A Livecoding Semantics for Functional Reactive Programming

Tom E. Murphy

vivid-synth.com

tom@vivid-synth.com

Abstract

Livecoding, while extremely powerful for the arts and beyond, has largely had its semantics tied to pervasive direct mutation of state. For the most part, livecoders have not been able to take full advantage of declarative programming, particularly when dealing with side-effects and the outside world. We present a semantics and implementation for functional reactive programming in the context of livecoding, with code hot-swap as a first-class operation. Programmers are freed from juggling low-level details, allowing them to write in a more declarative style and reap the benefits of pure functional programming.

Categories and Subject Descriptors J.5 [*Performing Arts*] ; J.5 [*Music*]; D.3.2 [*Applicative (functional) languages*]; D.3.2 [*Dataflow Languages*]

Keywords live coding, music, functional reactive programming

1. Introduction

Livecoding is a programming paradigm where the running program is changed by the programmer at runtime. Its core operation is *hot-swapping* components of the program - adding, removing, and updating the code at runtime. It is a paradigm that has found a home in many domains, including the arts and in particular music, where in avant-garde music its use is not uncommon.

Typically, however, the code is at least as imperative as it is functional. While there are powerful and expressive declarative languages embedded in purely-functional languages – like TidalCycles in Haskell (Mclean 2014) – they mainly steer clear of extensive interaction with *e.g.* hardware musical instruments. Most modern livecoding languages (SuperCollider, Extempore, Chuck) rely heavily on pervasive global mutable state, both for their hot-swap and for their interaction with the outside world.

There are, however, other ways to skin the cat. Functional reactive programming (FRP) is a way of – among other things – modelling stateful streams of data updates (*e.g.* from hardware musical instruments like MIDI control surfaces) in a way that allows the programmer to write pure functional code that operates over them. The user can write code that interacts with the world in terms of declarative combinators like `map`, `fold`, and `filter`.

FARM'16, September 24, 2016, Nara, Japan
ACM. 978-1-4503-4432-6/16/09...$15.00
http://dx.doi.org/10.1145/2975980.2975986

This paper describes both a semantics for livecoding with FRP, and a specific implementation: the Midair library for Haskell. [1]

Specifically, we present:

- A general semantics for FRP with hot-swap – of sub-components of the program, and at any time during its run – as a first-class operation (Section 3)
- An example implementation, Midair, with a more specific semantics (Section 4), and which is resource-efficient (Section 5)
- An outline of idioms for expressiveness, allowing livecoding to be truly practical

2. FRP Background

A powerful concept from FRP is the ability to abstract over time-varying values. One way this is done is by modelling time-varying values as functors. We think of a functor as a "context" in which values can exist. A property of any functor is that you can apply a function to every value within that context. In Haskell we use `fmap`:

```
fmap :: Functor f => (a -> b) -> f a -> f b
```

In the list context, applying to all values is `map`:

```
chars = ['a','b','c'] :: [Char]
fmap (=='a') chars :: [Bool]
   == [True, False, False]
```

In Haskell, `IO` is also a functor, where the context is the side-effecting computation that returns a value. With `IO`:

```
getChar :: IO Char
fmap (=='a') getChar :: IO Bool
```

Similarly, values that change over time can be thought of as having the time-dependent nature of their values as their context. In pseudocode, we can imagine a stream of key presses coming from the user's keyboard:

```
lastKeyPressed :: FRP Char
fmap (=='a') lastKeyPressed :: FRP Bool
```

The current value of this stream is `True` whenever the most-recent character pressed was `'a'`, and `False` all other times.

The user can also filter and "fold over" streams of data. For example, the user can have a streaming `Bool` value that toggles on/off each time the spacebar is hit – first by filtering out all key presses other than from the spacebar, then by "folding" with `not`.

In this way, like with `IO` in Haskell, we can simplify reasoning about our system by separating the bulk of our program that is referentially transparent from the tangle of state that interacts with the world. When there is a bug, for example, we can separate the

[1] Since the library is still under active development, it is worth noting we are talking here about Midair version 0.3.

task of inspecting our logic from inspecting the stateful machinery that runs it.

This gives us the ability to, in typical pure-functional programming style, write our program as mainly pure functions which have no notion of a global mutable state, and then use these functions in a world that is ever-changing.

