals by calling each global array node's notify method. The signal values essentially flow via direct method calls: new values are computed during the notify method calls.

While there exists some operations that work asynchronously, there are only few ways to run JavaScript code concurrently. One way to run code concurrently is to use Web Workers. Another option is to compile Elm code into interleavable JavaScript function calls, and let a custom built scheduler keep executing the function calls part by part. This approach is further explored in chapters 5 and 6.

5 Concurrency in a single execution thread

In order to enhance concurrency in Elm, its the runtime environment, the browser, must include some kind of support for concurrency. Most browsers already support some level of concurrency through events, but they have very limited support for threading.

In the last chapter we learned that Elm FRP semantics can be implemented using threads, which makes them a worthy target for further research. One way to approach the concurrency problem is to find out if threading can be built on top of an environment that doesn't support native threads. The first section of this chapter specifies an interface for building concurrent applications in a style that resembles traditional threaded computing. To support the ideas presented in the first section, the second section examines two different implementations for the new interface: one based on continuations and one based on free monads.

The data between Elm's signals is communicated using a channel-based data communication. The third section shows a way to integrate data communication into the new threading interface, as well as shows how it can be used to implement channels. The fourth section shows how Elm's FRP primitives can be implemented with the help of the new threading interface and the queues.

Most of the code shown in this chapter is written in Haskell. The code acts as intermediary between Elm code and JavaScript code. The final section of this chapter focus on describing how Elm code can be expressed using the Haskell code shown in the chapter.

5.1 Threading API

In this section, we explore the requirements for building interleavable applications and define a Haskell API based on those requirements. By separating the threading API from the final FRP implementation, the FRP primitives and the applications using the concurrency features can be built once, while the API implementation can be optimized as needed.

In order to create threads for a threadless environment, computations in a program must be interleavable in the context of its threading environment. One way to achieve this is by structuring an application in a way that allows capturing the next part of the computation after the previous one is completed, and letting the threading environment's custom scheduler choose which part to execute next. The final part in the computation chain produces the final result of the computation, which should be made available to other computations in order to be able to compose multiple computations.

Because the non-native threads can only be interleaved at the points where they pause their computation, the granularity of interleaving depends on how frequently a thread pauses itself. High granularity means that long-running threads are less likely to block other concurrent threads from executing, which can improve responsiveness. On the other hand, splitting a computation into too many parts can also bring overhead by causing unnecessary pauses. Thus the user of the threading API should be allowed to decide the granularity of the threads they build.

One way to model threads is to express them using data types. Modeling threads using types allows threads to be passed around and extended similar to other values. While the thread type shouldn't depend any particular interleaving method or expose how the thread is interleaved, it should include the information about the type of the final result. Thus the threads can be expressed using the type signature `Thread r`, where `r` is the type of the thread's final result.

In order to bring native computations into the threading environment, the API needs a function for creating threads out of normal computations. Because the native computations don't use the threading environment, they will naturally be indivisible when made into threads. The function `atom` creates a thread with a single computation, the given computation, which is also the thread's final result.

```
atom :: r -> Thread r
```

Atomic threads by themselves are not enough for creating interleavable threads, but a sequence of them can be interleaved with other sequences. A sequence of atomic threads is a thread in itself, thus both atoms and sequences share the same type `Thread`. This way the threads can share the same API for composition regardless of their structure.

One way to compose threads is to use the monad interface. Particularly, the `Thread` type should be a monad, where the bind method is used for composing threads together. The implementation of the monad interface depends on how the `Thread` type and interleaving is implemented. The benefit of using the monad interface as an abstraction for thread composition is that it allows separating the thread composition from the application logic. The monad interface is also well-supported in the Haskell environment, which offers a many functions that operate on the monad abstraction and an special notation for performing bind operations. Listing 5.1 demonstrates how a typical long-running using the monadic properties of threads and the atom function. The sum of the two previous fibonacci numbers is made atomic in the thread, which acts as the interleaving point for the rest of the computation. Because the monad's bind method allows capturing the result of the sub thread to form a new thread with a new result, the sum operation is able to use the captured results from the sub fibonacci calls.

Listing 5.1. Naive fibonacci algorithm written as an interleavable thread

```
naiveFibonacci :: Thread Int
naiveFibonacci n =
    if (n <= 1) then atom 1
    else do
        x <- naiveFibonacci (n - 1)
        y <- naiveFibonacci (n - 2)
        atom (x + y)
```

Programs such as the fibonacci in program in listing 5.1 can be interleaved with other threads, but there also needs to be a way to launch new threads. In the Elm semantics, a thread is spawned for each signal for processing the new changes (Czaplicki and Chong 2013). In the threading API, the processing functions for the signals could be built beforehand and passed on to the threading environment's scheduler for interleaving once all of them have been built.

An alternative option for building all threads beforehand would be to require the threading implementation to support creating new concurrent threads during the run of a another thread. The benefit of this is that it allows creating new concurrent threads while the scheduler is running. The function `fork` creates a thread that executes the given thread concurrent to other the thread the `fork` call was initiated in. The call to `fork` returns immediately after setting up the concurrent thread, thus the thread execution continues from whatever computation is bound next in the thread. Since the forked thread is executed concurrent to the rest of the thread, the `fork` call cannot return the final value of the given thread. Thus the type of the thread's final value is a unit value.

```
fork :: Thread r -> Thread ()
```

Because the final results of the concurrent threads created using `fork` cannot be captured from outside the thread, the threads have no method of passing information out of them. In the Elm semantics, the signal threads use message passing to convey information between threads (Czaplicki and Chong 2013). One way to solve problem would be to integrate the messaging API into the threading API in order to allow communication between threads.

The messaging API doesn't need to be part of the threading API, but it can be delegated to another monad instead. This way the threading API can focus only on delivering concurrency, while the messaging API can be made as elaborate as necessary. This can be achieved by allowing the computations in messaging monad to have an effect on how thread computations progress. In order to incorporate the messaging monad's capabilities into the threading API, the API functions and types need to be aware of the additional monad type. The threads can be expressed as type `Thread m r` where `m` is the type of the monad that delivers the messaging functionality and `r` is the type of the thread's final result. Similarly, the type signatures for `atom` and `fork` also change.

```
atom :: Monad m => m r -> Thread m r
fork :: Monad m => Thread m r -> Thread m ()
```

Instead of interleaving pure functions, the threads instead interleave the messaging monad computations. In order to drive the outcome of the threads, the messaging monad computations must be chained correctly as well. Therefore, thread's bind method is required to be able to chain both its own computations and the computations of the messaging monad that occur inside the thread. In other words, the thread monad must have both the monad (Wadler 1992) and monad transformer (Liang, Hudak, and Jones 1995) capabilities.

5.2 Implementations for the threading API

In this section, we introduce two threading monads that can be used for implementing the threading API. First, we introduce Claessen's continuation-based concurrency monad. Second, we introduce Gonzales's free monad based threading monad.

