Implementation of Digital Synthesis in Functional Programming*

Evan Sitt, Xiaotian Su, Beka Grdzelishvili, Zurab Tsinadze, Zongpu Xie
Hossameldin Abdin, Giorgi Botkoveli, Nikola Cenikj, Tringa Sylaj, and Viktória Zsók

Eötvös Loránd University, Faculty of Informatics
Department of Programming Languages and Compilers
H-1117 Budapest, Pázmány Péter sétány 1/C., Hungary
{sitt.evan, suxiaotian31, bekagrdzelishvili0, zukatsinadze, szumixie,
hossamabdeen17, botko.gio, nicola.cenic, tringasylaj}@gmail.com,
zsv@inf.elte.hu
– Project Paper –

Abstract. Digital synthesis is a cross discipline application used in fields such as music, telecommunication, and others. The nature of digital synthesis involving multiple tracks as well as parallel post-processes lends itself naturally to the functional programming paradigm. The paper demonstrates this by creating a fully functional, cross platform, standalone synthesizer application framework implemented in a pure lazy functional language. The application handles MIDI input and produces wav output played by any multimedia player. Therefore, it can serve as a preprocessor for users who intend to create digital signals before transcribing them into a digital or physical media.

Keywords: Functional Programming · Digital Synthesis · Waveforms · MIDI · wav

1 Introduction

Digital synthesis is a *Digital Signal Processing (DSP)* technique for creating musical sounds. In contrast to analog synthesizers, digital synthesis processes discrete bit data to replicate and recreate a continuous waveform. The digital signal processing techniques used are relevant in many disciplines and fields including telecommunications, biomedical engineering, seismology and others. Digital synthesis is an application typically implemented in C++ with many frameworks provided [5]; however, their algorithms and methods are less intuitive.

Our project proposes to explore the applications of functional programming and to demonstrate its features in a framework implementation that can be used in multiple disciplines.

Due to the parallel nature of processing multiple tracks of audio, the project is designed to replicate synthesis techniques by utilizing all of the features and advantages of a purely lazy functional paradigm. While some algorithms were referenced, we implemented it from scratch.

In this paper, after briefly presenting a general background of digital synthesis (section 2), the details of each project components are provided (section 3), which is followed by the summary of the results (section 4), by the conclusions (section 5), and by the future plans (section 6).

* This work was supported by the European Union, co-financed by the European Social Fund, grant. no **EFOP-3.6.3-VEKOP-16-2017-00002.**

2 Background

Digital synthesis is a field that was pioneered in the 1970s and it is still continuously innovated by the music industry. Digital synthesizers use the power of microprocessors to replicate analog synthesis. Among the techniques used are additive synthesis, wavetable lookup synthesis, and physical modeling.

Additive synthesis is a technique for creating waveforms via the summation of sine waves. A sine wave is a waveform of pure single-frequency value. By summing multiple sine waves at various frequencies, amplitudes, and phase shifts, it is theoretically possible to generate all types of sound waves.

Our application utilizes harmonic additive synthesis to create the basic waveforms commonly used to generate more complex synths. Harmonic additive synthesis involves using the Fourier series of a waveform to determine the weighted summation of sine waves in order to generate the target waveform. These sine waves are called *harmonics*, so-called because their frequencies are integer multiples of a standard fundamental frequency.

For example, in order to generate a sawtooth waveform, we use the Fourier series to determine the harmonics and their amplitudes for summation. In the case of the sawtooth wave, the k-th harmonic of the fundamental frequency f is $2 * k * f$ with amplitude $(-1^k)/k$.

After generating the sawtooth waveform, this waveform is used with subtractive synthesis, essentially additive synthesis with negative polarity, to subtract a phase-shifted sawtooth waveform to generate a pulse wave.

In order to generate the waveforms efficiently, digital waveform synthesis is typically implemented using wavetable lookup synthesis. In contrast to calculating a specific value of a waveform at a specific point of time, a waveform table is used to store one duty cycle of a waveform. The value of the waveform can be accessed by using the frequency to modify the access point of the waveform table, and then multiplying by the appropriate amplitude. With this method, it is far more efficient to generate a waveform by use of constant time array access instead of repeated calculations. In the following the details of each waveforms are given.

3 Project details

3.1 Wavetable Lookup Synthesis

In implementing the digital synthesis, we utilize a technique called *Wavetable Lookup Synthesis*, in which a certain waveform is stored in a wavetable, and it exploits the relation between frequency and sampling rate to quickly build new waveforms. Based on the methods for designing wavetables of [6], our implementation chooses to set the size of the table as 2205, i.e., we store a table of 2205 real numbers, representing consecutive amplitudes within one single vibration of the sound wave. Thus, achieving the minimum sound intensity that humans can hear, 20 Hz.

The single cycle sine wavetable, shown in figure 1, is the basis for our additive and subtractive synthesis, as the sine wave is the simplest of all waveforms and it contains only a single frequency and no harmonics. All the other waveforms can be efficiently generated from sine by utilizing Fourier series of sine functions [11].

An example of sawtooth can be seen in figure 2. The sawtooth waveform is then further used with subtractive synthesis to generate the pulse waveform shown in figure 5. For each wave form we generate list of indexes, which we need to sample from the wavetable, using the function `getIndexes`. This indexes depend on frequency and harmonic, they and are not necessarily integers.

The `getValues` function (listing 1.1) takes wavetable, frequency, harmonic and duration as parameters and it uses generated indexes, while linear interpolation solves the complication caused by real indexes.

```
getValues :: {Real} Frequency Int Samples → [Real]
getValues waveTable frequency harmonic dur = [(getValue i waveTable) \\ i ← indexes]
where
    indexes = getIndexes frequency harmonic dur
```

Listing 1.1: getValues

The wavetable is implemented as an array. Despite the fact that lists offer much more functionality, they are actually linked-lists, and they not give us access to the elements in constant time.

