Wãvy

A sound library written in Haskell

Date of compilation: March 16, 2013

Contents

Part I Introduction

WĂVY is a Haskell library entirely written by Daniel Díaz[1]. Things you can do with WĂVY include, but are not limited to, generate values containing sound waves info, manipulate sound waves as they were mathematical functions, decode sound files to values of the type already mentioned and encode them back to a sound file.

This text explains the library from the most basic usage to the details of its implementation. It tries to be as comprehensive as possible, but it is difficult to keep track of everything. In conjuction with the API reference, it should be enough to get a complete understanding of the library. Two different types of code will appear: code *using* the library and code *from* the library source code. The former is distributed along the whole text while the latter is concentrated in Part VII.

The library has been designed with some features in mind.

- **Easy-to-use:** This means that you can get some code working without any effort. It is well-documented and well-organized.
- **High-level:** So you can play with sounds without worrying about the actual implementation of the library. However, it lets you access to the low-level operations to let you tune up a very specific computation in a high-performance way.
- **Good performance:** High performance is one of the *goals* of the library. Here *goal* means that currently the library is not in an ideal state, but it works quite fine for a number of examples. No need to say that work in this *goal* will never stop.

This being said, since WĂVY is an open source project, remember that anybody interested can get involved and contribute to it. The source repository is hosted on GitHub.

`https://github.com/Daniel-Diaz/Wavy`

Feel free to fork! Also, install the latest release using Cabal.

```
cabal install wavy
```

Or download it manually from Hackage, where you can also find the API Reference online.

`http://hackage.haskell.org/package/Wavy`

[1]E-mail: `dhelta.diaz@gmail.com`.

Part II
Overview

This part contains an overview of how WĂVY works. Some of the types are described here and how to work with them, without going into particular details. Therefore the intention is just to give a vague idea about the use of the library.

1 First example

Let's start looking at a very simple example that shows how it works by itself. The aim is to generate a sine wave and encode it into a file. Here is the code:

```
import Data.Sound
import Data.Sound.WAVE

main :: IO ()
main = encodeFile "sine.wav" $ fromSound 16 s
 where
  s :: Sound
  s = sine 5 1 100 0
```

You may get confused by all the numbers applied to the `sine` function. Nothing to worry about. The actual sine wave is defined in the `where` clause.

- The first argument of `sine` is the duration in seconds.
- The second argument is the amplitude of the wave. Actually, the maximum amplitude the sine wave takes.
- The third argument is the frequency.
- Finally, the fourth argument is the *phase*.

So, in the example above, the defined sound is a 100Hz sine wave five seconds long. Then, we pass this sound as argument to the `fromSound` function. This function translates the sound to the WAVE format by choosing a *bit depth* (of 16 bits in the example) now ready to be encoded directly into a file via the `encodeFile` function. We chose `"sine.wav"` as the name for the file. And that's it! We have created and written our first sound! This example is included in the source distribution, along many others.

2 Types

Probably, the type you want to know first is `Sound`. This is the type where the sound info is stored. It contains information about how long is it, its sample rate, how many channels does it have and, perhaps the most sustantial part, the actual list of *samples* of the sound. Basically, a sound wave is stored as a (huge) list of signals. We cannot store a continuous wave, so the traditional method is to take a big list of samples equally-distributed in time (see Figure 1). Each