In the continuation-based concurrency monad, each continuation monad holds the future of the computation (the continuation) which is to be called at the end of the monad evaluation Claessen (1999). The concurrency monad implementation is shown in listing 5.3. This allows composing separate computations together by creating continuation computations that pass in the next step of the computation as the continuation for the earlier computation. Because these joining computations are continuation monads themselves, they can be further joined together with more computations.

The concurrency monad uses a separate datatype, `Action`, for driving the flow of the continuation Claessen (1999). There are three types of actions: `Atom`, `Fork`, and `Stop`. Each type of action contain the actions that follow it. The `Atom` action contains one action produced through an arbitrary monad computation, which allows effects to occur between action evaluation. The `Fork` action contains two branching actions, which allows splitting the computation into multiple paths. Finally, the `Stop` action contains no follow-up actions, which allows it to end the chain of actions.

The concurrency monad can be used directly as the threading API's thread monad. The interleaving can be achieved by evaluating one action at a time from each concurrent action. In order to compose the actions, the actions are wrapped into continuations where the follow-up actions are resolved through the continuations passed to them. This way the continuation monad's monad interface can be used for composing actions. The concurrency monad's `atom` and `fork` functions follow the same semantics as the corresponding functions in the threading API.

Non-native threads can also be implemented using a combination of free monads and a datatype, that acts as the instruction set for the thread control. The full implementation of the free monad threads are shown in listing 5.4. There are three types of instructions: `Yield`, `Fork`, and `Done`. Each instruction contains the instructions that follow it. `Yield` is used for yielding the execution turn to another thread, and it contains the instruction that should follow the yield. `Fork` creates two execution branches, thus it contains two follow-up instructions. `Done` is used for ending the thread, so it doesn't contain any follow-up instructions.

The instructions can be composed together with the help of the free monad. The thread monad type is a free monad consisting of `ThreadF` nodes. Thus binding functions into thread monad values works similar to how it works in any other free monads: the binding function is delegated the next step of the free monad tree until the binding function can be bound to the last step. This way multiple threads can be joined using the monad's bind method.

The free monad based threading monad can also be used as the threading API's thread monad. Threads are essentially free monad trees that can be evaluated one node at a time to achieve interleaving. The `fork` function shown in listing 5.4 is compatible with threading API's `fork` function. The execution turn is meant to be yielded with the special function `yield`, by introducing it into the thread at the point where the execution should be yielded. The `yield` function can be used for creating threading API's `atom` function as shown in listing 5.2: the execution is always yielded right after the given monad is evaluated.

Listing 5.2. Threading API's `atom` function implemented using `yield`

```
atom :: Monad m => m b -> Thread m b
atom m = do
    v <- lift m
    yield
    return v
```

The continuation-based concurrency monad and the free monad based threading monad have very similar approaches for building threads. Both express threads as trees of thread instructions, and both use the capabilities of another monad to built those trees. The techniques for building the threading trees is different, but the result is similar when the trees are evaluated.

Listing 5.3. Continuation based implementation of a concurrency monad (Claessen 1999) using the continuation monad from chapter 3. The concurrency monad uses its own implementation of a continuation monad. The function `rrScheduler` is a round robin scheduler for the concurrency monad.

```
data Action m = Atom (m (Action m))
              | Fork (Action m) (Action m)
              | Stop

type Conc m a = Cont (Action m) a

instance Functor (Cont r) where
    fmap f (Cont c) = Cont (\k -> c (\x -> k (f x)))

action :: Monad m => Conc m a -> Action m
action (Cont k) = k (\_ -> Stop)

atom :: Monad m => m a -> Conc m a
atom m = Cont $ \c -> Atom (m >>= \x -> return (c x))

par :: Monad m => Conc m a -> Conc m a -> Conc m a
par (Cont k1) (Cont k2) = Cont (\c -> Fork (k1 c) (k2 c))

fork :: Monad m => Conc m a -> Conc m ()
fork m = Cont (\c -> Fork (action m) (c ()))

stop :: Monad m => Conc m a
stop = Cont (\c -> Stop)

rrScheduler :: Monad m => [Action m] -> m ()
rrScheduler [] = return ()
rrScheduler (a:as) = case a of
    Atom m     -> m >>= \x -> rrScheduler (as ++ [x])
    Fork a1 a2 -> rrScheduler (as ++ [a1,a2])
    Stop       -> rrScheduler as

run :: Monad m => Conc m a -> m ()
run m = rrScheduler [action m]
```

Listing 5.4. Free monad based implementation for non-native threads (Gonzales 2013). The threads use free monad transformer implementation from `Control.Monad.Trans.Free` module. The `roundRobin` function is round robin scheduler for the free monad threads.

```haskell
{-# LANGUAGE DeriveFunctor #-}

import Control.Monad.Trans.Free
import Control.Monad
import Data.Sequence

data ThreadF next = Fork  next next
                  | Yield next
                  | Done
                  deriving (Functor)

type Thread = FreeT ThreadF

yield :: (Monad m) => Thread m ()
yield = liftF (Yield ())

done :: (Monad m) => Thread m r
done = liftF Done

cFork :: (Monad m) => Thread m Bool
cFork = liftF (Fork False True)

fork :: (Monad m) => Thread m a -> Thread m ()
fork thread = do
    child <- cFork
    when child $ do
        thread
        done

roundRobin :: (Monad m) => Thread m a -> m ()
  where
    go ts = case (viewl ts) of
        EmptyL   -> return ()
        t :< ts' -> do
            x <- runFreeT t
            case x of
                Free (Fork t1 t2) -> go (t1 <| (ts' |> t2))
                Free (Yield   t') -> go (ts' |> t')
                Free  Done        -> go ts'
                Pure  _           -> go ts'
```

5.3 Communication between threads in the threading API

Threads require a mechanism for communicating information between other concurrent threads. In this section, we introduce a mechanism for passing data between threads that's compatible with both the Elm semantics and the threading API introduced earlier in this chapter.

The Elm semantics uses unbounded queues and channels to convey information between threads (Czaplicki and Chong 2013). Multicast channels act as sources of information that can be subscribed to in order to receive updates from them. Each multicast channel subscription produces a new message channel where all the messages coming from the multicast channel are enqueued to. Threads can receive the information from multicast channels by reading updates from one of its channels. When a new value is read from the channel, the value is removed from the channel. When an empty channel is read, the execution of the thread should be blocked until a new value is received.

The channel-based messaging operations can be implemented as state changes. The state holds the registered multicast channels, their subscriptions, and the messages dispatched through the system. Each operation creates a new state with that incorporates the changes made by the operation. The new state is made available for the next computation, while the old state is dropped.

The monad interface can be used to encapsulate state changes, thus it can be made compatible with the threading API. The state changes made by the messaging operations are conveyed as part of the parametrized monad that is used in the threading API.