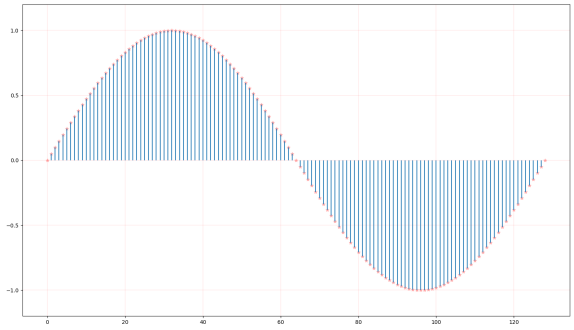

Fig. 1: Sine wavetable

Wave Forms

Four type of waveforms are the basis of any sound: sine wave, square wave, triangle wave, and sawtooth wave. Besides these four, the project also includes parameters to generate pulse, silence, and noise waves. In the implementation, a waveform type is represented as an algebraic data structure:

```
:: Wave = Sine | Square | Triangle | Noise | Pulse | Sawtooth | Silence
```

which is a parameter of our interface function, that generates wave as a list of `Real` numbers. Each waveform has a list of harmonics and a list of amplitudes. In case of square, triangle, sawtooth and silence these lists are easily defined, while for pulse and noise, there is a need for more sophisticated techniques.

Sawtooth

Frequency components are all harmonics, relative amplitudes are inverses of harmonic numbers and all harmonics are in phase (see Figure 2).

Square

Frequency components are odd numbered harmonics, relative amplitudes are inverses of squares of harmonic numbers and all harmonics are in phase. (see Figure 3).

Triangle

Frequency components are odd numbered harmonics, relative amplitudes are inverse harmonic numbers and every second harmonic is 180 degrees out of phase. (see Figure 4).

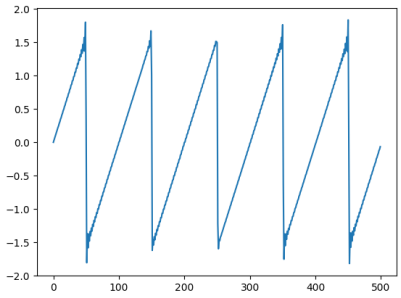

Fig. 2: Sawtooth waveform

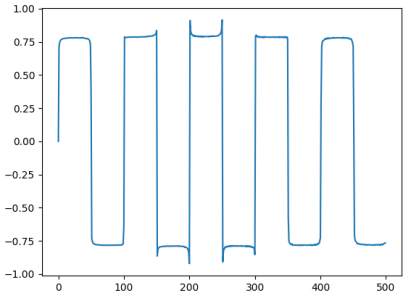

Fig. 3: Square waveform

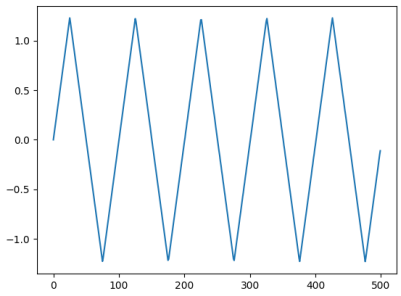

Fig. 4: Triangle waveform

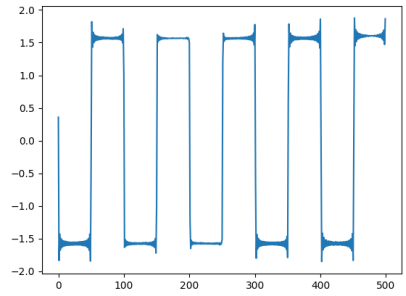

Fig. 5: Pulse waveform

Pulse

For the Pulse wave generation, Figure 5, Sawtooth wave and phase shifted version of itself are subtracted. For this, efficient helper function, `shiftLeft`, is defined. It moves every element of a list by given number to the left. (see Figure 5.)

Noise

For generating Noise wave, Figure 6, all amplitudes are 1 and harmonics are random numbers. Again using `shiftLeft` function, lists are shifted by random number of places before summing them up. Clean provides functions to generate pseudo random numbers using Mersenne Twister Algorithm [12] in the module `Math.Random`.

3.2 Envelope

In music, an envelope describes the varying level of a sound wave over time. It is the envelope of a wave which establishes the sound's uniqueness and has a significant influence on how we interpret music. Classic envelopes consist of 4 main parts: Attack, Decay, Sustain, and Release, where sustain refers to a level, while others represent time interval. Attack is the period of time for which sound needs to reach its peak from zero, after the key is pressed. After the attack, the decaying period starts, when sound level decreases to sustain level, and stays unchanged, during sustain phase, until the key is released. Final phase of the envelope is

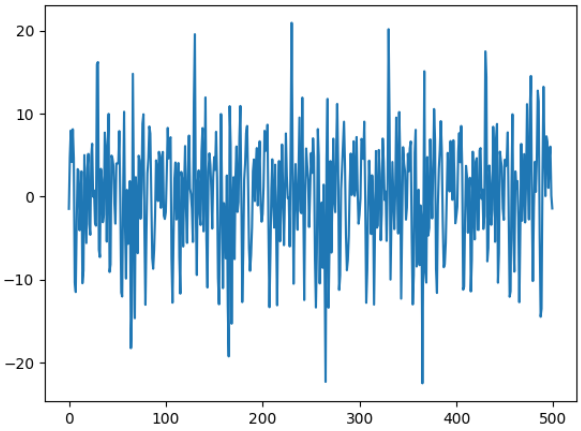

Fig. 6: Noise waveform

the release, which continues until sound fades to silence. Almost every musical instrument has it's own individual envelope. For example, a quick attack with little decay makes sound similar to an organ, while a longer decay is characteristic of a guitar. This application includes an envelope generator, which is a common feature of synthesizers and electronic musical instruments, used to control the different stages of sound.

Implementation of Envelopes using lists The purpose of the envelope generator function is to calculate values for the envelope according to the given parameters. These functions take note data and corresponding values for each step of the envelope as arguments and generate a list of `Real` numbers, which take values from -1.0 to 1.0.

