Push-Pull Signal-Function Functional Reactive Programming

Edward Amsden

Rochester Institute of Technology
eca7215@cs.rit.edu

Abstract. Functional Reactive Programming is a promising approach (RRN: systems – seems weird to identify it with the implementations... model, approach?) for writing interactive and time-dependent programs. Signal-function FRP is variant of FRP which provides advantages in modularity and correctness, but has proven difficult to implement efficiently.
The abstraction of signal vectors provides the necessary type apparatus to distinguish components of the input and output of signal functions which benefit from a push-based implementation from those which benefit from a pull-based implementation, and to combine both implementation strategies in a single system.
(RRN: Without understanding it a priori, the above para is hard to parse. How about the following?)
One important implementation trade-off is whether evaluation of signal functions is *push-based* or *pull-based*. The *signal-vector* technique addresses this by providing a type-level description of the inputs and outputs of signal functions, distinguishing components that benefit from a push-based implementation from those that benefit from a pull-based implementation. This makes it possible to combine both implementation strategies in a single system.
We describe a novel (RRN: is it novel? how? need clear contributions) signal-function FRP system which provides push-based evaluation for events, pull-based evaluation for signals, and a simple monadic evaluation interface that permits the system to easily integrate with one or more IO systems.

1 Introduction

Functional Reactive Programming (FRP) is a class of systems for describing reactive programs. Reactive programs are programs which, rather than batch-processing inputs to produce an output, instead map time-varying inputs onto outputs.

An FRP system provides a means of manipulating *behaviors* and *events*. (RRN: Switched to present tense. Don't need to add additional future or past tense...) Behaviors are often referred to as *signals* in FRP literature, but the definition is the same. Semantically, a behavior or signal is a function from time to a value. Whereas an event is a ordered sequence of discrete occurrences, carrying both values and timestamps.

(RRN: Looks like you should standardize early on **signal** and leave extra occurrences of **behavior** out of this intro.)

FRP systems can generally be categorized as "classic FRP," in which behaviors and events are manipulated directly as first-class values, or *signal-function FRP*. In signal-function FRP, the programmer cannot directly manipulate signals, rather, functions from input-signal to output-signal are the central abstraction. Thus signal functions are time-dependent, reactive transformers of signals, events, or combinations of signals and events. FRP *combines* behaviors and events through the use of *switching*, in which a behavior (in classic FRP) or a signal function (in signal-function FRP) is replaced by a new behavior or signal function carried by an event occurrence.

– TODO: it would be good to give SOME sense for how an FRP evaluation proceeds (e.g. a strawman) before getting into the next para.

Classic FRP was first described as a system for interactive animations [1]. Recent work on classic FRP has focused on efficient implementation (RRN: CITATION?). One approach to efficiency is to separate the evaluation of behaviors and events, since suitable strategies give best performance in each case. Push-based evaluation evaluates a system only when input is available, and is thus suitable for discrete inputs such as events. Pull-based evaluation evaluates the system as quickly as possible, polling for input, and is preferable for behaviors and signals. The initial implementations of FRP made use of pull-based evaluation for both behaviors and events. "Reactive" [2], as well as more recent systems such as "reactive-banana" [3], make use of push-based evaluation for events and pull-based evaluation for behaviors. This is known as "push-pull" evaluation.

All implementations of signal-function FRP to date [4–7] have used pull-based evaluation for both signals and events. This is due to the ease of implementation of pull-based evaluation, and because the types used for signal functions do not permit distinguishing signals and events, or constructing only part of the input (for instance, one event occurrence).

(RRN: What does it mean to construct one event occurrence? As opposed to?)

A recent extension of signal-function FRP called N-Ary FRP [7] describes a method of typing signal functions which, as we will show, enables the push-based evaluation of events in a signal-function FRP system. The notion of signal vectors allows the representation of signal function inputs and outputs as combinations of signals and events, rather than a single signal which may contain multiple values, including option values for events. Signal vectors are uninhabited types, which can be used to describe partial or full representations of the signal function inputs and outputs.

We present TimeFlies,[1] a push-pull signal-function FRP system. We hope to demonstrate the feasibility of such an approach to FRP, and provide a basis for further research into efficient implementation of signal-function FRP. We also describe a powerful evaluation interface for TimeFlies, which permits us to use TimeFlies to describe applications which make use of multiple and differing IO libraries.

