A Survey of Functional Reactive Programming

Concepts, Implementations, Optimizations, and Applications

Edward Amsden
Rochester Institute of Technology
eca7215@cs.rit.edu

Abstract

Functional Reactive Programming (FRP) provides a conceptual framework for implementing reactive systems. It is a relatively recent model of programming, but has already been explored, implemented, and optimized in several useful ways. We survey the literature on FRP, its implementation, optimization, and uses, and present ideas for further research, along with some examples.

Categories and Subject Descriptors []

General Terms Programming Languages, Functional Programming, Functional Reactive Programming

Keywords Programming, Functional, Denotational, Reactive, Time

1. Introduction

Functional Reactive Programming (FRP) is a method of modeling *reactive* (i.e. time-varying and responding to external stimuli) behavior in purely functional languages. It first appeared as a method of structuring a composable library for animation [7]. The model presented was already very general, and research quickly began to focus on precise definitions of the semantics as well as optimized implementations.

FRP permits the modeling of systems that must respond to input over time in a simple and declarative manner. It has been used in the construction of an animation library, an arcade-style video game, a modular sound synthesis library, and a robotics library. Possible further applications include general signal processing applications, GUI toolkits, simulators, and almost any embedded application. A program in an FRP language generally corresponds quite closely to a mathematical model of the system being implemented.

The goal of FRP implementations is to enable:

- Safe programming: Programs should have as much correctness checking as possible done by the compiler.
- Efficient programming: Since most FRP programs will be expected to respond in real-time, efficient operation and aggressive optimization are necessary.
- Composability: Building off of a functional framework, FRP permits programs to be built from smaller programs piecewise, rather than creating monolithic, problem-specific codebases.

Submitted for Independent Study in Functional Reactive Programming, Spring 2010-2011. Estimated hours: Average of 6 hours per week reading, for a total of 54 (no reading scheduled for 10th week), 15 for writing and editing, 12 for meetings, scheduled and unscheduled. Estimate total: 81 hours.

The primary concepts of FRP are *signals* or *behaviors* (time-varying values) and *events* (collections of instantaneous values, or time-value pairs). Every instance of FRP conceptually includes both, though they may not be first-class values, and one may be defined in terms of the other.

FRP achieves reactivity by providing constructs for specifying how signals or behaviors change in response to events. This may be the primary way of specifying and implementing behaviors (as in EFRP [21] and Frappé [5]) or behaviors may be semantic functions of time which are replaced on event occurrences (this is the model in most other implementations).

Several semantic models and variations on these models have been explored, and implementations of these models have usually been proposed alongside them. Several optimization techniques for these implementations have also been explored, both as static transformations and dynamic reconfigurations. Most of the optimizations are specific to a particular implementation, though one (Causal Commutative Arrows [14]) depends more on the semantic framework and less on the specifics of the underlying implementations.

2. Semantics

Much of the difficult work on FRP has been in the definition of suitable semantics. Three distinct semantic frameworks have emerged, each utilizing many of the same concepts, but with important differences. The main differences involve the treatment of signals and the representation of events.

Semantic definitions are necessary to permit reasoning about how to form FRP programs. At such a high level of abstraction, there is not a simple and obvious correspondence between the computations the computer does and the computation the programmer wishes to express. Thus it is important that the programmer have an exacting definition of what computation is expressed by which construct in an FRP library.

In Classic FRP (Section 2.1), signals (generally called behaviors) and events are first class values which are directly manipulated by various language constructs. Signal function semantics (Section 2.2) include the concept of signals, but do not include them as first class values or reactive constructs. Rather, functions on signals are manipulated and made reactive. Events are represented as a special case of signals, and manipulated with specialized signal functions. N-ary FRP (Section 2.3) abstracts further to signal vector functions, removing the ambiguity between an n-arity signal function and a 1-arity signal function on an n-tuple signal, and providing a mechanism to reintroduce events as a separate entity from signals [18].

