Functional Reactive Programming and the Web Audio API

Mike Solomon
klank.dev
mike@meeshkan.com

ABSTRACT

Functional Reactive Programming (FRP) is a way to model discrete and continuous signals using `events`, which carry information corresponding to a precise moment in time, and `behaviors`, which represent time-varying values. This paper shows how the behavior pattern can be used to build sample-accurate interactive audio that blends 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 polymorphic behavior and event types. It will then discuss the principal challenges of applying the behavior pattern to WebAudio, including named audio units, front-loaded evaluation, and scheduling precise temporal events using different clocks.

1. INTRODUCTION

Functional Reactive Programming (FRP) was first introduced as Functional Reactive Animation[2] (Fran) as a way to model 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 create a monadic context around values so that time is "carried" through a computation. *Yampa* has seen considerable traction in the audio community, including actings as the basis for a modular synthesizer.[3] As the need emerged to mix more heterogeneous signals and to use programming languages with varying degrees of functional expressivity, multi-language projects such as ReactiveX emerged that used the combinator-based approach over a pub-sub model via observables (emitters) and subscribers.[6] Some projects, like Elm, make FRP primitives first-class citizens[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 a set of FRP libraries with syntax 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 can be translated into calls to the WebAudio API.

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* with type `a` and performs any 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:

```
main :: Effect Unit
myEvent = do
  -- unsub is now Effect Unit after being called
  -- as part of the ‘bind‘ operation
  unsub <- makeEvent
  -- do some things, then return unsub
  -- which is Effect Unit - the type of ‘main‘
  -- in PureScript and Haskell (although it is
  -- called IO in Haskell)
  unsub
```

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

With `Event`, there is no notion of time. `Behavior`, on the other hand, introduces a notion of *sampling* an arbitrary value `b` based on a known entity `a`. While behaviors can be parameterized for any event type, it is customary to use the event definition above.

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

type Behavior a = ABehavior Event a
```

For a behavior to work, it must always contain a value of type `a` that can be used in a sampling function

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

that produces a value of *any* type. For example, one common behavior is the current time, with a signature `currentTime :: Behavior Number`. The current time can then be composed in an applicative fashion with other elements of a computation. All the consumer of `currentTime` knows is that it will used to construct a larger behavior that will act as an input to a rendering engine. The consumer has no idea what the type of this engine will be. In the example below, it is used to produce a new `Behavior` ten seconds in the future that can be further processed downstream or rendered by an engine.

```
tenSecondsIntoTheFuture :: Behavior Number
tenSecondsIntoTheFuture =
  (_ + 10.0) <$> currentTime
```

One way to think of behaviors is polymorphic procrastination. Because consumers of behaviors do not know how they will be rendered, they must allow for the rendering to be polymorphic. This is achieved using rank-two polymorphism, or existential qualification, to assert with the type system that a `b` exists such that for all `a`, `event (a -> b) -> event b`. This delays the resolution of the polymorphic type until the end of the computation, ie the rendering phase. The result is that behaviors can represent heterogeneous types through a computation and only specialized at the end - a feature of type-based existential qualification most commonly used in heterogeneous lists.[5] In our case, behaviors will represent time, mouse clicks, MIDI events, and a host of other input items that are blended together using the `Behavior` applicative functor.

2. AUDIO GRAPHS AS BEHAVIORS

Using the above definitions of the `Event` and `Behavior` types, we can construct a basic outline of an audio graph. The basic setup contains three steps:

- Subscribe to an event loop that is scheduled using the browser's `setTimeout`.
- In the subscription, sample the audio-graph behavior based on an arbitrary event.
- Use the resulting audio graph to drive the Web Audio API.

In pseudo-code, let's see how a behavior would look that incorporates information of if the mouse is clicked or not. We'll use a `clicked` behavior.

```
import FRP.Mouse(clicked, mouse getMouse)

audioGraph :: Mouse -> Behavior AudioUnit
audioGraph mouse = f <$> clicked mouse
  where
  f b = sinOsc (if b then 440.0 else 220.0)