State changes can be implemented using Haskell's IO monad and MVars. MVars are mutable locations for values. Each MVar holds either one or no values. The value can be changed to another value or read. All of the MVar operations occur in the IO monad because of the side effects that they cause. MVars essentially allow creating state changes in the IO monad.

Another way to state changes is to use a state monad. When state monad is used, all of the channel data passed along as the state value in the state monad. Using the state monad for state changes allows avoiding the use of the IO monad, which limits the range of possible side effects that can happen in the threading system. The examples in this article assume that channels are implemented using MVars and the IO monad.

A good example of a channel implementation are Haskell's FIFO channels, which allow state changes to occur using the IO and MVars. Haskell's channels are designed for multi-threaded environments, but they can also be used in a single thread environment as well. However, the channel read operation will block the current native thread if there is no value available in the channel. Blocking the thread is convenient when channels are used between multiple native threads, but it can potentially halt the whole environment when channels are read in a single threaded system. Channel read operations should block the threading API thread instead of the native thread.

When implementing the messaging API using IO and MVars, channels can be implemented using an MVar that holds a sequence of queued values. Listing 5.5 shows the data structure used for channels, and how the enqueuing and dequeuing operations can be implemented. Enqueuing a value appends the value to the end of the sequence. Arbitrary amount of values can be appended to the sequence without blocking the enqueue call, which makes the channel unbounded. Dequeuing removes and returns the value from the head of the sequence. Because channels can also be empty, dequeuing doesn't always return a value. Therefore, dequeue has to return a value that represents an empty value when an empty channel is dequeued.

Listing 5.5. Channels for the concurrency monad

```
import Data.Sequence as Seq
import Control.Concurrent.MVar

data Channel a = Channel (MVar (Seq a))

newChannel :: IO (Channel a)
newChannel = do ms <- newMVar Seq.empty
                return (Channel ms)

enqueue :: Channel a -> a -> IO ()
enqueue (Channel ms) a =
    modifyMVar_ ms (\s -> return (s |> a))

headMaybe :: Seq a -> Maybe a
headMaybe s =
    if Seq.null s then Nothing
    else Just (Seq.index s 0)

dequeue :: Channel a -> IO (Maybe a)
dequeue (Channel m) =
    modifyMVar m (\s -> return (Seq.drop 1 s, headMaybe s))
```

Instead of returning an empty value when the channel is empty, the desired effect would be to block the thread until the channel receives a new value. Blocking can be implemented by repeating the channel read operation when no value is available. Each channel read attempt has to be atomic in order to allow yielding between retrys. Listing 5.6 contains a `retry` function, which keeps on evaluation the given monad until it receives a value. Blocking channel read can be defined using the previously introduced dequeue and retry functions. Dequeue produces an IO monad that contains either the value from the channel or an empty value. Retry can use the result received from dequeue for deciding whether to block the read operation or not.

Listing 5.6. Retry function and blocking channel read operation

```
retry :: Monad m => m (Maybe x) -> Thread m x
retry m = do
    x <- atom m
    case x of
        Just v -> return v
        Nothing -> retry m

receive :: Channel a -> Thread IO a
receive = retry . dequeue
```

Multicast channels can also be implemented using IO and MVars. Listing 5.7 contains an implementation for multicast channels using MVars, where multicast channel is a storage that contains a collection of subscribed channels. When a new subscription is made to a multicast channel, a new subscribing channel is created and returned to the caller. Enqueuing a value to multicast channel causes the value to be enqueued to all of the subscribed channels. Because the enqueue operation doesn't involve any threading API operations, enqueueing is always atomic in the context of the threading API threads. Therefore, the values enqueued to the multicast channel will always appear in the same order in each subscribing channel regardless of which non-native threads enqueue the values.

Listing 5.7. An implementation of multichannels using Haskell's MVars

```
import Control.Concurrent.MVar

data MultiChannel a = MultiChannel (MVar [Channel a])

newMultiChannel :: IO (MultiChannel a)
newMultiChannel = do mcs <- newMVar []
                     return (MultiChannel mcs)

subscribe :: MultiChannel a -> IO (Channel a)
subscribe (MultiChannel mcs) = do
    c <- newChannel
    modifyMVar_ mcs (\cs -> return (c:cs))
    return c

multiEnqueue :: MultiChannel a -> a -> IO ()
multiEnqueue (MultiChannel mcs) a = do
    cs <- readMVar mcs
    mapM_ (\c -> enqueue c a) cs
```

5.4 Signals in the threading API

With the help of the threading API and channel-based thread communication, the implementation of Elm's FRP semantics becomes straightforward. Czaplicki and Chong (2013) described how the basic semantics of Elm can be translated to Concurrent ML Reppy (1993). The CML implementation provides good examples for implementing the same semantics using threads and channels.

The core structure for the FRP system is a signal. Each signal has a starting value, and they can be subscribed for changes. Each emitted change tells whether the value of the emitting signal has changed or not, and what the value for signal was at that point.

Czaplicki and Chong (2013) narrowed the signal data structure down to a tuple combined from a starting value and a mailbox of changes. Mailboxes are buffered non-blocking channels which other signal's can subscribe to. The signal's mailbox is essentially an output channel for the values that the signal produces. The incoming changes are copied from a signal's output channel to all of its subscribers.

In order to guarantee consistency between subscribing signals, all the changes must be fanned out to all the subscribers, and nothing should directly read new values from the signal's output channel. However, mailboxes themselves store the data until the data is read from the mailbox. Because the signal output channels are never read directly, every enqueued change will remain in the output channels, resulting in a memory leak.

An alternative choice to buffered channels would be to use multicast channels for all signal output channels. Values enqueued to multicast channels are broadcast to all of their subscribers and no values are retained in the multicast channel itself. This way the values accumulate only to signal subscribers' channels.

Because nothing needs to read the signal output channels directly, multicast channels can be used as a replacement for the output channel mailboxes effortlessly as shown in listing 5.8. The signal's output channel is replaced with a multicast channel, and the subscription and send functions are replaced with the multicast channel equivalents.

The `atom` functions is used for lifting the `subscribe` and `multiEnqueue` functions and the multicast channel constructor to thread monad. These functions can also be lifted to the thread monad without yielding the execution if it is supported by the thread monad.

Listing 5.8. The data structures and helper functions for signals and changes

```
mChannel :: Thread IO (MultiChannel a)
mChannel = atom newMultiChannel

data Change a = Change a | NoChange a

isChanged :: Change a -> Bool
isChanged (Change _) = True
isChanged (NoChange _) = False

bodyOf :: Change a -> a
bodyOf (Change a) = a
bodyOf (NoChange a) = a

data Signal a = Signal a (MultiChannel (Change a))

port :: MultiChannel a -> Thread IO (Channel a)
port = atom . subscribe

send :: MultiChannel a -> a -> Thread IO ()
send c a = atom (multiEnqueue c a)
```

Events from outside of the FRP system can be brought into the system with the help of a global event dispatcher and input signals. Listing 5.9 demonstrates how input signal primitive can be implemented for the threading API based FRP system. Each input signal is assigned with a signal ID, and a channel of input values. The input signal's listen to global event dispatcher's event notify channel (the `eventChan` parameter) for notifications on which signal the next emitted value belongs to. If the received ID matches the input signal's ID, the signal can read the next value from its input channel and enqueue it as a value change to the signal's output channel. If the IDs don't match, the signal will enqueue its existing value as a no-change value.