2.1 Classic FRP

The original FRP semantics were outlined as a model for Fran [7] and later given a formal denotative semantics [19]. Later papers [8,

```
-- Behavior (Function on time)
data Behavior a

-- Occurrence
newtype Occ a = Occ (Time, a)

-- Event
type Event a = [Occ a]

-- Lifting
lift0 :: a -> Behavior a
lift1 :: (a -> b) -> Behavior a -> Behavior b
lift2 :: (a -> b -> c) -> Behavior a -> Behavior b -> Behavior c
...

-- Reactivity, one of:
until :: Behavior a -> Event (Behavior a) -> Behavior a

switcher :: Behavior a -> Event (Behavior a) -> Behavior a
```

Figure 1. Types and combinators for Classic FRP

```
-- Functor
instance_sem Functor Behavior where
  fmap = lift1

-- Applicative Functor
instance_sem Applicative Behavior where
  pure = lift
  (<*>) = lift2 ($)
```

Figure 2. Semantic instances of Haskell typeclasses for behaviors

```
instance_sem Monad Event where
  return x = [(−∞, x)]
  occs (join ee) = foldr merge [] . map (uncurry delayOccs) (occs ee)

-- Makes each occurrence time at least the given time
delayOccs :: Time -> Event -> Event
```

Figure 3. Monad instance for events (simplified) [8].

18] refer to this model as "Classic FRP"; we continue this convention.

Classic FRP takes behaviors and events as first class values. Events are considered to be improving lists of occurrences[1], while behaviors are modeled semantically as functions of time [7].[2] Reactivity is modeled by combinators which take an initial behavior and an event whose occurrence values are new behaviors. The resulting behavior acts as the first behavior up to the first occurrence, and thereafter acts as the behavior encapsulated in the occurrence [7, 8, 19]. This combinator is variously called "until" [19], "switcher" [8], or some variant of those names, but the behavior is essentially the same (Figure 1).

The semantic definitions of behaviors given in most Classic FRP are very similar. Behaviors have lifting functions $lift_n$, which lift functions of arity n to functions of the same arity on behaviors ($lift_0$ or just $lift$ creates a constant-valued behavior), and an at function which given a behavior and time produces the value of the behavior at that time.

The implementers of Fran gave a formal semantics for Classic FRP.[3] An attempt was soon made at a correctness proof for the semantics of FRP [19]. This set of proofs introduced the notion of uniform convergence, which amounts to the limit of the result of the sampled implementation as the sampling interval goes to 0 being equal to the result of the semantic function.

The description of the *Reactive* implementation of Classic FRP provides semantic instances of functors and applicative functors

[1] An improving value is one in which more of the value becomes available as the computation proceeds. An occurrence is a value paired with a time.

[2] This model for behaviors is inefficient for direct implementation, since it provides no way of discarding unneeded information about past times, but it is useful semantically.

[3] The term FRP does not appear in the Fran paper, but seems to have come into common usage shortly thereafter, as it appears that every subsequent paper on the topic does refer to FRP.

```
-- Behavior
newtype Behavior a = Behavior (TimeStep -> (a, Behavior a))

-- Event (Same as semantic type)
data Event a
```

Figure 4. Sample implementation types for Classic FRP

for behaviors. These instances generalize to the standard semantic functions stated above, as given in Figure 2. A monad instance is also definable, but not considered useful. Semantic instances are also provides for events. Event is a monoid, with the identity being the never-occurring event and the operation being the merge[4] of the occurrence lists of both events. Events also form a functor, mapping a function over occurrence values without modifying occurrence times. Defining an applicative functor instance from the monad instance (below) would result in every value occurrence-function occurrence pair producing an occurrence in the event resulting from application. This is not considered useful. Finally, events form a monad instance (Figure 3). This permits a comfortable way to write transformations on events, since the sequencing operator will carry occurrence times along with transformed values [8].

2.2 Signal Functions

Signal functions were introduced as a means of avoiding time and space leaks (see below) and simplifying the use of external inputs as signals [16]. They are most often given as an instance of the Arrow framework [11], which provides a structured generalization of functions.

A signal function is a reactive construct which takes input and produces output, rather than simply producing output. Thus, a signal function conceptually corresponds to a behavior which carries a function. In signal function semantics, behaviors and events are not first class values [16]. Rather, signal functions are built into a reactive framework which is then given input and sampled. Reactivity is also defined at the signal function level. Rather than signals switching on events, signal functions switch on events. So rather than composing behavior values and having an implementation framework sample the resulting behavior, signal functions are composed and an evaluator provides input as it samples output.

Signal functions have not been given the formal semantic treatment provided for Classic FRP. It is helpful, however, to consider signal functions as they are usually implemented: A signal function is a function from a time step and input sample to an output sample and continuation (new signal function) for the next time step.

