Bithoven

Gödel Encoding of Chamber Music and Functional 8-Bit Audio Synthesis

Jay McCarthy
University of Massachusetts Lowell
jay@racket-lang.org

Categories and Subject Descriptors H.5.5 [*Sound and Music Computing*]: Methodologies and techniques

Keywords Audio Synthesis, Computer Music, NES Emulation

Abstract

Bithoven is a prolific composer of approximately 1.079363×10^{239} different compositions based on four-part harmony and basic chord progressions. It is combined with a purely functional audio synthesis engine based on the Ricoh RP2A03, found in the 1985 Nintendo Entertainment System (NES). The synthesis engine is parameterized over a band of instruments and styles of play, so each composition can be played in one of approximately 4.22234×10^{41} different arrangements or "NEStrations".

1. Introduction

As children of the nineteen eighties age, the cultural prominence of the nostalgic sights and sounds of their youth is growing more present. For example, media productions with tens of millions of consumers, like *Wreck-It Ralph*, *Ready Player One*, *Steven Universe*, and *Scott Pilgrim vs the World*, casually reference retro-computer culture that was only popular among a vastly smaller population. This is most prevalent in the popularity of so-called "video game beats" in music. For example, the best-selling single of 2010, *Ke$ha*'s *TiK ToK* features an 8-bit synthesized beat, while other acts like *Anamanaguchi* and *Deadmau5* use authentic eighties hardware in their live performances.

In this paper, we discuss the design and implementation of *Bithoven*, an algorithmic choral composer, and SRPNT, an audio synthesizer that was inspired by the Ricoh RP2A03, which was the audio processing unit (APU) used in the 1985 Nintendo Entertainment System (NES). SRPNT is purely functional synthesis that uses a series of DSLs to define and play 8-bit music. Bithoven is an enumeration of all choral music obeying certain basic principles of melody and rhythm, which we truncate to only include short songs. Bithoven's arranger is another enumeration of different sets of instruments and ways of playing music that interprets Bithoven's compositions into SRPNT music.

The paper's structure reflects a bottom-up perspective on the system. Section 2 reviews hardware audio synthesis and the RP2A03 in particular. Section 3 relates the details of the synthesis engine. Section 4 describes a DSL for programming digital instruments. Section 5 reviews some essential of music theory. Section 6 describes a DSL for music tracking. Section 7 reviews the theory of Gödel encoding. Section 8 describes our datatype for arrangements. Section 9 finalizes the presentation with our datatype for compositions. In section 10, we discuss some usage pragmatics. Section 11 summarizes some related work and we conclude in section 12.

The Racket source code for the system is available online at `https://github.com/jeapostrophe/srpnt`. Furthermore, the repository contains audio samples, sheet music, and other supplemental materials. We recommend listening to a few samples before reading to get an idea of what Bithoven produces.

2. Hardware Synthesis and the Ricoh RP2A03

Hardware audio synthesis is based on the understanding of sound as a mechanical wave of pressure and early synthesizers could only generate particular primitive waveforms from a set of templates. The Ricoh RP2A03 is typical in this way. It can generate five discrete audio waveforms: two pulse waves, one triangle wave, one noise channel, and one sample channel (Taylor 2004). Each channel was independently converted into an analog signal and combined in a non-linear way into a single monaural channel. We discuss the abstract operation of each channel in order of simplicity.[1]

The *sample* channel, called the DMC, outputs a 7-bit audio signal that is controlled via delta pulse-code modulation. The signal is initialized to a program controlled value after which it is adjusted by a stream of 1-bit delta values loaded from program memory. A high bit increments the value and a low bit decrements the value. The signal is clamped, rather than overflowing. Due to its inability to vary more than a single value each audio frame, arbitrary sounds cannot be encoded as samples, in addition fidelity is lost due to the 7-bit clamp. Nevertheless, the sample channel is useful for including otherwise hard to synthesize sounds, like voice samples and realistic drums.

The *noise* channel outputs a 4-bit audio signal that is controlled via a 1-bit pseudo-random number generator (*PRNG*). If the PRNG produces a high bit, then the 4-bit volume is produced, otherwise a zero signal is generated. The PRNG is implemented as a 15-bit linear feedback shift register. The program can influence the channel in three ways: first, it can control the volume produced; second, it can change the frequency at which the PRNG is stepped to one of sixteen different options; finally, it can change the feedback bit between bits 6 and 1, which produces a shorter period that tends

