Push-Based Signal-Function Functional Reactive Programming (Thesis Proposal)

Edward Amsden

May 21, 2012

Abstract

Functional Reactive Programming (FRP) is a promising class of abstractions for encoding interactive and time-dependent programs in functional languages, but has proven difficult to implement efficiently. Signal-function FRP is a subclass of FRP which does not provide direct access to the time-varying values by the programmer, but rather provides *signal functions* as first-class objects in the program. All signal-function implementations of FRP to date have utilized demand-driven or "pull-based" evaluation, producing output from the FRP system whenever the consumer of the output is ready. This greatly simplifies the implementation of signal-function FRP systems, but leads to inefficient and wasteful evaluation of the FRP system.

In contrast, an input-driven or "push-based" system evaluates the network whenever new input is available. This frees the system from evaluating the network when nothing has changed, and then only components necessary to react to the input are reevaluated. This form of evaluation has been partially applied to standard FRP systems [7] but not to signal-function FRP systems.

I propose the implementation of a push-based signal-function FRP system. The semantics of the system will be discussed, and its performance and expressiveness for practical examples of interactive programs will be compared to existing signal-function FRP systems.

1 Introduction

Most of the useful programs which programmers are asked to write must react to inputs which are not available at the time the program starts, and produce effects at many different times throughout the execution of the program. Examples of these programs include GUI applications, web browsers, web applications, video games, multimedia authoring and playback tools, operating system shells and kernels, servers, robotic control programs, and many others. This ability is called reactivity.

Functional reactive programming (FRP) is a promising class of abstractions for writing reactive programs in functional languages. FRP abstractions translate desirable properties of the underlying functional languages, such as higher-order functions and referential transparency, to reactive constructs, generally without modifications to the underlying language. This permits the compositional construction of reactive programs in purely functional languages.

Functional reactive programming was introduced, though not quite by name, with Fran [8], a compositional system for writing reactive animations. From the start, two key challenges emerged: the enforcement of *causality*[1] in FRP programs, and the efficient evaluation of FRP programs.

The first challenge was addressed by representing FRP programs, not as compositions of signals and events, but as compositions of *transformers* of behaviors and events called *signal functions*. By

[1] Causality is the property that a value in an FRP program depends only on present or past values. Obviously, a program which violates this property is impossible to evaluate, but in some FRP systems such programs are unfortunately quite easy to write down.

not permitting direct access to behaviors or events, but representing them implicitly, signal functions prohibit accessing past or future time values, only allowing access to values in the closure of the signal function and current input values. Signal function FRP programs are written by composing signal functions, and are evaluated using a function which provides input to the signal function and acts on the output values. This evaluation function is elided in existing literature on signal functions. A further advantage of signal function FRP programs is that they are more compositional, since additional signal functions may be composed on the input or output side of a signal function, rather than simply the output side in classic (non-signal-function) FRP systems.

The second challenge was addressed for classic FRP by the creation of a hybrid push-pull FRP system [7]. This system relied on the distinction between reactivity and time dependence to decompose behaviors into phases of constant or simply time-dependent values, with new phases initiated by event occurrences. However, this implementation did not address concerns of causality or compositionality. Further, the existence of events as first-class values in classic FRP forced the use of an impure and resource-leaking technique to compare events to determine which had the first occurrence after the current time. This technique made use of the Haskell `unsafePerformIO` function to fork two threads to compare the improving values of the next occurrence time for two events being merged. Further, since the classic FRP interface permits access to only the output of a behavior or event, and is always bound to its implicit inputs, the best the push-based implementation could do is to block when no computation needs to be performed. The computation cannot be suspended entirely as a value representation.

To address both challenges in a FRP implementation, it seems desirable to combine the approaches, creating a push-based signal-function FRP system. The current implementation approach for signal-function FRP is inherently pull-based [12], but recent work on N-ary FRP, a variant of signal-function FRP which enumerates inputs and distinguishes between events and behaviors, provides a representation which suggests a method of push-based evaluation. In previous implementations of FRP, events were simply option-valued signals. This approach has the advantage of being simple to implement in a pull-based setting, but does not have an obvious semantics for event merging, and necessitates pull-based evaluation, since there is no way to separate the events which may be pushed from the truly continuous values which must be repeatedly sampled.