Signal functions allow us to dispense with a large class of time and space leaks. In Classic FRP semantics, a behavior defines its value at every point in time and thus we can depend on past, present, and future values for our present value. In a signal function, the only way to preserve any past time information is to explicitly include it by having the continuation close over the values to be preserved. Since a signal function can be defined only in terms of its input and certain primitives (such as time), a signal function cannot evaluate another signal function at an arbitrary time. These restrictions make it much more difficult to produce time leaks within signal function programs.

Other advantages and disadvantages to this approach will be discussed in Section 3 on implementation.

2.3 N-ary FRP

N-ary FRP [18] is an improvement on signal functions where, conceptually, signal functions are not functions from signal to signal, but rather signal vector to signal vector. A signal vector is a conceptual group of signals. This avoids the ambiguity present in signal function semantics, where a tuple of signals and a signal carrying a tuple are indistinguishable. Signal vectors distinguish them by providing a separate construct for multiple signals. This then permits signal descriptors to be used to tag signals as continuous or discrete[5] , removing the ambiguities of event signals and permitting routing to be expressed at the signal function level rather than by lifting pure functions.

3. Implementation

3.1 Classic FRP

Classic FRP has been implemented in several different languages using several techniques. The original implementation made use of the monotonic nature of time sampling to discard information about past times in behaviors. A behavior was represented as a function from time to a value at that time, and a continuation (new behavior) with information for previous times discarded. This gives the ability to remove space leaks where behaviors accumulate information about past times that will not be used again, and time leaks where event-dependent behaviors must search an event list starting from time 0, to find the last event prior to the sampling time [7]. (Events are represented as simple lazy lists of occurrences.) Thus the definitions for the implementation might be as in Figure 4.

In Frappé [5] (implemented in Java), FrTime [4] (implemented in Scheme), RT-FRP [20], and EFRP [21] (both implemented as non-embedded languages), evaluation is driven by incoming events rather than the sampling of outputs. Rather than directly representing functions from time, behaviors represent computation trees which are recomputed on incoming events, with the resulting value pushed as output. Behaviors are thus represented as a dependency graph with references from dependencies to dependents. Integration (where implemented) is accomplished by having an evenly spaced time event source to update the time behavior.

The *Reactive* implementation introduces a normal form known as *reactive normal form*, in which every behavior is of the form b '$switcher$' e where b being a non-reactive behavior and all behaviors in the occurrences of the event e are also in reactive normal form. In this case, the reactive expression is only evaluated on event occurrences, with each resulting behavior spawning a thread which samples the behavior at successive time values. The implementation does include a representation of constant behaviors which are sampled only once [8].

3.2 Signal Functions

Signal functions are implemented as functions from a *time step* and input value to an output value and continuation (new signal function). Unlike behaviors, however, which use the continuation nature to explicitly eliminate information such as event occurrences which are no longer needed, signal functions use continuations

[4] Merge is the familiar sorting operation, comparing by occurrence time.

[5] Continuous signals are defined at every point in time, while discrete signals are defined at countably many points in time. This corresponds to behaviors and events.

to explicitly store state which will be required at the next time step [16].

The continuation-based representation is not exposed to the library user, but rather is used to implement primitives and combinators. General combinators from the arrow framework such as composition, splitting, and parallelization[6] [11] must be recursively defined to combine continuations of the combined signal functions and to distribute the time step input appropriately. Lifting of pure functions is done by creating a signal function which ignores time step input and returns itself as a continuation.

Integration, time, and other stateful primitives are accomplished by returning a recursively defined closure over the state as the continuation. For instance, the time signal function (which ignores the input and returns the "local" time or time since the signal function was switched into the network) closes over the sum of time steps up to the given point and returns as an output value that sum plus the input time step.

Switching is accomplished by simply returning the switched-into signal function as the continuation. This allows an important degree of flexibility in switching. A normal switch will evaluate the new signal function at the time interval of the event occurrence, returning its output and continuation. A decoupled switch, however, evaluates the old signal function at the moment of switching, merely returning the new signal function as a continuation. Since signal functions permit signal feedback, it is important to provide ways to make such feedback loops *well-founded*, that is, loops should not depend on the current time feedback to produce the current time output. Using a decoupled switch as the signal function and switching events as feedback is one method of ensuring the well-foundedness of these feedback loops.