[1] We recommend listening to the following examples before and after reading this section: `https://www.youtube.com/watch?v=la3coK5pq5w`.

```
(define DUTY (vector 0.125 0.25 0.5 0.75))
(define (pulse-wave n period volume %)
  (define freq
    (pulse-period->freq period))
  (define duty-cycle
    (vector-ref DUTY n))
  (define next-% (cycle%-step % freq))
  (define out
    (if (< next-% duty-cycle)
        volume
        0))
  (values out next-%))

(define CPU-FREQ-MHz 1.789773)
(define CPU-FREQ-Hz
  (* CPU-FREQ-MHz 1000.0 1000.0))
(define (pulse-period->freq period)
  (/ CPU-FREQ-Hz (* 16.0 (+ 1.0 period))))

(define (cycle%-step % freq)
  (define %step (/ freq 44100.0))
  (define next% (+ % %step))
  (- next% (floor next%)))
```

Figure 1: Pulse Wave Generator (from `srpnt/apu`)

to produce a "metallic" sound. The noise channel normally sounds like television static, but by quickly turning it on and off rhythmically, it can be used for percussion.

The *triangle* channel outputs a 4-bit audio signal with a defined pattern at a program-controlled 11-bit frequency. The signal has a 32-step inverted triangle pattern that starts high, goes low, then returns to high. Thus, the triangle has no volume control: it is either on or off. The interpretation of the frequency is between 55.9kHz (higher than the highest key on a standard 88-key piano) and 27.3Hz (slightly lower than the lowest A). The frequency of the triangle is versatile, but the lack of volume control and the pulse channel's inability to play low notes relegates it to the bass line in most compositions.

The two *pulse* channels behave identically. They output a 4-bit audio signal at a program-controlled 11-bit frequency. The signal value is set by the program, but the pattern is defined by the circuit. However, the program may select one of four patterns, called a *duty cycle*. Each pattern is eight frames long. The first has a single on bit, the second has two, the third has four, and the fourth is second inverted (thus, it sounds the same as the second.) The interpretation of the frequency is between 111.8kHz and 54.6Hz, so it cannot play the lowest octave on a standard piano. These tend to be used for the melody in most compositions.

Nearly all of the most popular NES music uses only the programmable waveforms and ignore the sample channel. The key to using these effectively is to quickly vary the programmable parameters to simulate different instruments, as we discuss in section 4.[2]

3. Functional Audio Synthesis

Our *synthesizer* is modeled after the Ricoh RP2A03, but is not a perfectly accurate simulation of it. We divide it into components: authentic waveform generators and an inauthentic mixer. The generators are called in a loop to produce individual channel samples and then combined by the mixer to produce a single sample. We generate 44,100 samples per second.

Each waveform generator `G` is represented via a function of type `(-> Parameters State (values Signal State))` where `Parameters` is a structure holding the varying parameters, `State` holds the internal state of the generator, and `Signal` is the output signal value. This model is essentially the compiled form of a functionally reactive program (Elliott and Hudak 1997).

Figure 1 shows the pulse wave generator, `pulse-wave`. There are three `Parameters`: the duty cycle identifier, the period, and the volume. A period is used rather than a frequency, because the RP2A03 actually uses a ticking counter running at its own frequency, which creates a strange discretization of the available frequencies. However, we internally convert the period into the appropriate frequency with `pulse-period->freq`, which divides the clock rate of the CPU by the appropriate amount. The `State` is simply an inexact value representing the percentage the generator is through a single cycle, which is adjusted by incrementing it by a fraction of the frequency (based on the sample rate). Finally, the output `Signal` is either 0 or the volume parameter.

The other generators work similarly, with appropriate differences for the particular waveform: the triangle's state is also a percentage, the noise's state is the register value and a percentage, while the sample channel is simplified into a vector of 7-bit samples and an offset to start at.

We cannot use our generators, however, unless we can control their parameters over time and combine the results into an audio buffer delivered to the underlying consumer, whether it be the operating system's audio interface or a recording. We fix a particular rate of audio generation at 60 audio frames per second. Since there are 44,100 samples per second, this means there are 735 samples per frame. This fixes the smallest quantum of musical change at about 16.6 milliseconds.

We define a structure for each kind of wave form to hold its parameters over the course of a single 735 sample audio frame:

```
(struct wave:pulse (duty period volume))
(struct wave:triangle (on? period))
(struct wave:noise (short? period volume))
(struct wave:dmc (bs offset))
```

Relating this model back to the RP2A03, it would take a synth frame consisting of two `wave:pulse` values and one each of the other values, then call the waveform generators in a loop 735 times, threading the state through each time and recording the output signals, mixing them, and then delivering the combined signal to the consumer. A sketch of this, specialized to just the single pulse wave, is as follows:

```
(struct synth-frame (p1))
(define (synth sf)
  (match-define (synth-frame p1) sf)
  (match-define (wave:pulse d p v) p1)
  (define sample-count 735)
  (define samples (make-bytes sample-count))
  (for/fold ([p1-% 0.0])
            ([i (in-range sample-count)])
    (define-values (p1 new-p1-%)
      (pulse-wave d p v p1-%))
    (bytes-set! samples i p1-d)
    new-p1-%)
  samples)
```

Our synthesizer is similar, but expands the capabilities of the RP2A03 slightly with a wider *synth frame*. We allow two each of the pulse and triangle, so that we can have four voices (as we discuss in section 9, our composer produces choral quartets) with

[2] We recommend listening to a sampler of tracks, such as: `https://www.youtube.com/watch?v=UNA3Laa3-_Q`.

two high and two low. Next, we provide three noise channels to simulate a traditional three piece drum kit. Finally, we provide two sample channels where one is for the left audio and the other is for the right audio, so that users of the synthesizer can produce stereo effects, such as approaching gunfire for gaming applications.

