Eventless Reactivity from Scratch

Gergely Patai

Budapest University of Technology and Economics, Budapest, Hungary
patai@iit.bme.hu

Abstract. Functional reactive programming is a conceptual extension to traditional functional programming that promises to make the description of interactive software easy to express in a declarative way. However, reactive libraries promoting an applicative style all seem to suffer from various implementation issues, which prevents programmers from exploring the design space of functional reactive *applications*, and relevant research effort is mostly limited to establishing the foundations of various libraries.
This paper presents Elerea, a simple reactive library created in order to fill this hiatus in the Haskell world. The library offers first-class continuous higher-order signals and the ability to resolve feedback loops without requiring the programmer to insert explicit delays. The signal network can be fully dynamic, and stateful signals are created through a monadic interface that makes their start time explicit. Initial experience suggests that Elerea manages to fulfil its goal and provides a comfortable environment for application-level experiments.

1 Introduction

Functional reactive programming (FRP) is a programming paradigm conceived with the aim of describing various kinds of interactive systems in a clean, declarative manner. As FRP has been proposed for a wide range of application domains, there are great differences between various systems. The common denominator is that these systems all try to hide the specifics of temporal behaviour behind some abstraction and deal with the whole life cycle of a time-varying value (usually referred to as 'signal' or 'behaviour' in FRP literature) as one entity instead of concentrating on separate points, and sometimes they also attempt to collapse related events to be labelled with a single name. For instance, 'current window size' can be a behaviour, while 'all key presses received by the main window' can be an event using this terminology.

While FRP is a language independent concept, most systems are built on top of Haskell. There are three important libraries to consider at the moment: Yampa [5], Grapefruit [13] and Reactive [6]. Yampa is an arrow-based library, which allows the user to describe the application as a data-flow network built from signal functions that consume and produce a stream and can optionally maintain an internal state. Reactive is intended to provide a functionally pure applicative interface, where signals are first class entities that can be combined just as freely

as ordinary values in any functional program. Grapefruit is conceptually the most complex system, where signals are used to connect effectful circuits through an arrow interface, but more complex signals can be derived from simpler ones using applicative combinators.

Out of the three, so far only Yampa has reached the level of practical usefulness, and it has indeed sparked interesting projects like Frag [3] or YampaSynth [11]. However, this means that there is no applicative style FRP library available to Haskellers to experiment with, while this style preserves most of the nice properties of functional programming in general. Therefore, it became a basic requirement for Elerea to give the user a similar experience and open the way for people to experiment with applicative reactive design. For simplicity, the library only deals with continuous signals, and it has no concept of event, hence its name Elerea – short for 'eventless reactivity'.

Elerea is mainly an experiment to see where we can get if we start the exploration from the user end instead of defining a clean core to build on. This paper introduces Elerea with the help of some examples, guides us through its internals and explains the thought process leading to the final design.

2 Signal Interface

2.1 Static Networks

An Elerea signal is a time-varying value of any type, and most importantly it is a first-class citizen. Conceptually, a value of type *Signal a* (denoted as $\mathsf{S}\ a$ from now on) can be thought of as a function of type $\mathit{Time} \to a$, which is sampled at non-decreasing points of time. *Time* is just a synonym for a suitable floating point type.

Signals are applicative functors [14]. In practice, this means that ordinary functions of any arity can be lifted to perform a point-wise application on signals, i.e. the *Applicative* instance of S behaves the same way as the corresponding instance of functions with a given input type (like *Time*).

For instance, given two signals s_1 and s_2, we can define their point-wise sum by lifting $(+)$ to act on them:

$$s = \mathit{liftA2}\ (+)\ s_1\ s_2$$

Elerea defines default instances for signals carrying certain numeric types, so several operations can be lifted automatically. In particular, the alternative definition $s = s_1 + s_2$ would also work in this case.

Of course the real power of FRP lies in manipulating stateful signals. The simplest way to define such a signal is provided by the *stateful* combinator, which can be used to define a source without an input:

$$\mathit{stateful} :: a \to (\mathit{DTime} \to a \to a) \to \mathsf{SM}\ (\mathsf{S}\ a)$$

DTime is a synonym for *Double*, and it denotes the time elapsed between consecutive sampling points. The return type reveals a vital design decision of Elerea:

stateful signal combinators return values in a monad called SM (*SignalMonad*), whose role is to make it clear when the resulting signals start. This has been one of the central problems of FRP, and various systems settle for different answers. We will see later what SM looks like inside, but from a user standpoint it is enough to think of it as the only possible source of stateful signals.