One important characteristic of signal functions is that it is much more obvious how to provide input from arbitrary sources during evaluation. In Classic FRP, input behaviors must be provided by the library implementer and generally rely on "shortcuts" such as Haskell's *unsafePerformIO* or *unsafeInterleaveIO* functions. Newer experimental implementations which do not rely on such shortcuts fail to observe sharing, that is, behaviors must be re-sampled each time they are evaluated. There is also no enforcement of synchronicity between input and output sampling, and thus no guarantee that a behavior will have a value available for a particular time when it is sampled at that time, even if time is monotonically increasing. Since signal functions have explicit inputs, there is a great deal of flexibility in defining those inputs. In signal function evaluation, a time step is computed along with input to the signal function, and output for that time step is observed in the same evaluation step. Yampa [16], for instance, has an IO evaluation loop which permits the user to define an arbitrary IO action to fetch the input sample and time step at each cycle.

3.3 N-ary FRP

Implementations of N-ary FRP follow the same structure as signal functions (Section 3.2). The semantics give rise to stronger type safety and the possibility of optimizations of events and routing, but these optimizations were not explored in the surveyed work.

4. Optimization

While the stated semantics of the various forms of FRP quite often provide a tantalizingly simple implementation, such naïve implementations are inefficient. Every practical implementation of FRP includes some form of optimization intrinsically. However, further and less obvious optimization techniques have also been demonstrated for Classic FRP and signal functions. Most optimizations are specific to a particular implementation technique, but the Causal Commutative Arrows optimization [14] (Section 4.2), while not (yet)[7] applicable to general signal functions, can optimize non-reactive signal functions which obey a certain set of laws related to arrow looping, regardless of the specifics of the actual implementation.

4.1 Signal Functions and GADTs

In signal function networks, a number of somewhat ad-hoc optimization opportunities arise. The addition of Generalized Algebraic Datatypes (GADTs) to the GHC compiler allowed somewhat simple exploitation of several of these opportunities. GADTs allow the data constructors of a particular type constructor to be given individual type signatures, thus permitting data constructors to specify arguments or constraints on arguments to the type constructor.

This definition permitted the optimization of the Yampa [16] signal function implementation to be optimized by dynamically combining signal functions at points of reactivity. Continuations are no longer represented as signal functions, but rather as a GADT marking special properties of continuations. Combined with pattern matching, this representation permits an efficient way to recognize optimization opportunities as continuations are composed into the signal function network.

For instance, the data constructor for the identity signal function enforces the type restriction that the input type is identical to the output type, thus permitting optimizations which discard the identity function rather than composing it into the signal function expression to pass typechecking. Standard algebraic datatypes would not permit this, as the compiler would not be able to identify from the data constructor that the types were in fact equal, and would produce a type error. In a similar manner, constant and stateless functions can also be eliminated either by single evaluation or function-level (rather than signal-function level) composition [15].

A disadvantage of this optimization is the overhead required every time a new continuation is produced (though the optimization itself eliminates many such productions) to compose the continuation into the network. Another disadvantage is the "small combinatorial explosion" of patterns required for more special cases and optimizations to be recognized. In benchmarks included with the original description, significant speedups were demonstrated for some applications, despite the overhead of pattern matching [15].

4.2 Causal Commutative Arrows

Causal Commutative Arrows is a generalized optimization for a subclass of the Arrow typeclass[8]. If an arrow instance can be shown to obey a small set of identities on the loop operator, and if it includes a "dec" operator which removes dependence on the current input, then any expression of this arrow instance can be statically reduced to a normal form consisting of an outer loop and a single, pure expression. Though it would seem that the reactive nature of full signal functions would prevent them from being a direct instances of this class, non-reactive stream transformers are proven to obey these identities. This static reduction then permits the full optimizing power of the compiler to work on the pure expression given [14].

It is not clear that an advantage would be gained by applying a similar optimization to signal functions. In particular, the reactive nature of signal functions seems to require dynamic optimization [15]. Thus little advantage would be seen from compile-time optimizations to a static reduction. However, it may be that the

[6] See Appendix A

[7] It is not yet clear whether the laws and normal form required for this optimization can be applied to reactive signal functions. The normal form transformation would almost certainly have to be made dynamic and recursive rather than static to handle reactivity.

[8] More specifically the ArrowLoop typeclass

CCA normal form (or an extended normal form which permits the expression of reactivity) holds intrinsic performance advantages regardless of compiler optimization.

4.3 Lowering

The FrTime implementation of FRP in Scheme takes advantage of the language-rewriting and imperative features of Scheme to construct a dataflow graph where sources signal sinks when they update. Since the reactivity in FrTime is implicit rather than explicit (primitive operators in Scheme are implicitly lifted to operate on signals), the unoptimized implementation of FrTime evaluates pure expressions as a dataflow graph rather than as pure Scheme expressions.