(RRN: Need a concrete contributions list. "The first system to apply push-pull implementation strategies to signal-function FRP", etc. It doesn't hurt for it to be bulletted.)

Section 2 describes design choices for the system, and provides an overview of the interface. Section 3 describes how the system is implemented, and how the separation of evaluation between events and signals is achieved. Section 4 is a discussion of the usefulness (RRN: more specific?) of our implementation. Section **??** describes the current and future work on this system. Section 5 gives an overview of related efforts. Section 6 concludes.

2 System Design and Interface

The purpose of FRP is to provide an efficient, declarative abstraction for creating reactive programs. To this end, there are three goals that the TimeFlies system is intended to meet. (RRN: What are the three!? Enumerate. Push based, composable, IO-system agnostic?)

Efficient evaluation is the motivation for push-based evaluation of events. Since FRP programs are expected to interact with an outside world in real time, efficiency cannot simply be measured by the runtime of a program. Thus, when speaking of efficiency, we are expressing a desire that the system utilize as few system resources as possible for the task at hand, while responding as quickly as possible to external inputs and producing output at a consistently high sample rate.

– At this point we still need a clear notion of what a pull-based evaluation of events would actually do.

A composable abstraction is one in which values in that abstraction may be combined in such a way that reasoning about their combined actions involves little more than reasoning about their individual actions. In a signal function system, the only interaction between composed signal functions ought to be that the output of one is the input of another. Composability permits a particularly attractive form of software engineering in which successively larger systems are created from by combining smaller systems, without having to reason about the implementation of the components of the systems being combined.

[1] The sentence "Time flies like an arrow." is a favorite quotation of one of the author's philosophy instructors, used to demonstrate the ambiguity of language. The origin of the quotation is unknown.

(RRN: Can the above be made into a slightly more technical claim? Composable abstractions enable composing $O(N)$ components while reasoning about the $O(N)$ local interactions, rather than the $O(N^2)$ global interactions between all components? Second, I think not needing to know the implementation is separate from "composability"; surely there is some name for that: ADTs, "encapsulation", information hiding, etc.)

Further, an FRP program is not a closed system; it interacts with the outside world. Since we cannot anticipate every possible form of input and output that the system will be asked to interact with, we must interface with Haskell's IO system. (RRN: First mention of Haskell... need some transition?) In particular, most libraries for user interaction (e.g. GUI and graphics libraries such as GTK+ and GLUT) and most libraries for time-dependent IO (e.g. audio and video systems) make use of the event-loop abstraction. In this abstraction, event handlers are registered centrally; once initiated, the running event-loop detects events and invokes handlers; and handlers are responsible for rendering output.

(RRN: Is the event loop really an *abstraction*? What is abstracted? Or is it just a "model" or an API?)

We would like for the FRP system to be easy to integrate with such IO systems, while being flexible enough to enable its use with other forms of IO systems, such as simple imperative systems, threaded systems, or network servers. (RRN: vague... not sure what these mean. Threaded = multiple event loop? Alternative to an event loop is... what?... blocking IO operations?)

2.1 Semantics

A rigorous semantics of signal-function FRP with signal vectors remains unattempted, though a categorical semantics for signal-function FRP using *fan categories* has been explored [**?**]. In both these semantics and their implementation, events are represented (following AFRP [4] and Yampa [6]) as option-valued signals. A push-based implementation of these semantics is proposed, but the unit being pushed is a *segment* of signals, and not event occurrences. This approach requires processing the whole signal function at each time step, whether an event occurs or not, and checking for its occurrence at each step.

(RRN: Might want to make it clear – the "whole" signal function means "all signal functions in the program" (i.e. which are composed to form the global signal function) right?)

Another concern is that this approach limits the event rate to the sampling rate. The rate of sampling should, at some level, not matter to the FRP system. Events which occur between sampling intervals are never observed by the system. (RRN: An FRP system has sampling intervals? Where were we told this!?) Since the signal function is never evaluated except at the sampling intervals, no events can otherwise be observed.

This raises another concern. Events are not instantaneous in this formulation. If events are signals, the sampling instant must fall within the interval where there is an event occurrence present for that event to be observed. If events are instantaneous, the probability of observing an event occurrence is zero. (RRN:

So.... how do existing systems/semantics compensate for this? They allow events to have temporal extent?)

