Functional Reactive Programming and the Web Audio API

Mike Solomon
Meeshkan
mike@meeshkan.com

ABSTRACT

Functional Reactive Programming (FRP) is a way to model temporal phenomena using *events*, which carry information corresponding to a precise moment in time, and *behaviors*, which represent time-varying values. This paper shows how FRP can be used to build reactive audio applications that blend the WebAudio API with other browser-based APIs, such as mouse events and MIDI events. It will start by presenting a brief history of FRP as well as definitions of the `Event` and `Behavior` types. It will then discuss the principal challenges of applying the behavior pattern to WebAudio and how these challenges can be solved by using induction on existentially-quantified and linearly-typed Indexed Cofree Comonads. An implementation of this approach is provided via the library `purescript-wags`.

1. INTRODUCTION

Functional Reactive Programming (FRP) was first introduced as Functional Reactive Animation[2] (Fran) as a way to animate physical phenomena using Hugs, a now-defunct variant of Haskell. Since then, a number of Haskell libraries, such as *reactive-banana* and *Elerea*, have provided robust implementations of FRP that are used in a number of time-based domains, including animation, user interface implementation, and signal processing. Newer libraries such as *Yampa* implement a point-free approach using the *arrow* pattern, which provides a group of combinators that "carry" time through a computation. *Yampa* has seen considerable traction in the audio community, serving as the basis for at least one modular synthesizer.[3] As the need emerged to mix more heterogeneous signals and to use programming languages with varying degrees of functional expressivity, Microsoft invested substantial resources in ReactiveX, which uses a pub-sub model based on observables (emitters) and subscribers.[4] Some projects, like Elm, made FRP primitives first-class citizens via the `Signal` type, which functions as an application-wide event bus[1].

This paper will explore how FRP can be used to pilot the WebAudio API. It will use PureScript, a web-friendly dialect of Haskell, and two FRP libraries whose syntaxes are close to the original *Fran* implementation. It will start by presenting the two basic FRP types — `Event` and `Behavior` — and will the show how these types can be translated into calls to the WebAudio API.

1.1 The Event type

The `Event` type is parameterized over a single type variable `a` that represents the type of an event, ie `MouseClick` or `KeyboardPress`. The signature for an event is as follows:

```
newtype Event a =
  Event ((a -> Effect Unit) -> Effect (Effect Unit))
```

Here, the `Event` constructor takes a single argument: a function that accepts a callback and returns an unsubscribe effect. The callback of form `(a -> Effect Unit)` accepts the *event* of type `a` and performs an arbitrary side effect in the `Effect` monad. This callback is called with the event payload, such as a key value or mouse coordinates. The return value of `Effect (Effect Unit)` is an unsubscribe effect. The double-effect acts as a closure so that the unsubscribe operation is not performed immediately but rather is passed to the consumer to be called at a later time, ie in the following manner:

```
-- subscribe ::
--   forall a r.
--   Event a ->
--   (a -> Effect r) ->
--   Effect (Effect Unit)
--
-- makeEvent ::
--   forall a.
--   ((a -> Effect Unit) -> Effect (Effect Unit)) ->
--   Event a

main :: Effect Unit
main = do
  unsubscribe <- subscribe (makeEvent \k -> do
    k "hello"
    pure $ pure unit) log
  unsubscribe
```

When the unsubscribe effect is invoked, the callback no longer receives events emitted by a source.

1.2 The Behavior type

A `Behavior` is a continuous function of time that is *sampled* by an event.

Web Audio Conference WAC-2021, July 5–7, 2021, Barcelona, Spain.

```
newtype ABehavior event a =
  ABehavior (forall b. event (a -> b) -> event b)

type Behavior a = ABehavior Event a
```

A `Behavior` must always be able to produce a value of type `a` that "unlocks" the value `b`. For example, one common behavior is `currentTime`, with a signature `currentTime :: Behavior Instant`. In the example below, the current time is sampled by an event that curries a mouse click and produces `Tuple Point Instant`.

```
timeAndClick :: Event (Tuple Point Instant)
timeAndClick =
  sample currentTime (Tuple <$> mouseClick)
```

2. AUDIO GRAPHS AS BEHAVIORS

An audio graph in the Web Audio API can be thought of as a continuous function of time. This means that, in Functional Reactive Programming, a good candidate for its type would be `Behavior Graph`. At a high-level, the set-up would be:

1. A behavior describing the current web audio graph (ie an OscillatorNode going to a GainNode going to a Context) is sampled by some event, ie a mouse click or the passage of time in a polling loop.
2. A subscriber listens to changes in the graph, rendering them to the imperative web-audio API.
3. Unsubscribing from the event cleans up resources like the context and any active `MediaRecorder` or `BrowserMediaStream` objects.