The key idea of lowering is that any signal may have a lowered form. For primitive expressions, such as literals or library functions, the lowered form is simply the unlifted expression. A lifted expression which has a lowered form if all of the elements of the expression have a lowered form. Lowered expressions are cached as they are discovered and the lifted signals given a reference to them. This is particularly key for lambda expressions, where the lambda may be lowered but the signal it is applied to may not be. Thus it is important to retain the signal version of the lambda, though it is itself internally optimized [3].

This optimization eliminates overhead from pure signals notifying other pure signals of updates, thus permitting the Scheme compiler to optimize expressions and removing steps from the computation. In this way it is similar to the CCA optimization described in Section 4.2). The GADT optimizations for signal functions (Section 4.1 are similar in recognizing adjacent pure expressions but operate in a more ad-hoc manner and dynamically rather than statically.

4.4 Push-Pull FRP

The *Reactive* implementation of Classic FRP can be considered an optimization, since it provides a normal form which permits efficient evaluation, and defines its primitives and combinators such that resultant programs are encoded in this normal form. The reactive normal form forces reactivity to be encoded at the top level of an expression, and maintains that guarantee through the lifecycle of a program run. This permits the framework itself to be evaluated only on event occurrences, resultant behaviors may then be sampled at increasing time values until the next event occurrence.

5. Demonstrations

Functional Reactive Programming has been demonstrated to work in practical application settings. The original statement of FRP was in terms of an animation library. Other demonstrations have included simulated robotics, a rewrite of the classic arcade game ”Space Invaders”, and a modular sound synthesizer.

5.1 Robotics

Signal function based FRP has been used to implement a demonstration robotics framework [10]. This framework permitted the definition of signal functions from a composite input datatype containing sensor events and signals to an output datatype which contained motor speeds. The implementation of the evaluator was not discussed, as the intent was to provide a fairly comprehensive tutorial on the constructs of signal functions in general and Yampa [16] in particular.

5.2 Space Invaders

A successful demonstration of the performance of signal function implementations was given by an implementation of the classic game ”Space Invaders” in Yampa [6]. The description of this implementation discussed in great detail a unique feature of the Yampa library known as ”dynamic collections”. This construct allows the creation of a collection of signal functions to which new elements may be dynamically added or removed. The resulting signal function appears as taking an input to a collection of outputs. A routing function defines how the input is distributed to the signal functions in the collection. Using this framework, together with looping combinators, the game was implemented in a manner analogous to object oriented programming, where each game object was represented by a signal function, and objects could pass (continuous and discrete) messages to each other, as well as respond to external input.

5.3 Modular Synthesizer

Yampa has also been used to define a modular synthesizer library, YampaSynth. Here, reactivity constructs were used to activate and deactivate synthesizer elements in response to MIDI events. The modular nature of Yampa corresponded directly to the modular nature of synthesis, easing both the definition of elementary synthesis constructs such as variable oscillators and the combination of these constructs into a full synthesizer. Yampa’s dynamic collections framework (Section 5.2) was used to support polyphonic synthesis [9].

6. Conclusions and Further Work

Functional Reactive Programming has already begun to deliver on the promise of safe, composable and efficient reactive software. However, many challenges and opportunities still exist. Except for such restricted subsets of FRP as EFRP [21], FRP implementations are still not efficient enough or predictable enough in performance to be used effectively in domains which require latency guarantees alongside short sampling intervals (for instance sound or dynamic control systems). There has also yet to be an implementation which combines statically checked safety guarantees such as termination of sample computation [18] with optimized performance.

As noted in Section 4.2, it would be interesting to explore an extended form of the Causal Commutative Arrow framework which traded dynamic optimizations for the current static optimization and permitted the expression of reactivity. One question is if any performance benefit is gained simply by reducing to the single-loop normal form, or whether compiler optimization is required for a performance gain. In either case, another intriguing question is that of the impact of JIT (Just-In-Time) compilation on such a dynamic optimization. This question also applies to the work on optimizations with GADTs (Section 4.1).

The approach taken by the ”lowering” optimization is an efficient and simple one, but a method of applying it in a purely functional setting is not obvious and would be an interesting problem.

