Push-Pull Signal-Function Functional Reactive Programming

Edward Amsden

Rochester Institute of Technology
eca7215@cs.rit.edu

Abstract. Functional Reactive Programming is a promising class of systems for writing interactive and time-dependent programs. Signal-function FRP is a subclass of these systems which provides advantages of modularity and correctness, but has proven difficult to efficiently implement.
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.
We describe a signal-function FRP system which provides push-based evaluation for events, pull-based evaluation for signals, and a simple monadic evaluation interface which permits the system to be easily integrated 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 taking a single input and producing a single output, must accept multiple inputs and alter temporal behavior, including the production of multiple outputs, based on these inputs.

An FRP system will provide a means of manipulating *behaviors* and *events*. Behaviors are often referred to as *signals* in FRP literature, but the definition is the same. A behavior or signal is, semantically, a function from time to a value. An event is a discrete, possibly infinite, and time-ordered sequence of occurrences, which are times paired with values.

FRP systems can generally be categorized as "classic FRP," which corresponds to the originally described FRP system in that behaviors and events are manipulated directly and are first-class values in the FRP system, or "signal-function FRP," in which behaviors (generally termed signals in this approach) and events are not first-class values, but signal functions are first class values. Signal functions are time-dependent and 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.

Classic FRP was first described as an system for interactive animations [1]. Recent work on classic FRP has focused on efficient implementation. Reactive is a system for push-pull FRP [2]. 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, 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.

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

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 type 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.

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 of our implementation. Section 5 describes the current and future work on this system. Section 6 gives an overview of related efforts. Section 7 concludes.

2 System Design

Our goal is to produce a composable and efficient FRP system. Signal function FRP has an advantage in terms of composability, because it permits the construction of self-contained objects through both input and output composition, rather than purely through output composition as in classic FRP. Signal-function

[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.

```
data SVEmpty
data SVSignal a
data SVEvent a
data SVAppend svLeft svRight
```

Fig. 1. Signal vectors.

FRP also avoids problematic properties common to classic FRP systems such as a large class of time and space leaks [8].

Efficient implementations of signal-function FRP have been approached through runtime optimization [6], but all implementations have been pull-based for both signals and events. A truly efficient implementation will likely combine run-time optimization with push-based evaluation for events.

The concept of N-Ary FRP was to encode additional safety properties into the types of a signal-function FRP system [7]. This system introduced the concept of *signal vectors* as input and output types for signal functions. Signal vectors are combinations of signals and events. In N-Ary FRP, signal functions are represented at the type level, and type-level functions (type families in Haskell) are used to specify the representations of signal functions.

In our system, we construct representations of signal vectors using Generalized Algebraic Datatypes [9, 10]. The use of GADTs enables us to use signal vectors to instantiate type parameters in the types of signal vector representations, as well as in the types of signal functions. GADTs are available as an extension in the Glasgow Haskell Compiler [11].

This approach permits us to construct partial inputs and outputs for signal functions. For instance, we can construct a representation for a single event in a signal vector, and another representation of updated values for a subset of the signals in the signal vector, and yet another carrying values for all signals in a signal vector. The signal vector types are shown in Figure 1.

With the ability to construct partial representations of signal vectors, we can represent signal functions with a datatype carrying multiple functions, one for each type of input. This allows us to separate the pull-based processing of signals from the push-based reaction to events.

The exposed interface is a set of combinators for constructing signal functions, as well as combinators for describing the evaluation of a signal function. The interface is shown in Fig. 2.

Signal functions are produced by combining primitive signal functions using the `>>>` (sequential composition), `first`, and `second` routing combinators. Other routing signal functions are provided, but they are intended to be combined with, rather than to modify, other signal functions.