main = do
  mouse <- getMouse
  unsub <- subscribe
    (interval 40) -- every 40 ms
    (const $ do
      event <- create -- make an arbitrary event
      audioNow <- sample_ (audioGraph mouse) event
      sendToWebAudio audioNow
    )
```

The power of the behavior abstraction is that it enforces a clear separation between the push/pull mechanism (`main`) and the building of the audio graph (`audioGraph`). This allows us to use a declarative style in the audio graph via `clicked mouse` without worrying about the temporality and implementation of mouse event listeners.

In the example above, there is some information that is propagated via behaviors (ie is the mouse `clicked`) whereas other information that is the result of an effectful computation (ie get the mouse via `getMouse`). This distinction is somewhat arbitrary: we could also write a mouse "behavior" that gets the mouse at any given time, and we could run an effectful `isClicked` computation whose result we pass to the audio graph instead of using a behavior. In this way, FRP is a leaky abstraction. Temporal events that are somehow "distant" from the calling code, like a mouse click, are often represented as behaviors because it allows for a succinct, declarative API like the one above. On the other hand, one-time events like `getMouse` as well as temporal phenomena that are tightly coupled with the calling code (like the current time of the audio clock of the context to which the calling code is associated) are often passed directly to a function.

3. AUDIO BEHAVIORS IN EXAMPLES

The following section contains three representative scenario of how temporal information can be used to construct audio graphs via FRP: approximate functions of time, precise functions of time, and integration over time.

3.1 Approximate functions of time

Approximate functions of time use quantized time at a given control rate. For example, on klank.dev, the default control rate is 50Hz. For many functions, such as gradual changes of amplitude and frequency, sampling at regular intervals produces the desired sonic effect.

As an example, consider the following audio behavior[1]:

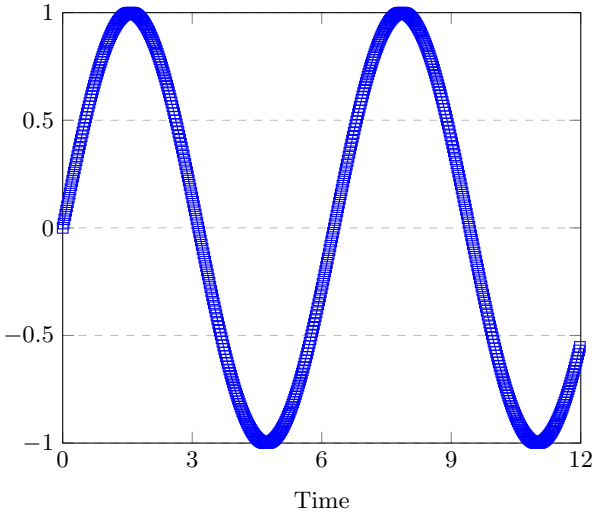
```
scene :: Number -> Behavior (AudioUnit D1)
scene time =
  pure
  ( speaker
         ((gain' (0.1 + (-0.1) * cos (0.5 * rad))
           (sinOsc $ 440.0 + 6.0 * (sin (0.1 * rad))))
         :| (gain' (0.1 + (-0.1) * cos (0.47 * rad))
           (sinOsc $ 330.0 + 4.0 * (sin (0.2 * rad))))
         : Nil
         )
  )
  where
  rad = pi * time
```

All of the control rate variables, meaning everything dependent on time, is changing at a rate of 50 Hz,

[1]https://klank.dev/?k&url=https://klank-share.s3.eu-west-1.amazonaws.com/K16086575564388849.purs&klank=https://klank-share.s3.amazonaws.com/klank16086575602675017.js

which means that the minima and maxima of the sine and cosine waves are truncated by a crude linear approximation. That said, the control rate is sufficiently granular to produce the sound of gradually undulating waves without any perceptible artifacts. For more examples, there is a growing body of documentation that can be found at https://discourse.klank.dev as well as https://github.com/mikesol/purescript-audio-behaviors.

Figure 1: Discretized sine wave at the `purescript-audio-behaviors` control rate.



3.2 Precise functions of time

When using control data to model precise rhythmic events, aliasing will often truncate inflection points, making them arrive to early or too late. To remedy this, it is possible to specify with sample-rate accuracy (44100 Hz) at what time a value should occur during a control cycle.

As an example, consider the following audio behavior[2]:

```
pwf :: Number -> Array (Tuple Number Number)
pwf x =
  join
    $ map
        ( \i ->
            map
              ( \(Tuple f s) ->
                  Tuple (f + x * toNumber i) s
              )
              [ Tuple 0.0 0.0,
              Tuple 0.02 0.7,
              Tuple 0.06 0.2 ]
        )
        (range 0 400)