Most FRP implementations, including all signal function implementations to date, succumb to continuous re-evaluation of event non-occurrences due to a ”pull-based” implementation where a system continuously resamples the FRP expression for output. The work on *Reactive* (Sections 3.1 and 4.4) purports to solve this problem for Classic FRP, but extending this work to signal functions has not yet been explored, and the simple operation of occurrence time comparison relies on a programmer-checked and arguably difficult to prove identity to retain referential transparency.

The semantics of N-ary FRP have been embedded in the dependently typed language Agda [18], but the surveyed literature did not include an embedding in Haskell or another similarly general functional programming language. In particular, the notion of heterogeneous type-level lists (used in the expression of signal vectors) presents a hurdle to the Haskell type system. It is unknown whether the use of a type-level list such as HList [13] would interfere with the type inference we desire from a Haskell implementation.

```
-- Yampa type for rpswitch
rpswitch :: Functor col =>
            (forall sf . (a -> col sf -> col (b, sf)))
            -> col (SF b c)
            -> SF (a, Event (col (SF b c) -> col (SF b c))) (col c)

-- Refined type for rpswitch with
-- rank-2 quantification of collection
-- type for routing function
rpswitch :: Functor col =>
            (forall sf . forall col'. (Functor col') => (a -> col' sf -> col' (b, sf)))
            -> col (SF b c)
            -> SF (a, Event (col (SF b c) -> col (SF b c))) (col c)
```

Figure 5. Comparison of two possible types for $rpswitch$.

One interesting discovery was that the type of the $rpswitch$ combinator in Yampa [15], described further in the paper referenced in Section 5.2, does not enforce quite what the description insists that it does. The rank-2 type of the routing function ensures that no new signal functions could be introduced. However, since the collection type is still rank 1, the function may instantiate the functor typeclass and thus manipulate the structure of the functor. For instance, the functor could be instantiated as a list, and the routing function could then duplicate or remove elements of the list. We explored the effects of making the type of the collection argument to the routing function rank-2 quantified with a Functor typeclass constraint. (Figure 5.) We verified that this does prevent changing the structure of the collection, but did not explore in detail whether the loss of expressiveness caused by such a change is acceptable. Note also the great complexity of both the initial and refined type signatures.

The concepts of FRP point to an attractive way of expressing reactive systems. A great deal of progress has already been made in optimizing performance. If performance problems continue to be solved, FRP will present a simple, powerful, and generalizable form of expressing such reactive systems.

Acknowledgments

Dr. Matthew Fluet was my advisor for this study. His insight, questions, and explanations were invaluable to me in forming my understanding of FRP.

The members of the *haskell-cafe* [1] and *yampa-users* [2] mailing lists provided a great deal of helpful feedback as I practiced implementing FRP libraries and programs.

The template for this document is the ACM SIGPLAN proceedings template.

A. Library Primitives and Combinators for a Signal Function Library

Figure 6 gives types for a general set of primitives and combinators for a signal function library. The majority of the functions are defined in the Yampa library [15]. The $rpswitch$ function has been retyped to enforce the hiding of the collection structure as well as the signal function structure. The reason for this is described in Section 6.

B. Example Programs

To assist understanding of the semantics and concepts of the two primary expressions of FRP, we provide two sample programs. Each implements an extremely simple version of the Karplus-Strong plucked string sound synthesis algorithm [12]. The two programs are not precisely equivalent. A version in Classic FRP is given in Figure 7 and a version expressed as a signal function is given in Figure 8.

The Karplus-Strong algorithm for plucked string synthesis is defined as an iterative algorithm, but the idea can be extended into FRP semantics. The idea is that a very simple feedback loop, when given a short pulse of noise, will cause the noise to decay very quickly into a uniform tone. This is a surprisingly accurate model for the sound of a plucked string on a musical instrument.

The feedback look is accomplished by passing the initial input into a delay line and then a "low pass filter" which feeds back into the delay line. In the original algorithm, the filtering was accomplished by averaging adjacent samples. In these examples, we assume the existence of a low-pass filter construct.

In both examples, the noise burst is constructed by merging the trigger event with the result of delaying the trigger event by a fixed amount. The non-delayed trigger event carries a behavior or signal function which produces or passes the noise, while the delayed event carries a constant 0 behavior or signal function. This creates a fixed length noise burst.

It is important to note the differences in the implementation of feedback in the two examples. In the Classic FRP example, feedback is accomplished by recursively referring to a delayed version of the resonator behavior. In the signal function example, feedback is accomplished using the "do rec" construct for arrows, which makes use of the "loop" construct of the ArrowLoop instance for signal functions, since a signal function could not recursively evaluate itself.