This implementation makes use of list comprehensions and lazy evaluation in calculations, Clean's two main advantages.

For each type of envelope a corresponding `Record` structure was implemented for easily manipulating envelope data. Each record contains information about the duration, level, and/or rate of each step according to which final list is created.

ADSR Envelope The `getADSR` function is used to generate an ADSR Envelope, figure 7. It has only basic 4 steps: Attack, Decay, Sustain, and Release. This function gets a beat, time signature, tempo, and ADSR record as parameters. At first the beat, time signature and tempo are used to calculate the duration of the note, time interval between pressing and releasing the key. `noteToSamples`, one of the utility functions, is used to convert these parameters to the number of samples in this time interval. After that, the number of samples for each step of the envelope is calculated. As the release is independent from the note duration it is enough to directly convert given release duration to samples, but the other 3 steps need different approach. Instead of directly using given duration of each step independently, the number of samples is calculated based on time offset from the starting time and subtracting sum of samples of the previous steps (As sustain does not have a fixed time interval, the total note duration can be used as offset). This is important to avoid losing samples during flooring of real numbers and to make sure that the number of total samples is equal to sum of each

step's samples. After calculating the list of samples, each step of the envelope is calculated independently using list comprehension. For the linear attack and decay, the value of sample is instantly calculated with index, number of samples and final value. Concatenating these lists produces the entire envelope excluding the release tail, however as the key may be released any time during the first 3 steps, including attack or decay, it might be necessary to shorten it. The clean built in function `take` is used for extracting the exact amount of samples needed. Finally the release tail list is generated in the same way as attack and decay and is concatenated to the others to get complete envelope.

DAHDSR Envelope The `getDAHDSR` function generates another type of envelope, which has two more steps than ADSR envelope: delay and hold. Delay is the time interval before attack, when sound stays silent, while hold phase comes after attack and indicates duration while sound maintains its peak. Functions implementation is similar to `getADSR` function and data is stored as a `::DAHDSR` record, shown bellow (listing 1.2). Each step is generated using list comprehensions and concatenated. Whole envelope is generated and only after that its prefix is taken to make sure that key can be released at any time.

```
:: DAHDSR = { delay :: Real
            , attack :: Real
            , hold :: Real
            , decay :: Real
            , sustain :: Real
            , release :: Real
            }
```

Listing 1.2: DAHDSR envelope record

Casio, 8 step Envelope Casio, figure 8, is a more modern type of envelope which allows more flexibility and vast variety. It is different from above mentioned types, like ADSR envelope (Figure 7), For each step, instead of providing duration and level, it has 8 steps, described by rate and level values, where level is the desired percentage to be reached at the end of the current phase, while rate denotes the percentage with which samples change per second. Rate and level pair make possible for same phase to be ascending or descending depending on the needs of users. Implementation of Casio envelope differs for other two envelopes, as the structure is different. `CasioCZ` record provides data, necessary for creating Casio envelope, it has rate and level values for each 8 step, first 5 step represent front part of the envelope, while last 3 steps are used to generate release tail after sustain. `generateLine` function is used to generate point values for line between two level using current rate. This function returns not list of points, but tuple of list and real value. Second return value plays important role in interpolation. Last value of line, may not have integer index, hence it can not be included in the list. Due to the above mentioned reason, instead of directly using previous endpoint as the beginning for the current line we need to recalculate it based on the second value of `generateLine` function using the formula: $\mathrm{casio.level1} - \mathrm{rt2} * (\mathrm{snd\ line1})$. At the end, similarly to other envelopes, we need to take exact amount of samples according to note duration.

Generalized Envelope Last type of envelope data structure is Generalized Envelope, which is similar to Casio, but provides even more flexibility during sound synthesizing. Both of them use rate and level values to describe each step, but generalized envelopes do not have fixed number of steps like other previous structures.

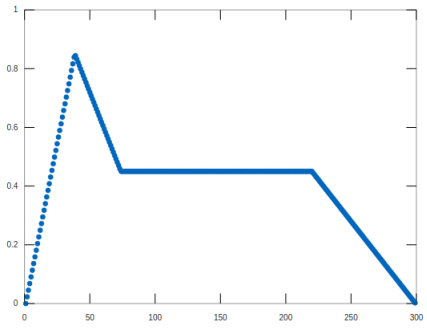

Fig. 7: ADSR envelope

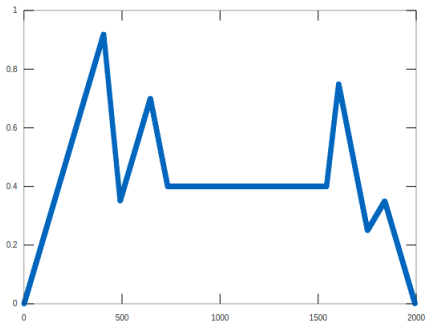

Fig. 8: 8-Step Casio envelope

`GenEnv` record uses list to store data, where each element is `EnvLevel` record type, containing rate and level values. Also, as generalized envelopes do not have fixed number of steps before release tail, `GenEnv` record contains value for index indicating sustain level. Generating data for each step is done similarly to Casio envelope, but rate and starting value can not be recalculated manually, so preprocessing data is needed before using it, therefore implementation, which is shown bellow (listing 1.3), is a bit different. `parseData` recursievely traverses inital list and generates new one, which can be directly used to generate lines for each step using similar way as in Casio envelope.

```
getEnvelope :: Real GenEnv → [Real]
getEnvelope duration envelope = envShortened ++ envRelease
where
    noteSamples = secondsToSamples duration
    sustL = toReal (hd [x.level \\ x ← envelope.levels &
                        d ← [1,2..(length envelope.levels)] | ind == envelope.sustainLevel])
    envIntroData = parseData (take envelope.sustainLevel envelope.levels) 0.0 0.0
    envIntro = [0.0] ++ (flatten [generateLine data \\ data ← envIntroData])
    envSustain = [sustL \\ x ← [1,2..(noteSamples-(length envIntro))]]
    envShortened = take noteSamples (envIntro ++ envSustain)
    envReleaseData = parseData (drop envelope.sustainLevel envelope.levels)
                               (last envShortened) 0.0
    envRelease = flatten [generateLine data \\ data ← envReleaseData]
```