msBetweenSamples = defaultEngineInfo.msBetweenSamples
kr = (toNumber msBetweenSamples) / 1000.0 :: Number
```

[2]https://klank.dev/?k&url=https://klank-share.s3.eu-west-1.amazonaws.com/K16087041970194799.purs&klank=https://klank-share.s3.eu-west-1.amazonaws.com/klank16087041988326746.js

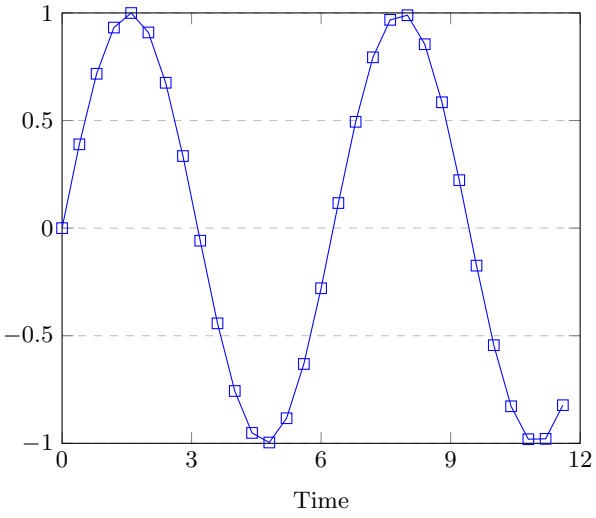
```
epwf = evalPiecewise kr

pulse t v p n s =
  gain_' (t <> "go") v
    (gainT_' (t <> "gi")
      (epwf (pwf p) s) $ sinOsc_ (t <> "sw") n)

scene :: Number -> Behavior (AudioUnit D1)
scene s =
  pure
    $ speaker_ "speaker"
        (map (applyFlipped s)
           (pulse "g0" 0.1  0.11 440.0
            :| pulse "g1" 0.1 0.13 660.0
            : pulse "g2" 0.1 0.15 990.0
            : pulse "g3" 0.1 0.9 220.0
            : pulse "g4" 0.05 0.14 1210.0
            : pulse "g4" 0.025 0.12 1580.0
            : Nil
        ))
```

The polymetric rhythmic events are placed with audio-rate accuracy using `evalPiecewise`, which is a function that uses *both* time and the control rate to make sure the peaks and troughs of a piecewise function are rendered at the nearest audio-rate sample.

Figure 2: Discretized sine wave at a lower control rate causes less-precise extremes and a narrower ambitus, requiring precise event placement between control-rate cycles.



3.3 Integration over time

The behavior pattern is stateless, which makes it difficult to integrate over time, as an integral (or `fold`) requires remembering an aggregate and modifying it. As mentioned previously, behaviors are existentially qualified - their signature reads *there exists an `x` such that, if you give me a function in the form `(a -> x)`, I'll give you an `x`.* The rendering function can parameterize `x` as a feedback unit that reports previous values (or an initial set of values) and retains new values for a next iteration.

When working with behaviors, integrals or folds are useful in two contexts:

1. When a function of time is easier to express as a differential equation. In this case, one can integrate over time to solve the equation. Phil Freeman's `purescript-behaviors` library uses this technique for many of the example animations, producing effects like simple harmonic motion.[3]

2. When a function needs to deal with indeterminacy, ie responding to a user interaction.

As an example of the latter, consider the following scene. When one clicks anywhere with a mouse, the pitch begins to rise linearly. In other words, it integrates over a constant (the slope) with the initial value being the point in time of the click.[4]. To perform the integration, it uses an accumulator of type `{ onset :: Maybe Number }`.

```
scene ::
Mouse ->
{ onset :: Maybe Number } ->
CanvasInfo ->
Number ->
Behavior (AV D2 { onset :: Maybe Number })
scene mouse acc@{ onset }
  (CanvasInfo { w, h }) time = f time <$> click
where
f s cl =
  AV
    ( Just
        $ dup1
            (
  (gain' 0.1 $ sinOsc
       (110.0 + (3.0 * sin (0.5 * rad))))
    + ( gain' 0.1
          $ sinOsc
              ( 220.0
                  + ( if cl then (
    50.0
    + maybe 0.0
        (\t -> 30.0 * (s - t))
        stTime
                        )
                      else
                        0.0
                    )
              )
      )
            ) \mono ->
            speaker
              $ ( (panner (-0.5)
                  (merger
                    (mono +> mono +> empty)))
                    :| Nil
                )
    )
    ( Just
        $ filled
            (fillColor (rgb 0 0 0))
            (circle (w / 2.0)
            (h / 2.0)
            (if cl then 25.0 else 5.0))
    )
    (acc { onset = stTime })
  where
  rad = pi * s

  stTime = case Tuple onset cl of
    (Tuple Nothing true) -> Just s
    (Tuple (Just y) true) -> Just y
    (Tuple _ false) -> Nothing