Therefore, TimeFlies employs the N-Ary FRP type formulation to represent signals and events as distinct entities in the inputs and outputs of signal functions. This means we are now free to choose our representation of events, and to separate it from the representation and evaluation of signals. This freedom yields the additional ability to make events independent of the sampling interval altogether. The semantics of event handling in TimeFlies is that an event occurrence is responded to immediately, and does not wait for the next sampling instant. (RRN: Could you flesh out how this is a *semantics* issue and not merely a more efficient or accurate implementation? Ah, that's answered in the next para, but might be good to use the word semantics somewhere and connect this explicitly.) This allows events to be instantaneous, and further, allows multiple events to occur within a single sampling interval. (RRN: Clarification—even in traditional approaches multiple **different** events could occur in one sampling round right? It's just that *each* event [stream] could only have one occurrence per sampling interval?)

There are two tradeoffs intrinsic to this approach. The first is that events are only partially ordered temporally. There is no way to guarantee the order of observation of event occurrences occurring in the same sampling interval. Further, the precise time of an event occurrence cannot be observed, only the time of the last (RRN: [signal?]) sample prior to the occurrence. (RRN: That last sentence had an ambiguous parse wherein the first time I read it I thought it was saying that you couldn't tell the timestamp of event N only of the preceding event $N-1$, which of course makes no sense, but it might be good to clarify what *sample* you're referring to.)

In return for giving up total ordering and precise observation of the timing of events, we obtain the ability to employ push-based evaluation for event occurrences, and the ability to non-deterministically merge event occurrences. When events being input to a non-deterministic merge have simultaneous occurrences, we arbitrarily select one to occur first. This does not violate any guarantee about time values, since they will both have the same time value in either case, and does not violate any guarantee about ordering, since no guarantee of their order is given. (RRN: Nice.)

2.2 Types

In a strongly and statically typed functional language, types are a key part of any interface. Types provide a mechanism for describing and ensuring properties of the interface's components and about the systems created with these components.

In order to type signal functions, we must be able to describe their input and output. In most signal function systems, a signal function takes exactly one input signal and produces exactly one output signal. Multiple inputs or outputs are handled by making the input or output a signal of tuples, and combinators which combine or split the inputs or outputs of a signal assume this. Events are represented at the type level as a particular type of signal, and at the value level as an option: either an event occurrence or not. (RRN: That's a bit confusing, because of the issue of whether we're talking about *event occurrences* or event streams. At runtime there is only a representation for the former, which is a simple `Maybe`, right?)

This method of typing excludes push-based evaluation at the outset. It is not possible to construct a "partial tuple" nor in general is it possible to construct only part of any type of value. Push-based evaluation depends on evaluating only that part of the system which is updated, which means evaluating only that part of the input which is updated.

In order to permit the construction of partial inputs and outputs, we make use of signal vectors. Signal vectors are uninhabited types which describe the input and output of a signal function. Singleton vectors are parameterized over the type carried by the signal or by event occurrences. The definition of the signal vector type is shown in Figure 1.

Having an uninhabited signal vector type allows us to construct representations of inputs and outputs which are hidden from the user of the system, and are designed for partial representations.

```
data SVEmpty    -- An empty signal vector component,
                -- neither event nor signal
data SVSignal a -- A signal, carrying values of type a
data SVEvent a  -- An event, whose occurrences carry values of type a
data SVAppend svLeft svRight -- The combination of the signal vectors
                             -- svLeft and svRight
```

Fig. 1. Signal vector types.

Signal functions with signal vectors as input and output types form a Haskell `GArrow` [8]. Specifically, the signal function type constructor (with the initialization parameter fixed) forms the arrow type, the `SVAppend` type constructor forms the product type, and the `SVEmpty` type constructor forms the unit type. The representation of signal functions is discussed in Section 3.1.

Our interface for evaluating signal functions takes the form of a monad transformer [9]. This formulation permits us to provide specific temporal actions to perform on a signal function in sequence, and in conjunction with actions taken in Haskell's `IO` monad, or another monad. Actions are provided in this monad to push an event to the signal function's input, step the signal function's time, or modify a signal on the signal functions input to be observed at the next time step. Outputs are handled by a vector (RRN: vector? in the same sense as a signal vector?) of actuation functions in the underlying monad, supplied by the user of the interface. This vector is structured by parameterizing it over the signal function's output signal vector. (RRN: huh? more concrete way to say it?)

```
-- Signal functions
-- init: The initialization type for
--   the signal function, always NonInitialized
--   for exported signal functions
-- svIn: The input signal vector
-- svOut: The output signal vector
data SF init svIn svOut

data NonInitialized

type svIn :~> svOut = SF NonInitialized svIn svOut
type svLeft :^: svRight = SVAppend svLeft svRight
```

Fig. 2. Signal function types.

