INF221 Project Report - SimpleSynth

Kristian Sørdal

May 5, 2023

Contents

List of Figures

1 Abstract

This report aims to provide an overview of the implementation details and functional techniques employed for a synthesizer and DFT visualizer written purely in Haskell. The report provides an overview of the program structure and a flowchart of the programs runtime. In depth explanations are provided for techniques and monads used, and how they are used. Background theory for the Discrete Fourier Transform is also provided.

2 Introduction

During my enrollment in the course *INF221 - Advanced Functional Programming*, we had to develop a project of our own choosing in Haskell (or any other functional programming language.). My project started with the intention of developing a visualizer of the Discrete Fourier Transform, and later evolved into an implementation of a synthesizer which allows the user to select between multiple oscillators and beats per minute, and allows for music creation by selecting of notes and number notes, aswell as their beat-length. The synthesizer writes the resulting sound to a WAVE file, which can be read by the DFT visualizer to plot the wave, and to apply effects provided by the equalizer. The project feels like a coherent and completed program, and went beyond what was orignally proposed. Although I could implement several other features, I have to cut it short here, due to time constraints.

This project is very interactive, and makes heavy use of the state and `IO` monad. In fact, the monad transformer `StateT`, which adds stateful computation to other monads, in this case the `IO` monad, is what makes up the most important monad in both the DFT visualizer, and the Synthesizer.

Other important libraries used is the `Graphics.Matplotlib`, which provides bindings to Pythons Matplotlib. `Data.WAVE` to read and write from .wav files. `Numeric.Transform.Fourier.FFT` to compute the DFT using the Fast Fourier Transform algorithm. `SDL.Mixer` and `SDL` is also used to play sound. Parsing is provided by `Text.Megaparsec` and `Control.Monad.Combinators.Expr`.

3 Program Structure

As mentioned, the program is very interactive, and relies on input from the user to advance. The program has two main components, these being the DFT Visualizer, and the Synthesizer. The flowchart illustrated in Figure 1 illustrates the program flow.

The two modes might look disconnected at first, however, the files written in the synthesizer can be read in the DFT visualizer, which allows for usage of the equalizer and the plotting of the wave generated by the sound.