N-ary FRP provides a model of a type system for a push-based signal-function FRP system. In order to represent signal functions as continuations on collections of inputs rather than on single values, we represent the signal vectors as types over which the signal function type is parameterized for both inputs and outputs. Indices into the signal vector are typed with the signal vector. These indices contain the actual input or output values and form elements for a collection of inputs or outputs. Obviously the collection need not have all possible elements, which forms the basis for evaluating only those parts of the network which depend on the updated values.

2 Background

The key abstraction of functional programming is the function, which takes an input and produces an output. Functions may produces functions as output and/or take functions as input. We say that functions are *first-class values* in a language.

Most reactive programs are written in an imperative style, using low-level and non-composable abstractions including callbacks or object-based event handlers, called by event loops. This ties the model of interactivity to low-level implementation details such as timing and event handling models.

FRP implies that a model should keep the characteristics of functional programming (i.e. that the basic constructs of the language should remain first-class) while incorporating reactivity into the language model. In particular, functions should be lifted to operate on reactive values, and functions themselves ought to be reactive.

The goal of FRP is to provide compositional and high-level abstractions for creating reactive

```haskell
-- Assumed primitives:
-- mousePosition, mouseLeftClick, microphoneAudio

-- onceE :: Event a -> Event a, gives only the first occurrence
-- restE :: Event a -> Event a, gives all except the first occurrence
-- switcher :: Behavior a -> Event (Behavior a) -> Behavior a,
--             act like the given behavior, then each successive
--             behavior in the event stream

-- Assumed pre-existing functions:
-- bandpassFilter :: Behavior Double -> Behavior Double -> Behavior Double -> Behavior Double
-- (Audio, low frequency, high frequency, audio output)

import Control.FRP -- Our theoretical classic FRP module
import Control.Applicative (<$>, <*>, pure) -- fmap and applicative functor application

mouseX :: Behavior Double
mouseX = (\(x, _) -> fromIntegral x / fromIntegral screenWidth) <$> mousePosition

mouseY :: Behavior Double
mouseY = fmap (\(_, y) -> fromIntegral y / fromIntegral screenHeight) mousePosition

highFreq :: Behavior Double
highFreq = fmap (\x -> 60 * exp (x * ln (20000 / 60)) mouseX

lowFreq :: Behavior Double
lowFreq = (\y highF -> 60 * exp (y * ln (highF / 60)) <$> mouseY <*> highFreq

audioOutput :: Behavior Double
audioOutput = let filtered = bandpass microphoneInput lowFreq highFreq
                  willStopFiltering =
                     (\evt -> filtered `switcher`
                              fmap (const (willStartFiltering $ restE evt))
                              $ onceE evt)
                  willStartFiltering =
                     (\evt -> microphoneInput `switcher`
                              fmap (const (willStopFiltering $ restE evt))
                              $ onceE evt)
              in willStopFiltering mouseLeftClick

main :: IO a
main = playBehaviorAsSound audioOutput
```

Figure 1: The example application expressed using theoretical classic FRP primitives.

```haskell
-- Assumed primitives:
-- arr :: (a -> b) -> SF a b
-- >>> :: SF a b -> SF b c -> SF a c
-- first :: SF a b -> SF (a, c) (b, c)
-- second :: SF a b -> SF (c, a) (c, b)
-- switch :: SF a (b, Event (SF a b)) -> SF a b
-- reactimateWithMouseAndSound :: SF (Event (), ((Int, Int), Double)) (Double) -> IO ()

-- Assumed given signal functions:
-- bandpass :: SF (Double, (Double, Double)) Double

import Control.AFRP -- Our theoretical AFRP

mouseCoordsToDoubles :: SF (Int, Int) (Double, Double)
mouseCoordsToDoubles =
  arr (\(x, y) -> (fromIntegral x / fromIntegral screenWidth,
                   fromIntegral y / fromIntegral screenHeight))

highFreq :: SF Double Double
highFreq = arr (\x -> 60 * exp( x * ln (20000 / 60)))

lowFreq :: SF (Double, Double) Double
lowFreq = arr (\(high, y) -> 60 * exp (y * ln (high / 60)))

lowHighFreq :: SF (Double, Double) (Double, Double)
lowHighFreq = first highFreq >>>
              arr (\(high, y) -> (high, (high, y))) >>>
              second lowFreq

mouseCoordsToLowHighFreqs :: SF (Int, Int) (Double, Double)
mouseCoordsToLowHighFreqs = mouseCoordsToDoubles >>> lowHighFreq

toggleFilter =
  let filterOn =
        switch (arr (\(evt, (mouse, audio)) ->
                       ((audio, mouse), fmap (const filterOff) evt)) >>>
                first (second mouseCoordsToLowHighFreqs >>> bandpass))
      filterOff = switch (arr (\(evt, (_, audio)) ->
                                 (audio, fmap (const filterOn) evt)))
  in filterOn

main :: IO ()
main = reactimateWithMouseAndSound toggleFilter
```