3. First-Class Hot-Swap

Traditional FRP (Elliot *et al.* 1997) models changing values over time as `Behaviors`, functions of `Time -> a`. For example, the position of a bouncing ball in a video game can be modeled as a function from a point in time to x- and y-coordinates on the screen.

FRP has typically been formulated without the notion of code that is changed as it runs, however. We present a semantics where changing of code at runtime is a first-class operation.

When we talk about `fold` in the context of FRP, we sometimes say that we are "folding over the past." Given a changing stream of input values, we "fold over" to return a changing stream of values whose current value is the result of folding with a given function over all values that have occurred so far. This allows us to accumulate state in a declarative way.

If, however, we strictly think of a fold as a function over all values over all time, we can quickly land in trouble in a livecoding scenario.

For example, imagine a piece of music whose volume is controlled by an FRP program. For every message from a Musical Instrument Digital Interface (MIDI) controller (say, an up or down button pressed), the current volume of the piece is multiplied either by 1.01 or by 0.99, respectively, gradually changing the volume.

```
data Direction = Up | Down

volChange :: Double -> Direction -> Double
volChange v Up   = v * 1.01
volChange v Down = v * 0.99
```

Using this 1.01/0.99 relationship, our hypothetical programmer plays a piece for a few minutes, accumulating 100 up-button messages and 50 down-button messages, resulting in a volume 163% of the starting volume.

We can think of the stream of events in terms of a list:

```
buttonHistory :: [Direction]

foldl volChange 1 buttonHistory == 1.65...
```

Imagine then that our programmer, partway through playing the piece, wants more responsiveness from a button press, and defines each press of the up/down buttons to correspond to a larger volume change, say a multiplication of 1.03 and 0.97. In traditional FRP, the correct semantics would be for that multiple to have been the "true" multiple all along. We would then recompute from the beginning of time (as a `Behavior` semantics of `Time -> a` would imply), and the volume would immediately more than double, jumping from 163% to 419% of the starting volume.

```
volChange' v Down = v * 0.97
volChange' v Up   = v * 1.03

foldl volChange' 1 buttonHistory == 4.19...
```

In this paper, we believe that the correct semantics for an FRP for livecoding is in fact different: what is typically useful for a livecoder is to describe the past as it did in fact happen, not as it "should have." In this paper, we refer to this changing of the accumulator function without changing the accumulated value as "pivoting."

An implication of this semantics is that we consider it a feature that, given two FRP functions `f` and `g`, an FRP graph that starts folding with `f` and then is hot-swapped to instead fold with `g` can actually output values that neither folding with `f` nor folding with `g` would (or could) have emitted on their own. This is a desirable property, although it has implications for writing code that are discussed later in the paper.

4. The FRP

Here we present an overview of how our FRP semantics is formulated in Midair. Many design decisions (choices of performance and expressiveness characteristics) have been made to fit the problem domain. This includes some limits in expressiveness, for the sake of fast and predictable performance.

It is important to note that the livecoding and hot-swap semantics presented here (and in particular hot-swap for FRP) are not at all limited to how they are implemented in Midair (and in particular its discrete-time, signal-based semantics).

4.1 Signals and Signal Flow

For Midair we use a discrete-time semantics, meaning that events only "fire" at specified times. This is in contrast to continuous-time semantics – as seen in Elliot *et al.* (1997) – where time is infinitely divisible.

For efficiency, Midair is also built atop Arrows (Hughes 2000), which is a common tool in the toolchest used to make efficient FRP (Hudak *et al.* 2003, Liu *et al.* 2007, Wan *et al.* 2002).

These design decisions, while handy for livecoding for other reasons, are not required for the hot-swap semantics described in this paper.

4.1.1 User-Facing Semantics

In its shortest form,

```
SFlow a b
```

represents signal flow from type `a` to type `b`. `SFlow` is analogous (and if you squint, can be read like) the `->` in the function type `a -> b`, although it operates over ever-changing streams of values.

Midair provides a set of core functions to construct `SFlows`, including:

```
sMap :: (a -> b) -> SFlow a b
sFold :: c -> (a -> c -> c) -> SFlow a c
sFilter :: (b -> Bool) -> SFlow b b
sCompose :: SFlow b c -> SFlow a b -> SFlow a c
```

Along with its core functions, Midair provides many helper functions such as `sAll`, which provides a boolean fold, and `sMapMaybe`, which combines mapping and filtration.