Due to these extensions, we cannot simulate an authentic RP2A03 mixer. Instead, we simply add the seven 4-bit values to produce a 7-bit quantity, then linearly mix it with the left or right sample channel to render a combined 7-bit sample. In summary, we generate 735 such samples and then return the state of each waveform generator for the next synth frame.

Finally, the audio system is provided as function that accepts of list of synth frames and delivers the corresponding set of samples to a recipient, which either immediately plays them or saves them.

4. Instruments

It is difficult to use the synthesizer by constructing lists of synth frames directly. First, it is painful to relate tones (the pitches we hear) to periods. Second, it is difficult to relate notes (the duration of tones) to numbers of frames. Finally, producing particular timbres (the texture of one instrument versus another) by hand is laborious and complex. We leave the second problem to future sections, but address tones and timbre in this section.

In Western music, tones are given names based on their *pitch*, i.e. their position in the scale (C through B), as well as their octave (a number). For example, the right-most key on a standard piano is **C8**, while the left-most key is **A0**. These tones are mapped to frequencies by tuning conventions. Although it was not always this way, at present we define **A4** as 440Hz. This definition gives the frequency of the `n`th key as `(* (expt 2 (/ (- n 49) 12)) 440)` Hz. From this, we can map the frequency to a period on either the pulse wave or the triangle wave by inverting the period to frequency function:

```
(define (pulse-freq->period freq)
  (define pre (/ CPU-FREQ-Hz (* 16.0 freq)))
  (round (- pre 1.0)))
```

We encapsulate this composition as a function, `pulse-tone->period`, and henceforth assume that we deliver tone names to waveform generators. We do the same for the triangle, but not for the noise, because its period is unrelated to its tone, but just controls the PRNG feedback.

Timbre refers to the qualities that make two signals of the same pitch and volume sound differently. For example, human voice, violin, and piano can all produce a **C4**, but may all sound differently when they do. A simplistic way to understand timbre is variation across the length of a note in pitch and volume. For example, a plucked string's volume tends to decay rapidly compared to drawing a bow across it, which produces a stable volume. Other examples include vocal vibrato, where the pitch is modulated across the note (notably exaggerated in our stereotypes about opera), and tremolo, which is a modulation of volume.

We deal with timbre in the context of our system in two steps. First, we define the type `Instrument` as a function of type `(-> Tone Nat (listof Synth-Frames))`, i.e. it accepts a tone to play, a number of frames to play it for, and it produces synth frames. This representation allows higher levels of abstraction to simply "play an A4 on a piano-like pulse for 33ms" by calling the appropriate function with the arguments `'A4` and `2`. Second, we define a DSL in the style of functional reactive programming (FRP) for writing instruments (Elliott and Hudak 1997).

Figure 2 shows the constructor for pulse wave instruments. The FRP-like values are called "specs" (for specifications) and are essentially signals defined over time. The constructor `i:pulse` receives one spec for each of the parameters of the pulse wave:

```
(define (i:pulse #:duty ds
                 #:period ps
                 #:volume vs)
  (λ (frames tone)
    (define d* (stage-spec ds frames))
    (define p* (stage-spec ps frames))
    (define v* (stage-spec vs frames))
    (define base-per
      (pulse-tone->period tone))
    (for/list ([f (in-range frames)])
      (define duty (eval-spec d* f))
      (define per
        (fx+ base-per (eval-spec p* f)))
      (define volume (eval-spec v* f))
      (wave:pulse duty per volume))))
```

Figure 2: Pulse Instrument (from `srpnt/nestration/instrument`)

the duty cycle, the period, and the volume. The duty cycle and volume specs are expected to evaluate to the actual value, while the period spec is interpreted as a delta from the period of the tone that is played. The body of the loop in the constructor does the obvious thing: it evaluates the spec for each frame and constructs the corresponding synth frame. The outside of the loop, however, must first "stage" the specification by informing it what the total number of frames will be. We elaborate on this below.

We proceed by providing examples of instruments and spec constructors. First, consider a trivial pulse with constant values for each parameter: a given duty, a constant period, and a fixed volume.

```
(define (i:pulse:basic duty)
  (i:pulse
   #:duty (spec:constant duty)
   #:period (spec:constant 0)
   #:volume (spec:constant 7)))
```

This uses the constant specification, `spec:constant`, which returns the same value for every frame.