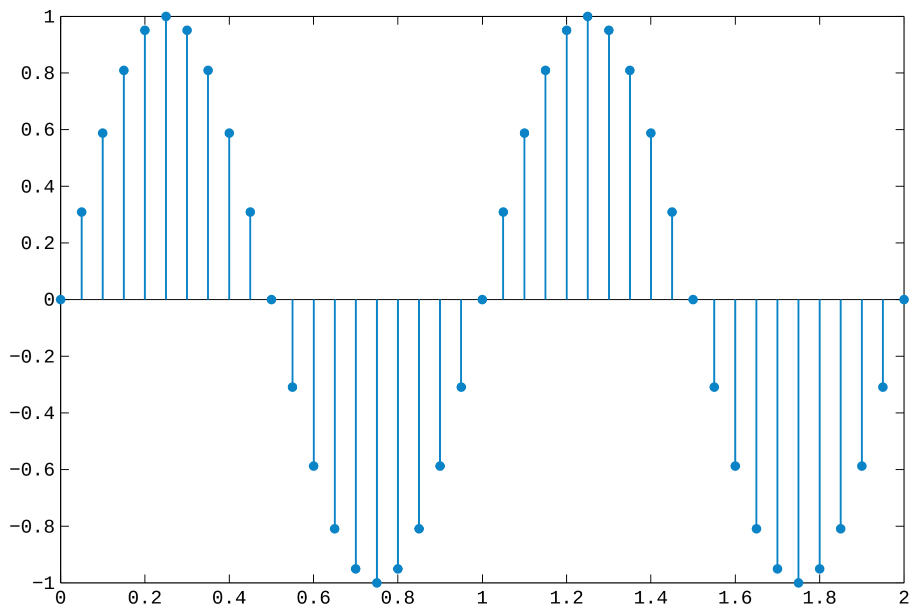

Figure 1: An approximation of a 1Hz sine wave using 20 samples per second.

value in these samples is represented by a `Double` number and they all lie in the interval $[-1,1]$.

Since the distance between the samples is fixed, there is also a fixed number of samples per second. This number is called the *sample rate* (note that the sound in Figure 1 has a sample rate of 20). So, how many samples has a value of type `Sound` one second long with a sample rate of 44100?[2] Easy: it has exactly 44100 samples! What if the sound was two seconds long? Just multiply by two the sample rate: $2 \cdot 44100 = 88200$. We see that the number of samples gets quickly big. Although it is just a linear growth, we usually take lot of samples per second, and this is the main reason why we do care so much about performance issues. In general, the number of samples N of a sound with sample rate R of duration D is given by the following equation:

$$N = \lfloor DR \rfloor \tag{1}$$

Note that we apply the floor function (denoted by $\lfloor \dots \rfloor$) because the number of samples is necessarily a non-negative integer (represented in WĀVY by a `Word32` value). If you think about it, it does not make any sense to have a fractional number of samples.

Reciprocally, we can also calculate how long a sound is from its number of samples. This is easily done following Equation 1.

$$D = \frac{N}{R} \tag{2}$$

[2]This is a commonly used sample rate and we are going to use it frequently.

Sometimes, we are interested in having several sounds running at the same time. For example, almost all the music recorded nowadays is recorded in stereo. That is, we listen to one sound through the left speaker and to another one through the right speaker. So each sample we take from this song has two values: the left value and the right value. The number of values per sample is the *number of channels*. Each sample is then stored as a list of `Double`s.

```
type Sample = [Double]
```

The *size* of a sound stands for the number of values stored in it and, if we denote by C the number of channels and restoring the above notation, the equation for the size (S) is as follows:

$$S = CN = C\lfloor DR \rfloor \tag{3}$$

Time is represented by values of the type `Time`, which is nothing but a type synonym for `Double`.

```
type Time = Double
```

So you can use any `Double` as a time parameter. But, in practice, most of the functions assume that the value of the `Time` argument is a non-negative number and this precondition is not checked, so you should be careful and always use non-negative times whenever it looks reasonable. For example, it is *not* reasonable to create a sound wave of negative duration.

The reason why we explain these properties of a sound is because we are going to use this terms and notation later in the text.

3 Workflow

Let's talk about the general workflow in WĂVY. In Figure 2 you can see a picture where the general workflow of the library is presented in a graphical way. This will give you an idea of how to work with WĂVY.

Essentially, you have three ways to obtain a new sound:

1. Generating the sound directly, which will give you a value of type `Sound`. Read more about generating sounds in Part III.
2. Reading the sound from a file. This will require a conversion in order to get a value of type `Sound`.
3. Transforming and combining previously generated sounds. Read more about sound manipulation in Part IV.