`SFlow` is an instance of the Arrow typeclass (Hughes 2000). The `(<<<)` operator from Haskell's `Control.Arrow` module can be used like `(.)` for function composition. For example, we can use `(<<<)` to build the example we described earlier: a `Bool` which starts `True` and toggles on/off every time the user hits the spacebar:

```
sFold True (\_ -> not) <<< sFilter (==' ')
   :: SFlow Char Bool
```

The `Applicative` nature of `SFlows` lets us easily combine them with each other and with pure code:

```
one :: SFlow x Double
two :: SFlow x Double
(*) <$> one <*> two
   :: SFlow x Double
```

In addition, the new `ApplicativeDo` (Marlow *et al.* 2016) semantics in Haskell allow graphs to be expressed in the form:

```
do o <- one
   t <- two
   pure (o * t)
```

4.1.2 Internal Representation

A `SFlow a b` program is represented as a tree, with some `SFlow` components containing others, to allow an input signal to "flow" through the tree.

The components of the tree are represented with generalized algebraic data types (GADTs). A simplified version of the representation is presented here:

```
data SFlow a c where
   SF_Map :: (a -> c) -> SFlow a c
   SF_Compose
      :: forall a b c.
      SFlow b c -> SFlow a b -> SFlow a c
   SF_FoldP :: c -> (a -> c -> c) -> SFlow a c
   SF_Filter
      :: Maybe a -> (a -> Bool) -> SFlow a a
   SF_NodeRef
      :: TVar (Maybe a) -> TVar (Maybe c)
      -> TVar (SFlow a c) -> SFlow a c
```

Most of these look familiar from the `sMap`, `sFold`, etc. combinators above. The one that will look most unfamiliar is `SF_NodeRef`, which we will see more of in the next section.

The user's FRP code runs only when there is a new input at the top-level of the tree – for example when a MIDI message is received. Each time we receive a new input, the graph "fires." Input is fed to the top-level `SFlow` component, which in turn feeds the inputs to its sub-components, to arrive at its output which is then returned.

To get an output `b` from an input `a` and a `SFlow a b`, we can essentially just walk the tree of `SFlow` components recursively.

There is in fact some trickiness that makes the recursive descent a bit more nuanced than we describe here (mainly updating the graph), but this is an accurate first approximation.

In this paper, we will leave the implementation no more specific than describing a fundamental function:

```
fireGraph :: TVar (SFlow i o) -> i -> STM (Maybe o)
```

The `Maybe` in the return value of `fireGraph` is present because a signal may be filtered, returning no value.

4.2 Hot-Swap

The liveness in livecoding comes from the user writing and modifying code at runtime. So far we have only seen static FRP graphs – how do we allow the user to update their code?

To do this, we introduce a few new functions:

```
mkNodeRef :: SFlow a b -> IO (SFNodeRef a b)
nRef :: SFNodeRef a b -> SFlow a b
hotswap :: SFNodeRef a b
        -> (Maybe a -> Maybe b -> SFlow a b)
        -> IO ()
```

`SFNodeRef`s can be used anywhere in a Midair program, including nested within each other.

Crucially, although a quick glance at the types of `mkNodeRef` and `nRef` might imply that `fmap nRef (mkNodeRef x) == pure x`, this is not the case. The result of `nRef`, while having the same type `SFlow a b` as the argument to `mkNodeRef`, contains a transactional variable that in turn contains a `SFlow a b` tree - in fact the same `SFlow a b` tree that was originally passed to `mkNodeRef`. This way, we can swap out one `SFlow a b` with an entirely different one anywhere that the `SFNodeRef a b` was used. This is a key piece of the semantics of Midair.