Listing 1.3: Generalized envelope generating function

Efficiency improvements Unlike other steps release does not have fixed starting value and it needs to be calculated based on the time of releasing key. First implementation of envelope generator function used `last`, built in functions for list, to calculate base value of release. As mentioned before, lists are implemented as linked lists and do not have direct access to the element, `last` function uses recursive approach which results in $O(N)$ time complexity and excessive use of memory. To avoid overfilling heap, release base value is calculated directly using constant time and memory. At first it is determined on which step generation was terminated and then value can be calculated directly according to it.

Implementation of Envelopes with Arrays Direct access to the elements might be essential to avoid linear time complexity and provide better memory management, therefore envelope generator functions were also implemented with arrays instead of lists. These implementations provide direct access to data and support array implementations of waves, which can be handful, for example, during summing up different waves, while this can not be done with lists, at first they need to be converted into arrays, then processed and reconverted to the lists again.

Rendering waves and applying envelope Rendering process consists of several steps. First step is to calculate whole length of the sound, as each wave can start at different moment of time and can have distinct lengths. This value will be used later, to generate silent track, which will act as the base during summing up all wave samples. Next step is to process data stored in each chunk to generate sound waves and sum up all of them. Each chunk stores wave type, time signature, tempo, envelope and other data, extracted from MIDI files, which are needed to generate wave and apply envelope to it, Figure 9. Values for each wave can be calculated using already developed functions for envelopes and sound synthesizing. After generating all waves we need to sum up them in the single list. If we use arrays we can use each wave's starting time as an index offset, but same approach is not useful with list implementation. To easily sum up lists they need to be same size, therefore appropriate amount of silence samples should be appended on the both sides of the list. Last step is normalization: converting values to $[-1.0, 1.0]$ range. After summing up lists some samples might go out of those bounds, that's why final list needs to be normalized at the end of the process. After normalization sound rendering is finished and it can be used for later processing.

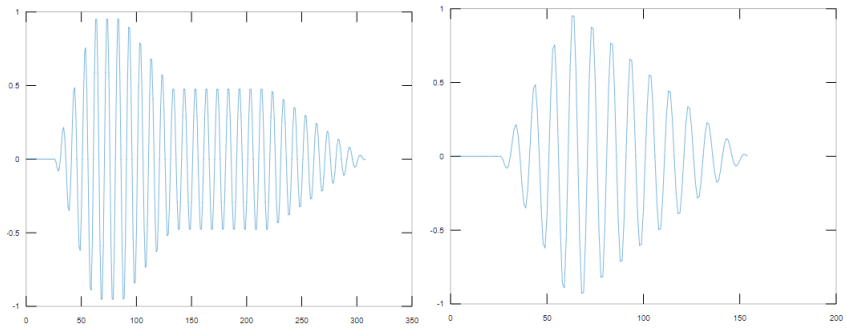

Fig. 9: Sine wave modified by different envelopes

Four data structures were created to support different types of envelopes: ADSR, DAHDSR, Casio and Generalized envelopes. A demonstration of DAHDSR followed by ADSR being applied to a sine wave are shown in fig 9. Several implementations and types of envelopes provide flexible environment during music generator development and sound synthesizing. Giving possibility to implement more efficient approaches during rendering process and creating envelopes to generate more complex and better sounds.

3.3 MIDI Input

MIDI is short for Musical Instrument Digital Interface which related audio devices for playing, editing and recording music. The MIDI file is just a stream of numbers, each of which is in the range from 0 to 255. The

bytes order is big-endian. MIDI files consist of chunks. There are two types of chunks. [7]

structure / **type**	type (4 bytes)	length (4 bytes)	data (variable length of bytes)
Header Chunk	MThd	6	<format><tracks><division>
Track Chunk	MTrk	<length>	<delta_time><events>...

Table 1: Three types of MIDI Events
length : length in bytes of the chunk data part

Information in MIDI file (as in [9])
<format> :

- format 0 : a header chunk + a single track chunk.
  The single track chunk contains all the note and tempo information.
- format 1 : a header chunk + one or more track chunks.
  All tracks being played simultaneously.
- format 2 : a header chunk + one or more track chunks.
  Each track represents an independent sequence.

<tracks> : The number of track chunks contained in MIDI file.
<delta_time> :

- a time value which specifies the duration between two events
- not optional
- 0 is a valid delta time

<events> : The default unit of delta-time for this MIDI file.

- midi events : any MIDI Channel message.
  - Channel Voice messages
  - Channel Mode messages
- sysex(system exclusive) events : include messages other than MIDI Channel.
- meta events : used for things like track-names, lyrics and cue-points, etc.

Challenges Unlike regular audio files like MP3 or WAV files, MIDI files don't contain actual audio data and are therefore much smaller in size. Due to this, they are more compact but this makes it more difficult to parse it since there are a lot of information to extract and store.

Delta time is represented by a time value which is a measurement of the time to wait before playing the next message in the stream of MIDI file data. Time values are stored as Variable Length Values (VLV:a number with a variable width) [8]. Each byte of delta time is consist of two parts: 1 continuation bit and 7 data bits. The highest-order bit is set to 1 if it needs to read the next byte, set to 0 if this byte is the last one in VLV.