Listing 5.9. Input signal primitive implemented using threads and channels

```
sendLoop :: MultiChannel (Change a)
         -> a -> (a -> Thread IO (Change a))
         -> Thread IO b
sendLoop c prev m = do
    msg <- m prev
    send c msg
    sendLoop c (bodyOf msg) m

inputSignal :: Eq id => MultiChannel id
            -> id -> MultiChannel a -> a
            -> Thread IO (Signal a)
inputSignal eventChan signalId scIn v = do
    cOut <- mChannel
    e <- port eventChan
    cIn <- port scIn
    fork $ sendLoop cOut v $ \prev -> do
        eId <- receive e
        if signalId == eId
            then fmap Change $ receive cIn
            else return $ NoChange prev
    return $ Signal v cOut
```

The functionality of the input signal primitive matches the functionality of the CML input signal translation. Both implementations produce the same results on same inputs.

However, unlike in the original CML implementation, only one subscription is established to the event notify channel for the input signal. The threading API based input signal primitive establishes the subscription to event notify channel before the loop function is started, thus only one channel is created for the signal to consume.

Input signals have a major role in the creation of asynchronous signals. Listing 5.10 demonstrates how the `async` primitive can be implemented for the threading environment. The primitive follows the same logic as the CML version of `async`: the primitive is used for creating events out of the values emitted by the given signal. The primitive creates an output channel for the values received from the given signal. The primitive also creates a new input signal with the given ID and the new output channel. For each new value coming from the given signal, the incoming value is send to the created output channel and an event send to the event channel with the ID of the new input signal. This way the new input signal receives the values from the given signal as events.

Listing 5.10. `async` primitive implemented using threads, channels, and the input signal primitive.

```
async :: Eq id => MultiChannel id -> id -> Signal a
      -> Thread IO (Signal a)
async eventChan sId (Signal d scIn) = do
    cIn <- port scIn
    cOut <- mChannel
    let loop = do
            msg <- receive cIn
            case msg of
                NoChange _ -> loop
                Change v -> do
                    send cOut v
                    send eventChan sId
                    loop
    fork loop
    inputSignal eventChan sId cOut d
```

Besides the `async` primitive, threads and channels can also be used for implementing other useful primitives. The implementation of FRP primitives `lift2` and `foldp` are demonstrated in listing 5.11.

Both of the `lift2` and `foldp` primitives and the primitives described earlier follow the same principles as their CML equivalents: channel subscriptions are established for the parent channels using `port` function, an output channel is created using `mChannel`, and a thread loop is started using `fork`. On each iteration of the signals' loop function, new changes are read from subscriptions using `receive`, a new change is computed, and the change is enqueued to the output channel using `send`.

Similarly, the return value of the primitives are signals. The signal is wrapped into a thread and joined with the rest of the computation in order to retain the effects from the forked thread. Monad transformer properties of the thread also makes sure that the effects from the channel operations are preserved in the return value.

Primitives such as `lift2` and `foldp` use a function given to them to compute the new change. Instead of the producing a pure value, the functions produce a thread instead. When the signals' loop function computes a new change, the received thread is joined into the loop thread's execution. This allows interleaving to also occur while computing new changes.

Listing 5.11. `lift2` and `foldp` implemented using threads and channels

```
lift2 :: (a -> b -> Thread IO c) -> Signal a -> Signal b
      -> Thread IO (Signal c)
lift2 f (Signal v1 sc1) (Signal v2 sc2) = do
    c1 <- port sc1
    c2 <- port sc2
    cOut <- mChannel
    v <- f v1 v2
    fork $ sendLoop cOut v $ \prev -> do
        msg1 <- receive c1
        msg2 <- receive c2
        if (isChanged msg1) || (isChanged msg2)
            then fmap Change $ f (bodyOf msg1) (bodyOf msg2)
            else return $ NoChange prev
    return $ Signal v cOut

foldp :: (a -> a -> Thread IO a) -> a -> Signal a
      -> Thread IO (Signal a)
foldp f v0 (Signal _ sc) = do
    c <- port sc
    cOut <- mChannel
    fork $ sendLoop cOut v0 $ \acc -> do
        sourceMsg <- receive c
        case sourceMsg of
            Change v -> fmap Change $ f v acc
            NoChange _ -> return $ NoChange acc
    return $ Signal v0 cOut
```

5.5 Translating Elm applications into interleavable FRP programs

The signal primitives and the threading API shown in the previous sections can be used for delivering Elm's FRP semantics in a single-thread. In order to make the platform useful for the Elm language, the compiler needs to translate the Elm applications into applications that use the threading API. In this section, we present some options for translating Elm applications.

As it was pointed out in section 4.4, program interleaving should occur automatically in order for it to be transparent for the Elm programmer. However, in the section 5.1 we pointed out that the interleaving must be decided manually. Therefore, the interleaving points would have to be decided during the translation from Elm to threading API programs.

However, deciding the optimal places for computation interleaving can be difficult. In the section 5.1 we also pointed out that interleaving brings additional overhead to the length of the execution. Thus the interleaving should optimally only occur when it helps the responsiveness of the application.

One way to approach automatic interleaving is to find the computations that usually take longer to execute and attempt to introduce interleaving points to them. Some of the typical long-running programs are programs where the running time depends on the size of the program's parameters. For example, each map and fold function call has to apply the given function for each element in the given collection, thus the running time of map and fold are tied to the length of the given list.

Interleavable `map` and `fold` functions are demonstrated in listing 5.12, where each step is made into an atomic operation. `mapM` and `foldM` allow passing the effects of the mapping/-folding function from one step to the next while computing new values. Yielding the execution between each step is a naive way to achieve interleaving as the benefits of interleaving can in many cases become visible only after several steps in the `map` or `fold` process. The `mapC` and `foldC` shown in the listing work as examples for approaching automatic interleaving by replacing some of the native functions with versions that have automatic yielding built into them.

Listing 5.12. Interleaving versions of `map` and `fold` higher order functions

```
mapC :: Monad m => (a -> b) -> [a] -> Thread m [b]
mapC f xs = mapM (atom . return . f) xs

foldC :: Monad m => (a -> b -> m a) -> a -> [b] -> Thread m a
foldC f a xs = foldM (\acc x -> atom . return $ f acc x) a xs
```

Having some of the pure functions in Elm translate into pure functions and the other into interleavable functions complicates the function composition in the translation target. Because the interleaving is done using the threading monad, functions calling an interleavable function must also produce a thread in order to retain the return value and effects of the interleavable function. At the same time, the pure functions that don't require interleaving should be translated into pure functions in order to avoid unnecessary execution yielding. Meanwhile, the composition of interleavable and non-interleavable functions should be transparent in Elm source code.