The signal created by *stateful* x_0 f emits x_0 when it is first sampled, and every time it is sampled it modifies its internal state (the next output) by applying f to the time elapsed since the previous sampling and the previous value. For instance, we can create a timer signal by simply accumulating the time differences:

$$
\begin{aligned}
&\mathit{makeTimer} :: \mathit{Double} \to \mathsf{SM}\ (\mathsf{S}\ \mathit{Double}) \\
&\mathit{makeTimer}\ t_0 = \mathit{stateful}\ t_0\ (+) \\
&\mathit{signalWithInternalTimer} = \mathbf{do} \\
&\quad \mathit{timer} \leftarrow \mathit{makeTimer}\ 0 \\
&\quad \ldots
\end{aligned}
$$

The *timer* signal can be thought of as the identity function, if we consider signals functions of time.

While these stateful signals are nice, they do not really allow us to build interesting systems that can react to user input in a stateful way even when used together with the applicative combinators. There is a more general construct, the transfer function, which also takes the value of another signal into consideration when calculating the next state:

$$\mathit{transfer} :: a \to (\mathit{DTime} \to t \to a \to a) \to \mathsf{S}\ t \to \mathsf{SM}\ (\mathsf{S}\ a)$$

The signal constructed by *transfer* x_0 f s has an initial internal state x_0, and every time it is sampled with a time difference dt, its next state is calculated by applying f to dt, the current value of s, and the previous state. It is slightly different from *stateful* in that the first output of this signal is the result of the first application, i.e. x_0 is not observed on the output (except for certain conditions, as we will see later). In other words, the value of signal s at any given moment affects the value of the constructed signal at the same moment.

A simple use for the transfer function is defining a generic integral combinator:

$$
\begin{aligned}
&\mathit{integral} :: \mathit{Fractional}\ a \Rightarrow a \to \mathsf{S}\ a \to \mathsf{SM}\ (\mathsf{S}\ a) \\
&\mathit{integral}\ x_0\ s = \mathit{transfer}\ x_0\ (\lambda dt\ x\ x_0 \to x_0 + x * \mathit{realToFrac}\ dt)\ s
\end{aligned}
$$

One can easily ignore the time step if the transfer function does not need it. For instance, a transfer function that remembers the maximum of its input signal can be defined in the following way:

$$
\begin{aligned}
&\mathit{keepMax} :: \mathit{Ord}\ a \Rightarrow a \to \mathsf{S}\ a \to \mathsf{SM}\ (\mathsf{S}\ a) \\
&\mathit{keepMax}\ x_0\ s = \mathit{transfer}\ x_0\ (\mathit{const}\ \mathit{max})\ s
\end{aligned}
$$

Even though *transfer* can be used for point-wise function application, it is better to use *fmap* for that purpose, since it is more efficient and easier to compose.

2.2 Automatic Delays

Integrals can be used to define trig functions from scratch:

$$
\begin{aligned}
&\mathit{trigs} :: \mathit{Fractional}\ t \Rightarrow \mathsf{SM}\ (\mathsf{S}\ t, \mathsf{S}\ t)\\
&\mathit{trigs} = \mathbf{mdo}\\
&\quad \mathit{sine} \leftarrow \mathit{integral}\ 0\ \mathit{cosine}\\
&\quad \mathit{cosine} \leftarrow \mathit{integral}\ 1\ (-\mathit{sine})\\
&\quad \mathit{return}\ (\mathit{sine}, \mathit{cosine})
\end{aligned}
$$

Note the **mdo** notation, which is an extension of **do** notation allowing value recursion [8]. Since SM is an instance of *MonadFix*, it is possible to define several signals in terms of each other, just as one can do with ordinary values. This feature is essential, because such a need seems to come up routinely in practice.