2.1 Examples of Signal Function Primitives

The most basic signal is the `identity` signal function, which, as its name suggests, simply passes its input to its output.

```
-- Signal Functions
type :~> svIn svOut

-- Infix type alias for SVAppend
type :^: svLeft svRight = SVAppend svLeft svRight

-- Basic signal functions
identity :: sv :~> sv
constant :: a -> SVEmpty :~> SVSignal a
never    :: SVEmpty :~> SVEvent a
asap     :: a -> SVEmpty :~> SVEvent a
after    :: Double -> a -> SVEmpty :~> SVEvent a

-- Lifting pure functions
pureSignalTransformer :: (a -> b) -> SVSignal a :~> SVSignal a
pureEventTransformer  :: (a -> b) -> SVEvent a :~> SVEvent a

-- Composition and routing
(>>>)           :: (svIn :~> svMiddle) -> (svMiddle :~> svOut)
                   -> (svIn :~> svOut)
first           :: (svIn :~> svOut) -> (svIn :^: sv) :~> (svOut :^: sv)
second          :: (svIn :~> svOut) -> (sv :^: svIn) :~> (sv :^: svOut)
swap            :: (svLeft :^: svRight) :~> (svRight :^: svLeft)
copy            :: sv :~> (sv :^: sv)
ignore          :: sv :~> SVEmpty
cancelLeft      :: (SVEmpty :^: sv) :~> sv
cancelRight     :: (sv :^: SVEmpty) :~> sv
uncancelLeft    :: sv :~> (SVEmpty :^: sv)
uncancelRight   :: sv :~> (sv :^: SVEmpty)
associate       :: ((sv1 :^: sv2) :^: sv3) :~> (sv1 :^: (sv2 :^: sv3))
unassociate     :: (sv1 :^: (sv2 :^: sv3)) :~> ((sv1 :^: sv2) :^: sv3)

-- Reactivity
switch :: (svIn :~> (svOut :^: SVEvent (svIn :~> svOut)))
          -> svIn :~> svOut

-- Feedback
loop :: ((svIn :^: svLoop) :~> (svOut :^: svLoop)) -> svIn :~> svOut

-- Time dependence
time  :: SVEmpty :~> Double
delay :: Double -> (SVEvent a :^: SVEvent Double) :~> SVEvent a
class TimeIntegrate

-- Joining
union          :: (SVEvent a :^: SVEvent a) :~> SVEvent a
combineSignals :: (a -> b -> c)
                  -> (SVSignal a :^: SVSignal b) :~> SVSignal c
capture        :: (SVSignal a :^: SVEvent b) :~> SVEvent a

-- Events
filter         :: (a -> Maybe b) -> SVEvent a :~> SVEvent b
```

Fig. 2. Signal function interface.

The `constant` signal function has an empty input, and produces a constant signal as its output.

The `asap` signal function produces an event at the first time interval after it begins, and never again.

The `filter` signal function applies a predicate to the values of event occurrences and passes them along only if the predicate is true of the value. The predicate outputs `Maybe` values to allow the user to avoid partial functions on types such as `Maybe` or `Either`.

The `associate` signal function is a routing function, which takes as input a combination of three signal vectors to which the signal vector append constructor is combined left-associatively, and produce the same signal vector as output, but with the signal vector append constructor combined right-associatively.

The `switch` function is the essential signal function for reactivity. It acts as the supplied signal function until that signal function produces an event on its right output. This event's value is a new signal function, which replaces the switch signal function.

The `loop` function permits a signal function to receive its own output as an input.

Implementations for these examples will be discussed in Sec. 3.3.

2.2 Evaluation Interface

The evaluation interface provides a monad [12, 13] for specifying input to signal functions and handling their output. This interface also allows input actions to be separated from output actions, and from each other, so that a signal function may receive input from multiple IO systems and have its output handled by multiple output systems.

The evaluation interfaceis a monad transformer [14]. This allows a signal function evaluation to be constructed from primitive evaluation actions (event pushing, input signal updating, and sampling) as well as actions from the underlying monad. These actions can then be run in the underlying monad to actuate the signal function. Because the actuation takes a signal function's state as input and produces a new state as output, in addition to the monadic side effects of handling the signal function's output, different evaluation actions may be taken on the same signal function from distinct locations within an application's code. This allows the evaluation interface to be easily integrated with event-loop style systems as well as traditional imperative IO systems.