Some of the translation scenarios are demonstrated in listings 5.13 and 5.14. Listing 5.13 contains a set of functions written in Elm, and listing 5.14 contains example translations of those functions. The examples assume that certain functions are translated to native counterparts in Haskell, and that the interleavable versions of `map` and `fold`, `mapC` and `foldC`, are used in the translation target. These examples ignore the differences in Haskell's and Elm's evaluation order.

The `hypotenuse` function is an example of a pure function that translates to a pure Haskell function. The function depends only on pure functions, thus it isn't required to produce a thread. Conversely, the `squares` function depends on `map` which translates to `mapC` which produces a thread.

Because the return type changes during the translation, every function using `squares` must be able to handle its resulting thread. For example, the `squaresSum` function passes the result of `squares` function to `fold`. In the Elm program, the composition occurs naturally. However in the translated program, the result of `squares` must be composed with the `foldC` function using monad binding in order for `foldC` to be able to use the actual return value of the `squares` function call. Similar to the `squares` function, the translated version of `squaresSum` must also return a thread because it interacts with other threads. Even functions such as `squaresSumPlus1` must return a thread even if it doesn't directly interact with `foldC` or `mapC`.

In some cases threads can become nested. For example in listing 5.13 and 5.14, the function `sums` uses the function `fold` as a part of `map` function's mapping function, which produces a thread wrapping a list of threads in the target program. In order to keep threads from nesting uncontrollably, nested threads can be joined to a single thread before the result is passed outside of the function. In the `sums` function of listing 5.14, the list of threads are joined together into a single thread using `sequence` function. There are several ways thread nesting can occur, thus there must also be several methods for the compiler to handle unnesting as well.

The approach for integrating automatic interleaving into Elm compiler shown here is inconclusive, but it highlights some of the issues that may occur when attempting to transparently compose interleavable code with non-interleavable code. Therefore, this topic is left open for further research. Meanwhile, parts of the threading API can be exposed to the Elm code in order to let Elm programmers to create interleavable programs themselves.

Listing 5.13. Elm example functions

```
hypotenuse : Float -> Float -> Float
hypotenuse a b = sqrt (a^2 + b^2)

squares : [Int] -> [Int]
squares xs = map (\x -> x^2) xs

squaresSum : [Int] -> Int
squaresSum xs = foldl (+) 0 (squares xs)

squaresSumPlus1 : [Int] -> Int
squaresSumPlus1 xs = squaresSum xs + 1

sums : [[Int]] -> [Int]
sums xs = map (\x -> foldl (+) 0 x) xs
```

Listing 5.14. Elm example translated to Haskell

```
hypotenuse :: Float -> Float -> Float
hypotenuse b = sqrt (a^2 + b^2)

squares :: Monad m => [Int] -> Thread m [Int]
squares xs = mapC (\x -> x^2) xs

squaresSum :: Monad m => [Int] -> Thread m Int
squaresSum xs = do
    v <- squares xs
    foldC (+) 0 v

squaresSumPlus1 :: Monad m => [Int] -> Thread m Int
squaresSumPlus1 xs = do
    v <- squaresSum xs
    return (v + 1)

sums :: Monad m => [[Int]] -> Thread m [Int]
sums xs = mapC (\x -> foldC (+) 0 x) xs >>= sequence
```

6 Translating non-native threads from Haskell to JavaScript

The FRP system described in chapter 5 is written in Haskell, but the goal is to be able to run it in a browser. Instead of inventing a similar FRP system for the browser environment, we can attempt to translate the ideas and implementations for the browser from the Haskell code.

Translating Haskell programs for the browser has its own challenges. The primary programming language for the browser, JavaScript, doesn't include many of the concepts that the Haskell implementation of the FRP system depends on. In the first section, we examine some of the challenges that can occur when translating programs from Haskell to JavaScript and provide solutions for these situations. Some of the solutions we provide only solve the problems partially. In the second section, we show how these solutions affect the threading API in JavaScript.

In order to support the FRP system, the browser needs its own implementation of the threading API. In the third section, we provide an implementation of the modified threading API based on free monads. In the fourth section, we demonstrate how the implementation can be integrated into the browser environment.

6.1 Translating programs from Haskell to JavaScript

Translating programs from Haskell to JavaScript isn't always straightforward because of the fundamental differences between the two languages. For example, JavaScript uses dynamic typing and strict evaluation while Haskell uses static typing and non-strict evaluation. In this section, we examine some of the challenges of translating Haskell code to JavaScript and provide solutions for them. We only focus on providing solutions for challenges that assist translating the Haskell threading environment from the previous chapter to JavaScript.

All values in Haskell use non-strict evaluation by default, while JavaScript uses strict evaluation. In non-strict evaluation, expressions are evaluated from the outside in (Hudak 1989), which means that the evaluation of values are delayed until they're explicitly needed. On the other hand, in strict evaluation, the innermost expressions are evaluated before the expressions that wrap them, which means that the values are evaluated before they're made available.

One way to create non-strict values in a strict language is to encapsulate values in parameterless functions. The body of the function acts as the value that is yet to be evaluated. The value can be evaluated by calling the function.

Non-strict values should also be composable. One of the key benefits of non-strictness is the ability to effortlessly build values based on other values without evaluating any of them. However, in a strict environment, composition of non-strict values must be managed manually. Listing 6.1 contains an example of how two non-strict values can be used for creating a third non-strict value without evaluating any of the values. The values `value1` and `value2` are both parameterless functions that contain an unevaluated computation, while the `lazyResult` is a parameterless function that composes the two values two create a new value. In its body, the `lazyResult` function calls the two values to evaluate them and sum both of their results. Because the computation is wrapped in a parameterless function, the whole computation isn't evaluated until the function is called. Thus the evaluation of both `value1` and `value2` are successfully delayed until `lazyResult` is evaluated.

Listing 6.1. Composing non-strict values using parameterless functions in JavaScript

```
var value1 = function() { return 9 + 1;  };
var value2 = function() { return 15 - 2; };

var lazyResult = function() {
  var v1 = value1();
  var v2 = value2();
  return v1 + v2;
};

var result = lazyResult();
```

While the parameterless functions provide a flexible method for handling non-strict values in a strict environment, they should be used cautiously. Each non-strict value creates an additional anonymous functions in the runtime, which may cause too much overhead when used for small computations. Instead of enforcing non-strictness everywhere, programs should use parameterless functions to bring non-strictness only where it matters.

Haskell uses type classes to support features such as monads. When a call is made to a type class function such as the monad bind, the call is dispatched to the correct implementation based on the type of the parameter.

JavaScript lacks the support for both the type classes and monads. In JavaScript, type classes' function dispatch can be emulated to some extent in the runtime using object methods. For example, `Monad` type class's bind function can be written as a method of the monad object. On the other hand, `Monad` type class's `return` function cannot be written as a method because it has no parameter to resolve the correct implementation from.