Figure 2: The example application expressed using theoretical signal-function FRP primitives.

programs. The key abstractions of FRP are behaviors or signals[2], which are time-dependent values defined at every point in continuous time, and events, which are values defined at countably many points in time. An FRP system will provide functions to manipulate events and signals and to react to events by replacing a portion of the running program in response to an event. Behaviors and events, or some abstraction based on them, will be first class, in keeping with the spirit of functional programming. Programs implemented in functional reactive models should of course be efficiently executable. This has proven to be the main challenge in implementing Functional Reactive Programming.

The two general approaches to FRP are "classic" FRP, where behaviors and signals are first-class and reactive objects, and "signal-function" FRP, where transformers of signals and events are first-class and reactive objects.

In order to illustrate the differences, it will be helpful to consider an example. In this case, we will consider a toy bandpass audio filtering application which sets the low- and high-pass frequencies based on mouse position.

We would like the low-pass frequency to be exponentially related to the y-coordinate of the mouse, and similarly for the high-pass frequency and x-coordinate. Further, left-clicking the mouse should toggle the effect.

2.1 Classic FRP

The first FRP system, Fran [8], was introduced as a system for compositionally describing interactive animations. Though the term "Functional Reactive Programming" was not used in this work, it was the first to introduce the concept of first-class behaviors and events as an abstract model for reactive programs. The key motivation was to separate the implementation of reactivity from the modelling of reactivity, so that a programmer could be freed from details such as sampling time-varying values, time-slicing to simulate updating values in parallel, and sampling of inputs.

The first implementation of FRP outside the Haskell language was Frappé, an FRP implementation for the Java Beans framework. Frappé built on the notion of events and *bound properties* in the Beans framework, providing abstract interfaces for FRP events and behaviors, and combinators as concrete classes implementing these interfaces. The evaluation of Frappé used Bean components as sources and sinks, and the implementation of Bean events and bound properties to propagate changes to the network [4].

The FrTime[3] [3] language extends the Scheme evaluator with a mutable dependency graph, which is constructed by program evaluation. This graph is then updated by signal changes. FrTime does not provide a distinct concept of events, and chooses branches of the dependency graph by conditional evaluation of signals, rather than by the substitution of signals used by FRP systems.

The Reactive [7] system is a push-based FRP system with first-class behaviors and events. The primary insight of Reactive is the separation of reactivity (or changes in response to events whose occurrence time could not be know beforehand) from time-dependence. This gives rise to *reactive normal form*, which represents a behavior as a constant or simple time-varying value, with an event stream carrying values which are also behaviors in reactive normal form. Push-based evaluation is achieved by forking a Haskell thread to evaluate the head behavior, while waiting on the evaluation of the event stream. Upon an event occurrence, the current behavior thread is killed and a new thread spawned to evaluate the new behavior. Unfortunately, the implmentation of Reactive uses a tenuous technique which depends on also forking threads to evaluate two haskell values concurrently in order to implement event merging. This relies on the library author to ensure consistency when this technique is used, and leads to thread leakage when one of the merged events is itself the merging of other events.

[2]Behaviors are generally referred to as *signals* in signal function literature. This is unfortunate, since a signal function may operate on events, signals, or some combination of the two.

[3]FrTime is available in the current version of Dr Racket (5.2.1).

The reactive-banana [2] library is a push-based FRP system designed for use with Haskell GUI frameworks. In particular, it features a monad for the creation of behaviors and events which may then be composed and evaluated. This monad includes constructs for binding GUI library constructs to primitive events. It must be "compiled" to a Haskell IO action for evaluation to take place. The implementation of reactive-banana is similar to FrTime, using a dependency graph to update the network on event occurences. Reactive-banana eschews generalized switching in favor of branching functions on behavior values, similarly to FrTime, but maintains the distinction between behaviors and events. Rather than a generalized switching combinator which allows the replacement of arbitrary behaviors, reactive-banana provides a step combinator which creates a stepwise behavior from the values of an event stream.