(RRN: Evaluation interface: Very cool. It sounds a bit your FRP implementation itself is kind of "scriptable" rather than black box...)

2.3 Combinators

TimeFlies includes basic signal functions and a library of signal function combinators, falling into these categories: lifting operations from pure functions to signal functions, routing, reactivity, feedback, event processing, joining, and time dependence. (RRN: Eek, I get thrown off by a definition of combinators which is so broad as to include everything. I like the definition here http://stackoverflow.com/tags/combinators/info : *A combinator is a higher-order function that uses only function application and earlier defined combinators to define a result from its arguments.* Does everything in the above list meet that definition?)

The basic signal functions provide very simple operations. The `identity` signal function, as expected, simply copies its input to its output. The `constant` signal function produces the provided value as a signal at all times. The `never` signal function has an event output which never produces occurrences. The `asap` function produces an event occurrence with the given value at the first time step after it is switched into the network. The `after` function waits for the specified amount of time before producing an event occurrence.

Two combinators are provided to lift pure functions to signal functions. The `pureSignalTransformer` combinator applies the pure function pointwise to a signal. The `pureEventTransformer` combinator applies the function to the value carried by each occurrence of an input event.

The routing combinators are used to combine signal functions, and to rearrange and discard components of signal vectors in order to connect signal functions. Only those routing combinators which modify or combine signal functions (`(>>>)`, `first`, `second`) are reactive (*i.e.* may replace themselves in response to events, and then only if they inherit their reactivity from the signal function(s) which are inputs to combinator. The rest do not react to or modify the input in any way, except to re-arrange it, copy it, or discard it altogether.

Reactivity is introduced by means of the `switch` combinator. The design of this combinator allows modular packaging of reactivity. A signal function can determine autonomously when to replace itself, based only on its input and state, by emitting an event occurrence carrying its replacement. The combinator consumes and hides the event carrying the replacement signal function, so the reactivity is not exposed by the resulting reactive signal function.

It is often useful for a signal function to receive a portion of its own output as input. This is especially useful when we have two signal functions which we would like to mutually interact. We cannot just serially compose them, we must also bring the output of the second back around to the first. Many signal-processing algorithms also depend on feedback. This ability is provided by the `loop` operator.

This combinator provides decoupling for signals (the input signal is the output signal at the previous time-step) but not events (event occurrences are supplied to the combinator immediately). This means that the programmer has the responsibility to ensure that feedback does not generate an infinite sequence of events in a single time-step. (RRN: Presumably that is different from previous formulations of FRP? Because ... there was no notion of multiple events in a sampling interval?)

The `filter` and `filterList` combinators are stateless signal functions which apply a function to each event occurrence. The `filter` combinator expects an option-typed output, while the `list` combinator expects a list output. (RRN: and input right?) If the function yields a non-empty output, one or more output occurrences are produced with the elements of that output. If the output is empty, there are no output occurrences.

The `accumulate` and `accumulateList` functions are similar, except that they maintain a state value which is replaced after each input event occurrence by an additional output from the function, and supplied to the function in addition to the value of each event occurrence. (RRN: Murky prose definition...)

The joining combinators provide the ability to combine two event streams, two signals, or a signal and an event stream. (RRN: How does that work temporally? For event/event is this a zip that matches up the left events w/ the right events?)

The `union` combinator is a non-deterministic merge of two event streams. Simultaneity of events is only observable up to the sampling rate, but events can occur in between sampling intervals. Thus, if two events occur simultaneously, one occurrence is arbitrarily selected to be first in the output stream.

The `combineSignals` combinator applies a binary function pointwise to two signals, and produces the result of this application as a third signal. The combining function is necessary because we will have two input samples at each time step, and must produce exactly one output sample. (RRN: Wait that sounded like it was covered by join?)

The `capture` combinator adds the last-sampled value of a signal at the time of an event occurrence to that event occurrence. (RRN: Again, what is a signal-event **join** if not that?)

These three combinators (RRN: Which?) together provide the ability to combine elements of signal vectors. By combining these combinators, along with the `cancelLeft` and `cancelRight` routing combinators, arbitrary signal vectors can be reduced.