The evaluation interface consists of the monad transformer type, functions for constructing event inputs, initial input signal samples, and input signal updates, functions for constructing the vector of output handlers for a signal function, and evaluation actions. The full interface is shown in Fig. 3.

```
-- Input helpers
class SVRoutable -- Instances: SVSignalUpdate,
                 -- SVEventOccurrence
svLeft :: (SVRoutable r) =>
          r svLeft -> r (SVAppend svLeft svRight)
svRight :: (SVRoutable r) =>
           r svRight -> r (SVAppend svLeft svRight)

-- Signal inputs
data SVSignalUpdate
data SVSample
sample :: a -> SVSample (SVSignal a)
sampleEvt :: SVSample (SVEvent a)
sampleNothing :: SVSample SVEmpty
combineSamples :: SVSample svLeft -> svSample svRight
                  -> svSample (svLeft :^: svRight)
svSig :: a -> SVSignalUpdate (SVSignal a)

-- Event inputs
data SVEventInput
svOcc :: a -> SVEventInput (SVEvent a)
-- Actuation
update :: (Monad m) => SVSignalUpdate svIn
           -> SFEvalT svIn svOut m ()
push   :: (Monad m) => SVEventInput svIn
          -> SFEvalT svIn svOut m ()
sample :: SVEvalT svIn svOut m ()
-- Running
data SVHandler
emptyHandler :: SVHandler m SVEmpty
eventHandler :: (a -> m ())
                -> SVHandler m (SVEvent a)
signalHandler :: (a -> m ())
                 -> SVHandler m (SVSignal a)
combineHandlers :: SVHandler m svLeft
                   -> SVHandler m svRight
                   -> SVHandler m (svLeft :^: svRight)
initSFEvalT :: SVHandler m svOut -> SVSample svIn
               -> Double ->(svIn :~> svOut)
               -> SFEvalState m svIn svOut
runSFEvalT :: SFEvalT svIn svOut m a
              -> SFEvalState m svIn svOut
              -> m a
```

Fig. 3. Evaluation interface.

```
data SVSample sv where
  SVSample    :: a -> SVSample (SVSignal a)
  SVSampleEvt :: SVSample (SVEvent a)
  SVNothing   :: SVSample SVEmpty
  SVBoth      :: SVSample svLeft -> SVSample svRight
                 -> SVSample (svLeft :^: svRight)

sample         :: a -> SVSample a
sampleEvt      :: SVSample (SVEvent a)
sampleNothing  :: SVSample SVEmpty
combineSamples :: SVSample svLeft -> SVSample svRight
                  -> SVSample (svLeft :^: svRight)
splitSample    :: SVSample (svLeft :^: svRight)
                  -> (SVSample svLeft, svSample svRight)
sampleValue    :: SVSample (SVSignal a) -> a
```

Fig. 4. Signal sample representation.

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 Input and Output Representations

As discussed in Section 2, our implementation requires representations of the inputs and outputs of signal functions. Since signal functions are typed using signal vectors, the representation types must be parameterized over signal vectors. But signal vectors are uninhabited types, and in ordinary Haskell we cannot declare a data constructor which uses a component of a type parameter. Further, we cannot restrict which types may instantiate a type parameter for a particular data constructor.

GADTs lift these restrictions, permitting type constraints to be applied to individual data constructors. In the declaration of a GADT, a type signature is provided for each data constructor, including the types of the constructor's parameters. When a GADT is pattern-matched, the types of the parameters are inferred using the constraints given in this signature. This is called type refinement.

Using GADTs, we can construct representations of signal vectors to use in our implementation. The first such representation carries a value for each signal component of a signal vector, and is shown in Figure 4.

A value of this type represents the entire vector, though it has nullary constructors at event leaves and empty leaves. For event occurrences, we want to represent only a single point in the vector, as shown in Figure 5. This representation uses two constructors, each of which leaves one or the other of the type variables in the signal vector `SVAppend` constructor unconstrained.

```
data SVOccurrence sv where
  SVOccurrence :: a -> SVOccurrence (SVEvent a)
  SVOccLeft    :: SVOccurrence svLeft
                  -> SVOccurrence (svLeft :^: svRight)
  SVOccRight   :: SVOccurrence svRight
                  -> SVOccurrence (svLeft :^: svRight)

occurrence        :: a -> SVOccurrence (SVEvent a)
occLeft           :: SVOccurrence svLeft
                     -> SVOccurrence (svLeft :^: svRight)
occRight          :: SVOccurrence svRight
                     -> SVOccurrence (svLeft :^: svRight)
chooseOccurrences :: SVOccurrence (svLeft :^: svRight)
                     -> Either (SVOccurrence svLeft) (SVOccurrence svRight)
fromOccurrence    :: SVOccurrence (SVEvent a) -> a
```