We can implement a pulse with tremolo, which is a modulation of volume. We accept a parameter `freq` for the frequency of the modulation and replace the `#:volume` argument with `(spec:% (spec:modulate freq 7 4))`. When evaluated, `spec:%` divides the frame by the total number of frames (provided during staging) to produce the percentage through the note. It then delivers this value to the spec given by its argument. For example, if note will be played for 10 frames, then `spec:modulate` will be called with `0.1`, `0.2`, and so on until `1.0`. `spec:modulate` receives three arguments: the modulation frequency, the base value, and the width of the modulation. Essentially it views the percentage argument as a radian and evaluates sine at the given frequency and position to produce a value between `-1.0` and `1.0`, which we multiply with the width and then add to the base. Thus, the volume oscillates between `3` and `11`, according to the frequency.

We can implement a basic decaying (or strengthening) signal with a linear interpolation using `spec:%`. For example, `#:volume (spec:% (spec:linear 7 0))` smoothly interpolates from volume `7` to silence over the whole note. `spec:linear` simply multiplies the initial value by the remainder of the percentage and adds it to the multiplication of the final value and the percentage.

Finally, we can implement a traditional ADSR specification (Pinch and Trocco 2004). ADSR (Attack-Decay-Sustain-Release) is an early theory of instrument synthesis going back to the Novachord and Moog synthesizers. It divides the synthesis of a particular note

into four stages: the Attack, where it linearly increases; the Decay, where it linearly decreases; the Sustain where it is constant; and the Release, where it linearly decreases again. This notion can, of course, be generalized to arbitrary many stages and arbitrary specifications on each stage. For instance, the Casio CZ has 8-stages with three Delay slopes and two additional Attacks, one before the Sustain and one in the middle of the Release. The entire configuration is referred to as the Envelope in the synthesis literature.

We define a `spec:adsr` combinator that accepts one sub-spec argument for each stage, as well as four other arguments that define how long each stage is, and a final argument that specifies which argument receives any left over frames. For example, the following volume specification, which we refer to as "plucky" (because it sounds like a plucked string), gives four frames to each stage, makes the Attack a constant 14, the Decay goes from 14 to 7, the Sustain holds at 7, and the Release decreases from 7 to 0, for a total of 16 frames. If fewer than 16 frames are given, then the frame counts are used as percentages. For instance, if 8 frames were available, then each would be given 2 frames. On the other hand, if more than 16 frames were given, then any extra frames would be allocated to the Release stage. This makes it so that as the length of the note increases, it will be trailed by more silence.

```
(spec:adsr
 'release
 4 (spec:constant 14)
 4 (spec:linear 14 7)
 4 (spec:constant 7)
 4 (spec:linear 7 0))
```

In most cases, we use this specification language for controlling the volume of a note, which changes the overall shape of the waveform, but leaves the tone perfect.

The same specification language is also used for components of the drum kit. For example, through intense experimentation, we have found configurations that resemble a hihat, bass, and snare (shown in figure 3).

In summary, we define a simple embedded DSL for describing instruments as particular configurations of the primitive waveforms over time. We define a repertoire of these instruments which will be used in section 8 for determining how to play Bithoven's compositions.

5. Essentials of Music Theory

In our discussion so far, we have gone from sound to notes and now we come to music. There are five main concepts that we employ in the rest of our development: notes, tempo, time signatures, scales, and chords. Below, we define each of this. Of course, these definitions are not ours and not universally agreed upon, as the concepts of music theory are ancient and have considerably variability over time (Hewitt 2008). Leaving that aside, we provide our technical definitions and explanations as to how they will be used later.

Note. A *note* is a relative measure of time and therefore length of a tone. We define notes as non-positive powers of two. For example, a half-note is twice as long as a quarter-note and half as long as whole note. Traditional music admits other notes, but we do not consider them. A note is a relative measure and not an absolute measure, because it depends on a tempo to concretize it.

Tempo. A *tempo*, or *metronome*, is a pair of a note, called the *beat unit*, and the number of beats per minute. For example, `(cons 1/4 76)` is a tempo called *andante*. The key thing that a tempo gives us is the ability to convert notes into frames by simply multiplying out the quantities while tracking units. For example, at `(cons 1/4 60)`, a quarter note lasts 60 frames. The following function does this:

```
(define (frames-in-note me note)
  (match-define
    (cons beat-unit beats-per-minute)
    me)
  (define beats-per-second
    (/ beats-per-minute 60.0))
  (define beats-per-frame
    (/ beats-per-second 60.0))
  (define frames-per-beat
    (/ 1.0 beats-per-frame))
  (define beats-in-note
    (/ note beat-unit))
  (define frames-in-note
    (* beats-in-note frames-per-beat))
  frames-in-note)
```

We use this function inside of the tracker (section 6) to determine the argument to the instruments function.