Steps: To get an integer number represented by a VLV

1. convert the first byte in VLV to integer
   - if it is greater than 128, put it into a list and read next byte recursively
   - if not, just put this byte into a list and end the recursion
2. convert the list of bytes into one integer number
3. return not only an integer of delta time but also the length of byte of it

There are three main different kinds of events that can occur in track chunk, each type has different number of bytes to store the information. We are not able to know the length of each specific event until we reach its status byte which stores the information indicating what type it is.

structure / **event type**	status byte	byte2	byte3	byte4
midi events	0x8n - 0xEn	data	(data)	–
sysex events	0xF0 and 0xF7	length	data	–
meta events	0xFF	type	length	data

Table 2: Three types of MIDI Events

Running Status: Different types of events already make it hard to parse information, in addition to that, the midi events use a data-thinning technique which is running status. If the first (status) byte is less than 128 (hex 80) which implies that running status is in effect, and that this byte is actually the first data byte (the status carrying over from the previous MIDI event). This can only be the case if the immediately previous event is also a MIDI event, because system exclusive events and meta events interrupt (clear) running status.

Solutions The length of track chunk is useful since it is not only help for processing normal chunks but also make it easier to deal with unexpected chunk types – just by skipping that amount of byte then we can continue to process next chunk.

event type	status byte		byte2	byte3
Note On	0x8	Channel	Note Number(frequency)	velocity
Note Off	0x9	Channel	Note Number(frequency)	velocity

Table 3: Events we handle so far

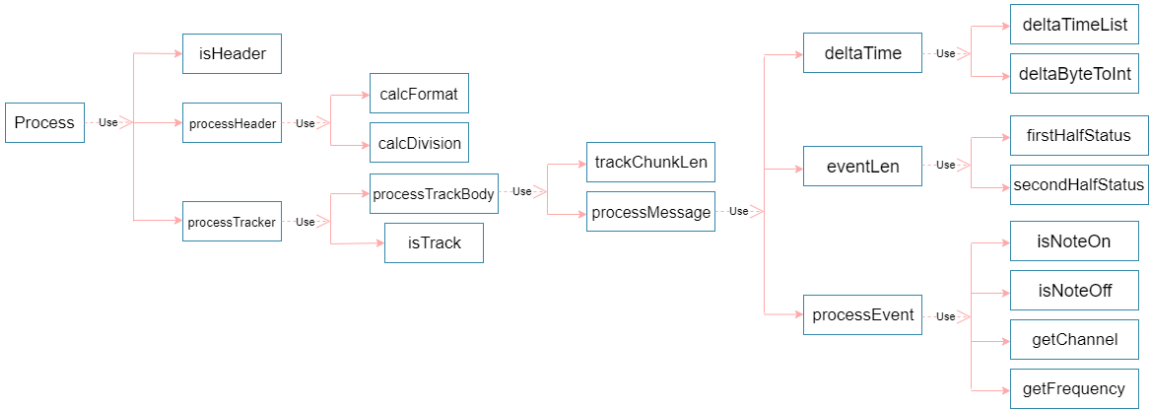

Fig. 10: functions

Function `process`, (listing 1.4), parses a MIDI file from scratch, accepting a list of Char (i.e. bytes) and returning an `Info` record which contains information about header and track chunks. `isHeader` takes the first four elements of a list of bytes and sees if it is type of header chunk which is `MThd`.
The first six bytes of the list give information about format, number of track chunks in the file, and division.
`procesHeader`, (listing 1.5), stores the first and third value in the `HeadInfo` record.
`processTrack`, (listing 1.6), uses `isTrack` function to see if currently beginning of a track chunk is being processed, and if yes, it drops first four elements which contain the information of chunk type and continues processing.

```
process :: [Char] → Info
process l
  |length l > 14 && isHeader (take 4 l) =
    {
      headerInfo = processHeader (drop 8 l),
      trackInfo = processTrack (drop 14 l)
    }
  = abort "not enough information"
```

Listing 1.4: process function

```
processHeader :: [Char] → HeaderInfo
processHeader l =
  {
    format = calcFormat (take 2 l),
    division = calcDivision (take 2(drop 4 l))
  }
```

Listing 1.5: processHeader function

```
processTrack :: [Char] → [TrackInfo]
processTrack [] = []
processTrack l
  //4bytes:type of chunk(mtrk)
  |isTrack l = processTrackBody (drop 4 l)
  = processTrackBody l
```

Listing 1.6: processTrack function

3.4 Transcoding

After the synthesis part, the signal needs to be converted into a form that can be recorded onto physical or digital media. This process is also known as transcoding. In days of analogous signal synthesis, recording equipment transcoded the electrical signals using various mechanical or electromagnetic methods. With digital synthesis, the applications have to transcode the digital waveforms into bits in order to store them into the appropriate file.

Similarly, in this implementation, once the program obtains the wavetable examined in Section 3.1,the next step is to write this sound data to a WAV file. As discussed in Section 3.5, three main components separate the WAV file: the `RIFF` chunk, the `fmt` sub-chunk and the `data` sub-chunk. The `data` sub-chunk contains the sound information, which is stored in bits. In consequence to that, it was necessary to find a way to convert the result of the wavetable into appropriate data for the file, hence there are transform functions implemented.

Initially only the `transform8` function was created which takes the wavetable and its maximum value and converts the values to fit the 8 bits range. In other words,the wavetable values are converted into an interval from 0 to 255. Later on, as a precondition for increasing the quality of the generated sounds, the function `transform16` was added, which alters the values to 16 bits samples stored into the interval 0 to $2^{16} - 1$, and the function `transform32`, which alters the values to 32 bits samples stored into the interval 0 to $2^{32} - 1$.

In order to make the project more flexible with the number of channels received input, two versions were made for the transform functions. In the default case the sound data obtained as the input will represent only one channel, meaning that the wavetable could be correctly represented as a list of Reals. On the other hand, in case the data has two or more channels then a better representation would be a list of lists of Real. For that reason, in addition to `transform8`, `transform16` and `transform32`, three more functions, called `transform8_multiChannel`, `transform16_multiChannel` and `transform32_multiChannel` respectively, were created to work with lists of lists where the new functions map their previous respective functions to each sub-list.