Since *transfer* is immediate, the above definition looks like it would cause an infinite loop as soon as we request a sample from either trig signal. In other systems, this problem is normally solved by inserting a delay element in every feedback loop. However, Elerea is capable of inserting such delays automatically, during runtime. There is one limitation, mainly for the sake of efficiency: these delays can only affect *transfer* nodes, i.e. loops that are built from solely stateless combinators are not resolved. Actually, if they were, that would violate the laws of applicative functors.

It is not possible to tell in advance where exactly these extra delays will be placed, because this operational detail is deliberately left undefined. If a library user does not like this uncertainty, they can prevent it by inserting explicit delays with the following combinator:

$$\mathit{delay} :: a \rightarrow \mathsf{S}\ a \rightarrow \mathsf{SM}\ (\mathsf{S}\ a)$$

In short, *delay* takes a value to be output upon the first sampling, and a signal to delay by one step. As far as the values of the streams are concerned, $\mathit{delay}\ x_0$ is equivalent to $\mathit{transfer}\ x_0\ (\mathit{const}\ \mathit{const})$. However, the latter would not prevent automatic delays, since the evaluator cannot tell from an opaque function that the current output depends only on past input, therefore *delay* is a primitive combinator.

2.3 Embedding and Animating Signals

In order to get to the top-level signal, Elerea provides a conversion function, which allows the programmer to embed the reactive code in an imperative framework:

$$\mathit{createSignal} :: \mathsf{SM}\ a \rightarrow \mathit{IO}\ a$$

If we have a signal in our hands, we can sample it using the *superstep* function, which expects a signal and a time step, and produces a sample in return:

$$\mathit{superstep} :: \mathsf{S}\ a \rightarrow \mathit{DTime} \rightarrow \mathit{IO}\ a$$

It is up to the user of the library to execute *superstep* in a loop and supply the time differences. Elerea does not provide any higher level construct for this purpose, so it can easily interface with basically any imperative framework.

Since we are programming reactive systems, we need a way to provide external input to the signal network. The *external* primitive returns a signal and an *IO* action that can be used to feed it.

$$external :: a \rightarrow IO\ (\mathsf{S}\ a, Sink\ a)$$

Sink a is just a synonym for $a \rightarrow IO\ ()$. Whenever the sink is passed a value, the corresponding signal will be delivering that value until the next call to the sink. The user-defined loop is usually the best place to update external signals.

It would be difficult to include a meaningful example using these embedding functions in the paper. However, the library comes with a fully functional and richly commented, yet still rather small OpenGL Breakout example in the *elerea-examples* package available through cabal-install[1], which shows what a real-life Elerea application might look like.

2.4 Dynamic Networks

Static networks can solve simple problems, but a complex task is often easier to model with signals whose life cycles are shorter than that of the whole network. Stateless combinators pose no problem, as their creation time does not affect their behaviour. In fact, they can be completely lazy just like any ordinary value. On the other hand, it is essential that we know exactly when a stateful signal comes to life, e.g. which moment the *time* signal defined above is measuring from.

Using a monadic interface can provide an convenient solution to this problem. Elerea defines a single construct that can be used to construct arbitrary signals on the fly:

$$generator :: \mathsf{S}\ Bool \rightarrow \mathsf{S}\ (\mathsf{SM}\ a) \rightarrow \mathsf{S}\ (Maybe\ a)$$

A *generator* takes a stream of booleans and a stream of SMs, and extracts the latter at every sampling point the former evaluates to true. The resulting signal has an option type: if the boolean was false, it carries *Nothing*, otherwise it carries the value produced by the monad wrapped in *Just*. The concrete type in place of the *a* can be any complex data structure containing one or more signals.

For instance, we can deliver a new timer signal every time a boolean condition holds ($cond :: \mathsf{S}\ Bool$); we will use the *makeTimer* function defined above:

$$timers :: \mathsf{S}\ (Maybe\ (\mathsf{S}\ Double))$$
$$timers = generator\ cond\ (pure\ (makeTimer\ 0))$$

[1] Available at `http://hackage.haskell.org/package/elerea-examples`. If you have the Haskell Platform installed, just type `cabal unpack elerea-examples`, and check the `doc` directory.