All the transformers and combiners in WĂVY work only with values of the type `Sound`. These are defined in the `Data.Sound` module. The encode and decode functions as well as the conversion functions are defined in submodules of `Data.Sound` in such way that each submodule corresponds to a different encoding. For instance, `Data.Sound.WAVE` is the module where the PCM WAVE

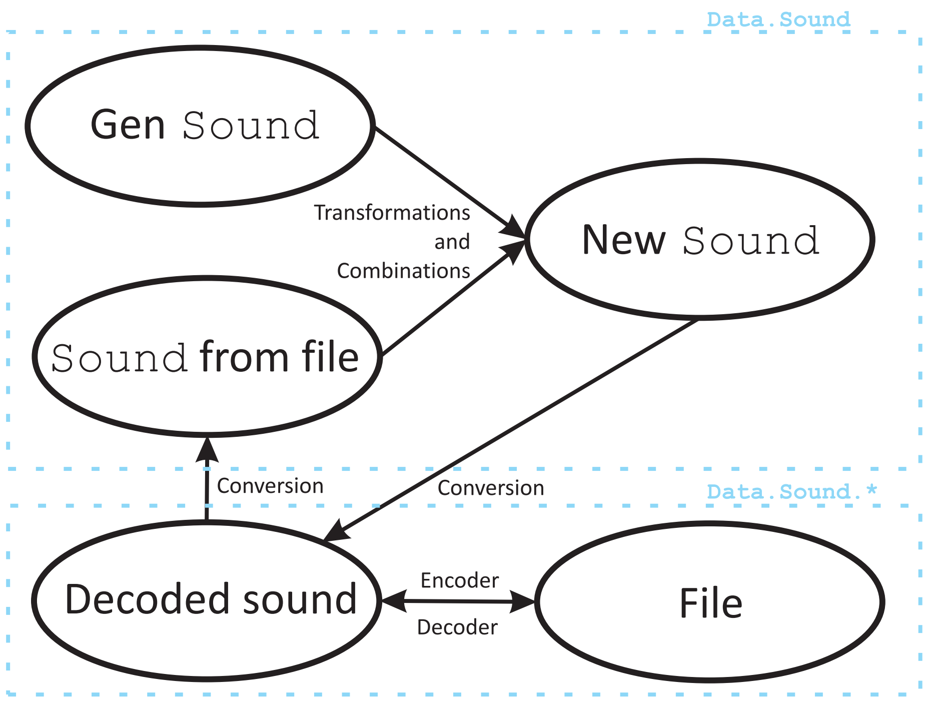

Figure 2: WĂVY general workflow.

encoders and decoders are defined. All these submodules have certain function names in common (like `encode` and `decode`) to make easier switch from one encoding to another[3]. Therefore, we work at two different levels: the encoding level, close to the wave representation level, and the manipulation level, close to the wave interpretation level.

Part III

Generating waves

The first basic task you might want to learn is how to generate new waves. There are several ways to do it. Namely, using basic wave generators, using functional wave generators or using other more sofisticated generators.

In this section, we will provide an explanation of how to build new wave signals using WĂVY.

4 Predefined waves

We will take a look at the basic wave generators predefined in WĂVY. These correspond to the most basic sound waves: the zero signal and the sine, sawtooth,

[3]Read more about encodings in Part V.

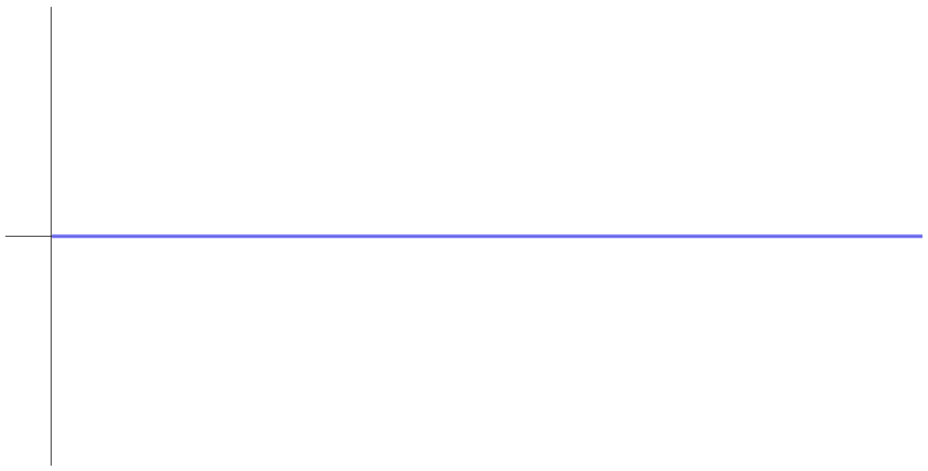

Figure 3: Zero signal.

square and triangle waves. They all work in a very similar way. Supplying a certain number of arguments to a generator function will give you a value of type `Sound` containing the desired sound wave. These are the basic wave generators explained one by one:

`zeroSound` The most simple but necessary generator. It creates a silence, a constantly zero valued sound.

```
zeroSound :: Time -> Sound
```

As indicated, it takes a `Time` argument and returns a zero sound with the given duration. Useful to create silence between other sounds. Also used internally to implement other functions and operators. See Figure 3.