A set of combinators are provided for making use of time-dependence in a signal function. These combinators allow the modification of signals and events with respect to time, and the observation of the current time value. The simplest time-dependent combinator is `time`, which simply outputs the time since it began to be evaluated. This does not necessarily correspond to the time since the global signal function began to be evaluated, since the signal function in which the `time` combinator is used may have been introduced through `switch`.

(RRN: If that's the simplest what are the more complex ones? If not worth mentioning it seems easier to just describe **time**.)

The `delay` signal function allows events to be delayed. An initial delay time is given, and event occurrences on the right input can carry a new delay time. Event occurrences on the left input are stored and output when their delay time has passed. Changing the delay time does not affect occurrences already waiting.

The `integrate` combinator outputs the rectangle-rule approximation of the integral of its input signal with respect to time.

– What's novel here? After this section is de-listified, need to emphasize that.

2.4 Evaluation Interface

(RRN: Is the thing that really sets TimeFlies apart? (Apart from being the first push/pull for SFFRP.))

The evaluation interface provides a modified state monad which holds a signal function, together with some additional information, as its state. Rather than monadic instructions to put and get the state, the monad provides instructions to trigger an input event, update an input signal, and trigger sampling of signals in the signal function. Additional state includes the current set of modifications to the input signals (since the last sample) and a set of handlers which actuate effects based on output events or changes to the output signal.

Sets which correspond to signal vectors are built with "smart" constructors. For instance, to construct a set of handlers, (RRN: is this "set" the same as the earlier "vector of actuation functions"?) individual handling functions are lifted to handler sets with the `signalHandler` and `eventHandler` functions, and then combined with each other and `emptyHandler` leaves using the `combineHandlers` function. (RRN: Examples, examples, examples. There's a lot of prose here that gives the reader some idea, but doesn't show them exactly what it would look like. Too much guesswork required to fill in the gaps.)

Building the initial input sample is similar, but `sampleEvt` leaves do not carry a value.

In order to initialize the state, the user must supply a set of handlers, the signal function to evaluate, and initial values for all of the signal inputs.

This state can then be passed to a monadic action which will supply input to the signal function. Inputs are constructed using a simple interface with functions to construct sample updates and event occurrences, and to specify their place in the vector.

The `SFEvalT` monad is actually a monad transformer, that is, it is parameterized over an underlying monad whose actions may be lifted to `SFEvalT`. In the usual case, this will be the `IO` monad.

`SFEvalT` actions are constructed using combinators to push events, update inputs, and step time, as well as actions lifted from the underlying monad (used to obtain these inputs). (RRN: That sounds introductory... I thought that was supposed to be what we were learning about throughout this subsection?) An action in the underlying monad which produces the result and a new state is obtained with the `runSFEvalT` function.

3 Implementation

We now turn our attention to the implementation of the signal function system. We will discuss representations of inputs, outputs, and signal functions, as well as the implementations of specific signal function combinators.

3.1 Signal Functions

The design of signal functions specifies a family of types for the inputs and outputs of signal functions. Signal functions are not functions in the purest sense, however. They are not mappings from a single instance of their input type to a single instance of their output type. They must be implemented with respect to the temporal semantics of their inputs and outputs.

We therefore start by creating a set of concrete datatypes for the inputs and outputs of signal functions. These datatypes will be parameterized by the input and output types of the signal function, and will not be exposed to the user of the library. Rather, they will specify how data is represented during the temporal evaluation of signal functions. We then implement signal functions as records of functions from these concrete types to these concrete types paired with new signal functions. The evaluation system for signal functions then maintains this record for the signal function it is evaluating, calls the correct function for the current input, actuates the outputs (using supplied functions from the outputs to monadic actions) and replaces the record with the newly supplied signal function record.

In Section 2 we presented signal vectors as a set of types. In order to be completely principled, we should isolate these types into their own *kind* (a sort of type of types); however, the Haskell extension for this was far from stable at the time this system was created.

Using the Glasgow Haskell Compiler's extension for Generalized Algebraic Datatypes [10–12], we specify three concrete datatypes which are parameterized over signal vectors, and represent information about one temporal point in a signal function's evaluation.

The first type carries the instantaneous values of all signals in a signal vector. There is precisely one constructor for each type constructor of a signal vector (`SVEmpty`, `SVSignal`, `SVEvent`, and `SVAppend`). The constructor for `SVSignal` carries a value typed with the type parameter of the `SVSignal` constructor, the rest do not. Thus, in this representation, for any signal vector, there is a node in the representation for each node in the signal vector, and precisely the signal nodes have values.