Fig. 5. Event occurrence representation.

```
data SVDelta sv where
  SVDelta        :: a -> SVDelta (SVSignal a)
  SVDeltaNothing :: SVDelta sv
  SVDeltaBoth    :: SVDelta svLeft -> SVDelta svRight
                    -> SVDelta (svLeft :^: svRight)
delta         :: a -> SVDelta (SVSignal a)
deltaNothing  :: SVDelta sv
combineDeltas :: SVDelta svLeft -> SVDelta svRight
                 -> SVDelta (svLeft :^: svRight)
updateSample  :: SVDelta sv -> SVSample sv -> SVSample sv
```

Fig. 6. Signal delta representation.

The final representation "signal deltas," carries replacement values for some points in a signal vector. Here, we combine the unconstrained type variable from the occurrence representation with the "Both" constructor from the sample representation. The unconstrained type variable will be placed in a separate constructor, which allows the construction of empty signal deltas, as is shown in Figure 6.

3.2 Signal Function Representations

With representations for signal function inputs and outputs in place, we turn to the representation of signal functions. A signal function must be able to respond to time increments and new signal samples, as well as event occurrences. Since we wish to be able to "push" event occurrences independently of sampling, events are handled using the last-updated time and signal sample.

When a signal function is asked to respond to either of these types of inputs, it must produce an output consisting of zero or more event occurrences, a new

```
data Initialized
data NonInitialized

type :~> svIn svOut = SF NonInitialized svIn svOut

data SF init svIn svOut where
  SF :: (SVSample svIn -> (SVSample svOut, SF Initialized svIn svOut))
        -> SF NonInitialized svIn svOut
  SFInit :: (Double -> SVDelta svIn
             -> (SVDelta svOut, [SVOccurrence svOut],
                 SF Initialized svIn svOut))
            -> (SVOccurrence svIn -> ([SVOccurrence svOut],
                                      SF Initialized svIn svOut))
            -> SF Initialized svIn svOut
```

Fig. 7. Signal function representation

signal function with the same type (to enable reactivity and state), and, if it is responding to a time and sample update, a delta which represents updates to the sample of its output signals.

Finally, there is a special case for a signal function at time 0. It must be provided with an initial input sample before it can respond to incremental time and sample updates or to event occurrences.

Signal functions are represented as a datatype with two constructors. The first constructor wraps a function from an input sample to an output sample and an initialized signal function. The second wraps two functions. The first accepts a time delta and a signal delta, and produces a signal delta, a list of event occurrences, and a new signal function as the output. The second accepts an event occurrence and produces a list of event occurrences and a new signal function as the output. The signal function representation is shown in Figure 7.

This representation allows a signal function to respond to events and time updates separately, and does not enforce the representation of events during time updates, as previous signal function systems do.

3.3 Signal Function Implementations

Now that we have established representations for signal functions and their inputs and outputs, we show several examples of how signal function combinators are implemented.

The `identity` signal function's initialization function takes a sample, and returns that sample as its output sample. The initialized function returned is the `identityInit` function, which is not exported by the module. The time and sample function ignores the provided time delta, produces the input signal delta as the output signal delta, and returns `identityInit` as the new signal function. The event function takes an event occurrence and returns a singleton

```
identity :: sv :~> sv
identity = SF (\sample -> (sample, identityInit))

identityInit :: SF Initialized sv sv
identityInit = SFInit (\_ delta -> (delta, [], identityInit))
                      (\occ -> ([occ], identityInit))
```

Fig. 8. `identity` signal function implementation.

list containing that occurrence, along with the `identityInit` function as the replacement signal function. The implementation is shown in Figure 8.