Elerea also provides a simple latch defined in terms of *transfer* that takes an initial value and a *Maybe* stream, and holds on the last input that was wrapped in *Just*:

```
storeJust :: a → S (Maybe a) → SM (S a)
```

With its help, we can get rid of the *Maybe* layer:

```
timers′ :: SM (S (S Double))
timers′ = do
  init ← makeTimer 0
  storeJust init timers
```

But this is a higher-order signal, and neither of the combinators mentioned so far can get to the samples of the inner signal. This is where the *sampler* primitive can be used:

```
sampler :: S (S a) → S a
```

A *sampler* can collapse a higher-order signal by exposing the inner signal delivered at the given moment. This definition suggests that *sampler* might be used as *join* to define a *Monad* instance of S. However, it is not clear yet whether this is a valid assumption. In any case, the current implementation does not support such a substitution – this was tested by attempting to redefine the *Applicative* instance in terms of *sampler* and *fmap*.

The use of higher-order signals also uncovers a basic operational detail of Elerea: a signal is only *aged* (its internal state updated) if the signal sampled by *superstep* depends on its current output. Let us consider the following example:

```
toggleTimer :: S Bool → SM (S Double)
toggleTimer sel = do
  timer1 ← makeTimer 0
  timer2 ← makeTimer 10
  let selectTimer b = if b then timer1 else timer2
  return (sampler (fmap selectTimer sel))
```

What happens if we sample a *toggleTimer* applied to some boolean signal? Depending on the current value of the *sel* signal we can observe the current value of either $timer_1$ or $timer_2$, and when *sel* is toggled, we can see that the timer we ignored so far has not advanced.

If this is not the behaviour we want, we have to sample the internal timers at the same level as the enclosing *toggleTimer*. The straightforward way to achieve this is not using higher-order signals at all (`$` stands for inline *fmap* and `⊛` is lifted function application):

```
toggleTimer′ :: S Bool → SM (S Double)
toggleTimer′ sel = do
  timer1 ← makeTimer 0
```

```
  timer2 ← makeTimer 10
  let ifte b x1 x2 = if b then x1 else x2
  return (ifte <$> sel <*> timer1 <*> timer2)
```

However, if we still want to model our network with higher-order signals, we can use the *keepAlive* construct, whose type is $\mathsf{S}\ a \to \mathsf{S}\ t \to \mathsf{S}\ a$, and it simply ensures that both signals passed to it are aged while only delivering the value of the first. So the last line of *toggleTimer* could be modified the following way:

```
return $ sampler (fmap selectTimer sel)
  `keepAlive` timer1 `keepAlive` timer2
```

It is useful to keep in mind that automatic delays work seamlessly even in dynamic networks as long as all cyclic dependencies have at least one stateful node somewhere in the loop.

2.5 Signal Collections

We do not need any additional construct to be able to model a dynamic collection of entities as a dynamically changing collection of signals. As a simple example, we will define a list of three timers started at different points, each of which is removed after reaching a preset limit. We use *delay* to be able to initialise the collection ($timers_0$) and describe its next state ($timers'$) in terms of the current one (*timers*). Both *timers* and $timers'$ have type $\mathsf{S}\ [\mathsf{S}\ Double]$, i.e. they are signals carrying a collection of timers.

```
timerList :: SM (S [Double])
timerList = mdo
  timers0 ← mapM makeTimer [40, 20, 70]
  timers ← delay timers0 timers'
  let ts = sampler (sequenceA <$> timers)
      timers' = map snd ∘ filter ((<80) ∘ fst) <$> (zip <$> ts <*> timers)
  return ts
```

Note that we take advantage of the fact that lists are instances of *Traversable*, which provides the *sequenceA* operation that essentially turns the structure inside-out by swapping the traversable and the applicative layer:

```
sequenceA :: (Traversable t, Applicative f) ⇒ t (f a) → f (t a)
```

In this particular context, the traversable layer is the list and the applicative layer is the signal, so the specialised type of *sequenceA* is $[\mathsf{S}\ Double] \to \mathsf{S}\ [Double]$. Under the hood, this function simply traverses the data structure and replaces every constructor with its lifted version, thereby moving it behind the applicative abstraction.

In order to construct $timers'$, we simply pair up timers with their current values and keep only those that have not yet reached the limit.