click :: Behavior Boolean
click = map (not <<< isEmpty) $ buttons mouse
```

For a larger example of this integration/accumulator pattern, you can refer to https://bit.ly/silent-night-kdv (https://bit.ly/silent-night-klank in developer mode), which uses an accumulator containing information about user interactions to shape the work over time.

4. CHALLENGES AND INITIAL SOLUTIONS FOR FRP WEB AUDIO

Using a stateless functional language like PureScript to model the stateful, imperative WebAudio API presents several challenges. In this section, I will discuss three difficult aspects of using FRP in web audio along with initial solutions implemented in `purescript-audio-behaviors`.

4.1 Persistent audio units

When an audio context is created in JavaScript, it persists until there is no longer a reference to it, at which point the context and all live audio units are destroyed. The audio context is stateful, meaning that when audio units are connected or disconnected, it retains the state of the audio graph until it is either changed again or destroyed. In addition to being a useful abstraction for programmers, this also provides the necessary internal memory to nodes like `DelayNode`, `BiquadFilterNode` and `ConvolverNode` that must retain a previous state and/or previous audio.

The behavior pattern, on the other hand, is stateless: it does not remember what the contents of a prior behavior were. Even when using an accumulator, there is no way to precisely correlate objects in an audio graph with WebAudio API objects. For example, consider the following behavior that turns on a second oscillator after 10 seconds.

```
scene time = pure $
  gain 0.2 (sinOsc 440.0 :|
    (if time > 10.0
      then
      pure (sinOsc 880.0)
      else Nil))
```

There is no information in the graph that would tell the renderer "do not turn disconnect or reassign the 440Hz oscillator at the ten-second mark." The library could choose to assign the previous 440Hz oscillator to the 880Hz oscillator and create a new oscillator at 440Hz. Or it could choose to destroy the previous oscillator and create two new ones. All of these scenarios have different sonic properties, but the intent is most likely to persist the 440Hz oscillator to avoid clicks related to change in phase.

[3]https://github.com/paf31/purescript-behaviors
[4]https://klank.dev/?k&ec&url=https://klank-share.s3.amazonaws.com/K1608660252992183.purs&klank=https://klank-share.s3.amazonaws.com/klank1608660256958970.js

Fortunately, due to the deterministic nature of PureScript's evaluation, it is most often the case that the order of object creation is the same across cycles, leading to a natural persistence of audio units from one cycle to the next. However, subtle changes can provoke entirely different orders of evaluation, leading to noticeable jank. To mitigate this, one strategy used by many PureScript libraries, including `purescript-audio-behaviors`, is the *slot* pattern. In the slot pattern, an additional tag of an arbitrary type (often a string) is assigned to each object created. This tag is then used to group similar objects. Below, two slots are used to disambiguate the two oscillators:

```
scene time = pure $
  gain 0.2 (sinOsc_ "osc0" 440.0 :|
    (if time > 10.0
      then pure (sinOsc_ "osc1" 880.0)
      else Nil))
```

4.2 Front-loaded evaluation

The 50Hz control-rate deadline for JavaScript evaluation is more than enough for most simple audio graphs. However, as the graphs get more complicated, 50Hz risks becoming too slow, leading to dropped frames.

One way to solve this issue is to pre-compile as much of the graph as possible as an array that can be indexed by time, very much like a lookup table for sine waves, but at the audio-graph scale. For instance, the previous pulse example can be almost entirely pre-compiled, as it stops computation after 400 iterations. That leads to a 97% reduction in evaluation time of the graph at the expense of upfront compilation time. The reduction becomes more substantial as the graph becomes larger [5].

```
maxPulse = 18000 :: Int

pwf :: Number -> Array (Tuple Number Number)
pwf x =
  join
    $ map
        ( \i ->
            map
              ( \(Tuple f s) ->
                  Tuple (f + x * toNumber i) s
              )
              [ Tuple 0.0 0.0
              , Tuple 0.02 0.7
              , Tuple 0.06 0.2
              ]
        )
        (range 0 400)

kr =
  (toNumber defaultEngineInfo.msBetweenSamples)
    / 1000.0 ::
    Number

epwf ::
  Array (Tuple Number Number) ->
  Number ->
  AudioParameter
epwf = evalPiecewise kr

pulse t v p n s =
  gain_' (t <> "go") v
    ( gainT_' (t <> "gi")
        (epwf (pwf p) s)
        $ sinOsc_ (t <> "sw") n
    )