The `constant` signal function is initialized with a value, which it then produces as its output forever. Thus, the initialized version of `constant` never produces any output other than itself as a replacement signal function, and an empty signal delta. The implementation of `constant` is shown in Fig. 9.

```
constant :: a -> SVEmpty :~> SVSignal a
constant x = SF (\_ -> (constantInit, sample x))

constantInit :: SF Initialized SVEmpty (SVSignal a)
constantInit = SFInit (\_ _ -> (deltaNothing, [], constantInit))
                      (\_ -> ([], constantInit))
```

Fig. 9. `constant` signal function implementation.

The `asap` signal function implementation requires a parameter for the initialized version of the signal function, to specify the value of the event occurrence. It replaces itself with the initialized version of the `never` signal function after the first time step. The implementation is shown in Fig. 10.

```
asap :: a -> SVEmpty :~> SVEvent a
asap x = SF (\_ -> (sampleEvt, asapInit x))

asapInit :: a -> SF Initialized SVEmpty (SVEvent a)
asapInit x = SFInit (\_ _ -> (deltaNothing, [occurrence x], neverInit))
                    (\_ -> (never))
```

Fig. 10. `asap` signal function implementation.

The `filter` function accepts events, applies a `Maybe` predicate to their values, and produces the value produced by the predicate as an event occurrence, or no occurrence if the value is `Nothing`. It is implemented by having the event

```
filter :: (a -> Maybe b) -> SVEvent a :~> SVEvent b
filter p = SF (\_ -> (sampleEvt, filterInit p))

filterInit :: (a -> Maybe b) -> SF Initialized (SVEvent a) (SVEvent b)
filterInit p = SFInit (\_ _ -> (deltaNothing, [], filterInit p))
                      (\evtOcc -> (maybe [] ((:[]) . occurrence) $
                                   fromOccurrence evtOcc,
                                   filterInit p))
```

Fig. 11. `filter` signal function implementation.

occurrence response function apply the `maybe` function from the Haskell prelude to the value returned by the predicate, as shown in Fig. 11

The `associate` signal function is a routing signal function. It transforms a signal vector which is left-associated at the top level to one that is right-associated at the top level. Its implementation is representative of all of the routing functions, and is shown in Fig. 12.

The `switch` function is the basic combinator used to introduce reactivity. It is given a signal function whose output is the append of two signal vectors. The left side of the signal vector is passed on as the output of the reactive signal function, and the right side is an event carrying signal functions. When an event occurrence is present on the right-side event output, the signal function carried by this occurrence replaces the signal function constructed by switch. Due to space concerns, the implementation is elided, but a brief description will suffice. The signal function stores the input sample provided during initialization, and updates it with deltas. When an event occurrence carrying a signal function is produced by either the event or time function of the wrapped signal function, the stored signal sample is used to initialize the new signal function. If this occurs while handling an event input, the sample output by the new signal function's initialization is stored and is combined with its first output delta to produce the output delta at the next time step.

The `loop` function allows a signal function to see a component of its own output as input. This is primarily useful when a signal function has components which mutually depend on each others outputs, such as in physics simulations or games. Care must be taken that the feedback output is not immediately dependent on the feedback input, or sampling the signal function will not terminate. Loop is implemented by generating a recursive list (which will be infinite if the feedback is not decoupled) of event inputs, and by splitting the output signal delta and using the right side output delta as the right side input delta within a recursive let-binding. (The let-binding in Haskell always admits recursive bindings.) If the system is not completely decoupled, this will result in non-termination during an evaluation step. Decoupling can be achieved using `delay` primitive.

```
associate :: SVAppend (SVAppend sv1 sv2) sv3
             :~> SVAppend sv1 (SVAppend sv2 sv3)
associate =
  SF (\sigSample -> let (sigSampleLeft, sigSampleRight) =
                          splitSample sigSample
                        (sigSampleLeftLeft, sigSampleLeftRight) =
                          splitSample sigSampleLeft
                    in (combineSamples sigSampleLeftLeft
                        (combineSamples sigSampleLeftRight sigSampleRight),
                        associateInit))

associateInit :: SF Initialized (SVAppend (SVAppend sv1 sv2) sv3)
                                (SVAppend sv1 (SVAppend sv2 sv3))
associateInit = SFInit (\_ sigDelta -> let (sigDeltaLeft, sigDeltaRight) =
                                             splitDelta sigDelta
                                           (sigDeltaLeftLeft,
                                            sigDeltaLeftRight) =
                                             splitDelta sigDeltaLeft
                                       in (combineDeltas sigDeltaLeftLeft
                                           (combineDeltas sigDeltaLeftRight
                                                          sigDeltaRight),
                                           [], associateInit))
                       (\evtOcc -> (case chooseOccurrence evtOcc of
                                      Left leftOcc ->
                                        case chooseOccurrence leftOcc of
                                          Left leftLeftOcc ->
                                            [occLeft leftLeftOcc]
                                          Right leftRightOcc ->
                                            [occRight $ occLeft leftRightOcc]
                                      Right rightOcc ->
                                        [occRight $ occRight $ rightOcc],
                                    associateInit))
```