The next representation contains a constructor for `SVEvent` nodes, carrying a value typed by the type parameter of `SVEvent`, and two constructors, each of which carries a representation for one child of a `SVAppend` node (the left or right), and is parametric in the other. Thus, an instance of this representation is a path from the root of a signal vector to an event node, carrying exactly one value for the event.

The last representation is used for efficient implementation, and has three constructors. One constructor represents a `SVSignal` node and has a value typed by its type parameter. The second carries representations of both children of a `SVAppend` node. The third is parametric in its signal vector, and thus represents the absence of information for an arbitrary signal vector. This representation contains values for all, some, or none of the `SVSignal` nodes of a signal vector.

Signal functions are implemented by combining two strategies for temporal updates. The first such strategy is sampling. A signal function has a continuation which may be invoked at regular intervals, using the partial representation of signals described above as input and producing such a representation as output. This amounts to repeated sampling of the output signals of a signal function.

The other approach is notification. Signal functions have a second continuation which is only invoked when there is an event occurence on their input. The input to this continuation is, of course, the representation of event occurrences. If events were signals, they would be sampled as above, and the sampling interval would not match the event occurrence interval. It would thus be necessary to represent these non-occurrences. By implementing event handling separately from signal updates, we eliminate the need for a representation of event non-occurences and the need for invoking event-handling code at every time step.

We represent signal functions as a GADT with three type parameters and two constructors. The first type parameter represents the initialization state, and is specialized to `Initialized` or `NonInitialized` depending on the constructor. It would of course be possible to represent these as two distinct datatypes, but this representation communicates the intuition that these are two states of the same object, rather than separate objects. The other two type parameters are the input and output signal vectors, respectively. The signal functions that a user will compose are non-initialized signal functions. They must be provided with an initial set of input signal values, which are considered the sample for time zero, and represented by the full representation of signals described above.

When provided with this input, they produce their time-zero output and an initialized signal function.

Initialized signal functions carry the two continuations described above. The first continuation takes a time differential and a set of signal updates (the partial representation of signals) and returns a set of signal updates, a collection of event occurrences, and a new initialized signal function of the same type. This is the continuation called when sampling.

The second continuation takes an event occurrence, and returns a collection of event occurrences and a new signal function of the same type. This continuation is only called when there is an event occurrence to be input to the signal function.

This is the type and framework for our signal function implementation. Combinators are implemented as functions which return these types with specific implementations of the continuations. Space precludes a full discussion of the implemenation of each combinator, but some of the examples will be discussed.

The simplest example of the implementation of is the `identity` combinator. This signal function simply passes all of its inputs along as outputs. The initialization function simply passes along the received sample and outputs the initialized version of the signal function. The initialized version of the input is similar, but is self-referential. It outputs itself as its replacement. This is standard for simple and routing combinators which are not reactive, and simply re-arrange, discard, or combine signals and events.

In order for our primitive signal functions to be useful, we need a means of composing them. Serial composition creates one signal function from two, by using the output of one as the input of the other. The serial composition combinator is styled `(>>>)`. The implementation of this operator is one place where the advantage of responding to events independently from signal samples becomes clear.

This is the only primitive combinator which takes two signal functions, and thus, it is the only way to combine signal functions. Parallel, branching, and joining composition can be achieved by modifying signal functions with the `first` and `second` combinators and composing them with the routing and joining combinators.

Combinators which take one or more signal functions as input must recursively apply themselves, as is shown in the implementation of serial composition. They must also handle initialization, retaining the initialized signal functions and passing them to the initialized version of the combinator.

The switch combinator is the means of introducing reactivity into a signal function. This combinator allows a signal function to replace itself by producing an event occurrence. The combinator wraps a signal function, and observes an event on the right side of the output signal vector. At the first occurrence of the event, the signal function carried by the occurrence replaces the signal function.

The switch combinator stores the input sample provided during initialization, and updates it with the input signal updates. When the wrapped signal function produces an occurrence carrying a new signal function, that signal function is initialized with the stored input sample. It is then ”wrapped” by another function which closes over its output sample, and outputs the sample as a signal update as the next time step. After this, it acts as the new signal function. This wrapping has some performance implications, which are discussed in Section 4.