In practice, imagine a program which takes a MIDI note number input from a MIDI instrument, constructs a chord from the note, transposes the chord, then plays it. Using pseudocode for the musical actions:

```
ref <- mkNodeRef $ sMap (transpose 2)
   :: IO (SFNodeRef Chord Chord)
```

We can then use this reference in a signal flow graph:

```
sMap playChord <<< nRef ref <<< sMap makeChord
   :: SFlow Note (IO ())
```

Later, we can change how many semitones our graph transposes the chord by:

```
hotswap ref $ \_ _ -> sMap (transpose 4)
```

The ignored arguments to the function above are the most-recent input and output of the piece being hot-swapped. If instead of mapping from the input value we want to start folding over inputs, we can start *e.g.* counting "where we left off".

Going back to our earlier example with the volume changes (from Section 3):

```
ref <- mkNodeRef $ sFold 1 volChange
hotswap ref $ \_ lastOut ->
   sFold (fromMaybe 1 lastOut) volChange'
```

The `fromMaybe` above is used in case the earlier `sFold` has never fired and thus does not have a last output. In practice this is often unnecessary – the programmer will know at the time they call `hotswap` whether the piece they're hot-swapping has already fired. In these cases they can just pattern-match on the output `Maybe`:

```
hotswap ref $ \_ (Just lastOut) ->
   sFold lastOut volChange'
```

4.2.1 Restructuring

`SFNodeRef`s, as we have seen, can be used anywhere in a Midair program. However, they are embedded in a larger `SFlow` structure – what if the user builds their FRP graph in a tree shape that they then want to alter? The cleanest solution is to use the fact that `SFNodeRef`s can be nested to create references of finer and coarser granularity. The user can then hot-swap out larger and smaller parts of the graph as needed, including reusing the smaller parts when restructuring the larger – the existing state will persist.

Sometimes however, a programmer may feel their graph needs to be completely rewritten. For these cases, there is always a (potentially drastic) escape measure: when Midair runs an FRP graph, it returns a `SFNodeRef` for it – so the whole graph can be swapped out if need be.

4.2.2 Idioms

Midair provides several `hotswap` convenience functions. Perhaps the most basic is `hotswapFold`, whose use we can demonstrate with the volume change example from a previous section:

```
hotswapFold ref 1 volChange'
```

`hotswapFold` is a key function in Midair: it provides the canonical "pivot" that comes with first-class hot-swap.

Many similar functions exist, from the simpler `hotswapMap` and `hotswapFilter`, to more-specific ones like `hotswapMax` and `hotswapMapInsert`.

Another workhorse hotswap function is hotswapHold. This continually emits its last-outputted value, each time it receives input.

We can demonstrate its use using sCount:

```
sCount :: Num n => SFlow x n
   == sSum <<< sConst 1
   == sFold (+) 0 <<< sMap (const 1)
```

We can, given keyboard input, count every key press of a digit character:

```
ref <- mkNodeRef sCount

nRef ref <<< sFilter isDigit
   :: SFlow Char Int
```

Then stop counting for a while:

```
hotswapHold ref
```

Then, after more time, start counting where we left off:

```
hotswapCount ref
```

Even after switching back to counting fires with "count," none of the key presses that occurred while the "hold" was in effect are part of the sum emitted from the graph.

4.2.3 Internal State

We use software transactional memory (STM) (Harris *et al*. 2005) to ensure that hot-swaps occur deterministically: a function can only be swapped out between firings of the graph, as updates mid-fire could lead to unpredictable behavior. These STM actions are wrapped as IO actions for convenience but are accessible to the programmer also.

SF_NodeRef contains three values, each a STM transactional variable (TVar). The first two are the most recent input and output respectively; the third is the inner SFlow which can be hot-swapped.

```
SF_NodeRef :: TVar (Maybe a) -> TVar (Maybe c)
           -> TVar (SFlow a c) -> SFlow a c
```

SFNodeRef itself is a small wrapper for type safety. When we use nRef we still are able to swap it out because the definition of SFNodeRef exactly mirrors its SFlow cousin:

```
data SFNodeRef a c
   = SFNodeRef_Internal
      (TVar (Maybe a))
      (TVar (Maybe c))
      (TVar (SFlow a c))
```

An SFNodeRef can only be created with mkNodeRef, and its only uses are with hotswap and nRef.

4.2.4 Caveat

As discussed in a previous section, an implication of our semantics is that an FRP node whose inner function is swapped from folding with function f to folding with function g can output values that neither the f fold nor the g fold could have output on their own. This is desirable in general, but requires caution when there are invariants on the codomain of the function that are not expressed in the type (when its image is smaller than its codomain).