Fig. 12. `associate` signal function implementation.

3.4 Evaluation Interface Implementation

The evaluation interface must maintain state including the current signal function, the current time (to produce time deltas), updates to the input sample which have yet to be sampled, and the output handlers.

The output handlers are stored in a structure similar to that for a signal sample. The difference is that both signal and event leaves contain values, and these values are functions from the leaf type to another type. The handler datatype is shown in Fig. 13.

We now need a datatype to hold the various components of the evaluation state. This type is shown in Fig 14.

The evaluation interface itself is a monad transformer, which we implement as a Haskell newtype wrapping the `StateT` monad transformer. We do not ex-

```
data SVHandler out sv where
  SVHandlerEmpty  :: SVHandler out SVEmpty
  SVHandlerSignal :: (a -> out) -> SVHandler out (SVSignal a)
  SVHandlerEvent  :: (a -> out) -> SVHandler out (SVEvent a)
  SVHandlerBoth   :: SVHandler out svLeft -> SVHandler out svRight
                     -> SVHandler out (SVAppend svLeft svRight)
```

Fig. 13. Handler datatype.

```
data SFEvalState m svIn svOut
  = SFEvalState {
                  esSF :: SF Initialized svIn svOut,
                  esOutputHandlers :: SVHandler (m ()) svOut,
                  esLastTime :: Double,
                  esDelta :: SVDelta svIn
                }
```

Fig. 14. Evaluation state datatype.

port the raw `put` and `get` actions of the state monad, but instead implement the push, update, and sampling operations for signal function evaluation using `put` and `get`. The evaluation monad transformer is shown in Fig. 15. Instances of typeclasses including `Monad` and `MonadTrans` are derived using the "GeneralizedNewtypeDeriving" extension to the Glasgow Haskell Compiler.

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

```
newtype SFEvalT svIn svOut m a = StateT (SFEvalState m svIn svOut) m a
```

Fig. 15. Signal function evaluation monad transformer.

to change these representations without the need to further constrain the input and output types.

5 Ongoing and Further Work

TimeFlies, the system described here, has been implemented, but not extensively tested. The immediate goal is to create a real-time application which will permit a performance and implementation comparison of TimeFlies with Yampa, the current state-of-the-art pull-based signal-function system.

In the future, we hope to apply run-time optimizations, using the technique used for Yampa, to create a push-pull self-optimizing signal-function system. Further, we hope to use this system as a basis for exploring signal-function FRP as a basis for general-purpose application frameworks.

6 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].

7 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. Liu, H., Hudak, P.: Plugging a space leak with an arrow. Electronic Notes in Theoretical Computer Science **193**(0) (2007) 29–45
9. Cheney, J., Hinze, R.: First-class phantom types. Technical report, Cornell University (2003)
10. Xi, H., Chen, C., Chen, G.: Guarded recursive datatype constructors. SIGPLAN Not. **38**(1) (January 2003) 224–235
11. 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
12. 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
13. Peyton Jones, S.: Tackling the awkward squad: Monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In Hoare, T., Broy, M., Steinbrüggen, R., eds.: Engineering Theories of Software Construction. Volume 180 of NATO Science Series: Computer & Systems Sciences. IOS Press, Amsterdam, The Netherlands (2001) 47–96
14. Liang, S., Hudak, P., Jones, M.: Monad transformers and modular interpreters. In: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. POPL '95, New York, NY, USA, ACM (1995) 333–343