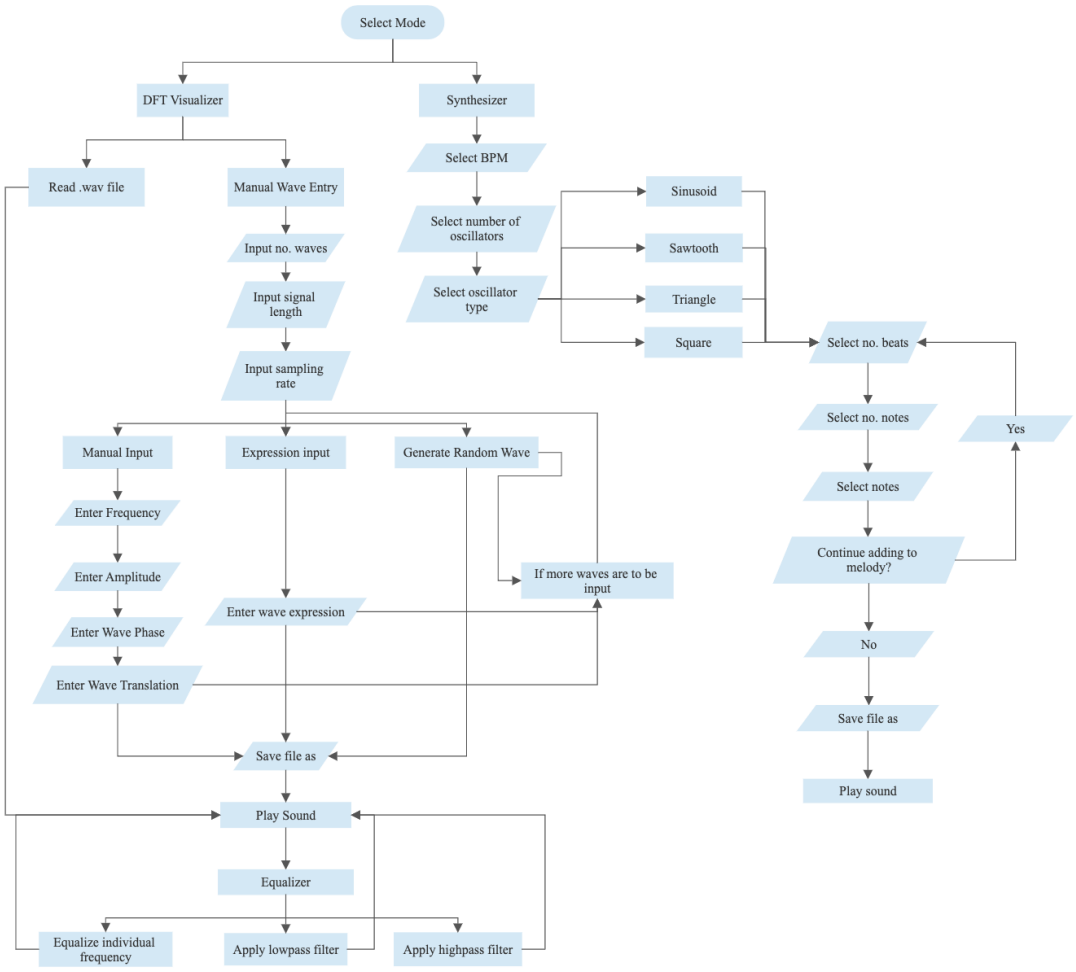

Figure 1: Interactive program flowchart

4 Functional Programming Techniques

When writing code in Haskell, we can employ functional programming techniques to make our code more readable and safe. This section goes into detail about the functional programming techniques used in this project.

When writing Haskell, we are able to alias types to a more descriptive name. This is also done in this project. It's worth taking a look at the various type aliases used in this project, as shown in Figure 2. These aliases increases readability by making type definitions more descriptive as to what a functions arguments represent.

```haskell
type BPM = Integer
type Beats = Float
type Frequency = Float
type Note = String
type SignalLength = Float
type SampleRate = Float
type Sample = (Float, Float)
```

Figure 2: Type aliases used in this project

4.1 IO Monad

The `IO` monad is used throughout the program to recieve user input, plotting through bindings to `Matplotlib`, writing WAVE files with `Data.WAVE`, and playing sounds through bindings to `SDL`.

4.1.1 User input

If we want to process user input in Haskell, we have to employ the `IO` monad. Using this monad, we get access to functions such as `getLine`, which reads input from the user, and `putStrLn, putStr` and `print`, which prints content to the terminal.

Using the `IO` monad by itself is rather straight forward, however, using it in monad stacks have some caveats we have to pay attention to. When we want to use the `IO` monad in our monad stack as shown in Figure 8, we have to use the function `liftIO`. This function has the type `liftIO :: IO a -> m a`. It has the effect of lifting a computation done in the `IO` into the context of another monad `m`, and is what allows the `IO` monad to be used in the monad stack.

In the `Utils` module provided, some `IO` functions have been combined with the `liftIO` function, to prevent repetition, as shown in Figure 3.

```haskell
inputFloat :: String -> IO Float
inputFloat s = do
  putStrLn s
  putStr "$ "
  input <- getLine
  case (readMaybe input :: Maybe Float) of
    Just num -> return num
    Nothing -> do
      putStrLn "Incorrect type, try again."
      inputFloat s

putStrLn' s = liftIO $ putStrLn s

putStr' s = liftIO $ putStr s

inputFloat' x = liftIO $ inputFloat x
```

Figure 3: `IO` functions combined with `liftIO`

4.1.2 Matplotlib

Another component of the program which uses the `IO` monad is the `Graphics.Matplotlib` module. As described earlier, this module provides bindings to Pythons Matplotlib library, and is what takes

care of plotting the waves. The function `onscreen` provides the actual plotting, and it has the type `Matplotlib -> IO ()`. The `Matplotlib` datatype is the wrapper used for communication with Matplotlib. In the case of the DFT visualizer, we have defined the `plotFigure` function, shown in Figure 4, which when given the samples of a wave, and the samples of its DFT, plots the figure to the screen.

```
plotFigure :: [Sample] -> [Sample] -> IO ()
plotFigure samples fftArr = do
  let xCoords = map fst samples
      yCoords = map snd samples
      xCoordsFFT = map fst fftArr
      yCoordsFFT = map snd fftArr

  onscreen $
    subplots
      @@ [o2 "nrows" 2, o2 "ncols" 1]
      % setSizeInches 10 8
      % setSubplot 0
      % title "Time Domain"
      % xlabel "Time [s]"
      % ylabel "Amplitude"
      % plot xCoords yCoords
      @@ [o2 "linewidth" 0.5]
      % setSubplot 1
      % title "Frequency Domain"
      % xlabel "Frequency Bins [Hz]"
      % ylabel "Magnitude"
      % plot xCoordsFFT yCoordsFFT
      @@ [o2 "linewidth" 1]
```

Figure 4: Function for plotting

Note that the operator `%` is used to sequence two plots, and the `@@` operator is used to specify options for the plot.