Time Signature. A *time signature* is a pair of a natural number, the *beats per bar*, and a note, the *beat unit*. (The beat unit in a time signature does not need to match the beat unit in the tempo, but it often will.) A common time signature is `(cons 4 1/4)`, which is referred to as `4:4`. All the music Bithoven produces is in `4:4` and it allows produces complete bars, as discussed in section 9.

Scale. *Scales* are used to define sets of tones that "sound good" together and produce harmony. Music of the Western "common practice period"[3] typically only used tones from a single scale. We only consider scales with seven tones.

A *detached scale* is a sequence of pairs of a pitches and an octave offsets. For example, the diatonic **C** major scale is **C,0**, **D,0**, **E,0**, **F,0**, **G,0**, **A,0**, and **B,0**; while the diatonic **D** major scale is **E,0**, **F#,0**, **G#,0**, **A,0**, **B,0**, **C#,1**, and **D#,1** (notice that the last two elements have a positive octave offset so they are higher than the **D#** adjacent to the first **E**).

A *fixed scale* is a sequence of tones. A detached scale can be transformed into a fixed scale by providing an initial octave and applying each octave offset.

An *abstract scale* is a function from a pitch, called the *key*, to a detached scale. Abstract scales are typically defined as a sequence of *intervals* that are added to the key to produce the detached scale. For example, the diatonic major scale is defined by the intervals `(2 2 1 2 2 2 1)`. There are a variety of common abstract scales. Most people tend to find the diatonic major to be "happy", while the harmonic minor is "sad".

An *abstract tone* is an integer that is interpreted relative to a detached scale. When it is between `0` and `6`, it is an index into the scale. If it is outside of this range, than it refers to the entry modulo seven, but with the octave offset added to the quotient of the tone by seven. For example, `-1` in the diatonic **C** major detached scale is **B,-1** and `7` is **C,1**.

The tracker (section 6) accepts detached tones and fixes them differently for each of the four voices, so that, for example, the bass plays lower than the soprano. Meanwhile, the arranger (section 8) selects an abstract scale, a key, and applies it to the abstract tones produced by Bithoven (section 9), which allows Bithoven's compositions to be transposed into any scale or key.

Chord. A *chord* is a set of elements of a scale that "sound good" when played simultaneously. For example, in diatonic **C** major scale, **C**, **E**, and **G** are a chord. A *chord kind* is a function from a scale and an offset to a chord. For example, the triad is a chord kind that when given `n`, selects the `n`th, `n+3`nd, and `n+5`th elements of a scale; the previous example chord is a triad where `n` is `0`.

Bithoven (section 9) selects tones from chords of scale of abstract tones.

[3] What a lay person might call "classical music".

```
(define i:drum:hihat
  (i:noise
   #:mode
   (spec:constant #f)
   #:period
   (spec:constant 12)
   #:volume
   (spec:adsr
    'release
    1 (spec:constant 4)
    2 (spec:constant 3)
    4 (spec:constant 2)
    4 (spec:constant 0))))
```

```
(define i:drum:bass
  (i:noise
   #:mode
   (spec:constant #f)
   #:period
   (spec:constant 9)
   #:volume
   (spec:adsr
    'release
    1 (spec:constant 10)
    2 (spec:constant 7)
    4 (spec:linear 4 2)
    4 (spec:constant 0))))
```

```
(define i:drum:snare
  (i:noise
   #:mode
   (spec:constant #f)
   #:period
   (spec:constant 7)
   #:volume
   (spec:adsr
    'release
    1 (spec:constant 11)
    4 (spec:linear 11 6)
    8 (spec:linear 6 2)
    4 (spec:constant 0))))
```

Figure 3: Drum Instruments (from `srpnt/nestration/instruments`)

6. Music Tracker

The music *tracker* compiles a sheet-music-like DSL into a sequence of synth frames that can be played by the synthesizer.

It provides a function called `song->commands` that returns a `(listof synth-frame?)`, which (recall) is the input type of the simulator. It has a number of arguments:

- `#:me` — a tempo that is used to convert note lengths into frame counts when instruments are rendered.
- `#:instruments` — A vector of four instruments, two of which are pulses and two of which are triangles. Each instrument is also paired with its initial octave. This is added to octave-deltas that are part of the abstract tones produced by Bithoven. Typically, the first two are the pulses.
- `#:drum` — Similar to `#:instruments`, this keyword provides a vector of three noise instruments, which are used when synthesizing the drum track. Typically, the first is a hihat, the second is a bass, and the third is a snare.
- `#:drum-measure` — A list of measures of drum notes, which are patterns of drum beats to synthesize with the drums. They will be the backing beat to each corresponding measure of tones. A drum measure is three lists of notes where the first list uses the first drum in `#:drum`, and so on. An example is shown in figure 5.
- `#:measures` — The normal list of measures of a song. Each measure is a list of simultaneous tones. Each such simultaneous group has a single note length and then one detached tone for each of the four instruments. In place of a tone, `#f` is allowed for that instrument resting. An example is shown in figure 4.