`sine` One of the most fundamental sound waves. The pure tone with no harmonics. It produces a smooth sound with a clear pitch. Figure 4 shows how it looks like. The sine wave generator has the following type signature:

```
sine :: Time -> Double -> Time -> Time -> Sound
```

We give here an argument-by-argument explanation.

`Time` Duration. You must provide a *positive* number.

`Double` Amplitude. Any value between -1 and 1 is OK here. Note that a negative value has the same effect than the positive counterpart but with the phase inverted.

`Time` Frequency in Hz, the number of periods per second. This should be a positive number too since a negative frequency does not make much sense, but the function will work anyway.

`Time` Phase, the angle where the wave starts. Figure 5 shows the same sine wave of Figure 4 but phased by π. Note that is moved half period to the left, since the period of the sine function is 2π.

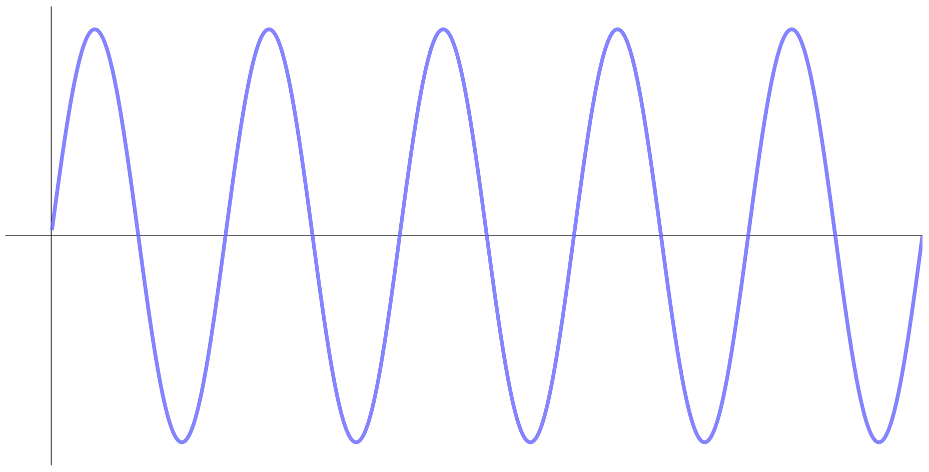
Figure 4: Sine wave.

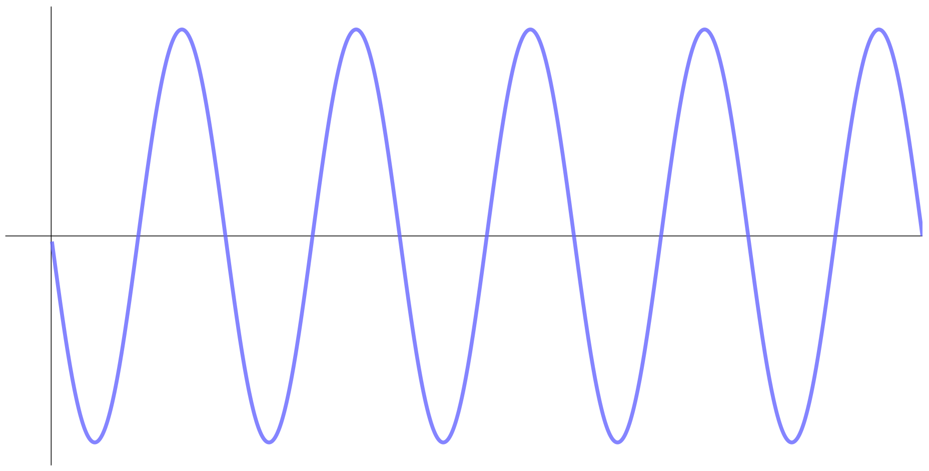
Figure 5: A sine wave phased by π.