4.1.3 Reading and writing WAVE files

The `Data.WAVE` [8] modules exposes the `WAVE` datatype, and `WAVEheader` datatypes. These datatypes are used to describe the contents of a WAVE file. They have the following constructors.

```
data WAVE = WAVE { waveHeader :: WAVEHeader,
                   waveSamples :: WAVESamples }

data WAVEHeader = WAVEHeader { waveNumChannels :: Int
                             , waveFrameRate :: Int
                             , waveBitsPerSample :: Int
                             , waveFrames :: Maybe Int
                             }
```

Figure 5: WAVE and WAVEheader datatypes.

The constructors describe the follwing:

- `waveSamples` - The samples of the waves.
- `waveNumChannels` - The number of channels of the WAVE file.
- `waveFrameRate` - The sample rate of the wave.
- `waveBitsPerSample` - Significant bits (in our use case this is 32).

The function `putWAVEfile` is used to write a new WAVE file. It has the type `String -> WAVE -> IO ()`. The string specifies the filepath (and name) of the file, and the `WAVE` argument are the wave samples, along with the header of the wave we have created. We also use the function `getWaveFile`, which when given a filepath, returns the WAVE at this locations.

4.1.4 Playing sound

`SDL.Mixer` [3] is used to get access to the system audio. This module provide Haskell bindings to the SDL2 Mixer library, which is a very simple multi-channel audio mixer [7]. It is used in close conjunction with the `Data.WAVE` module, as after we have written a WAVE file, we can access it with `SDL.Mixer`, to play its audio contents.

When using `SDL.Mixer`, we have to specify the audio frequency, format, and output through the `Audio` data type. However, this class is an instance of the `Default` class, which allows us, using the function `def` from the module `Data.Default.Class`, to intialize this audio with `Mix.openAudio def 256`. This provides us with the default audio configuration, and we can load files and play them as we please, using the `load` and `play` functions.

4.2 State monad transformer

When I first started writing this project, I noticed that I was reusing alot of the same variables in different parts of the program. Information about waves such as its sample rate, length, and sample points where being passed around from function to function, leading to very messy type signatures with many arguments, making it hard to keep track of which variables was available where. It also led to the functions being hard to extend, when I needed to expand functionality. This led me to exploring the use of the `State` monad.

The state monad is a monad made available through the `Control.Monad.State` module [6]. It opens up for state management in Haskell. State is a collection of one or more variables which are used throughout the program to perform various calculations. All variables in the state are usually not needed all the time, but all are used at some point. In an imperative language, state can be manged through either global variables, initalized to some default value, or through a class with field variables defining the state of the program.

Haskells state monad provides much of the same functionality. We are able to get the state in any function with the appropriate type, using the `get` function. We are also able to modify the state using the `put` or `modify` function.

One problem is that we also need to modify the state based on input from the used, and as we know, user input is provided in Haskell by the `IO` monad. Combination of the `State` and `IO` monad is not possible by default. However, by utilizing monad transformers, this functionality is achievable. The `StateT` monad has the type

```
StateT s (m :: Type -> Type) a
```

which has the following constructors

- `s`: The state

- `m`: The monad to combine the `State` monad with.

To define the state we want to use for the DFT visualizer, we first define the state we want to use, as shown below.

```
type WaveVars =
  ( SignalLength,
    SampleRate,
    [[Sample]],
    FilePath,
    Mode
  )
```

Figure 6: Variables for the WaveState

Similarily for the state of the synthesizer, as shown in figure Figure 7

```
data Oscillator
  = Sinusoid {sample :: [Sample]}
  | Sawtooth {sample :: [Sample]}
  | Triangle {sample :: [Sample]}
  | Square {sample :: [Sample]}
  deriving (Show, Eq)

type SynthVars =
  ( BPM,
    Beats,
    [Oscillator],
    [Frequency],
    [[Sample]],
    [[Sample]]
  )
```

Figure 7: Variables used for the synthesizer state

As we now have variables making up our desired states, we can create types for the monad transformer combining the `IO` monad and the `State` monad, as shown below.

```
type WaveState = StateT WaveVars IO ()
type SynthState = StateT SynthVars IO ()
```

Figure 8: The WaveState and SynthState type

This monad is used heavily throughout the DFT Visualizer and the Synthesizer, and it provides all necessary functionality for recieving user input, and managing the state.