After doing some research regarding the opportunities CLEAN offers and discussing about the best approach possible the group made a decision to use the concept of vertical graph shifting and multiplying. The simplest vertical graph transformation involves adding a positive or negative constant to a function. In other words, adding the same constant k to the output value of the function regardless of the input shifts the graph of the function vertically by k units. In order to give a more detailed explanation of the implementation, it is a good idea to handle the 8, 16, and 32 bits cases separately.

Regarding the 8 bit case, the first step is dividing each number from the list with the maximal possible value the input values can reach. In the case of real numbers from $[-0.5, 0.5]$ this value is 0.5. The next step is adding the $\max(0.5)$ and then multiplying each element by 255 ($2^8 - 1$) in order to get real numbers in $[0, 255]$ interval. After that the function `toInt` (build in function in Clean) handles the conversion of the data from real to integer. As a last step `toChar` (build in function in Clean) converts the list of integers into list

of characters which are represented in 8 bits. If the input is list of lists, mapping the same transformation to each and every sub list of the input does the proper conversion.

Moving on,in the 16 bit case the first thing done is applying a function which converts the input from list of real numbers to integers.

```
transform16 :: [Real] Real → [Char]
transform16 list max=flatten (map (λx = intToBytesLE 2 x) (map (λx = aux16 x max) list))
```

Listing 1.7: transform16

In this case (Listing 1.7) instead of using `toInt` it was more appropriate to created a function which takes two real numbers (the number that to convert (lets name it x) and the maximal possible value the numbers in the list can reach (lets name it max)) and returns $2^{15} - 1$ if the number equals to max or otherwise the lower integer part of x multiplied by 2^{15} and divided with max. After getting the list of integers mapping a function which converts each integer into a list of bits (1's and 0's) and then concatenating all of sub lists concludes the transformation. If the input is list of lists, mapping the same transformation to each and every sub list of the input gives the expected output.

Similarly, on the 32 bit,the first step is mapping a function which converts the input from list of real numbers to integers but now instead of working with 2^{15} the function mapped works with 2^{31} but the concept is exactly the same. The following step is converting each integers into a list of bits (1's and 0's) of length 32 and in the end just concatenating all of the 32-length lists into one. If the input given is a list of lists, handling it is done correspondingly to the 16 bit conversion.

3.5 .wav Output

Description of WAV File

The WAV file is an instance of a *Resource Interchange File Format* (RIFF) defined by IBM and Microsoft. The RIFF format acts as a `wrapper` for various audio coding formats.

Though a WAV file can contain compressed audio, the most common WAV audio format is uncompressed audio in the linear pulse code modulation (LPCM) format. LPCM is also the standard audio coding format for audio CDs, which store two-channel LPCM audio sampled at 44,100 Hz with 16 bits per sample. Since LPCM is uncompressed and retains all of the samples of an audio track, professional users or audio experts may use the WAV format with LPCM audio for maximum audio quality. WAV files can also be edited and manipulated with relative ease.

The WAV format supports compressed audio using, on Microsoft Windows, the Audio Compression Manager. Any ACM codec can be used to compress a WAV file.

Beginning with Windows 2000, a *WAVE FORMAT EXTENSIBLE* header was defined which specifies multiple audio channel data along with speaker positions, eliminates ambiguity regarding sample types and container sizes in the standard WAV format and supports defining custom extensions to the format chunk.

There are some inconsistencies in the WAV format: for example, 8-bit data is unsigned while 16-bit data is signed, and many chunks have duplicate information found in other chunks.

Specification of File Structure

A RIFF file is a tagged file format. It has a specific container format (a chunk) that includes a four character tag (FourCC) and the size (number of bytes) of the chunk. The tag specifies how the data within

the chunk should be interpreted, and there are several standard FourCC tags. Tags consisting of all capital letters are reserved tags. The outermost chunk of a RIFF file has a RIFF form tag; the first four bytes of chunk data are a FourCC that specify the form type and are followed by a sequence of subchunks. In the case of a WAV file, those four bytes are the FourCC WAVE. The remainder of the RIFF data is a sequence of chunks describing the audio information.

The advantage of a tagged file format is that the format can be extended later without confusing existing file readers. The rule for a RIFF (or WAV) reader is that it should ignore any irrelevant tagged chunk. The reader won't be able to use the new information, but the reader should treat it as valid input and ignore it.

The specification for RIFF files includes the definition of an INFO chunk. The chunk may include information such as the title of the work, the author, the creation date, and copyright information. Although the INFO chunk was defined in version 1.0, the chunk was missing from the formal specification of a WAV file. If the chunk were present in the file, then a reader should know how to interpret it, but many readers had trouble. Some readers would abort when they encountered the chunk, some readers would process the chunk if it were the first chunk in the RIFF form, and other readers would process it if it followed all of the expected waveform data. Consequently, from an interchange standpoint, it was safest to omit the INFO chunk and other extensions and send a lowest-common-denominator file. There are other INFO chunk placement problems.

RIFF files were expected to be used in international environments, so there is CSET chunk to specify the country code, language, dialect, and code page for the strings in a RIFF file. For example, specifying an appropriate CSET chunk should allow the strings in an INFO chunk (and other chunks throughout the RIFF file) to be interpreted as Cyrillic or Japanese characters.