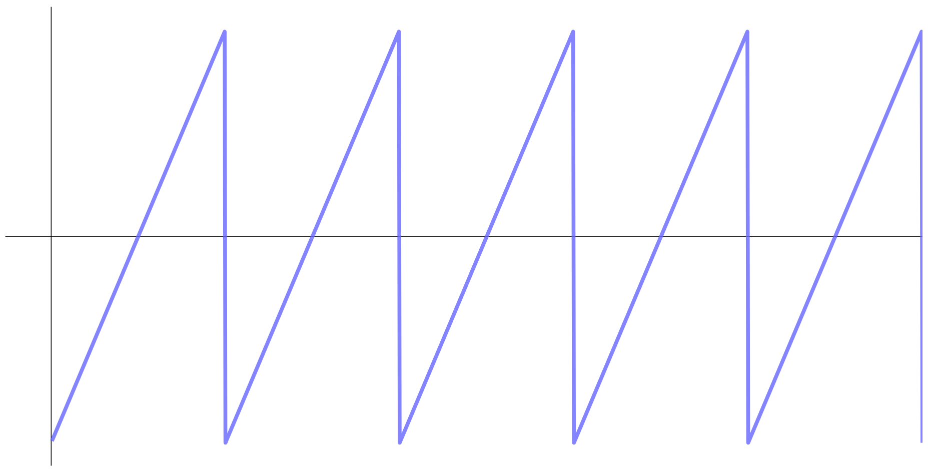

Figure 6: Sawtooth wave.

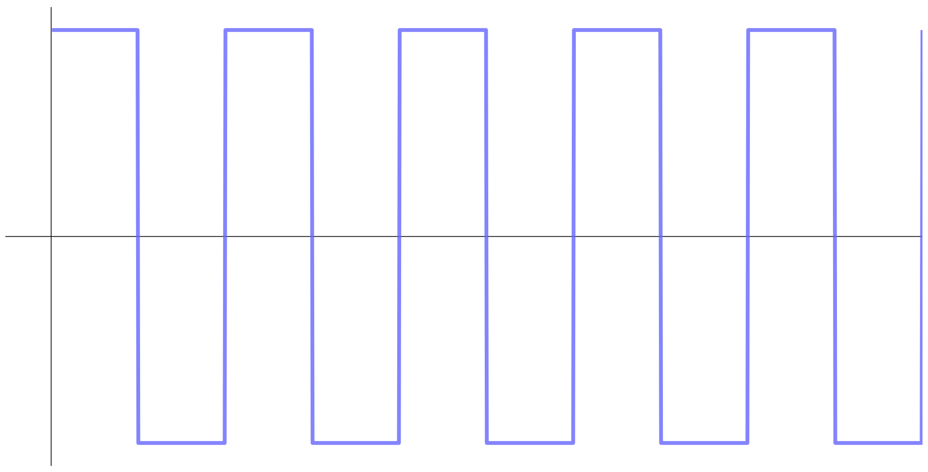

Figure 7: Square wave.

`sawtooth` Sawtooth non-sinusoidal wave.

The type signature coincides with the `sine`'s as well as the purpose of each one of its arguments. See Figure 6.

`square` Square non-sinusoidal wave.

The type signature coincides with the `sine`'s as well as the purpose of each one of its arguments. See Figure 7.

`triangle` Triangle non-sinusoidal wave.

The type signature coincides with the `sine`'s as well as the purpose of each one of its arguments. See Figure 8.

The output of the basic generators is always *mono*. Also, an alternative generator with an additional argument is supplied for all of them. This extra argument