4.3 Parsing

As the DFT Visualizer allows for wave inputs in the form of mathematical expressions, we need a way to parse these expressions. To do this, we utilize the module `Text.Megaparsec` from the

`megaparsec` package [2]. We also use the module `Control.Monad.Combinators.Expr` from the package `parser-combinators` [1].

When parsing in Haskell, it's a good idea to first define an abstract syntax tree, or AST, to represent the structure of the "language" of our mathematical expressions. The AST defined for this project is not an extensive tree describing all mathematical operations, but is rather limited to operations useful for creating a wave expressions. The AST defined in this project can be seen below.

```
data WaveExpr
  = Lit Float
  | Var String
  | Add WaveExpr WaveExpr
  | Sub WaveExpr WaveExpr
  | Mult WaveExpr WaveExpr
  | Div WaveExpr WaveExpr
  | Exp WaveExpr WaveExpr
  | Sin WaveExpr
  | Cos WaveExpr
  | Asin WaveExpr
  | Acos WaveExpr
  | Mod WaveExpr WaveExpr
  | Floor WaveExpr
  | Signum WaveExpr
  deriving (Show, Read, Eq)
```

Figure 9: AST defined for wave expressions

Now, we can start writing our parser. To do this, we define the type for the parser, as seen in Figure 10.

```
type Parser = Parsec Void String
```

Figure 10: The parser type

Next, using the power of the `Control.Monad.Combinators` module, we define our operator table. This is done using the datatype `Operator`, which has the type `data Operator m a`, where:

- `m` : The parser monad we'll use, in our case: `Parser`
- `a` : The type we want to parse into

In our operator table, we'll use the following constructors:

- `InFixL`: Left-associative infix
- `Prefix`
- `Postfix`

It is worth going over the type definition of these constructors. The `InFixL` constructor has the type defintion

```
m(a -> a -> a)
```

where `m` is a monad, and `a -> a -> a` is a function taking two arguments of type a, returning a. The parentheses represent the fact that the function between them is wrapped in the monad `m`. In our case, `m` represents the type `Parser`, and `a` represents the type `WaveExpr`. This is the constructor we use for binary operators, such as addition, subtraction, multiplication and division.

The `Prefix` and `Postfix` constructors have a similar, but simpler type, as shown below

```
m(a -> a)
```

where again, `m` represents a monad, and `a -> a` represents a function which takes one argument of type a, and returns a. Again this functions is wrapped in the monad, and the letters represent the same types as mentioned above. The prefix operator is used for unary operators, such as sin, cos, asin and acos. The postfix operator is used to allow multiplication of real numbers and variables to be parsed correctly without including the multiplication sign, i.e. $2 \cdot x = 2x$.

With the explanation provided above, it should be easy to understand the operator table used in the project, as shown in Figure 11. Our operator table is a list of lists of type `Operator Parser WaveExpr`. Note that the ordering of this list is vital, as it describes the precedence of the operators.

```
opTable :: [[Operator Parser WaveExpr]]
opTable =
  [ [ Postfix
        ( do
            x <- some alphaNumChar
            return $ Mult (Var x)
        )
    ],
    [ Prefix (string "sin" $> Sin),
      Prefix (string "cos" $> Cos),
      Prefix (string "asin" $> Asin),
      Prefix (string "acos" $> Acos),
      Prefix (string "floor" $> Floor),
      Prefix (string "sgn" $> Signum)
    ],
    [InfixL (string "^" $> Exp)],
    [ InfixL (string "*" $> Mult),
      InfixL (string "/" $> Div),
      InfixL (string "%" $> Mod)
    ],
    [ InfixL (string "+" $> Add),
      InfixL (string "-" $> Sub)
    ]
  ]
```

Figure 11: Operator Table used in the parser

As an example, lets take a look at our defintion of the sin function: `Prefix (string "sin" $> Sin)`. This is a prefix operation, and for the parsing to succeed, the parser has to parse the string "sin". We then use the operator `$>`, which is a function of type `($> :: f a -> b -> f b)`. This function takes a functor `a`, which it ignores the value of (given no errors are thrown), and returns the value of type `b`, wrapped in a functor. In our case, it returns `Sin`, which is a constructor of the `WaveExpr` datatype, taking 1 argument.