RIFF also defines a JUNK chunk whose contents are uninteresting. The chunk allows a chunk to be deleted by just changing its FourCC. The chunk could also be used to reserve some space for future edits so the file could be modified without being rewritten. A later definition of RIFF introduced a similar PAD chunk

The top-level definition of a WAV file is:

```
<WAVE-form> → RIFF('WAVE'
                    <fmt-ck>             // Format
                    [<fact-ck>]          // Fact chunk
                    [<cue-ck>]           // Cue points
                    [<playlist-ck>]      // Playlist
                    [<assoc-data-list>]  // Associated data list
                    <wave-data> )        // Wave data
```

Listing 1.8: RIFF Header

The definition shows a top-level RIFF form with the WAVE tag. It is followed by a mandatory `<fmt-ck>` format chunk that describes the format of the sample data that follows. The format chunk includes information such as the sample encoding, number of bits per channel, the number of channels, the sample rate. The WAV specification includes some optional features. The optional fact chunk reports the number of samples for some compressed coding schemes. The cue point (`cue`) chunk identifies some significant sample numbers in the wave file. The playlist chunk allows the samples to be played out of order or repeated rather than just from beginning to end. The associated data list allows labels and notes (`lii` and `note`) to be attached to cue points; text annotation (`ltxt`) may be given for a group of samples (e.g. caption information). Finally, the mandatory wave data chunk contains the actual samples (in the specified format).

The WAV file definition does not show where an INFO chunk should be placed. It is also silent about the placement of a CSET chunk (which specifies the character set used). In the application, the PCM format generated omits this chunk because the omission has no effect on the functionality.

The RIFF specification lacks the formality in the specification, but its formalism lacks the precision seen in other tagged formats. For example, the RIFF specification does not clearly distinguish between a set of subchunks and an ordered sequence of subchunks. The RIFF form chunk suggests it should be a sequence container. The specification suggests a LIST chunk is also a sequence: `A LIST chunk contains a list, or ordered sequence, of subchunks.` However, the specification does not give a formal specification of the INFO chunk; an example INFO LIST chunk ignores the chunk sequence implied in the INFO description. The LIST chunk definition for `<wave-data>` does use the LIST chunk as a sequence container with good formal semantics.

The WAV specification allows for not only a single, contiguous, array of audio samples, but also discrete blocks of samples and silence that are played in order. Most WAV files use a single array of data. The specification for the sample data is confusing:

The `<wave-data>` contains the waveform data. It is defined as follows:

```
<wave-data>  → { <data-ck> | <data-list> }
<data-ck>    → data( <wave-data> )
<wave-list>  → LIST( 'wavl' { <data-ck> | // Wave samples
                              <silence-ck> }... ) // Silence
<silence-ck> → slnt( <dwSamples:DWORD> ) // Count of silent samples
```

Listing 1.9: Wave data

In the wave data chuck (Listing 1.9) produced by the application, the implementation changes `<data-list>` to `<wave-list>` in line 1 and `<wave-data>` to `<bSampleData:BYTE>` in line 2. These changes are done in order to avoid any possible recursion of `<wave-data>` contained in a `<data-ck>`.

WAV files can contain embedded IFF `lists`, which can contain several `sub-chunks`.

File Format Limitation

The WAV format is limited to files that are less than 4 GB, because of its use of a 32-bit unsigned integer to record the file size header. Although this is equivalent to about 6.8 hours of CD-quality audio (44.1 kHz, 16-bit stereo), it is sometimes necessary to exceed this limit, especially when greater sampling rates, bit resolutions or channel count are required. The W64 format was therefore created for use in Sound Forge.

Its 64-bit header allows for much longer recording times. The RF64 format specified by the European Broadcasting Union has also been created to solve this problem.

Based on the file specification of the WAV file format, a set of functions were implemented to create a framework for writing the data into a file. These functions have been enumerated in detail below 3.5.

File manipulation in Clean

Due to Clean being a purely functional language, side effects such as I/O operations are performed through uniqueness typing to preserve referential transparency [1]. Types which are marked as unique with $*$ cannot be used multiple times for any path of a function execution. It guarantees that at the time a function with unique arguments is executed, the arguments have no more than one reference each pointed to them.

`World` is an abstract type representing the environment, or the state of the world [2]. A program doing I/O is given an environment and produces a new environment containing the changes. A more complicated

I/O program thus need chains the subprograms together by passing the environment from one function to another.

```
echo :: *World → *World
echo world0
  | c == '\n' = world2
              = echo world2
  where
  (c, world1) = getChar world0
  world2      = putChar c world1
```

Listing 1.10: An example of passing the environment explicitly

The Clean `StdEnv` supports basic file manipulation in the `StdFile` module. It provides operations for the `File` type, which can also be a unique type. Opening a file requires an argument for a mode, which distinguishes read, write, and append, and between text files and binary data files. We will be using `FWriteData` mode for writing binary data.

There are several operations for writing data, though most of them are not easy to work with for binary data. The smallest unit we can write is a byte, however the byte is represented as a Clean `Char`. We will assume a `Char` in Clean is a byte, we can denote it with a type synonym as in Listing 1.11.

```
:: Byte :== Char
```

Listing 1.11: Type synonym in Clean

There is a function for writing a string (which are unboxed `Char` arrays in Clean) to a file, however lists are easier to work with most of the time, so we will define a function to write a list of `Char`s to a file as in Listing 1.12.

```
writeBytes :: ![Byte] !*File → *File
writeBytes []     f = f
writeBytes [b:bs] f
  #! f = fwritec b f
  = writeBytes bs f
```

Listing 1.12: Writing a list of bytes into a file

`!` in the type specifies that the arguments of the function are strict, this can improve program efficiency in places where laziness is not needed. `#!` is a strict let notation, assigning the output of `fwritec b f` to `f`. This `f` is not the same variable as the `f` in the line before. In fact, it introduces a new scope and shadows the previous variable. This is encouraged in Clean with unique types, it makes the program somewhat resemble imperative programs besides the explicit passing of the unique file.

We will also define a function to convert a nonnegative integer to a list of bytes in little-endian order for later use (Listing 1.13). It will take and argument to specify in how many bytes the number should be represented, e.g. if the argument is 2, then the output will represent a 16-bit word, the rest of the number is truncated. The function uses simple recursion and basic operators from `StdEnv`.

```
// The first parameter is the number of bytes
// The second parameter is the integer to be converted
uintToBytesLE :: !Int !Int → [Byte]
uintToBytesLE i n
  | i ≤ 0 = []
          = [toChar (n bitand 255) : uintToBytesLE (i - 1) (n >> 8)]
```

Listing 1.13: Converting an integer to a list of bytes

Interface

We will implement writing to a Wave file in (L)PCM format due to its simplicity.