The behavior of this function is relatively straight-forward: it loops over the `#:measures` and `#:drum-measures` lists to construct a list of synth frames after attaching each tone to the octave given in `#:instruments`, and uses the `#:drums` and `#:instruments` values to compute the synth frames for one note, as determined by `#:me`.

This function imposes some heavy constraints on the music: for example, there are no instrument changes mid-song and there are no tempo changes. It would be easy to change all of these things, but this DSL is primarily designed for Bithoven to target and it is difficult to change Bithoven to produce such things.

In addition to producing the synth frames, the track can also produce sheet music using Lilypond (Nienhuys and Nieuwenhuizen 2003).

```
(((1/4 ((C 0) (C 0) (C 0) (C 0)))
  (1/4 ((D 0) (D 0) (D 0) (D 0)))
  (1/4 ((E 0) (E 0) (E 0) (E 0)))
  (1/4 ((F 0) (F 0) (F 0) (F 0))))
 ((1/4 ((G 0) (G 0) (G 0) (G 0)))
  (1/4 ((A 0) (A 0) (A 0) (A 0)))
  (1/4 ((B 0) (B 0) (B 0) (B 0)))
  (1/4 ((C 1) (C 1) (C 1) (C 1))))
 ((1/4 ((B 0) (B 0) (B 0) (B 0)))
  (1/4 ((A 0) (A 0) (A 0) (A 0)))
  (1/4 ((G 0) (G 0) (G 0) (G 0)))
  (1/4 ((F 0) (F 0) (F 0) (F 0))))
 ((1/4 ((E 0) (E 0) (E 0) (E 0)))
  (1/4 ((D 0) (D 0) (D 0) (D 0)))
  (1/4 ((C 0) (C 0) (C 0) (C 0)))
  (1/4 ((D 0) (D 0) (D 0) (D 0)))))
```

Figure 4: An example track playing the diatonic **C** major scale

```
(define beat:straight-rock
  '((1/8 1/8 1/8 1/8 1/8 1/8 1/8 1/8)
    (1/4 1/4 1/4 1/4)
    (1/4 1/4 1/4 1/4)))
```

Figure 5: An example drum measure for the straight rock beat (from `srpnt/nestration/instruments`)

7. Gödel Encoding

We are now on the cusp of describing how Bithoven actually produces music. In the section, we describe some of the key principles. First, Bithoven is not a random process. Second, it does not use any sort of learning process to determine what "good" music might be. Instead, we inductively define a set, along with a bijection between that set and a prefix of the natural numbers.

We use Racket's `data/enumerate` module for this (New et al. 2016). It provides combinators (suffixed with `/e`) that construct bijections. For example, `string/e` is a bijection between the naturals and strings. It forms a `to-nat` function which projects a string into a natural. For example, `(to-nat string/e "Bithoven")` is 29446701124374494676409535373198388243249180. It

provides a `from-nat` function as well which projects a natural into a string. For example, `(from-nat string/e 42)` is "Q". `from-nat` is particularly convenient, because we can select a number at random and call `from-nat` on it to generate a random element of the set. These functions are defined such that they run in time approximately linear to the number of bits; in other words, they are very efficient.

Using this library, we could simply construct the bijection for valid input to the tracker and have an infinite set of songs. However, most of the elements of this set are uninteresting swill with no discernible melody or harmonies.

Instead, we construct a very particularly defined set where most elements sound pleasing, or at least typical of electronic music from the era of the Nintendo Entertainment System.

8. Arrangements

In order to warm up, we first define the set of arrangements of Bithoven composition. We call these arrangements "NEStrations", as they represent a particular strategy of putting the composition to life on our pseudo-NES audio synthesizer. These are necessary, because Bithoven only produces four sequences of abstract tones. These must be made concrete, attached to instruments, given a tempo, and so on.

The arrangement is a vector of many components:

- The key, drawn from the set of pitches, used to root a scale.
- The abstract scale, drawn from a small of common scales, used to produce a detached scale given the key.
- A tempo, drawn from the range of traditional tempos (about 60 to 200), used to quantize the notes.
- The instruments, drawn from the set of defined pulse, triangle, and drum instruments, used to generate the synth frames by the tracker.
- An assignment of each of the instruments to a particular voice, drawn from permutations of length four, to determine which instrument will play, e.g. the melody.
- An octave for the harmony to be attached to, drawn from near the middle of the piano.
- Octave offsets of the other three parts which are interpreted so that the melody is higher, tenor lower, and bass even lower. These are drawn from small numbers.
- A drum measure for each measure of the song drawn from the available hand-written drum measures.