3. AUDIO GRAPHS AS STREAMS

If the `Behavior` type provides a way to represent Web Audio graphs that are sampled over time, there is still the practical question of how to produce these graphs.

One approach would be to have a function `Environment -> Graph` that, for any given environment (ie the current time, if a mouse is clicked, etc), emits a graph. The issue with this approach is that it is stateless: there is no way to know what the graph was at time $n-1$, which makes working with named resources (ie starting and stopping precise oscillators) impossible. Another issue is that it requires the rebuilding of a graph every time the function is called, which is computationally expensive.

A second approach would be to pass an accumulator to the function, ie `Accumulator -> Environment -> Graph`. This allows the function to have access to the past, but it still requires rebuilding an entire graph. It is also inefficient in that it requires traversing an accumulator twice - once to build it and once to read it at the next timestamp.

A *cofree comonad*, or *stream*, is one way to construct a `Behavior` that solves these two issues. The signature for cofree comonad is `Cofree f a`, where `f` is an arbitrary `Functor` and `a` is a type annotating the functor as it branches. This allows one to retrieve the annotation (current value, or head) of the stream as well as the functor (tail) of the stream *ad infinitum*, which is the classic comonadic `extract/extend` pattern.

As an example, consider the following cofree comonad that acts as an easing algoritm that increases if `adj` is negative and decreases to 20 if `adj` is positive:

```
easingAlgorithm :: Cofree ((->) Int) Int
easingAlgorithm =
  let
    fOf initialTime =
      mkCofree
        initialTime
        \adj -> fOf $ max 20 (initialTime - adj)
  in
    fOf 20
```

The previous `initialTime` is passed to the future `(->) Int`, allowing the function to know if an adjustment should be made (`max 20 (initialTime - adj)`) without any need for an accumulator that passes along `initialTime`.

This construction is representative of a general pattern that applies to all cofree comonads - to produce an infinite stream of values, any constructor of a cofree comonad must make a recursive call to itself at some point. Otherwise, the program would be infinitely long. In calling itself, it can transmit information from a current state to a future state. This solves the problem of retaining state: the state is retained in the construction of the cofree comonad itself.

Furthermore, the state does not need to be packaged and "passed" to the future - it can be cached directly in the future, allowing small parts of a state to be split off and passed into alternate versions of the future. For example, if `Cofree f a` needs a `String` *or* a `Number` at time $t+1$, there's no need to create an `Either String Number` — you can return a different `mkCofree` based on a branching condition that uses either the `String` or the `Number`, obviating the need to create an `Either String Number`.

```
easingAlgorithm :: Cofree ((->) Int) Int
easingAlgorithm =
  let
    hello s = 21
    world q = 22
    fOf initialTime =
      mkCofree
        initialTime
        \adj -> fOf $ max
          -- we can work directly on "hello" or 42.0
          -- via caching it in the returned function
          -- so we don't need to send an
          -- Either String Number
          -- to (->) Int
          (if initialTime > 20
            then hello "hello" else world 42.0)
          (initialTime - adj)
  in
    fOf 20
```

Modeling Web-Audio graphs this way helps reduce needless memory operations by transmitting graph information directly to the point of construction of the graph at a future timestamp.

Returning to behaviors, a cofree comonad can be converted to a behavior by providing a function to ‘‘explore", or actualize, the underlying functor.

```
streamToBehavior ::
  forall f.
  (f ~> Identity) ->
  Cofree f ~>
  Behavior
streamToBehavior explore cf = behavior \aToB ->
  makeEvent \k -> do
    r <- Ref.new cf
    subscribe aToB \e -> do
      i <- Ref.read r
      k (e $ head i)
      Ref.write (unwrap <<< explore $ tail i) r
```

In this way, streams of web audio graphs can act as behaviors that are sampled by arbitrary events, streaming graphs frame-by-frame in the same manner as audio itself is streamed.

4. EXISTENTIALLY-QUANTIFIED STREAMS

For frequently-changing audio graphs, it is inefficient to have a single `Graph` type because it is needlessly broad. For example, if at a given moment in time a single sine-wave oscillator is playing but the full audio graph could be one of thousands of outcomes, some form of pattern-matching will be needed to "rule out" the other outcomes and work with the single oscillator. Obtaining the current audio graph by many calls to `if/then` statements slows down computations and causes missed rendering deadlines.