4.3 Input/Output

So far we have talked about the programs a user can write in Midair, but not the context in which they are run. We will not discuss this context much in this paper, but it is worth outlining. The typical structure of a Midair program is one or more top-level graphs with a type like:

```
SFlow MIDI (IO ())
```

To a first approximation, we simply call fireGraph (from Section 4.1.2) when we have new MIDI data and, if an IO () action is returned, we run it.

This is a bit of a simplification. In fact, we define a wrapper around IO, called Fx. This adds, among other things, a Monoid instance so effects can be combined in a larger FRP graph.

The input to the graph is all of the time-varying data that the program depends on. Midair (or a custom runner function) will handle the stateful computations required to provide the inputs to the FRP graph, and also run the effects that are returned.

5. Space and Time

A very pleasant side-effect of the choice to pivot values instead of recomputing from the beginning of time is that pivoting is extremely cheap: we only need to know the very last values the components of our signal graph have input and output.

By contrast, a "traditional" FRP semantics would require us to hold on to every previous state it had seen if it anticipated a need to recompute. If the signals that the FRP graph is processing are frequent, *e.g.* mouse position data or sensor data, holding on to the entire history of the program could become prohibitively expensive.

Livecoding is a domain in which an efficient implementation is crucial, for various reasons:

- Onstage during a performance, if execution speed is not both fast and predictable, there is a real risk of a long calculation (say, from a space leak) to cause the program to miss a musical cue.
- Code is usually sent to an interactive interpreter (in the context of Haskell, the usual livecoding environment is GHCi). For speed of compilation, these interpreters do many fewer optimizations by default than when compiling for maximal performance. When livecoding, compile time *is* runtime, and slow compilation can effectively be thought of as slow runtime performance.
- The transient nature of livecoding makes it typically more fragile than much other code. Unlike many programs – which save enough state to recover from a restart (*e.g.* a web server) – in livecoding, references to running processes and data are held in the interactive interpreter. If the interpreter were to crash, there would be no way to gracefully recover.

Midair sidesteps many of these issues simply by avoiding any features for which performance is an active research area. This, of course, is not a 100% win - the user could very well want to use features we have chosen not to support. In this paper we assume that many programmers in a livecoding context prefer a lightweight semantics with predictable performance.

5.1 Keeping History

While Midair itself does not hold on to older history, if the user wants to store history they are always able to. The simplest version of this is just to collect every input we have ever seen:

```
sFold [] (:) :: SFlow x [x]
```

This, of course, grows indefinitely and consumes more memory with every input.

More likely (and more efficiently), the user might want to hold on to some bounded amount of history. The user may define their own functions to do so, and in addition Midair provides some sugar. For example, Midair defines:

```
sTake :: Int -> SFlow x [x]
```

Given a signal flow function named `sf`, to continuously output the largest of the 100 most-recent values of `sf`'s output, the user would write:

```
sMap maximum <<< sTake 100 <<< sf
```

6. Related Work

6.1 Signal-Based FRP

Many previous FRP systems have limited their semantics to discrete-time signals for more-predictable performance and resource usage.

Real-Time FRP (RT-FRP) (Wan *et al*. 2001) and, later, Event-driven FRP (E-FRP) (Wan *et al*. 2002) use similar discrete-time signal-only semantics for efficiency.

Arrowized FRP (Hudak *et al*. 2003) has similarly seen fairly wide use for efficiency, including in Midair.

Elm (Czaplicki 2012), discussed below, has also used signal-only FRP for efficiency.

6.2 Hot-Swap

TidalCycles (Mclean 2014), a powerful declarative livecoding music language also in Haskell, borrows ideas from FRP in the way that it represents musical patterns. In particular, it uses a concept similar to the "classic" FRP notion of `Behaviors` as functions from time to values. TidalCycles' hot-swaps are for these FRP-inspired pure functions. There is not Midair-style support for hot-swapping FRP for interacting with the outside world.

Elm, a robust ML-style language and framework for web development, provides hot-swapping, although at a coarser granularity. It does not allow the user to hot-swap arbitrary components of the FRP graph. Instead, an Elm program is structured like a large `sFold` in Midair, which holds its global state.