Given the available options and the components, in our current implementation there are approximately 4.22234×10^{41} members of this set.

However, we actually define a family of such sets where the set each parameter is drawn from comes from is a parameter. This allows us to define well-known subsets based on an aesthetic criteria, which we call *styles*. For example, compositions that are played around 200bpm in a diatonic major scale tend to sound very exciting, so we name that set as `style:happy`, but if we switch to a lower tempo (around 120bpm) and only use minor scales, then it sounds very depressed and sad, so we name it `style:sad`.

In principle, we could make the set of arrangements arbitrarily sized by removing the bounds on octaves, not hand-designing instruments, and so on. Many such arrangements would be absurd, however, because they may be outside of the range of human hearing, for example. In contrast, the very large set we have chosen tends to be made up of reasonable music.

9. Compositions

Each of the earlier pieces of the system build upon the last: the APU produces waveforms, the instruments produce tones, the tracker organizes notes, and the arranger makes concrete decisions about how each of the other pieces will be used. The final piece is to compose the music for the arranger. This entails producing musical structure, harmony, and rhythm. Each of these imposes constraints on the possible set of tones to create patterns that listeners can identify and understand.

Like the arranger, Bithoven does not randomly choose these three things, nor does it choose them based on learning from a sample of quality music, instead we define a set of all possible structures, harmonies, and rhythms. Nevertheless, it is useful to think of Bithoven has making these choices randomly, because they are dependent and it explains the flow of information well.

First, Bithoven selects one of many possible overall song structures. For example, it can choose to generate a symmetric rondo with three parts, A, B, and C arranged as ABACABA; or, it could generate a typical pop song with five parts arranged as ABCBCDCCE. The available song structures comes from a typical list. By having repetition in the overall structure of the song, the music is more palatable because the listener can predict what is going to happen and sense the patterns. During testing of Bithoven, we always listen to a song multiple times to get a sense of whether the repetition is working.

Next, Bithoven uses chords exclusively for harmony. It never chooses tones that are not in a chord with one-another. Furthermore, its music contains harmonic rhythm through choosing a chord progression for the entire song from a different database of possible progressions. A chord progression is a sequence of offsets to use with a chord kind. For example, the progression `(0 3 0 4)` means to use the first triad, the fourth triad, the first again, and then the fifth. Many popular songs are primarily made up of a single chord progression. For example, the progression `(0 5 3 4)` underpins *Blue Moon*, *Donna*, and *All I Want for Christmas Is You*, among many others.

Bithoven replicates this single chord progression in each part of the song, but may use it differently in each one. This produces a slightly repetition structure, where similar sounds appear through the piece without being perfectly duplicated.

Bithoven determines the length (in measures) of each part by the length of the chord progression. For example, if the chord progression has four chords, then each part will have four measures. Bithoven then enumerates every possible way of dividing up the half-notes of the measures into each chord, such that each chord gets at least one half-note. This also ensures that chord changes only happen on half-notes, which creates a slight sense of accenting in the music. This accenting is reinforced by the tracker which increases the volume by one unit on all accented notes (a detailed not mentioned above.)

Within the notes of a particular chord, it chooses a permutation of half-notes and quarter-notes, such that half-notes never cross a measure boundary. This is a complicated permutation to code, because of the dependency in the details, but it is intuitive to define.

Finally, for each note, Bithoven chooses one of the available permutations of notes of the chord for the four parts. For instance, if the chord is `(0 3 5)`, then it will first select one value to be duplicated (to give four values), then choose a permutation of the four values and assign the first to the harmony, the second to the melody, the third to the tenor, and the last to the bass.

Each layer of the set construction adds an enormous range of possibility which ultimately leads to our 1.079363×10^{239} unique compositions. However, many of the compositions are very bland. For example, there are many compositions where the exact same notes of the chords are repeated through the entire song

without changes. Nevertheless, there are fewer of these than there are compositions where there is variability, so if you randomly choose a composition, you are unlikely to get such a bad one.

Like the arrangements, it would be possible to increase the number of compositions arbitrarily. For example, we could simply select a natural number for the length of a part, rather than having it equal the number of chords in the progression. However, we believe this would be detrimental to the aesthetic quality of Bithoven's compositions, because it is harder to detect enjoyable repetition as the song gets longer. Similarly, we believe that the current length hits a nice sweet spot because there are enough notes for a chord to get multiple notes before there's a chord change.

10. Pragmatics

This section covers a few miscellaneous pragmatic aspects of using Bithoven.

Implementation Dependencies. This project is almost entirely self-contained using only standard Racket libraries, except for two exceptions. First, as mentioned earlier, we use the `data/enumerate` library for building enumerating bijections. We co-developed this library in part to build Bithoven, however. Second, we use a *portaudio* FFI to emit raw 8-bit unsigned audio streams. We use no other libraries for higher-level audio or music concepts.