One way to solve this is for information to be encoded with a precise type that is erased when the stream is consumed. In functional programming, this pattern is called *existential types*, or *rank-n types*. It tells the compiler "on the inside of this computation, a type will exist that is not available to the outside scope."

When using this pattern, cofree comands no longer suffice because they do not contain an existential type. Instead, a tweaked version is needed:

```
makeStream ::
  forall a b.
  Frame a b ->
  (Frame a b -> Stream a) ->
  Stream a
```

Here, the type `b` is the existentially-quantified type that is erased in `Stream a`. That means that it can change on every invocation of `makeStream`. For example, imagine that we have `Frame String b` as our `Frame a b` and `Stream String` as our `Stream a`. We can write:

```
freeze :: forall a b. Frame a b -> String a
freeze s = makeStream s freeze

myStream :: Stream String
myStream =
  makeStream
    (Frame "hello" 1)
    \(Frame s i) ->
      makeStream (Frame s (toNumber i)) freeze
```

Here, the `b` in `Frame String b` is initially occupied by an `Int` (1) and is then occupied by a `Number` (1.0). However, the output stream of type `Stream String` knows nothing about the precise type of `b` - all it knows is that it exists.

In web-audio speak, this can be translated roughly as `Frame Sound Graph` and `Stream Sound`. The ear does not care about the intermediary graphs that are produced. They are erased over the course of the computation, leaving only the sound. Using this strategy, a computation can work off a precise graph type (ie a type representing one oscillator and one gain node) and then change the type downstream instead of pattern-matching against a mega-type, saving precious computation time.

Like cofree comands, at some point this strategy will need a recursive call to sustain itself into the future. `freeze` is a particularly brutal choice because it closes off the possibility for the graph to update. In real-world scenarios, loops usually span multiple functions with multiple branching-conditions, allowing for complex state machines. An example of a recursive stream can be found at https://github.com/mikesol/purescript-wags/blob/main/examples/kitchen-sink/KitchenSink/Piece.purs, which loops over time by feeding its own signature `LoopSig` to itself.

5. LINEAR TYPES FOR GRAPH OPERATIONS

Using the strategy above, we can achieve substantial gains in speed by working off of a specific type that does not need heavy pattern-matching. Another way to think of this is that the pattern-matching happens at compile time, plugging in the correct function as the compiler traverses the tree, instead of at runtime.

While this works on a conceptual level, it runs into challenges when making it work on top of an imperative API like Web Audio. This is because Web Audio itself is stateful, which means that a `Frame` at time t may have the same type as a frame at time $t+1000$, but the web audio graph itself may be different because the actual nodes have changed and occupy different locations in memory.

One way to solve this problem is through the use of *linear types*. If a resource has a linear type, it must be consumed exactly once. Thinking in terms of frames and streams, we want each frame in `makeStream` to be consumed only once — namely, in the continuation function that produces the new head of the stream. We *don't* want this frame to be consumed again in 10 minutes, even if the type would otherwise allow for it. To fix this, we can tweak our function to accept a proof term:

```
makeStream ::
  forall p0 a b.
  Frame p0 a b ->
  (forall p1. Frame p1 a b -> Stream a) ->
  Stream a
```

Now, there is no way the continuation function can be invoked with the prior frame because the types do not align: `p0` is a different type than `p1` and can *never* be the same because we can never know anything about `p1` other than the fact that it exists (we have another existential type).

Working with linear types means that one of the bread-and-butter functional classes — `Applicative` — is no longer possible. Applicative allows you to pull a frame "out of thin air" using a function called `pure`. However, and unlike many judicial systems in the world, we do not want to fabricate proof out of thin air. Rather, proof is encoded in the type itself, and while it can be passed onwards via a `map` or `apply` or `bind` operation, it cannot be fabricated via `pure`. This leads to the rare but interesting outcome where `Frame` can implement `Functor`, `Apply` and `Bind` but cannot implement `Applicative`. In other words, it cannot be a `Monad` even though it is a can use `do` notation via the implementation of `Bind`. The extrication of `Bind` from `Monad` is a fairly recent development in functional programming and has led to a plethora of monad-like constructs that omit some monadic laws. Bringing it back to audio, we delegate the creation of proof-of-time (Applicative) exclusively to the engine, but we leave it to artists to map, apply and bind temporal objects in order to create audio graphs.