The rest of the parser is neatly wrapped together with the functions shown in Figure 12, where the `parseVar` function parses our defined variables, "x" or "pi", into the AST. The `term` function

parses either a float, variables, or the rest of a wave expression between a pair of parentheses. Here, the `<|>` operator is used, often dubbed the "choice" operator, as it tries in order, the listed parsers, and if one fails, the next one is attempted until a parser succeeds, or the list is exhausted, and an error is thrown. The `waveExpression` wraps it all together by bringing the operator table and the `term` function together in one parser.

```
waveExpression :: Parser WaveExpr
waveExpression = makeExprParser term opTable

parseVar :: Parser WaveExpr
parseVar = do
  v <- string "x" <|> string "pi"
  return (Var v)

term :: Parser WaveExpr
term = Lit <$> float <|> parseVar <|> betweenParen waveExpression
```

Figure 12: Various functions used in the parser

After this parser is ran on valid input, we'll obtain the AST of the expression, which we can easily write a recursive evaluation function for, as shown in Figure 13

```
eval :: WaveExpr -> Float -> Float
eval (Lit x) _ = x
eval (Var n) v
  | n == "pi" = pi
  | otherwise = v
eval (Add x y) v = eval x v + eval y v
eval (Sub x y) v = eval x v - eval y v
eval (Mult x y) v = eval x v * eval y v
eval (Exp x y) v = eval x v ** eval y v
eval (Div x y) v = eval x v / eval y v
eval (Sin x) v = sin (eval x v)
eval (Cos x) v = cos (eval x v)
eval (Asin x) v = asin (eval x v)
eval (Acos x) v = acos (eval x v)
eval (Mod x y) v = eval x v `mod` eval y v
eval (Floor x) v = fromIntegral $ floor (eval x v) :: Float
eval (Signum x) v = signum (eval x v)
```

Figure 13: Evaluation function used to evaluate the AST

This function is used to compute the value of the AST at a given $x-$coordinate in order to obtain the $y-$coordinates of the wave. It takes both the AST and the value of the current $x-$coordinate as values. It is mapped across all $x-$coordinates.

5 DFT Visualizer

The DFT visualizer component of this program allows us to see the Fourier Transform of a wave. This is a novelty in the sense that its interesting in its own right to have a look at the varying frequencies

that make up a wave, and how much they contribute. It is also useful when using the equalizer provided, which allows us to filter these frequencies with a low pass or high pass filter, or equalize individual frequencies.

5.1 Discrete Fourier Transform using FFT

A Fourier Transform is a transform which converts a function from the time domain, to the frequency domain, i.e. to a description of the frequencies and amplitudes that constitute the wave [10]. The Discrete Fourier Transform is, as the name implies, an algorithm to perform a Fourier transform on a function, given a sequence of equally spaced samples. Given a set of n complex numbers x_n, we can use the DFT to convert these samples into another set X_k of complex numbers. X_k is defined as

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-\frac{i2\pi}{N}kn} \tag{1}$$

We can perform this algorithm on a set x_r where $x_r \in \mathbb{R}$, however, the resulting DFT will be symmetrical, i.e. $\{x_0 \dots x_{N/2-1}\} = \{x_{N-1} \dots x_{N/2+1}\}$. This is taken care of in the implementation of the `rfft` function, however we have to double the amount of samples in the `xCoords`, or else the frequencies would be doubled.

The algorithm used to perform the DFT is the FFT algorithm. This algorithm is provided by the `Numeric.Transform.Fourier.FFT` module [5]. The `rfft` function, takes in an array of real numbers (x_n), and returns an array of complex numbers (X_k). In order to extract useful information from these complex numbers, we map the `magnitude` function, provided from `Data.Complex` [4] over the elements in the array. The magnitude provided is equal to half the amplitude of the wave at the given frequency.

```
calcFFT :: [Float] -> Int -> Int -> [Sample]
calcFFT yCoords sampleRate signalLength = zip xCoords magnitudes
  where
    yCoords' = listArray (0, length yCoords - 1) yCoords
    fftArr = rfft yCoords'

    xCoords = [0, 1 .. (fromIntegral sampleRate)]

    magnitudes =
      getElems $
        map
          (\x -> (magnitude x :: Float) / fromIntegral sampleRate)
          (take (signalLength `div` 2) (elems fftArr))
```

Figure 14: Function used to compute the DFT using the FFT algorithm.

This function is used regardless of the input method for the wave, be it expression, manual, read WAVE files or randomly generated waves. As the samples generated from these input methods have the same type. During development, There were two separate functions to calculate the FFT. This was due to the fact that a separate python script was called from Haskell in order to read the WAVE file, and the resulting samples were different from the other samples. This was rectified when the `Data.WAVE` library was employed, and led to only one of the functions being necessary.