Bijections. As previously mentioned, both NEStrations and Bithoven compositions are in bijection with a prefix of the natural numbers (i.e., an enumeration.) It is vital to clarify that we do *not* have a bijection between either audio streams or tracker input and a natural prefix. In the case of compositions, we are enumerating lists of parts, so it is possible for two compositions to sound the same if the same notes are selected for each part. For instance, an ABA composition can produce the same notes as an A composition play three times, although the sheet music would indicate there are two parts played in a certain sequence.

Performance. The experience of using Bithoven is that it generates music instanteously on commodity hardware, while the audio synthesis fills the audio buffer without glitching or gaps. For a more quantitative evaluation, we constructed a simple benchmark and ran it on a 2015 Macbook Pro with a 3.1GHz Interl Core i7 processor. It takes approximately **37ms** to load the library and **30ms** to initialize it. We generated 100 compositions and rendered their audio to `/dev/null`, timing each step of the process. The average cost of each stage is as follows: each composition takes about **8ms** to decode from the enumeration and produce tracker input, NEStrations are produced in about **3ms**, and the instruments are evaluated and synthesis frames produced in **40ms**. Finally, the raw audio frames are generated in **670ms**, although in common use this does not happened batched, but incrementally as the song is played. In our main usage scenario, retro-style video games running at 60 frames-per-second, we can generate a new song every three frames, which is exceedingly reasonable.

Randomness. Although we have stressed that Bithoven is not random, of course it is most often that elements of the set are selected randomly via composition of `from-nat` and `random`. As a consequence, some compositions are extremely unlikely to be found with Bithoven. For example, as the number of parts in a musical structure increases, the likelihood of selecting a composition using that structure increases drastically: there are far more compositions with five-parts than with two-parts, holding the number of chords fixed, so it is very unlikely to randomly choose one with just two parts.

We had interesting experiences with non-random selection as well. For example, we have produced a demo video game where a single composition is chosen for a level, but its tempo and key change over the course of the level based on the actions of the player, and on the next level, we vary the index of the composition randomly by about 10%, so we tend to select a song in the same ball-park as the last level, creating a feel of cohesion across the game.

11. Related Work

The field of audio synthesis and computer generated music is vast. As well, here are many existing emulations of the Nintendo Entertainment System with cycle-perfect recreation of the exact waveforms produced by the real Ricoh RP2A03. The closest work, although it is far more involved and thorough, is Euterpea (Hudak et al. 2016) and its supporting textbook, the *Haskell School of Music* (Hudak 2015). While not aimed at recreation of any particular synthesizer, this text and library provide an extensive spectrum of computer music and audio synthesis tools from high-level music composition to instrument design and sound synthesis.

12. Conclusion

We have produced a purely functional implementation of a full-stack music synthesis system: we synthesize primitive waveforms, assemble them from instruments defined in a DSL, combine them through a tracking description language, arrange them through a projection from a large space of possible arrangements, and compose music through a rigorously defined combinatorial method. In our experience, while the music Bithoven produces is unlikely to win awards, it is plausible to most listeners as being hand-made in the era of the RP2A03.

We believe that there is a lot more that could be done to improve the quality of the set of Bithoven compositions. The most glaring problem is its restriction to only use chords. We could, for example, change it so that in each set of notes for a chord, it predominately uses the chord but is allowed to use any tone from the scale. We have yet to experiment with this or other modifications.

Acknowledgments. We are indebted to the NESdev community for their excellent documentation on the Ricoh RP2A03, which was essential to implementing the synthesis engine. We are grateful for Max New and Robby Findler's excellent work on the `data/enumerate` Racket module, which my project builds on.

References

Conal Elliott and Paul Hudak. Functional Reactive Animation. In *Proc. International Conference on Functional Programming*, 1997.

Michael Hewitt. *Music Theory for Computer Musicians.* 2008.

Paul Hudak. The Haskell School of Music. (Version 2.6), 2015.

Paul Hudak, Eric Cheng, Hai (Paul) Liu, Donya Quick, and Dan Winograd-Cort. Euterpea - A Haskell library for music creation. 2016.

Max New, Burke Fetscher, Jay McCarthy, and Robby Findler. Fair Enumeration Combinators. Unpublished, 2016.

Han Wen Nienhuys and Jan Nieuwenhuizen. LilyPond, a system for automated music engraving. In *Proc. Colloquium on Musical Informatics*, 2003.

Trevor Pinch and Frank Trocco. *Analog Days: The Invention and Impact of the Moog Synthesizer*. Harvard University Press, 2004.

Brad Taylor. 2A03 Technical Reference. 2004. `http://nesdev.com/2A03%20technical%20reference.txt`