A recent thesis [6] described Elm, a stand-alone language for reactivity. Elm provides combinators for manipulating discrete events, and compiles to Javascript, making it useful for client-side web programming. However, Elm does not provide a notion of switching or of continuous time behaviors, though an approximation is given using discrete-time events which are actuated at repeated intervals specified during the event definition. The thesis asserts that Arrowized FRP (signal-function FRP, Section 2.2) can be embedded in Elm, but provides no support for this assertion.

In classic FRP systems, it is common for the Behavior type to form an applicative functor, and for the Event type to form a functor. To construct our example application, we lift the exponential function, with a tuning constant, to a behavior, which we then apply to the x and y values of the mouse. Finally, we assume a function which takes a behavior for audio data, lowpass frequency, and highpass frequency, and produces a behavior for the filtered audio data. This function would be written in terms of the integration primitive common to most FRP libraries. Figure 1 provides an example.

2.2 Signal Function FRP

An alternative approach to FRP was first proposed in work on Fruit [5], a library for declarative specification of GUIs. This library utilized the abstraction of Arrows [9] to structure *signal functions*. Arrows are abstract type constructors with input and output type parameters, together with sequential (`>>>`) and parallel (`first` and `second`) composition, branching (`***`), and lifting (`arr`) functions to the Arrow type. Signal functions are the first-class abstraction in this approach, they represent time-dependent and reactive transformers of events and signals, which are themselves not first class, since such values cannot be directly manipulated by the programmer. This approach has two motivations: it increases modularity since both the input and output of signal functions may be transformed, as opposed to signals or events which may only be transformed in their output, and it avoids a large class of time and space leaks which have emerged when implementing FRP with first-class signals and events.

Similarly to FrTime, the netwire [15] library eschews dynamic switching, in this case in favor of *signal inhibition*. Netwire is written as an arrow transformer, which, together with the Kliesli arrow instance for the IO monad[4], permits it to lift IO actions as sources and sinks at arbitrary points in a signal function network. Signal inhibition is accomplished by making the output of signal functions a monoid, and then combining the outputs of signal functions. An inhibited signal function will produce the monoid's zero as an output. Primitives have defined inhibition behavior, and composed signal functions inhibit if their outputs combine to the monoid's zero.

Yampa [11] is an optimization of the Arrowized FRP system first utilized with Fruit (see above). The implementation of Yampa makes use of Generalized Algebraic Datatypes to permit a much larger class of type-safe datatypes for the signal function representation. This representation, together with "smart" constructors and combinators, enables the construction of a self-optimizing arrowized FRP system. Unfortunately, the primary inefficiency, that of unnecessary evaluation steps due to pull-based evaluation, remains. Further, the optimization is ad-hoc and each new optimization requires the

[4] Any monad forms a Kliesli arrow.

addition of new constructors, as well as the updating of every primitive combinator to handle every combination of constructors. However, Yampa showed noticeable performance gains over previous Arrowized FRP implementations.

A recent PhD thesis [14] introduced N-Ary FRP, a technique for typing Arrowized FRP systems using dependent types. The bulk of the work consisted in using the dependent type system to prove the correctness of the FRP system discussed. However, a key typing construct called signal vectors was presented, which permits the distinction of signal and event types at the level of the FRP system, instead of making events merely a special type of signal. This distinction is important to the approach I am proposing, as will be made clear in Section 3.

In the signal-function FRP example, we create a signal function with inputs for the audio data, mouse coordinates, and mouse clicks. These inputs are then routed using the composition combinators to the switching combinator (for toggling the effect) or the input of the filter (Figure 2).

2.3 Comparison of Examples

In the example for classic FRP (Figure 1), we note pure values which are not referentially transparent, since they depend on real-world values. While notation in this manner is appealing, the behavior and event streams must be implemented either by "compiling" to a dependency graph with continuations for sampling, as in reactive-banana, or by using backdoors which allow IO actions to be treated as pure values, as in most other classic FRP systems. Further, note that event streams, in particular, must be explicitly split in order to avoid time leaks and violation of causality. Finally, our program is not modular, since it is not parameterized on its inputs, and for the same reason,it cannot be suspended as a state value without reference to these input sources. This requires evaluation to take place in a single event loop.