6. TYPE-LEVEL LENSES FOR GRAPH ACCESS

Now that we have a stream comprised of type-safe graphs, we need a way to focus on part of those graphs to do operations, like changing a frequency or turning off buffered playback. In functional programming, the conventional way to zoom into a graph is using a Lens. Here, however, lenses are onerous for two reasons:

1. Because the type of the audio graph changes, the type of the lens must change as well.

2. ADTs can't have feedback loops and lenses can't have feedback loops, whereas audio graphs can.

To get around this problem, we can use type-classes to extract information from a type that acts as a lens and then use a different mechanism, ie an adjacency matrix, to effectuate graph operations that may be recursive. Adjacency matrices also have the advantage of *O(log(n))* lookup-time, which is significantly faster than traversing a deeply-nested graph.

For example, consider the type:

```
Gain
  (Function
    (Proxy MyGain)
    (PlayBuf
      /\ Delay (Gain (Proxy MyGain))
      /\ Unit))
```

A type-class can pick apart this type at compile-time to find that the top-level `Gain` feeds back onto itself lower in the graph. It can then use this information to determine the correct pointer in an adjacency matrix when it has to look up `MyGain`. Note that this type-traversal happens at compile-time, which means that at runtime, it has already produced a pointer that can be fed directly to an adjacency matrix. This means that, in a complex polyphonic structure, we can operate on a single audio unit without a runtime graph traversal, allowing for very fast responses to events like MIDI onsets.

7. INDEXED BINDS

In the previous examples, we were working off of a type `Frame proof Sound Graph` where the graph type was an existential type. By doing this, we've lost one of the core advantages of functors: the ability to hold onto *any* type. This means that all of our computations need to be encoded in the graph. Let's say, for example, we want to hold onto an arbitrary floating-point value. We'd have to stash it somewhere in an audio graph and then ignore it at rendering time.

A better solution is to move the graph itself to the type-level and allow the functor to carry an arbitrary type. This can be achieved through an *indexed* signature, ie `Frame proof Sound InGraph OutGraph a`. In this setup, the graph is moved to the type-level and its representation is cached within the frame type. Managing the graph manipulations as a monadic side effect checked by indexed binds has the added advantage of allowing for validation at every `bind` to disallow illegal operations on the graph, like disconnecting audio that was never connected in the first place.

8. ENCODING NON-STRUCTURAL GRAPH CHANGES USING INDUCTIVE TYPES

While indexed binds can enforce graph correctness for changes like connecting and disconnecting audio units, we still haven't found a way to account for non-structural graph changes. For example, when we set the frequency of an oscillator, the graph's structure stays the same even though the sound changes. In these cases, we may have many changes within a single timestep, and we need a way to indicate these changes in the type system so that the same change is not accidentally applied multiple times. Simply marking the graph as "changed" does not suffice, as we may change it multiple times. One solution is to encode this inductively using a change-bit.

Inductive types are types that are incremented by a successor type. The classic example is the Peano representation of integers:

```
foreign import data Z :: Type
foreign import data Succ :: Type -> Type

type Zero = Z
type One = Succ Zero
type Two = Succ One
type Three = Succ Two
-- etc
```

Including an inductive type in an indexed bind disallows map and apply operations on types at different stages of induction. This is a compile-time check, so there is no runtime penalty.

9. PURESCRIPT-WAGS

We've now shown how to build FRP-compliant behaviors out of type-safe audio streams with type-level lenses for efficient lookup operations and indexed binds that enforce graph correctness, allow individual frames of a stream to be a functor, and encode the correct order of non-structural graph changes.

`purescript-wags` is a PureScript library that implements this strategy. It is available at https://github.com/mikesol/purescript-wags, and has links to several code and audio examples.

10. ACKNOWLEDGMENTS

I'd like to thank Phil Freeman for creating both PureScript and `purescript-behaviors`, without which this paper would not have been possible. I'd also like to thank the PureScript community for their helpful answers to my questions as I learned the language and built `purescript-wags`.

11. REFERENCES

[1] E. Czaplicki and S. Chong. Asynchronous functional reactive programming for guis. *ACM SIGPLAN Notices*, 48(6):411–422, 2013.

[2] C. Elliott and P. Hudak. Functional reactive animation. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming*, pages 263–273, 1997.

[3] G. Giorgidze and H. Nilsson. Switched-on yampa. In *International Symposium on Practical Aspects of Declarative Languages*, pages 282–298. Springer, 2008.

[4] A. Maglie. Reactivex and rxjava. In *Reactive Java Programming*, pages 1–9. Springer, 2016.