makeGraph :: Number -> AudioUnit D1
makeGraph s =
  speaker_ "speaker"
    ( map (applyFlipped s)
        ( pulse "g0" 0.1 0.11 440.0
            :| pulse "g1" 0.1 0.13 660.0
            : pulse "g2" 0.1 0.15 990.0
            : pulse "g3" 0.1 0.9 220.0
            : pulse "g4" 0.05 0.14 1210.0
            : pulse "g4" 0.025 0.12 1580.0
            : Nil
        )
    )

pulses :: Array (AudioUnit D1)
pulses =
  map
    (makeGraph <<< (_ * kr) <<< toNumber)
    (range 0 maxPulse)

defaultGraph :: AudioUnit D1
defaultGraph = makeGraph (toNumber maxPulse * kr)

getPulse :: Number -> AudioUnit D1
getPulse time =
  fromMaybe defaultGraph
    $ index pulses (floor $ time / kr)

scene :: Number -> Behavior (AudioUnit D1)
scene = pure <<< getPulse
```

4.3 Scheduling

As is the case with all realtime audio processing, Web Audio scheduling must achieve a balance between look-ahead, which safeguards against dropped frames, and JIT evaluation, which allows for reactive responses to user input.[8][7] There is no silver-bullet to achieve low-millisecond responsiveness without dropped frames, and the audio graph will only be as responsive as:

- The periodicity of the polling function discussed in Section 1. In most real-world applications, 20 milliseconds is an achievable gap between polls.
- The amount of time it takes to build the graph and report it to the web-audio API. By definition, this should not exceed the amount of time between polls, and ideally should be less (ie 15 milliseconds).
- The amount of look-ahead time, which is often 20-40 milliseconds.

[5] https://klank.dev/?k&url=https://klank-share.s3.amazonaws.com/K16087046614232954.purs&klank=https://klank-share.s3.amazonaws.com/klank16087046653836736.js

This leads to a delay of 60-80 milliseconds for responsive input, which is almost always perceptible.[6] For mouse clicks, this tends to be less obvious, as psychologically we expect to see the results of a click only after lifting the mouse, whereas processing can start on a `mousedown` event. However, for MIDI events, this is longer than the minimum latency needed for musicians to feel comfortable playing a digital instrument (roughly 15ms).[4]

In `purescript-audio-behaviors`, there is ongoing work to improve audio scheduling so that it achieves the 15 millisecond deadline needed for real world musical instruments. This includes:

- Using event listeners to trigger the construction of the audio graph in addition to a pure polling solution. This eliminates the 20ms of between-poll latency, achieving sub-millisecond triggering of the graph construction.
- By pre-rendering substantial chunks of the graph, the building of the audio graph can take less than 10ms. It is possible to shrink this time even further by pushing the pre-rendering to include the list of imperative, assembly-like instructions given directly to the WebAudio API.
- Including instructions to ignore look-ahead rendering for certain events, scheduling them for immediate execution.

These changes are intended to be part of the `0.0.0` release of `purescript-audio-behaviors` that will hopefully arrive in mid-2021.

5. CONCLUSIONS

FRP and the *behavior pattern* is an attractive way to turn the imperative WebAudio API into a declarative data structure that is calculated as a function of several input streams such as the `currentTime` of the audio context, mouse events and MIDI-events. Because the behavior pattern requires the construction of an entire audio graph based on polling, it tends to be ill-suited to reactive, realtime audio applications. This can be mitigated by several strategies, including using the DOM's native event listeners, pre-compiling substantial parts of the graph and mixing look-ahead with immediate execution.

6. 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-audio-behaviors`.

7. 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] R. H. Jack, A. Mehrabi, T. Stockman, and A. McPherson. Action-sound latency and the perceived quality of digital musical instruments: Comparing professional percussionists and amateur musicians. *Music Perception: An Interdisciplinary Journal*, 36(1):109–128, 2018.

[5] O. Kiselyov, R. Lämmel, and K. Schupke. Strongly typed heterogeneous collections. In *Proceedings of the 2004 ACM SIGPLAN workshop on Haskell*, pages 96–107, 2004.

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

[7] C. Pendharkar, P. Bäck, and L. Wyse. Adventures in scheduling, buffers and parameters: Porting a dynamic audio engine to web audio. In *Proceedings of the Web Audio Conference, Paris, France*. Citeseer, 2015.

[8] N. Schnell, V. Saiz, K. Barkati, and S. Goldszmidt. Of time engines and masters an api for scheduling and synchronizing the generation and playback of event sequences and media streams for the web audio api. 2015.

[6] See https://twitter.com/stronglynormal/status/1326136187116023812 and https://twitter.com/stronglynormal/status/1316756584786276352 for examples of this latency