The *timerList* signal can be easily tested by repeated sampling:

```
timerTest :: IO [[Double]]
timerTest = do
  tl ← createSignal timerList
  replicateM 20 $ superstep tl 3
```

3 Implementation

3.1 Data Structures

Each signal constructor corresponds to a node in the network, therefore the dataflow network has essentially the same structure as the call graph on the user end. The nodes are mutable variables, and their value changes whenever the signal in question is aged. The actual type behind each signal is the following:

```
newtype S a = S (IORef (SignalTrans a))
```

SignalTrans is a wrapper that keeps track of the sampling phases during each superstep. Its definition is the following:

```
data SignalTrans a = Ready (SignalNode a)
                   | Sampling (SignalNode a)
                   | Sampled a (SignalNode a)
                   | Aged a (SignalNode a)
```

The *SignalNode* type is a disjoint union where each alternative corresponds to a signal combinator, and it contains references to other signals of type S *a*, thereby closing the loop. The primitives simply create the appropriate node in *Ready* state and wrap it in the signal structure. Since the interface has to look pure to support the applicative style demonstrated in the examples, the library relies on *unsafePerformIO* to create the references on demand in the case of the stateless combinators. Only *stateful* and *transfer* signals need to be created in the SM monad, as all the others are stateless.

The SM monad is in fact just a wrapper around *IO* that does not provide *liftIO*, therefore no effectful calculation can be executed in it other than creating stateful signals.

```
newtype SM a = SM { createSignal :: IO a }
  deriving (Monad, Applicative, Functor, MonadFix)
```

The only advanced language extension needed by the library is existential types, without which it would not be possible to define the applicative combinators and the *transfer* primitive, because they have to hide the types of the signals they depend on. Also, the functions traversing the signal network need to have existential types, because the nodes of the graph are heterogenous, even if they are related in well-defined ways.

3.2 Execution Mechanism

Each superstep consists of three separate phases: sampling, aging, and finalisation.

Every node starts out in the *Ready* state. Whenever their value is requested, they enter the *Sampling* state and cause all their dependencies to be sampled. As soon as their output is computed, they are marked as *Sampled*, and the node contains both the current sample and the old version of the signal.

If the sampling function encounters a signal in *Sampling* state, it assumes that it found a dependency loop without a delay. In this case, the signal in question is sampled with an alternative function, which delays *transfer* signals by reusing their previous output, but it acts as normal on all the other kinds of nodes. Therefore, loops composed of solely stateless primitives are not resolved by the system. The *delay* primitive plays a significant role here, because the evaluator knows that it does not need to sample its input signal in order to get the current sample of its delayed version, hence it does not touch the input before the aging phase. If *delay* was defined in terms of *transfer*, the evaluator could not tell that it only depends on past inputs.

The delayed samples are eventually overwritten by the call that put them in *Sampling* state originally, so no unnecessary delays are left in the system by the time everything is computed.

Afterwards, signals can be aged transitively using their current structure and the freshly produced sample where applicable. Only the stateful node types (*stateful*, *transfer*, and *delay*) change during this step. There is only one thing to keep in mind: the state of *stateful* and *transfer* is evaluated to weak head normal form regardless of whether they are needed or not, in order to prevent huge thunks from building up. At the end of this phase, all *Sampled* signals become *Aged*. The samples are still retained at this point, because they might be needed by dependency loops.

The final phase is a third traversal, where samples are discarded and the aged signals are wrapped in *Ready* for use in the next superstep.

3.3 Applicative Optimisation

Using only the two basic applicative combinators can easily result in a large number of nodes, since each function and each argument requires a separate node. It is known that all expressions built up using *pure* and $\circledast$ have a canonical form where the pure parts are factored out and united in a single monolithic function; an informal description of this process is given in [10]. Elerea uses the following equivalence to flatten function applications as much as possible:

$$
\begin{aligned}
&(f \mathbin{\textcircled{\$}} x_1 \circledast \ldots \circledast x_m) \circledast (g \mathbin{\textcircled{\$}} y_1 \circledast \ldots \circledast y_n) \\
\equiv\ &h \mathbin{\textcircled{\$}} x_1 \circledast \ldots \circledast x_m \circledast y_1 \circledast \ldots \circledast y_n \\
&\quad \textbf{where } h\ x_1 \ldots x_m\ y_1 \ldots y_n = f\ x_1 \ldots x_m\ (g\ y_1 \ldots y_n)
\end{aligned}
$$

While this equation can be derived from the applicative laws, its intuitive meaning should be easy to see: the right argument of $\circledast$ is the last argument of the function on the very left (the left pair of parentheses is superfluous), so this is the behaviour the united function has to emulate.