When creating waves with the DFT visualizer, the user will be prompted to input amount of waves, the length of the waves in seconds, and the sampling rate. It is recommended that the sampling

rate is high $\geq$ 30kHz, especially when sampling complicated waveforms, in order to preserve detail. After sample rate is input, the user can select input mode. Waves can be input manually, where variables such as frequency, amplitude, translation and phase is specified, or by expression input, where mathematical expressions are parsed into waves. Waves can also be randomly generated. The sound of randomly generated waves will result in white noise.

After the user has input all waves, a plot will be generated using `Graphics.Matplotlib`, Figure 15 shows what the DFT of a Sawtooth wave with a frequency of 440Hz would look like. Note that the x-axis is zoomed in to better show the wave. After the plot is shown on screen, the sound of the waveform will be played, and the user will be given the option to equalize the wave.

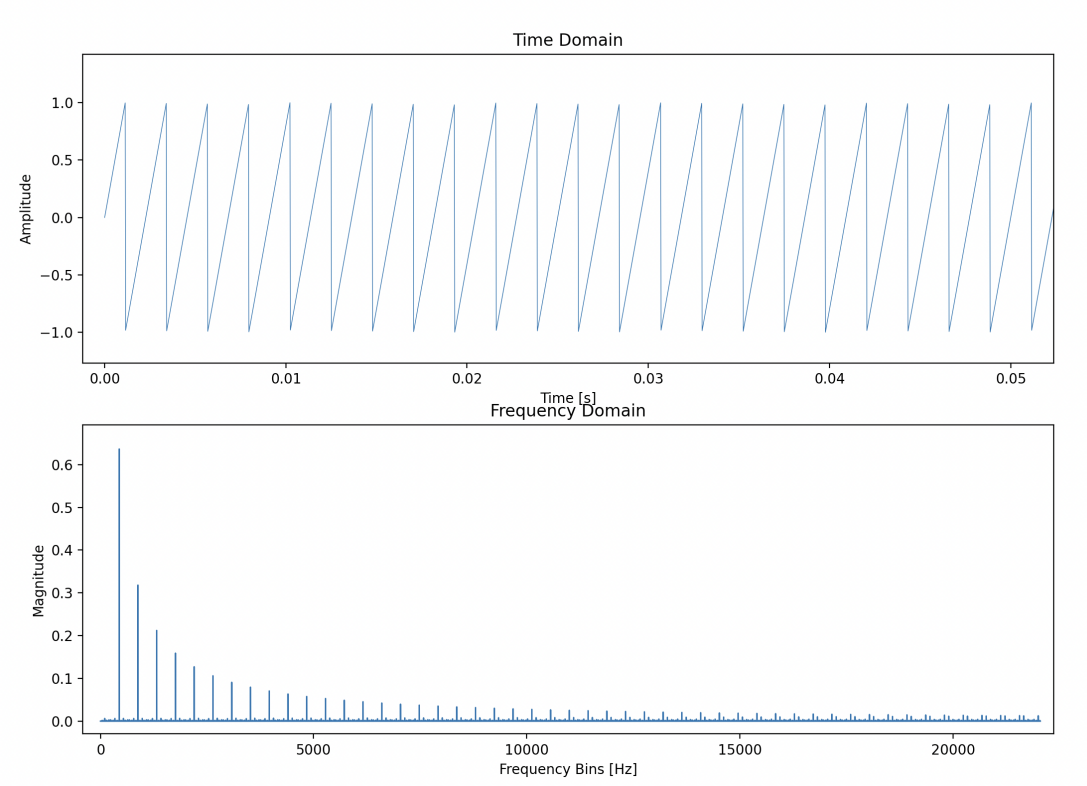

Figure 15: DFT of a Sawtooth wave at 440Hz

5.2 Equalizer

An equalizer provides the possibility to adjust the volume of certain frequencies in an audio signal [9]. The equalizer provided in this program has three different options. As mentioned, we can filter away unwanted frequencies using a low or high pass filter, where the low pass filter, or LFO, only allows waves with frequencies lower than the specified cutoff to pass through, and vice versa for the HFO. The implementations of this functionality, including equalization of individual frequencies is rather simples, as shown in Figure 16.

```
lowpass :: [Wave] -> Float -> [Wave]
lowpass w cutoff = filter (\(Wave _ freq _ _) -> freq <= cutoff) w

highpass :: [Wave] -> Float -> [Wave]
highpass w cutoff = filter (\(Wave _ freq _ _) -> freq >= cutoff) w

individual :: [Wave] -> Int -> Float -> [Wave]
individual ws i reduction = ws'
  where
    wave = ws !! (i - 1)
    equalizedWave =
      wave
        { amplitude = amplitude wave * (reduction / 100),
          frequency = frequency wave,
          phase = phase wave,
          translation = translation wave
        }
    ws' = take (i - 1) ws ++ [equalizedWave] ++ drop i ws
```

Figure 16: Functions for the three equalization options.