An example that Elm provides is a tiny piece of a video game, where the user can move the Mario character with key presses. The state that the Elm program holds on to contains Mario's current position, direction of motion, velocity, etc.

The input the program takes are the current key presses: whether the left/right and up/down arrows are pressed.

The program also provides `step`: a function that takes the user keyboard input, Mario's current state (velocity, etc.), and applies physics and game rules to it, returning output and a new state.

In Midair this would look something like:

```
data Direction = Pos | Neg | Zero

data MarioState = MarioState {
     x  :: Double, y  :: Double
   , vx :: Double, vy :: Double
   , marioDir :: Direction }

data KeyPress = KeyPress {
     leftRight :: Direction, upDown :: Direction }

startState :: MarioState
step :: KeyPress -> MarioState -> MarioState
```

The Elm-style top-level fold can be created with

```
ref <- mkNodeRef $ sFold startState step
    :: SFNodeRef KeyPress MarioState
```

We can then – while the game is being played – redefine `step` (for example, to increase the gravity in Mario's world):

```
hotswapFold ref startState step'
```

Elm also provides a time-travelling debugger, allowing the FRP graph state to be saved, rewound, and replayed. This is a very powerful tool and does not currently exist for Midair.

7. Further Work

There are several avenues of further work that appear promising:

- The proof of an ergonomic system is in its use, and some of the most fruitful further work will likely result from gathering experience reports from actual livecoders on their use of FRP within livecoding.
- Discrete-time, event-based signals are very important for the performance of Midair, but it is tempting to add support for some sort of continuous signals, even if not how they are implemented in "traditional" FRP. As an example, the programmer might want to access the current clock time without having a clock signal which pushes its values at a specified time interval.
- This paper has made multiple references to the fact that hot-swap in FRP is not tied to how it is implemented in Midair. It would be very interesting to see what first-class hot-swap would look like in FRP systems which made significantly different design decisions.
- Access to randomness has proven a stubborn challenge to many FRP systems. Midair does not currently attempt to solve this problem but it is arguably the largest current impediment to musical expressivity.

8. Conclusions

We have demonstrated a practical FRP for livecoding, with hot-swap of part or all of the FRP graph as a first-class operation. We allow the programmer to "pivot" their computations, creating a semantics tailored to a livecoder's needs. In addition, the implementation we present in Midair is efficient and terse: both important requirements for the problem domain. Livecoders are able to write in a clean, comprehensible declarative style without giving up the ability to work in a world of ever-changing streams of data.

Acknowledgments

I would like to thank Mark Santolucito and Evan Czaplicki for interesting discussions that inspired this line of research. I would also like to thank this paper's reviewers for their thoughts and suggestions.

References

Czaplicki, E. Elm: Concurrent FRP for Functional GUIs. *Thesis*, Harvard University (2012).

Elliot, C., and P. Hudak. Functional Reactive Animation. *International Conference on Functional Programming* (1997) 163-173.

Harris, T., S. Marlow, S. P. Jones, and M. Herlihy. Composable memory transactions. *ACM Conference on Principles and Practice of Parallel Programming* (2005).

Hudak, P., A. Courtney, H. Nilsson, and J. Peterson. Arrows, Robots, and Functional Programming. *Summer School on Advanced Functional Programming*, Oxford University LCNS 2638 (2003) 159-187.

Liu, P., and P. Hudak. Plugging a space leak with an arrow. *Electronic Notes in Theoretical Computer Science* (2007) 193(1):29-45.

Hughes, J. Generalising Monads to Arrows. *Science of Computer Programming 37* (2000) 67-111.

Marlow, S., S. P. Jones, E. Kmett, and A. Mokhov. Desugaring Haskell's do-notation Into Applicative Operations. *Haskell Symposium, International Conference on Functional Programming* (2016).

Mclean, A. Making programming languages to dance to: Live Coding with Tidal. *Conference: proceedings of the 2nd ACM SIGPLAN International Workshop on Functional Art, Music, Modelling and Design* (2014).

Wan, Z., W. Taha, and P. Hudak. Real-Time FRP. *International Conference on Functional Programming* (2001).

Wan, Z., W. Taha, and P. Hudak. Event-Driven FRP. *Practical Aspects of Declarative Languages)* (2002).