By contrast, the signal-function FRP program has somewhat less appealing notation (an issue addressed for Arrows [13]), but requires no explicit mechanism to avoid time leaks. Further, the program is completely modular, each signal function may be further composed sequentially either on its input, its output, or both; or in parallel. Finally, note that the evaluation function, not the signal function, depends on the real-world inputs, thus, given a stepping rather than looping evaluation function, it would be entirely possible to suspend a signal function as a state value.

2.4 Outstanding Challenges

At present, there are two key issues apparent with FRP. First, while signal-function FRP is inherently safer and more modular than classic FRP, it has yet to be efficiently implemented. Second, the interface between FRP programs and the many different sources of inputs and sinks for outputs available to a modern application writer remains ad-hoc and is in most cases limited by the implementation. One key exception to this is the reactive-banana system, which provides a monad for constructing primitive events and behaviors from which an FRP program may then be constructed. However, this approach is still inflexible as it requires library support for the system which the FRP program will interact with. Further, classic FRP programs are vulnerable to time leaks and violations of causality due to the ability to directly manipulate reactive values.

3 Proposed Work

Recent work on types for signal-function FRP systems has yielded N-ary FRP [14], which types signal function inputs and outputs with *signal vectors* rather than types of values. Signal vectors are types: a signal or event type constructor taking a value type, or a product type of signal vectors. A signal function then uses signal vectors for its input and output types. Since the actual values which are supplied to the representation of a signal function also parameterize over signal vectors, this representation permits us to restrict the type of values which may be supplied to the signal function,

```haskell
-- Signal Vectors
data SVEmpty
data SVSignal :: * -> *
data SVEvent  :: * -> *
data SVAppend :: * -> * -> *

-- Signal Vector indexes
data SVIndex :: (* -> *) -> * -> * where
  SVISignal :: a -> SVIndex p (SVSignal p a)
  SVIEvent  :: a -> SVIndex p (SVEvent p a)
  SVILeft   :: SVIndex p svl -> SVIndex p (SVAppend svl svr)
  SVIRight  :: SVIndex p svr -> SVIndex p (SVAppend svl svr)
```

Figure 3: Examples data types for push-based signal functions

```haskell
-- "Identity" type constructor
newtype Id a = Id a

-- Signal Function (initialized) datatype
data SFInitialized :: * -> * -> * where
  SFInitialized :: (DTime -> ([SVIndex Id svOut], SFInitialized svIn svOut)) ->
                   (SVIndex Id svIn -> ([SVIndex Id svOut], SFInitialized svIn svOut)) ->
                   SFInitialized svIn svOut
```

Figure 4: Example data type for push-based signal functions

even though no values take the type of the signal vector as their whole type. This permits individual components of a signal function's input or output to be designated as an event or as a signal, and provides precisely the flexibility necessary to create a push-based implementation.

When signal functions are typed using value representations, where events are simply option values, the entire input and output must be constructed for every step of a signal function. In particular, there is no obvious mechanism for producing signal values which contain only the updated components of the signal, together with actually occurring events.

By making use of phantom types which designate components of a signal vector, together with Generalized Algebraic Datatypes (GADTs) which permit the specification of the input and output types of a signal function and type-safe indexing of the input and output components, individual components of the input of a signal function may be processed as they become available, and may produce individual components of the output.

I will create a library of combinators for creating signal-function FRP systems which may be evaluated when input is provided, and which is typed with phantom types describing signal vectors (Figure 3). These combinators will produce continuations which accept values together with signal vector indexes and produce replacement continuations together with collections of signal vector indexed values (Figure 4). The indexes allow combinators to use only the continuations necessary to process a specific input, as opposed to requiring the entire input to be constructed for a signal function. In particular, sequential composition will detect output changes or event occurrences from the left-side signal function and only evaluate the right-side signal function as necessary, parallel composition (`first` and `second`) will only evaluate their left or right side signal function depending on the index, and routing combinators will produce outputs for a given index in the output signal vector only when there is an input presented on the corresponding index in the input signal vector.