6 Synthesizer

The synthesizer provided in the program allows the user to create music and melodies, through input specifying the types of oscillators, beats per minute, beat length, amount of notes, and which notes to be played.

The user first sets the tempo of the melody, after which the user can pick from up to four different oscillators, these being the Sinusoid, Sawtooth, Triangle and Square waveforms. If multiple oscillators are picked, the resulting waveform will be the interference of the oscillators. In order to take the interference of the oscillators, we need to generate samples for the individual wave forms. Equation (2), (3), (4), and (5), show the mathematical equation for calculating samples of the Sinusoid, Sawtooth, Triangle and Square waveform, respectively, where f is the frequency of the wave. Figure 17 shows the Haskell equivalent functions to generate these samples.

$$y(t) = A \sin(2\pi f t) \tag{2}$$

$$y(t) = 2\left(ft - \left\lfloor \frac{1}{2} + ft \right\rfloor\right) \tag{3}$$

$$y(t) = 2\left|2\left(ft - \left\lfloor \frac{1}{2} + ft \right\rfloor\right)\right| - 1 \tag{4}$$

$$y(t) = \operatorname{sgn}(\sin(2\pi f t)) \tag{5}$$

```
sinusoidWaveSample :: Float -> Float -> Float -> Float -> [Sample]
sinusoidWaveSample start freq len sampleRate = zip xCoords yCoords
  where
    xCoords = [start, start + 1 / sampleRate .. start + len]
    yCoords = map (\x -> sin (2 * pi * x * freq)) xCoords

sawtoothWaveSample :: Float -> Float -> Float -> Float -> [Sample]
sawtoothWaveSample start freq len sampleRate = zip xCoords yCoords
  where
    xCoords = [start, start + 1 / sampleRate .. start + len]
    yCoords = map (\x -> 2 * (freq * x - fromIntegral (floor (1 / 2 + freq * x))))
        xCoords

triangleWaveSample :: Float -> Float -> Float -> Float -> [Sample]
triangleWaveSample start freq len sampleRate = zip xCoords yCoords
  where
    xCoords = [start, start + 1 / sampleRate .. start + len]
    yCoords = map (\x -> 2 * abs (2 * (freq * x - fromIntegral (floor (1 / 2 + freq *
        x)))) - 1) xCoords

squareWaveSample :: Float -> Float -> Float -> Float -> [Sample]
squareWaveSample start freq len sampleRate = zip xCoords yCoords
  where
    xCoords = [start, start + 1 / sampleRate .. start + len]
    yCoords = map (\x -> signum (sin (2 * pi * freq * x))) xCoords
```

Figure 17: Functions to generate wave samples.

After the oscillators have been selected, the user is prompted to choose how many beats the notes should last. Next, the amount of notes needs to be selected, and finally which notes should be played. The notes are input in scientific pitch notation, i.e. Middle C will be input as "C4". This note is then looked up in the map provided in the `Utils` module, which maps the note to its respective frequency. If multiple notes are selected, the resulting wave will be the interference between the waves generated by all the notes. If there are multiple oscillators, the resulting wave will be the interference again, this time of the waves generated by the oscillators. The `interference` function is shown in Figure 18.

```
interference :: [[Sample]] -> [Sample]
interference s = zip (map fst (head s)) yCoords
  where
    yCoords = map sum (transpose $ map (map snd) s)
```

Figure 18: Function to compute interference

After this, the user is prompted whether or not to continue adding notes to the melody, and the process repeats until the user is satisfied, at which point the resulting melody will be played on a loop. The user can then restart the program, select the DFT visualizer, and read the newly generated wave file in order to have a look at its DFT, aswell as equalize the wave.