Instead of attempting to create a complete solution for the type class system, the type classes can be resolved manually for JavaScript. Creating a full implementations of the Haskell's type class system would require implementing a separate type system for the JavaScript. A more convenient solution would be to manually replace the calls to type class functions with their actual implementations. For example, if a function depends on some data type's monadic capabilities, the JavaScript translation of that function can be hard coded to use the implementation specific to the data type.

The disadvantage of manually resolving type classes is that it prevents reusing functions between different implementations of the same type class. If a function depends on a type class function, that function has to be implemented separately for each type class that are used in combination with function in the final program.

Haskell enforces purity by restricting where side-effects can occur in the code using constructs such as the IO monad, while JavaScript doesn't restrict side effects anywhere in the code. Because purity is not enforced in JavaScript, there is no guarantee on whether any function in JavaScript is pure or not. Therefore, there is no need to built a separate construct for encapsulating all the side-effects either.

Haskell supports sum types. A sum type is a data type that consists of several variants. Each sum type can have only a fixed number of variants, and each instance of a sum type can only be of one variant. Each variant can have their own set of fields. Listing 6.2 contains an example of how shapes can be expressed using a sum type. In the `Shape` type, each variant is a different kind of shape. The variants for the `Shape` type are `Circle`, `Rectangle`, and `Triangle`.

Listing 6.2. Shapes expressed as Haskell sum types.

```
type Point = (Int, Int)
type Radius = Int

data Shape = Circle Point Radius
           | Rectangle Point Point
           | Triangle Point Point Point
```

JavaScript doesn't include support for sum types, but they can be partially emulated using JavaScript objects. The emulation can be achieved by using a separate field, a discriminator, to indicate the variant. The discriminator field is stored along with the rest of the variant's fields in a JavaScript object. Programs can determine what fields are available in an object by inspecting the discriminator field. Listing 6.3 contains a JavaScript translation of the example shown in listing 6.2. Each shape has a function for creating an instance of the shape. Each of the functions set the field `shape` to indicate the shape's variant.

Listing 6.3. Sum types emulated using JavaScript objects.

```
function circle(point, radius) {
  return { shape: 'circle', point: point, radius: radius };
}
function rectangle(point1, point2) {
  return { shape: 'rectangle', point1: point1, point2: point2 };
}
function triangle(point1, point2, point3) {
  return {
    shape: 'triangle', point1: point1,
    point2: point2, point3: point3
  };
}
```

6.2 Adjusting the threading API for the JavaScript

The threading API we presented in section 5.1 depends on Haskell features that are not present in JavaScript. Therefore, the API cannot be translated to JavaScript as it is. In this section, we present a modified version of the threading API that is adjusted for JavaScript.

The lack of type classes limit the ability to support the messaging API through arbitrary monads. Because type classes are not supported in JavaScript, the monad operations for the messaging API must be delivered through other means. Instead of attempting to support any monad, the threading API can alternatively support only one method of messaging.

Instead of using monads to integrate a messaging API to the threading API, messaging can be delivered using unpure code in JavaScript. Listing 6.4 contains an example on how a messaging system can be implemented using JavaScript's mutable arrays. The messaging system follows the semantics of the channel-based messaging system shown in section 5.3. The `makeMultiChannel` and `makeChannel` functions create new arrays for the data they store. Array's `push` method is used in `subscribe` and `enqueue` functions to add a new element to the end of the array. Array's `shift` method is used in `dequeue` function to get and remove the first element from the start of the array. Both of these array methods are destructive: they modify the array they operate on. Thus the changes will be automatically available to anyone with a reference to the modified arrays.

Listing 6.4. Channel-based messaging system implemented in JavaScript using mutable arrays

```
function makeMultiChannel() { return []; }
function makeChannel() { return []; }

function subscribe(multiChannel) {
  var channel = makeChannel();
  multiChannel.push(channel);
  return channel;
}

function enqueue(channel, value) { channel.push(value); }

function multiEnqueue(multiChannel, value) {
  multiChannel.forEach(function(channel) { enqueue(channel, value) });
}

function dequeue(channel) {
  return channel.length > 0 ? channel.shift() : null;
}
```

Using unpure code instead of monads for messaging changes type signatures for the thread data type and the API functions. The thread type in the updated API does not include a type parameter for the messaging monad. The `atom` function in the updated API creates a thread from a plain computation, a parameterless function, instead of a monad value. On the other hand, the type signature for the `fork` function remains the same in the updated API.

```
function atom(value) { ... };
function fork(thread) { ... };
```

Threading API requires its own function for composing threads. Because the monad interface is not available in JavaScript, the composition function needs to defined manually. The function used for composition, `bindThread`, is the equivalent of monad's bind function. It combines given thread with the given function to create a new thread. The function `makeThread` is the equivalent of monad's return function, which can be be used for creating new threads from other values. The use of the composition method is demonstrated in listing 6.9. The fibonacci function follows the same principles as the function shown in section 5.1: each sum is made into an atomic operation to make it interleavable.

Listing 6.5. A naive version of the fibonacci algorithm implemented using the JavaScript threading API.

```
function bindThread(thread, f) { ... };
function makeThread(value) { ... };

function naiveFibonacci(n) {
  if (n <= 1) {
    return atom(function() { return 1; });
  } else {
    return bindThread(naiveFibonacci(n - 1), function(left) {
      return bindThread(naiveFibonacci(n - 2), function(right) {
        return atom(function() { return left + right; });
      });
    });
  }
}
```

6.3 Implementing the updated threading API

The updated version of the threading API needs an implementation to back the applications that use the API. Instead of creating a new implementation from scratch, the implementation of the API can be based on existing implementations such as the ones shown in section 5.2. In this section, we state the limitations that have to be dealt with when translating the free monad thread to JavaScript, and propose solutions that help make the translation compatible with both the JavaScript language and the updated threading API. Based on the solutions we present, we shown an example implementation of the updated threading API based based on the free monad thread.

In order to translate the free monad thread to JavaScript, the implementation has to be adjusted for JavaScript's limitations. The free monad transformer relies on the functor and monad type classes, which can only be supported partially in JavaScript. The free monad thread relies on non-strictness to be able to chain threads together without evaluating them. The non-strictness has to be built into the JavaScript implementation manually.

Free monad threads can be implemented for JavaScript by implementing only the minimum amount of features from free monad threads. The free monad transformer only needs to support one functor, the thread instruction data type, and one monad for messaging to support free monad threads. By replacing the calls to the functor and monad related functions with specific implementations, the free monad threads can be implemented without type classes.

Parameterless functions can be used in place of the free monad thread's monad parameter. The free monad transformer uses it's monad type parameter's bind function in its own bind function to carry the underlying monad's effects along the free monad information. If the monad parameter's bind function is replaced with a function that composes parameterless functions, then the free monads could be composed without evaluating their underlying computations. Listing 6.6 contains the function `bindLazy` which follows the monad bind function's principles for composing parameterless functions. In the function, the result of the parameterless function `value` is chained with function `f` to create a new parameterless function that produces the output of the function `f`. The effects from the parameterless functions can be carried over through side effects, which is compatible with the updated threading API.