Since we need a separate case for each arity, the flattening process has to be limited at some point. Besides, the number of rules is proportional to the square of the maximum arity handled by the system, so Elerea can only completely collapse function applications of up to five arguments.

4 Design Rationale

4.1 Parting with Events

The original idea was to create a library that can be programmed essentially like Reactive, but some paper experimentation suggested that Reactive-style events add a lot of complexity to the design. It is also typical to generate events from boolean conditions over the current state of the system, and the only way to achieve this in Reactive is to take some periodic event source – e.g. one that is sync with the sampling process of the top-level behaviour – and drop events when the condition does not hold. This is basically the same as Yampa-style event handling, i.e. events degrade into Maybe-like behaviours. Therefore, events are modelled with continuous boolean-valued signals. This reduces the generality of the library, making it incapable of efficiently handling event-driven systems like graphical user interfaces, but this is an acceptable trade-off in return for the reduced complexity.

4.2 Going Impure

While it would be straightforward to have each primitive construct some pure data structure, bringing this structure to life would be impossible without leaving the pure world. The heart of the problem is that we cannot age the network without knowing which nodes are physically identical, since the knots would be untied and we would get into an infinite loop if there is at least one cycle. However, we cannot detect sharing by looking at the values, unless the values are guaranteed to be all distinct. Needless to say, given the desired interface, there is no pure way to tag them with unique identifiers either.

As long as the network is static (it contains no higher-order constructs), there is a half-solution to this problem in the Haskell world: we can check the pointers of the nodes in a preparation phase and use them as unique identifiers to build an indexed map of nodes. Naturally, nodes would also be changed to refer to their dependencies through such indices. In other words, we would introduce a level of indirection. The resulting representation of the network could be properly updated in each superstep, even though it would be a cumbersome process: we cannot just map an aging function over the collection but have to traverse the graph in the same order as if it was the original pure structure – but at least we can cut the cycles short by checking whether some result is already available.

In the general case, the network is dynamic, which complicates matters even further. If new signals are synthesised along the way, they are also pure structures that have to be processed and inserted into the live network in a similar fashion as the already existing nodes. These structures can also contain references to the pure versions of some live nodes, and we have to find the index of the appropriate live node knowing only the reference to its original pure representation. Besides, we have to implement our own garbage collection mechanism to get rid of signals no-one can possibly reference any more.

All in all, maintaining an explicit structure requires a lot of book-keeping and it duplicates functionality that the runtime implements much more efficiently. This realisation led to the idea of representing each node directly with a mutable variable and let the runtime keep track of them.

4.3 Introducing Dynamism

The original goal was to create a purely applicative style library, and there was no monadic interface in the plans. To achieve this, the previous major version of the library (0.x.x) had a construct called *latcher*, which was superseded by *generator* and *sampler* in the current version. It combined control through a boolean signal with collapsing higher-order samples, and had the following type:

$$latcher :: \mathsf{S}\ a \to \mathsf{S}\ Bool \to \mathsf{S}\ (\mathsf{S}\ a) \to \mathsf{S}\ a$$

The signal constructed by *latcher* s_0 b *ss* started out as signal s_0, and altered its behaviour by sampling the stream of behaviours *ss* whenever the boolean control input b was true. Just like *transfer*, it was also undelayed, i.e. if the control signal evaluated to true at the moment, the new behaviour replaced the old one before producing a sample.

The basic idea behind this construct was that new stateful signals would be created on demand, as dictated by lazy evaluation. However, this interface broke referential transparency, since the behaviour of the network would depend on when signal expressions were evaluated. Most importantly, the start times would be different depending on the optimisation options used by the compiler, because let-floating (moving an expression to the outermost scope possible given its data dependencies) and common subexpression elimination (CSE) could easily interfere with the intent of the programmer.