This combinator checks the outputs of the wrapped signal function for an event occurrence from which an uninitialized signal function is extracted. The switch combinator stores the full sample for its input vector (which is identical to the input vector of the supplied signal function) to initialize the new signal function. This also demands that it add a wrapper to the new signal function which waits for the next sampling interval and actuates the sample output at initialization as an output set of changes to the signal. This has some performance implications, which are discussed in Section 4.

Most of the routing combinators are simple to implement. The only task is to add remove, or replace routing constructors on signal updates and event occurrences. Since these signal functions are stateless and primitive, they can simply return themselves as their replacements.

The looping feedback combinator is particularly tricky. As it is currently implemented, the initial sample for the right side of the input signal vector to the supplied function is the right side of the output sample. This is acceptable, given Haskell's non-strict evaluation strategy, but it is necessary that the right side of the signal function's output not be immediately dependent on its input. The feedback combinator makes use of Haskell's lazy evaluation to feed events back into the combinator, and stores signal updates until the next sample. Signal samples are thus automatically decoupled after initialization. The implementation makes use of the recursive nature of the `let` construct in Haskell, and the non-strict evaluation of Haskell, to implement feedback.

Time dependence is introduced by the `time`, `delay`, and `integrate` combinators. The time combinator simply sums the time updates and provides the sum as a signal output. The `delay` combinator keeps a table of events which have come in, along with their schedule occurrence time, and produces them as output when time advances far enough. The integrate combinator performs rectangle-rule integration on signal samples with respect to time.

The implementation strategy leaves room for optimizations. In particular, an additional constructor for time-independent signal functions would allow portions of a signal function to forgo evaluation during time steps unless they had signal updates. Optimizations in the style of Yampa, observed by keeping an updated AST for the signal function and pattern-matching on it when switching, might further improve performance. In particular, collapsing nested or serially-composed versions of the `switchWait` step when switching would remove at least some of the observed dependence of performance on sampling rate. Nevertheless, this implementation performs quite well as it currently exists, as we demonstrate in Section 4.

3.2 Evaluation Interface

The evaluation interface provides the means of evaluating a signal function with inputs and producing effects in response to the signal function's outputs. We would like to produce a set of constructs that interacts well with Haskell's system for external IO.

The evaluation interface translates between signal functions and the standard Haskell construct for sequencing effects and external inputs, namely, *monads* [13]. The inspiration for monads is drawn from the rather esoteric domain of category theory, but the concept as applied to programming languages is actually rather simple.

By formulating the evaluation interface as a monad transformer, we need only define the operations relevant to the evaluation of signal functions, and we can depend on the constructs of the IO monad to interact with whatever inputs and outputs are necessary. In some cases, we may not wish to use the IO monad at all (e.g for testing or simulation). In this case, we can parameterize over another monad, such as the Identity monad (which has no special operations and whose context is just the value), or the State monad (which maintains an implicit state accessible by `get` and `put` operations).

The `SFEvalState` type constructor parameterizes over input types for signal functions, and underlying monads for a monad transformer, but is not itself a monad transformer. It describes the state of signal function evaluation. It consists of a record with four members: the current signal function, the set of handlers for outputs, the current input signal delta, and the last sample time.

The `SFEvalT` monad transformer does not make the `get` and `put` actions available, but uses them in its implementation of the `push`, `update`, and `sample` actions.

The `push` action pushes an event onto the input of the signal function, resulting in immediate evaluation of the relevant components signal function and possible actuation of handlers specified for output events. It is implemented by fetching the `SFEvalState`, applying the event continuation of the signal function contained by the `SFEvalState` to the pushed event, thus obtaining a new signal function and a list of events, applying the handlers contained in the `SFEvalState` to the output events, replacing the signal function in the `SFEvalState`, and replacing the `SFEvalState`.

The `update` action modifies the current input signal update, which will be sampled at the next `step` action. It has no immediately observable effect. It simply updates the value of one signal in the input signal vector.

The `step` action takes a time delta, and calls the signal continuation of the stored signal function with the time delta and the stored signal update. It actuates the outputs using the stored handlers, replaces the stored delta with an empty delta, and stores the resulting new signal function in the state.

4 Discussion

The system presented here, TimeFlies, demonstrates how using signal vectors to type inputs and outputs enables push-based evaluation of events in a signal-function system. We take advantage of this representation in several ways.

First, by separating components of inputs and outputs in the types, we are free to create distinct, and often partial, representations of the input or output of a signal function. This enables us to represent only the event occurrence being pushed at that time.