References

[1] Haskell-cafe mailing list ¡http://www.haskell.org/mailman/listinfo/haskell-cafe¿.

[2] Yampa-users mailing list ¡http://mailman.cs.yale.edu/mailman/listinfo/yampa-users¿.

[3] K. Burchett, G. H. Cooper, and S. Krishnamurthi. Lowering: A static optimization technique for transparent functional reactivity. In *In ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 71–80. ACM Press, 2007.

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

[5] A. Courtney. Frappé: Functional reactive programming in java. In *In Proceedings of Symposium on Practical Aspects of Declarative Languages. ACM*, pages 29–44. Springer-Verlag, 2001.

[6] A. Courtney, H. Nilsson, and J. Peterson. The yampa arcade. In *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, Haskell '03, pages 7–18, New York, NY, USA, 2003. ACM. ISBN 1-

58113-758-3. doi: http://doi.acm.org/10.1145/871895.871897. URL http://doi.acm.org/10.1145/871895.871897.

[7] C. Elliott and P. Hudak. Functional reactive animation. *SIGPLAN Not.*, 32:263–273, August 1997. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/258949.258973. URL http://doi.acm.org/10.1145/258949.258973.

[8] C. M. Elliott. Push-pull functional reactive programming. In *Proceedings of the 2nd ACM SIGPLAN symposium on Haskell*, Haskell '09, pages 25–36, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-508-6. doi: http://doi.acm.org/10.1145/1596638.1596643. URL http://doi.acm.org/10.1145/1596638.1596643.

[9] G. Giorgidze and H. Nilsson. Switched-on yampa: declarative programming of modular synthesizers. In *Proceedings of the 10th international conference on Practical aspects of declarative languages*, PADL'08, pages 282–298, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3-540-77441-6, 978-3-540-77441-9. URL http://portal.acm.org/citation.cfm?id=1785754.1785773.

[10] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson. Arrows, robots, and functional reactive programming. In J. Jeuring and S. Jones, editors, *Advanced Functional Programming*, volume 2638 of *Lecture Notes in Computer Science*, pages 1949–1949. Springer Berlin / Heidelberg, 2003. URL http://dx.doi.org/10.1007/978-3-540-44833-4_6. 10.1007/978-3-540-44833-4_6.

[11] J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37:67–111, 1998.

[12] K. Karplus and A. Strong. Digital synthesis of plucked-string and drum timbres. *Computer Music Journal*, 7(2):43–55, 1983.

[13] O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, Haskell '04, pages 96–107, New York, NY, USA, 2004. ACM. ISBN 1-58113-850-4. doi: http://doi.acm.org/10.1145/1017472.1017488. URL http://doi.acm.org/10.1145/1017472.1017488.

[14] H. Liu, E. Cheng, and P. Hudak. Causal commutative arrows and their optimization. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, pages 35–46, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-332-7. doi: http://doi.acm.org/10.1145/1596550.1596559. URL http://doi.acm.org/10.1145/1596550.1596559.

[15] H. Nilsson. 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, pages 54–65, New York, NY, USA, 2005. ACM. ISBN 1-59593-064-7. doi: http://doi.acm.org/10.1145/1086365.1086374. URL http://doi.acm.org/10.1145/1086365.1086374.

[16] H. Nilsson, A. Courtney, and J. Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, Haskell '02, pages 51–64, New York, NY, USA, 2002. ACM. ISBN 1-58113-605-6. doi: http://doi.acm.org/10.1145/581690.581695. URL http://doi.acm.org/10.1145/581690.581695.

[17] R. Paterson. A new notation for arrows. In *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, ICFP '01, pages 229–240, New York, NY, USA, 2001. ACM. ISBN 1-58113-415-0. doi: http://doi.acm.org/10.1145/507635.507664. URL http://doi.acm.org/10.1145/507635.507664.

[18] N. Sculthorpe and H. Nilsson. Safe functional reactive programming through dependent types. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, ICFP '09, pages 23–34, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-332-7. doi: http://doi.acm.org/10.1145/1596550.1596558. URL http://doi.acm.org/10.1145/1596550.1596558.

[19] Z. Wan and P. Hudak. Functional reactive programming from first principles. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 242–252, New York, NY, USA, 2000. ACM. ISBN 1-58113-199-2. doi: http://doi.acm.org/10.1145/349299.349331. URL http://doi.acm.org/10.1145/349299.349331.