The original solution to this problem was to inject dependencies in the code by hand, because disabling optimisations altogether would have detrimental effects on performance. In practice, this meant passing around dummy parameters, so the optimiser would keep every stateful signal in its original scope. Naturally, this was an error prone workaround that went straight against the goal of practicality.

Introducing the SM monad solved the problem of scoping by basically replacing the explicit dummy parameters. At the same time, it does not really interfere with the applicative style, since most signal combinators are stateless – even *sampler* and *generator*, which perform subtasks of the original *latcher* –,

so they can be combined just as easily as pure values. Thanks to the *MonadFix* instance, definitions can be moved around just as freely as usual in functional programs. The biggest difference is that stateful signals need to be extracted from the SM monad, i.e. one has to write ← instead of = when defining them.

5 Evaluation

Even though Elerea is at best in the embryonic stage, it can already be used to create functional applications. This is mostly explained by the fact that getting some reasonably complex examples to run had priority over laying down theoretical foundations, arguing that if something works well, we can try to find out later why. Of course, this is only possible as long as the system is not too complex for such analysis. Fortunately, it took only a few simple functions and a straightforward data structure to get a useful system up and running, whose complexity has not grown much either for the time being, so there is a good chance that the specifics of its behaviour can be thoroughly analysed.

The system has some nice properties that make it a comfortable playground for experimenting with reactivity. First of all, the initial experience suggests that a primarily applicative interface can indeed be very convenient. Elerea does a good job supporting recursion and dynamism, providing great freedom to the user. Thanks to the use of mutable variables in the core, performance is predictable, and Elerea programs tend to play nice with resources. The overall simplicity of the library also makes it easy to embed signals into different frameworks, and it is even straightforward to create a feedback behind the scenes, i.e. let the reactive program affect the future readings of its own external inputs.

On the other hand, it must be noted that the heavy use of IORefs most likely severely limits performance. This problem is hard to solve without significant changes to the interface, because mutable variables were needed to deal with the structures allowed by general recursion. Also, the semantics of the constructs are not properly defined.

6 Related Work

It is getting more and more difficult to give an exhaustive list of FRP implementations, but it is customary to note that the concept was introduced with Fran [7], a language to define animations in a purely declarative way. The interface of Fran is applicative, very clean and high-level, but it is difficult, if not impossible to implement efficiently in Haskell. Still, it could be used as the reactive basis in projects like Frob [16], a domain specific language for controlling robots, and FVision [17], a reactive interface over a C++ library for visual tracking. Real-time FRP [19] attacked the performance issues by concentrating on the operational semantics, and factoring reactivity and pure calculations into different language layers. The Hume language [12] was designed at about the same time with a focus on giving static resource bounds while allowing the user to directly describe the system as a data-flow network, and it is also based on a similar

separation of concerns. FunWorlds [18] does away with global time and adopts the stream-of-residuals view, meaning that it explicitly breaks up behaviours at each sampling point on the conceptual level in order to describe reactions locally. React [1] disallows recursive definitions and provides a single looping construct instead that adds an infinitesimal delay.

The system that is probably closest to Elerea in spirit is FrTime [4]. It is a REPL-friendly system living in the DrScheme environment [9], whose signals can be manipulated and probed in an interactive session. Signal expressions are processed with a custom-made evaluator, and the system also incorporates applicative optimisations, albeit under the name *lowering* [2].

As for the three systems mentioned at the beginning, Elerea resembles Yampa [5] the most, because both are pull-based with only continuous signals and no 'real' events. Even though Reactive [6], carrying the legacy of Fran, looks more similar on the surface, its signal concept is completely different: a signal expression in Reactive has no identity, as it is a self-contained description of the whole history of a changing value. Consequently, interaction between signals has to be organised in fundamentally different ways. Grapefruit [13] focuses on describing event-driven systems, although it provides continuous signals too. It solves the problem of let-floating by having signals refer to their consumers through uninstantiated type parameters, which makes it clear when signals have to start.

7 Conclusions and Future Work

Elerea is an experiment to see what we get when we try to implement a reactive interface that feels intuitive. The essence of this intuition is laziness extended in the temporal dimension, where signals come to life whenever their output is needed by others. Intuitiveness is naturally highly subjective, but early experience with the library suggests that it can indeed give clear and concise descriptions for dynamic networks of continuous signals.