Further, I will create an evaluation interface for signal functions created with this library, which will provide monadic actions to push events into the signal function, activating predefined callbacks

```haskell
-- SF Evaluation monad transformer

data SFEvalMonadT :: * -> * -> * -> * -> *

instance Monad (SFEvalMonadT svIn svOut m)

instance SFMonadTransformer (SFEvalMonadT svIn svOut)

initSF :: SF svIn svOut -> SFState svIn svOut

runSFEvalMonadT :: (EventOnly svIn, EventOnly svOut, Monad m) =>
                   SFState svIn svOut ->
                   [SVIndex (-> m ()) svOut] ->
                   SFEvalMonadT svIn svOut m a ->
                   m (a, SFState svIn svOut)

push :: (EventOnly svIn, EventOnly svOut) =>
                SVIndex Id svIn ->
                SFEvalMonadT svIn svOut m ()
```

Figure 5: Example functions and monadic combinators for evaluation of push-based signal functions

on output events (Figure 5). Signals will not be permitted on the inputs or outputs of evaluated signal functions, but only internally to the signal function.

I will provide a formal semantic description for the combinator library and evaluation interface which will permit me to argue that the new implementation solves a particularly thorny problem with "merging" events.

This implementation will be compared to the existing pull-based signal-function implementation, Yampa, by implementing a simple 2-dimensional game with both Yampa and the new implementation, combining both with OpenGL [1] and GLUT [10] for graphics. CPU usage, framerate, and response latency will be measured for both systems and comparisons given. In addition, I expect the proposed implementation to be usable with GLUT and other event-loop based libraries, or even combinations of event-based libraries, simply and without an additional thread for the FRP system, something not possible with the present evaluation function in Yampa or with current classic FRP systems.

3.1 Timeline

April 2, 2012	Submit draft proposal with proposed work, timeline, and introduction for review by Professor Fluet.
April 23, 2012	Complete initial implementation of FRP library, begin debugging and testing.
April 30, 2012	Finish proposal including background. Submit final proposal to Professor Fluet, Professor Nunes-Harwitt and Professor Butler to convene committee.
May 31, 2012	Submit ongoing work writeup to Haskell Symposium. Include semantics and an interactive demonstration.
June 11, 2012	Complete demonstration application (2D graphical game) in new FRP library and in Yampa.
June 18, 2012	Complete testing and comparisons of demonstration application.
June 25, 2012	Thesis draft submission.
July 8, 2012	Thesis defense.

References

[1] Akeley, K., and Segal, M. *The OpenGL Graphics System: A Specification*, 2011.

[2] Apfelmus, H. reactive-banana library. `http://hackage.haskell.org/package/reactive-banana`.

[3] Cooper, G. H., and Krishnamurthi, S. Embedding dynamic dataflow in a call-by-value language. In *European Symposium on Programming* (2006), pp. 294–308.

[4] Courtney, A. Frappé: Functional reactive programming in Java. In *Proceedings of the Third International Symposium on Practical Aspects of Declarative Languages* (London, UK, UK, 2001), PADL '01, Springer-Verlag, pp. 29–44.

[5] Courtney, A., and Elliott, C. Genuinely functional user interfaces. In *In Proceedings of the 2001 Haskell Workshop* (2001), pp. 41–69.

[6] Czaplicki, E. Elm: Concurrent FRP for functional GUIs. `http://www.testblogpleaseignore.com/wp-content/uploads/2012/03/thesis.pdf`, 2012.

[7] Elliott, C. Push-pull functional reactive programming. In *Haskell Symposium* (2009).

[8] Elliott, C., and Hudak, P. Functional reactive animation. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming* (New York, NY, USA, 1997), ICFP '97, ACM, pp. 263–273.

[9] Hughes, J. Generalising monads to arrows. *Science of Computer Programming 37*, 13 (2000), 67 – 111.

[10] Kilgard, M. J. *The OpenGL Utility Toolkit (GLUT) Programming Interface API Version 3*, 1996.

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

[12] Nilsson, H., Courtney, A., and Peterson, J. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell* (New York, NY, USA, 2002), Haskell '02, ACM, pp. 51–64.

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

[14] Schulthorpe, N. *Towards Safe and Efficient Functional Reactive Programming*. PhD thesis, University of Nottingham, UK, 2011.

[15] Söylemez, E. netwire library. `http://hackage.haskell.org/package/netwire-3.1.0`.