Listing 6.6. A composition method for parameterless function similar to monad's bind function.

```
function bindLazy(value, f) {
  return function() {
    return f(value())();
  };
}
```

Parameterless functions can also be used as threads in the updated threading API. The free monad transformer data type is a wrapper for a single monad value that produces a free value. Because the data type contains only one value and the type classes are resolved manually, the data type can be replaced with type it wraps. Because the JavaScript free monad threads use parameterless functions in place of the monad type parameter, the thread type can be replaced with a parameterless function that returns a free value.

The instruction and free types can be built using JavaScript objects. In listing 6.7, the `makeInstruction` function is used for creating new instructions. The function creates a new object that contains two fields, `mode` and `next`. The field `mode` acts as the discriminator field for the instructions, which is used to distinguish yield, fork, and done instructions from each other. The field `next` contains all of the next instructions. Because all of the instructions can only contain zero or more instructions depending on the instruction type, an array can be used for containing the next instructions. Similar approach can be used for the free type as well. The function `makeFree` in listing 6.7 is used for creating a new instance of free. The field `pure` separates free and roll values from each other, while the field `value` contains the actual value.

Listing 6.7. Functions for creating instruction and free values.

```
function makeInstruction(mode, next) {
  return { mode: mode, next: next };
}

function makeFree(pure, value) {
  return { pure: pure, value: value };
}

function pure(value)       { return makeFree(true,  value);       }
function roll(instruction) { return makeFree(false, instruction); }
```

Listing 6.8 contains the implementation for threading API's `bindThread` and `makeThread` functions. Both of the functions follow the same algorithm as their Haskell counterparts, but with slight changes. The JavaScript implementation uses parameterless functions in place of threads and the function `bindLazy` as their composition function. The function instructionMap is used in place of the instruction's map function.

Translations for the atom and fork functions are straightforward. Listing 6.9 contains the implementations for threading API's atom and fork functions. Both of the functions follow the same algorithms as the Haskell versions of the functions.

Listing 6.8. JavaScript implementations for threading API's bindThread and makeThread functions.

```
function bindThread(thread, f) {
  return bindLazy(thread, function(free) {
    if (free.pure) {
      return f(free.value);
    } else {
      return wrap(instructionMap(free.value, function(v) {
        return bindThread(v, f);
      }));
    }
  });
}

function makeThread(value) {
  return function() { return pure(value); };
}

function wrap(instruction) {
  return function() { return roll(instruction); };
}

function instructionMap(instruction, f) {
  return makeInstruction(instruction.mode, instruction.next.map(f));
}
```

6.4 Scheduler for the threading API implementation

The threading API implementation shown in the last section is enough for building interleavable application, but it requires a scheduler to be able to execute the applications. At the same time, the scheduler should not interfere with other tasks in the browser environment. In this section, we present a scheduler for the threading API implementation that's compatible with the browser environment.

The free monad thread scheduler shown in section 5.2 can be translated to JavaScript with slight changes. The original scheduler is a tail-recursive function, which evaluates a single

Listing 6.9. JavaScript implementations for threading API's `atom` and `fork` functions.

```
function atom(lazyValue) {
  return bindThread(lift(lazyValue), function(v) {
    return bindThread(yield(), function() {
      return makeThread(v);
    });
  });
}

function fork(thread) {
  return bindThread(cFork(), function(child) {
    return when(child, bindThread(thread, function() {
      return done();
    }));
  });
}

function yield() {
  return liftF(makeInstruction('yield', [null]));
}
function done() {
  return liftF(makeInstruction('done', []));
}
function cFork() {
  return liftF(makeInstruction('fork', [false, true]));
}

function when(p, thread) {
  return p ? thread : makeThread(null);
}

function lift(lazyValue) {
  return bindLazy(lazyValue, makeThread);
}

function liftF(instruction) {
  return wrap(instructionMap(instruction, makeThread));
}
```

step from a collection of threads, creates an updated version of the thread collection, and calls itself with the updated collection. Because tail-call optimization is not guaranteed in JavaScript, the scheduler must be made iterative instead of recursive in order to avoid call stack from growing uncontrollably.

The scheduler should not block browser events. Threading API applications should be able to respond to browser events such as mouse clicks and button presses. Thus the browser events need to be interleavable with the scheduler process. However, the events and the scheduler cannot be executed simultaneously. Instead, either the event or the scheduler must finish before the other one can be executed. Because threading API applications can potentially run endlessly, the scheduler must occasionally be paused in order to let the browser process events.

Listing 6.10 contains a scheduler implementation for the JavaScript free monad threads. The scheduler function, `run_`, evaluates each thread from the given array. The function removes each thread from the array one by one starting from the head of the array until the array is empty. Each of the removed threads are evaluated by calling the thread. If a thread produces new threads, the new threads are placed to the array. Instead of creating a new array on each evaluated thread, the same array is mutated. By mutating the array, the loop for evaluating the threads can be written without recursion.

In order to let the browser process events, the scheduler pauses itself after a fixed amount of steps. In listing 6.10, the scheduler function `run_` sets a counter for the amount of threads it evaluates. The counter is decremented by one for each evaluated thread. When the counter reaches zero, the function stops the evaluation loop and sets up another scheduler call to occur after a delay using browser's `setTimeout` function. The delayed scheduler call is given the remaining threads that are to be evaluated. The `setTimout` function ensures that the browser events have a change to be processed before the scheduler is started again.

Listing 6.10. Round robin scheduler for the JavaScript threading API implementation.

```
function run_(initialStepCount, threads) {
  var stepCount = initialStepCount;

  while(threads.length > 0 && stepCount > 0) {
    var thread = threads.shift();
    var freeValue = thread();

    if (!freeValue.pure) {
      var instruction = freeValue.value;
      var next = instruction.next;
      if (isYield(instruction)) {
        threads.push.apply(threads, next);
      } else if (isFork(instruction)) {
        threads.unshift(next[0]);
        threads.push.apply(threads, next.slice(1));
      }
    }
    stepCount -= 1;
  }

  if (threads.length > 0) {
    delay(function() { run_(initialStepCount, threads); });
  }
}

function run(startThread) {
  run_(20, [startThread]);
}

function delay(action) {
  setTimeout(action, 0, []);
}

function isYield(instruction) { return instruction.mode === 'yield'; }
function isFork(instruction)  { return instruction.mode === 'fork';  }
function isDone(instruction)  { return instruction.mode === 'done';  }
```

7 Conclusions and future work

We examined why concurrency is important to Elm and what are the requirements for introducing it to Elm's FRP system. Concurrency in a FRP system makes it possible to introduce long-running computations to the system without sacrificing the responsiveness of the rest of the system. We determined that interleaving should occur even during the computations for computing signal values regardless of how concurrency is implemented.