7 Reflection on implementation and techniques

As I wanted to be able to plot waves on the screen, I knew I needed to find a library capable of this, sadly the options are limited. I conidered using both `gloss` and `Chart`. The former needs a lot of work in order to be able to properly provide the plotting capabilities necessary to be usable, and the latter I coulnd't even get working on my machine. I ended up using the `Graphics.Matplotlib` library. This was in my opinion a suboptimal choice, but I knew it would get the job done. As this library is described as having next to no type safety, and the fact that it generates Python code, I was not very excited about having to use this library. The library also contains a few bugs and oddities. Whenever a plot with subplots is to be generated (as in the case for the DFT visualizer), two windows opens, where one is empty. I was unable to find a fix for this.

I am quite satisfied with the way I decided to implement the Synthesizer, as it was written with the `StateT` monad transformer in mind. This resulted in clean and consice code. In contrast, when I first wrote the DFT visualizer in the `Main` module, I did not use the `StateT` monad transformer, instead, it was converted to use this monad transformer at a later date. This conversion did end up yielding better and more clear code, however I wasn't quite satisfied with the way it turned out. I ended up rewriting what was in the `Main` module in a new module named `DFTVisualizer`. This proved to be very beneficial, and it improved the code structure greatly.

Using the State monad transformer proved to be a very good choice for this project, as managing the state of the variables was something I already did before I employed this monad, but in a more verbose and comlicated manner. Although daunting at first, the necessary functions were simple to understand and employ, and improved both code quality and extensibility of the project.

The parser that was implemented turned out to work better than expected. In our lectures, we started implementing a basic parser for arithmetic expressions. This parser was used as a starting point, but it had to be improved in order to prevent parentheses between all operations. This seemed difficult to implement at first, however employing the usage of `Control.Monad.Combinators.Expr` was very beneficial. This module allowed me, as mentioned previously to define an operator table, which allowed me to specify both the syntax of the expressions, and their precedence. The operator table is very extensible, so adding new functionality was a breeze. I was honestly quite amazed of the power of this module, and it will definitely be my goto any time I'm in need of parsing mathematical expressions.

Another aspect that would have made the implementation smoother, would have been reading further up on theory before writing my code. I noticed that I struggled getting the FFT samples to properly be displayed. It wasn't until far out in the project I realized it even was wrong. The issue was that the frequencies values were doubled, and I spent a long time implementing a fix. I do believe that had I read the theory behind the algorithm more throughouly beforehand, this wouldn't have happened.

As audio samples usually have a high sampling rate, usually in the range of 48kHz, it can take quite some time to perform fourier transforms on these waves. This especially applies to waves such as the sawtooth wave seen in Figure 15, where there are many frequencies with very small magnitudes. When attempting to equalize these waves, a very large number of samples have to be regenerated, and it is not very efficient. Given more time, I would look into options for optimizing this, perhaps through memoization. I can imagine there are effective methods achieve an effective implementation of an equalizer when the amount of waves and samples grow large, but due to time contraints I have not been able to look into this.

References

[1] Mark Karpov , Alex Washburn. *Lightweight package providing commonly useful parser combinators*. https://hackage.haskell.org/package/parser-combinators-1.3.0. May 2, 2023.

[2] Paolo Martini , Daan Leijen , Contributors. *Monadic parser combinators*. https://hackage.haskell.org/package/megaparsec-9.3.0. May 2, 2023.

[3] Vladimir Semyonov , Sinisa Bidin , Daniel Firth. *Haskell Bindings to SDL2 Mixer*. https://hackage.haskell.org/package/sdl2-mixer. May 4, 2023.

[4] *Data.Complex*. https://hackage.haskell.org/package/base-4.18.0.0/docs/Data-Complex.html. May 4, 2023.

[5] Matt Donadio. *Haskell Digital Signal Processing*. https://hackage.haskell.org/package/dsp-0.2.3. May 4, 2023.

[6] Andy Gill. *Control.Monad.State*. https://hackage.haskell.org/package/mtl-2.3.1/docs/Control-Monad-State.html. May 2, 2023.

[7] Libsdl. *SDL Mixer*. https://github.com/libsdl-org/SDL_mixer. May 4, 2023.

[8] Bart Massey. *Data.WAVE Source Code*. https://hackage.haskell.org/package/WAVE-0.1.6/docs/src/Data.WAVE.html. May 3, 2023.

[9] Wikipedia contributors. *Equalizer (Audio) - Wikipedia*. https://en.wikipedia.org/wiki/Equalization_(audio). May 4, 2023.

[10] Wikipedia contributors. *Fourier Transform - Wikipedia*. https://en.wikipedia.org/wiki/Fourier_transform. May 4, 2023.