[20] Z. Wan, W. Taha, and P. Hudak. Real-time frp. In *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, ICFP '01, pages 146–156, New York, NY, USA, 2001. ACM. ISBN 1-58113-415-0. doi: http://doi.acm.org/10.1145/507635.507654. URL http://doi.acm.org/10.1145/507635.507654.

[21] Z. Wan, W. Taha, and P. Hudak. Event-driven frp. In *PADL: Practical Aspects of Declarative Languages, LNCS 2257*, pages 155–172. Springer, 2002.

```
-- Signal function type
data SF a b

-- Pure function lifting
arr :: (a -> b) -> SF a b

-- Signal Function composition
(>>>) :: SF a b -> SF b c -> SF a c

-- Parallel pass-through
first :: SF a b -> SF (a, c) (b, c)
second :: SF a b -> SF (c, a) (c, b)

-- Splitting
(&&&) :: SF a b -> SF a c -> SF a (b, c)

-- Parallel combination
(***) :: SF a b -> SF c d -> SF (a, c) (b, d)

-- Integration (by rectangle or trapezoid rule)
integral :: (VectorSpace a) => SF a a

-- Differentiation (ideally, we would have integral >>> derivative = identity
-- and derivative >>> integral = identity)
derivative :: (VectorSpace a) => SF a a

-- Delay: given an initial value and constant delay time, delay the incoming signal
-- by the given time
delay :: a -> Time -> SF a a

-- Single-event reactivity
switch SF a (b, Event c) -> (c -> SF a b) -> SF a b

-- Continuing reactivity
rswitch :: SF a b -> SF (a, Event (SF a b)) b

-- Dynamic reactivity
rpswitch :: Functor col =>
            (forall sf . forall col'. (Functor col') => (a -> col' sf -> col' (b, sf)))
            -> col (SF b c)
            -> SF (a, Event (col (SF b c) -> col (SF b c))) (col c)

-- Convenience routing function:
-- Pairs same input with every output
broadcast :: Functor f => f sf -> a -> f (a, sf)
```

Figure 6. Primitive and combinator types for a signal function library.

```
-- Delay functions
delayE :: Time -> Event a -> Event a
delayB :: Time -> a -> Behavior a -> Behavior a

-- Switch on every occurrence, not just the first (defined recursively using switcher)
stepper :: Behavior a -> Event (Behavior a) -> Behavior a

-- Event merging (Left occurrence if conflict)
mergeL :: Event a -> Event a -> Event a

-- Event source to trigger string plucks
triggerEvt :: Event ()

-- Noise source
noise :: Behavior Double

-- Noise bursts
noiseBursts :: Behavior Double
noiseBursts = lift0 0 `stepper` (mergeL (fmap (const noise) triggerEvt)
                                        (fmap (const \$ lift0 0) (delayE 0.005 triggerEvt)))

-- Low pass filter
lowPass :: Behavior Double -> Behavior Double

-- Resonator
resonator :: Behavior Double -> Behavior Double
resonator = lift2 (\x y -> (x + y)/2) (delayB 0.005 0 (lowPass resonator))

-- Karplus-strong string plucking
karplus :: Behavior Double
karplus = resonator noiseBursts
```

Figure 7. Example of Karplus-Strong string synthesis in Classic FRP

This example uses the arrow syntax now implemented in GHC [17].

```
-- Replaces the value in an event occurrence
tag :: b -> Event a -> Event b

-- Merge the occurrences of two events, taking the left one for simultaneous occurrences
lmerge :: Event a -> Event a -> Event a

-- Randomly generated white noise (range [0, 1])
noise :: SF a Double

-- Low-pass filter
lowPass :: SF Double Double

-- Noise burst on an incoming event
burst :: SF (Event ()) Double
burst = proc e -> do
  n <- noise -< ()
  let onEvt = tag identity e
  let offEvtEvt = tag (after 0.005 (constant 0)) e
  offEvt <- rswitch never -< ((), offEvtEvt)
  let onOffEvt = lmerge offEvt onEvt
  rswitch (constant 0) -< (n, onOffEvt)

-- Resonator
resonator :: SF Double Double
resonator = proc input -> do rec
  let output = (input + filterOutput) / 2
  delayOutput <- delay 0 0.005 -< output
  filterOutput <- lowPass <- delayOutput
  returnA -< output

-- Karplus-strong synthesizer:
karplus :: SF (Event ()) Double
karplus = burst >>> resonator
```

Figure 8. Example of Karplus-Strong string synthesis in signal function FRP