We produced a solution for interleaving programs in a single-threaded system in both Haskell and JavaScript. The solution includes an API that allows building interleavable applications in a style that resembles traditional multi-threaded programming style. We demonstrated how the API can be implemented in Haskell, translated the solution for JavaScript, and it integrated it to the browser environment. We also showed how Elm's FRP primitives can be implemented in Haskell using the threading API.

The translation from Elm to interleavable programs is yet to be solved. We briefly demonstrated the issues in translating Elm programs to automatically interleaved programs, and left the subject open for future research. As a temporary solution, we proposed exposing parts of the threading API to the developers in order to let developers build their own interleavable applications.

Bibliography

Canou, Benjamin, Emmanuel Chailloux, Jérôme Vouillon, et al. 2012. “How to run your favorite language in web browsers”. *WWW2012 Developer Track.*

Claessen, Koen. 1999. “A poor man’s concurrency monad”. *Journal of Functional Programming* 9 (03): 313–323. ISSN: 1469-7653. http://journals.cambridge.org/article_S0956796899003342.

Consortium, World Wide Web, et al. 2010. “HTML5 specification”. *Technical Specification, Jun* 24:2010.

Czaplicki, Evan. 2012. “Elm: Concurrent FRP for Functional GUIs”. *Master’s thesis, Harvard.*

Czaplicki, Evan, and Stephen Chong. 2013. “Asynchronous Functional Reactive Programming for GUIs”. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation,* 411–422. New York, NY, USA: ACM Press.

Dybvig, R.Kent, and Robert Hieb. 1989. “Engines from continuations”. *Computer Languages* 14 (2): 109–123. ISSN: 0096-0551. doi:http://dx.doi.org/10.1016/0096-0551(89)90018-0. http://www.sciencedirect.com/science/article/pii/0096055189900180.

Elliott, Conal, and Paul Hudak. 1997. “Functional reactive animation”. In *ACM SIGPLAN Notices,* 32:263–273. 8. ACM.

Flanagan, Cormac, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. “The essence of compiling with continuations”. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation,* 237–247. PLDI ’93. Albuquerque, New Mexico, USA: ACM. ISBN: 0-89791-598-4. doi:10.1145/155090.155113. http://doi.acm.org/10.1145/155090.155113.

Ganz, Steven E., Daniel P. Friedman, and Mitchell Wand. 1999. “Trampolined style”. *SIGPLAN Not.* (New York, NY, USA) 34, number 9 (): 18–27. ISSN: 0362-1340. doi:10.1145/317765.317779. http://doi.acm.org/10.1145/317765.317779.

Gonzales, Gabriel. 2013. *From zero to cooperative threads in 33 lines of Haskell code.* `http://www.haskellforall.com/2013/06/from-zero-to-cooperative-threads-in-33.html`.

Harrison, William L, and A Procter. 2005. "Cheap (but functional) threads". *Submitted to Journal of Functional Programming.*

Haynes, Christopher T, and Daniel P Friedman. 1984. "Engines build process abstractions". In *Proceedings of the 1984 ACM Symposium on LISP and functional programming,* 18–24. ACM.

Haynes, Christopher T., Daniel P. Friedman, and Mitchell Wand. 1984. "Continuations and Coroutines". In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming,* 293–298. LFP '84. Austin, Texas, USA: ACM. ISBN: 0-89791-142-3. doi:`10.1145/800055.802046`. `http://doi.acm.org/10.1145/800055.802046`.

Hieb, Robert, R.Kent Dybvig, and III Anderson ClaudeW. 1994. "Subcontinuations". *LISP and Symbolic Computation* 7 (1): 83–109. ISSN: 0892-4635. doi:`10.1007/BF01019946`. `http://dx.doi.org/10.1007/BF01019946`.

Hudak, Paul. 1989. "Conception, Evolution, and Application of Functional Programming Languages". *ACM Comput. Surv.* (New York, NY, USA) 21, number 3 (): 359–411. ISSN: 0360-0300. doi:`10.1145/72551.72554`. `http://doi.acm.org/10.1145/72551.72554`.

Kumar, Sanjeev, Carl Bruggeman, and R Kent Dybvig. 1998. "Threads yield continuations". *Lisp and Symbolic Computation* 10 (3): 223–236.

Levy, Yair, and Timothy J. Ellis. 2006. "A Systems Approach to Conduct an Effective Literature Review in Support of Information Systems Research." *Informing Science* 9:181–212. ISSN: 15214672. `http://search.ebscohost.com/login.aspx?direct=true&db=afh&AN=23852131&site=ehost-live`.

Liang, Sheng, Paul Hudak, and Mark Jones. 1995. "Monad transformers and modular interpreters". In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages,* 333–343. ACM.

Maki, Daisuke, and Hideya Iwasaki. 2007. “JavaScript multithread framework for asynchronous processing”. *IPSJ Transactions on Programming* 48 (12): 1–18.

Meyerovich, Leo A, Arjun Guha, Jacob Baskin, Gregory H Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. 2009. “Flapjax: a programming language for Ajax applications”. In *ACM SIGPLAN Notices,* 44:1–20. 10. ACM.

Nilsson, Henrik, Antony Courtney, and John Peterson. 2002. “Functional reactive programming, continued”. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell,* 51–64. Haskell ’02. Pittsburgh, Pennsylvania: ACM. ISBN: 1-58113-605-6. doi:10.1145/581690.581695. http://doi.acm.org/10.1145/581690.581695.

Peterson, John, Valery Trifonov, and Andrei Serjantov. 2000. “Parallel functional reactive programming”. In *Practical Aspects of Declarative Languages,* 16–31. Springer.

Ploeg, Atze van der. 2014. “Monadic functional reactive programming”. *ACM SIGPLAN Notices* 48 (12): 117–128.

Reppy, John H. 1993. “Concurrent ML: Design, application and semantics”. In *Functional Programming, Concurrency, Simulation and Automated Reasoning,* 165–198. Springer.

Swierstra, Wouter. 2008. “Data types à la carte”. *Journal of functional programming* 18 (04): 423–436.

Swierstra, Wouter, and Thorsten Altenkirch. 2007. “Beauty in the beast”. In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop,* 25–36. ACM.

Wadler, Philip. 1992. “The essence of functional programming”. In *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages,* 1–14. ACM.

Wan, Zhanyong, and Paul Hudak. 2000. “Functional reactive programming from first principles”. In *ACM SIGPLAN Notices,* 35:242–252. 5. ACM.

Wan, Zhanyong, Walid Taha, and Paul Hudak. 2001. “Real-time FRP”. In *ACM SIGPLAN Notices,* 36:146–156. 10. ACM.

Yoo, Danny, and Shriram Krishnamurthi. 2013. “Whalesong: running racket in the browser”. In *Proceedings of the 9th symposium on Dynamic languages,* 97–108. ACM.