Both the theoretical and the practical aspects of the system can be explored even further. The project is mostly focused on the latter, and the short-term plans are to create a complex application with a highly dynamic structure, which is necessary to test composability on a large scale and also to observe typical usage patterns that can drive the further design of the library. The library would also benefit from precise semantics, which should also be worked out in the future.

Acknowledgements

I owe thanks to Peter Verswyvelen and Péter Hanák for their helpful comments, as well as the anonymous reviewers of the Haskell Symposium, who gave an exhaustive analysis of an old version of this paper.

References

1. Daniel Bünzli. React (OCaml) 2009. `http://erratique.ch/software/react`
2. Kimberley Burchett, Gregory H. Cooper, and Shriram Krishnamurthi. Lowering: A Static Optimization Technique for Transparent Functional Reactivity. In *Proceedings of the 2007 ACM SIGPLAN symposium on Partial Evaluation and Program Manipulation (PEPM '07)*, pages 71–80. ACM Press, 2007, Nice, France
3. Mun Hon Cheong. Functional Programming and 3D Games. Master's Thesis, 2005, University of New South Wales, Sydney, Australia.
4. Gregory H. Cooper and Shriram Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *European Symposium on Programming*, 2006, pages 294–308
5. Antony Courtney, Henrik Nilsson, and John Peterson. The Yampa Arcade. In *Proceedings of the 2003 ACM SIGPLAN Haskell Workshop (Haskell'03)*, pages 7–18. ACM Press, 2003, Uppsala, Sweden.
6. Conal Elliott. Simply efficient functional reactivity. 2008. `http://conal.net/papers/simply-reactive/`
7. Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, June 1997, pages 263–273.
8. Levent Erkök. Value Recursion in Monadic Computations PhD Dissertation, Oregon Graduate Institute School of Science Engineering, OHSU, October 2002.
9. Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: A programming environment for Scheme. In *Journal of Functional Programming*, 12(2), pages 159–182, 2002.
10. Jeremy Gibbons and Bruno C. D. S. Oliveira. The essence of the Iterator pattern. In *Mathematically-structured functional programming*, Conor McBride and Tarmo Uustalu (eds), 2006.
11. George Giorgidze and Henrik Nilsson. Switched-on Yampa: Declarative Programming of Modular Synthesizers. In *Proceedings of 10th International Symposium on Practical Aspects of Declarative Languages*, San Francisco, CA, USA, January 7-8, 2008.
12. Kevin Hammond and Greg Michaelson. Hume: A Domain-Specific Language for Real-Time Embedded Systems. In *Generative Programming and Component Engineering, Second International Conference*, Erfurt, Germany, September 22-25, 2003, pages 37-56. Lecture Notes in Computer Science, Springer, 2003.
13. Wolfgang Jeltsch. Improving Push-based FRP. *9th Symposium on Trends in Functional Programming*, May 26–28, 2008.
14. Conor Mcbride and Ross Paterson. Applicative programming with effects *Journal of Functional Programming*, 18(1), pages 1–13, 2008.
15. Silvio Romero de Lemos Meira. On the Efficiency of Applicative Algorithms. PhD thesis, Computing Laboratory, The University of Kent at Canterbury, March 1985.
16. John Peterson, Paul Hudak and Conal Elliott. Lambda in Motion: Controlling Robots with Haskell. In *First International Workshop on Practical Aspects of Declarative Languages (PADL)*, January 1999
17. John Peterson, Paul Hudak, Alastair Reid and Greg Hager. FVision: A Declarative Language for Visual Tracking. In *Proceedings of PADL'01: 3rd International Workshop on Practical Aspects of Declarative Languages*, pages 304–321, January 2001

18. Claus Reinke. FunWorlds/HOpenGL – Functional Programming and Virtual Worlds. draft paper presented at the 14th International Workshop on the Implementation of Functional Languages (IFL 2002), Madrid, Spain, September 16–18, 2002.
19. Zhanyong Wan, Walid Taha, and Paul Hudak. Real-time FRP. In *Proceedings of Sixth ACM SIGPLAN International Conference on Functional Programming*, Florence, Italy, September 2001. ACM.