Second, this separation also permits us to separate the process of gathering the input to a signal function, and the process of handling its output, into different points in a program. Using the evaluation interface described, an event occurrence may be pushed onto one input of a signal function from one point in a program (e.g. a mouse click handler), an input signal may be updated in another (e.g. a mouse movement handler), and finally the system may be sampled in a third place (e.g. an animation or audio timed callback).

Finally, this approach enables further work on the implementation of the signal function system to be separated from changes in the interface. By enabling differing representations of the inputs and outputs of signal functions, we are free to change these representations without the need to further constrain the input and output types.

5 Related Work

Signal Function FRP was introduced as a model for Graphical User Interfaces [4]. The system was originally termed "AFRP" (Arrowized FRP). Yampa is a rewrite of AFRP where signal functions apply a number of ad-hoc optimizatons to themselves as they evolve. Yampa demonstrated a modest performance improvement over AFRP [6].

Reactive is a classic FRP system which implements push-based evaluation for events by transforming behaviors to "reactive normal form," where a behavior is a non-reactive behavior running inside a switch, whose event stream carries behaviors in reactive normal form. The system is evaluated by forking a Haskell thread to repeatedly sample the non-reactive behavior, and then blocking on the evaluation of the first occurrence in the event stream. When this occurrence is yielded, the evaluation thread for the behavior is killed and a new thread forked to evaluate the new behavior [2].

An alternate push-based system, based on *signal segments*, was presented in work on the Curry-Howard correspondence between a form of temporal logic (LTL) and FRP [**?**]. However, this system continues to represent events as option-valued signals. It therefore cannot be push-based in its response to individual events, and cannot be push-pull, evaluating signals by pull-based evaluation and events by push-based evaluation. It is push-based not in the sense of evaluating events only when they occur, but rather in that it provides a means to push signal segments (samples of a signal over intervals of time) to the system and wait on its response.

6 Conclusion

We have presented TimeFlies, a system for push-pull signal-function Functional Reactive Programming, and have shown how the use of a signal vectors as input and output types for signal functions, together with GADT-based representations of the inputs and outputs, permits the implementation of a push-pull system.

We have also described a general and flexible monadic evaluation interface for TimeFlies, which permits us to interface the TimeFlies system with different styles of IO systems, including multiple IO systems in the same application.

This opens up the exciting possibility that a signal-function FRP could become an efficient and general framework for writing interactive applications.

References

1. Elliott, C., Hudak, P.: Functional reactive animation. In: Proceedings of the second ACM SIGPLAN international conference on Functional programming. ICFP '97, New York, NY, USA, ACM (1997) 263–273
2. Elliott, C.: Push-pull functional reactive programming. In: Haskell Symposium. (2009)
3. Apfelmus, H.: reactive-banana library. http://hackage.haskell.org/package/reactive-banana
4. Courtney, A., Elliott, C.: Genuinely functional user interfaces. In: Proceedings of the 2001 Haskell Workshop. (2001) 41–69
5. Nilsson, H., Courtney, A., Peterson, J.: Functional reactive programming, continued. In: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell. Haskell '02, New York, NY, USA, ACM (2002) 51–64
6. 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. ICFP '05, New York, NY, USA, ACM (2005) 54–65
7. Schulthorpe, N.: Towards Safe and Efficient Functional Reactive Programming. PhD thesis, University of Nottingham, UK (2011)
8. Megacz, A.: Multi-level languages are generalized arrows. CoRR **abs/1007.2885** (2010)
9. Jones, M.P.: Functional programming with overloading and higher-order polymorphism. In Jeuring, J., Meijer, E., eds.: Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, May 24-30, 1995, Tutorial Text. Volume 925 of Lecture Notes in Computer Science., Springer (1995) 97–136
10. Cheney, J., Hinze, R.: First-class phantom types. Technical report, Cornell University (2003)
11. Xi, H., Chen, C., Chen, G.: Guarded recursive datatype constructors. SIGPLAN Not. **38**(1) (January 2003) 224–235
12. Peyton Jones, S., Vytiniotis, D., Weirich, S., Washburn, G.: Simple unification-based type inference for GADTs. In: Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming. ICFP '06, New York, NY, USA, ACM (2006) 50–61

13. Peyton Jones, S.L., Wadler, P.: Imperative functional programming. In: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. POPL '93, New York, NY, USA, ACM (1993) 71–84