The type of the function for writing a Wave file is given in a `dcl` file (definition module). It takes some parameters that specifies the structure of the file, and a list of bytes as the binary data in the data chunk.

```
:: PcmWavParams =
  { numChannels    :: !Int // Number of channels
  , numBlocks      :: !Int // Number of samples (for each channel)
  , samplingRate   :: !Int // Sampling rate in Hz (samples per second)
  , bytesPerSample :: !Int // Number of bytes in each sample
  }

writePcmWav :: !PcmWavParams ![Byte] !*File → *File
```

Listing 1.14: Interface of writing a Wave file in PCM format

All data that the Wave file needs can be calculated from these parameters. `numBlocks` represents the total number of blocks in the data chunk, where each block contains `numChannels` samples. `bytesPerSample` is how many bytes each sample contains.

Implementation

The main function is composed of three smaller functions. The first one writes the RIFF header into the file as in Listing 1.15.

```
writeHeader :: !Int !*File → *File
writeHeader l f
  #! f = fwrites "RIFF" f
  #! f = writeUint 4 l f
  #! f = fwrites "WAVE" f
  = f
```

Listing 1.15: Writing RIFF header into a file

The first argument is the length of the whole file minus the first eight bytes. It will be calculated in the main function with 4 (the bytes `WAVE`) + 24 (size of the format chuk) + 8 (header of the data chunk) + `bytesPerSample` × `numChannels` × `numBlocks` (size of the binary data) + (1 or 0 depending on whether the size of the bynary data is odd or even).

`writeUint` is a utility function that combines `writeBytes` and `uintToBytesLE`.

The second function writes the format chunk, which contains writing in the following data [10], using the same method as the previous function:

- The bytes `fmt`
- 16 as a 32-bit number: the size of the format chunk after the first eight bytes
- 1 as a 16-bit number: this specifies the audio format to be PCM
- `numChannels` as a 16-bit number
- `samplingRate` as a 32-bit number
- `samplingRate` × `bytesPerSample` × `numChannels` as a 32-bit number: the amount of bytes processed per second
- `bytesPerSample` × `numChannels` as a 16-bit number: the amount of bytes each block contains
- 8 × `bytesPerSample` as a 16-bit number: bits per sample

The last function takes in the length and the list of the binary data, and writes it into the file using `writeBytes` after writing in the chunk header. It also takes care of adding a padding byte if the size of the data in bytes is odd.

The main function composes the smaller functions and evaluates the size of the binary data so that it does not need to be calculated more than once in the sub-functions (Listing 1.16).

```
writePcmWav :: !PcmWavParams ![Byte] !*File → *File
writePcmWav p d f
  #! l = p.bytesPerSample * p.numChannels * p.numBlocks
  #! f = writeHeader (l + if (isEven l) 36 37) f
  #! f = writeFormat p f
  #! f = writeData l d f
  = f
```

Listing 1.16: The main function for writing Wave files

4 Results

In the initial test runs of the application, a notation file of Beethoven's Für Elise is used as input. Für Elise was chosen as an initial test input as the notation involved only a single instrument and the melodic and harmonic lines contained only monophonic lines. The initial test render of Für Elise took a total amount of time ranging between 900 - 1000 seconds to complete. Further iteration on the application, in which the wavetable implementation was changed from lists to arrays, resulted in a subsequent rendering time of 4-6 seconds.

5 Conclusion

The digital synthesizer application successfully demonstrated another possible application of functional programming. There were some challenges in the process. These included creating the framework for writing to wav, the conversion of data to bit format, and accomodating the variety of specifications and conventions within the MIDI and WAV file formats.

The team was successful in implementing full featured frameworks for importing MIDI files, writing to WAV files, and creating synths via additive synthesis, subtractive synthesis, and envelopes.

6 Further Work

The application can be easily extended in the future with further functionality to become more competitive with current offerings within the digital synthesis ecosystem. Support can added for more import file types such as MusicXML and export file types such as .mp3, .flac, and .ogg.
Additionaly functionality can be added with filters based on frequency such as passes, shelves, and EQ, effects based on amplitude such as compression, gate, and distortion, and effects based on time such as delay, reverb, and chorus.
Lastly, adding support for live MIDI input, sample banks, VST3 support, and a graphical user interface will further bring the application in line with other digital synthesizers.

References

1. Clean Language Report, https://clean.cs.ru.nl/download/doc/CleanLangRep.2.2.pdf.
2. Achten, P., Plasmeijer, R.: The Ins and Outs of Clean I/O. *Journal of Functional Programming*, 5(1), 81-110, 1995.
3. Hudak, P., Quick, D.: Haskell School of Music – From Signals to Symphonies, *Cambridge University Press*, 2018.
4. Thompson, S.: The Haskell: The Craft of Functional Programming, *Addison-Wesley Professional*, 3rd edition, 2011.
5. Szanto, G.: C++ Audio Library Options *Superpowered.com*, 2018. https://superpowered.com/audio-library-list
6. Zuurbier, E.: Organ Music in Just Intonation, https://www.ji5.nl/.
7. MIDI Files Specification http://somascape.org/midi/tech/mfile.htm
8. Variable Length Value http://www.ccarh.org/courses/253/handout/vlv/
9. Guide to the MIDI Software Specification http://somascape.org/midi/tech/spec.html
10. Kabal, P.: Wave file specifications, http://www-mmsp.ece.mcgill.ca/Documents/AudioFormats/WAVE/WAVE.html
11. J.-C. Risset, Computer music experiments, 1964—. . ., Computer Music Journal, vol. 9, no. 1, pp. 11–18, 1985.
12. Mersenne Twister Algorithm http://www.math.sci.hiroshima-u.ac.jp/ m-mat/eindex.html