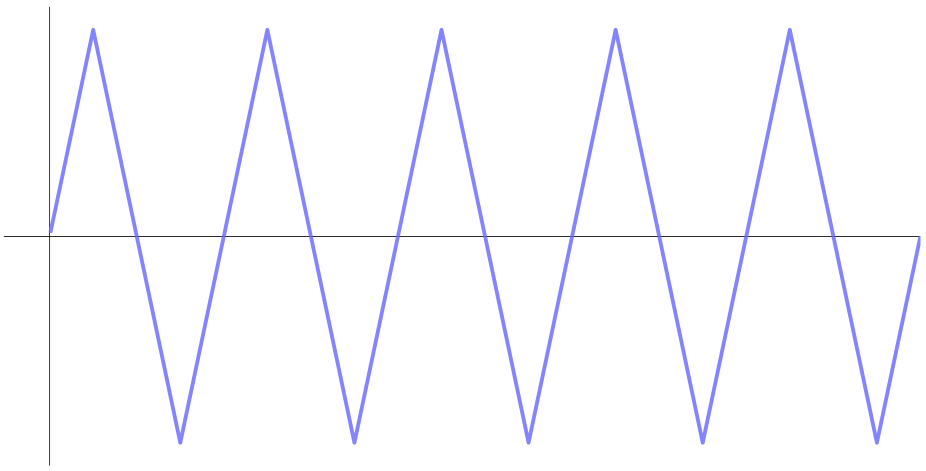

Figure 8: Triangle wave.

allows you to specify the sample rate of the generated sound. Otherwise, it will be defaulted to 44100 samples per second.

5 From functions

To give complete freedom when generating new waves, you can create new sounds from regular functions. Using `fromFunction` you are going to be able to create sound waves specifying the desired amplitude for each time.

5.1 General functions

It is possible to generate sound waves from time-dependent functions. Therefore, given the function $f(t) = 1$, you can easily create a sound with constant value 1. This can be done in the following way:

```
onlyOne :: Sound
onlyOne = fromFunction 44100 1 Nothing $ \t -> [1]
```

In this example, 44100 corresponds to the sample wave, 1 is the duration in seconds and the function is specified in the last argument. Note that it returns a list instead of a single value. The reason is because a sound with multiple channels will need more than one value for a given time. For instance, if you want to have silence in the left channel and constantly one in the right channel you should write:

```
rightOne :: Sound
rightOne = fromFunction 44100 1 Nothing $ \t -> [0,1]
```

More precisely, the type signature of `fromFunction` is:

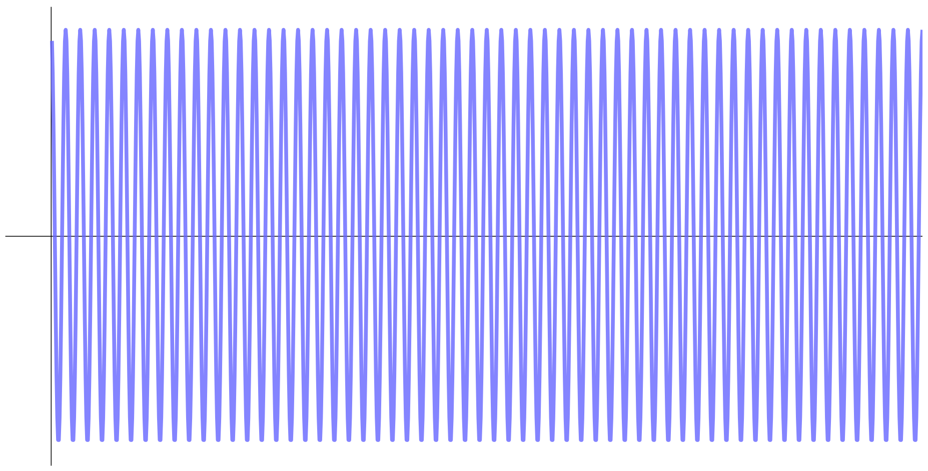
Figure 9: 2 seconds of a 30Hz cosine wave. Only one period has been calculated.

```
fromFunction :: Word32 -> Time -> Maybe Time -> (Time -> Sample)
             -> Sound
```

Remember that a `Sample` is a list of `Double` s.

5.2 Periodic functions

Recall the type signature of `fromFunction` :

```
fromFunction :: Word32 -> Time -> Maybe Time -> (Time -> Sample)
             -> Sound
```

We have ommited an explanation for the `Maybe Time` argument. It is an optional argument which purpose is to make the wave generation more efficient in the special case of periodic signals. This way, if you know that the signal you want to generate has period p, you can specify this period to help `fromFunction` to save time. Once the period is specified, `fromFunction` will only calculate the samples from 0 to p, and then it will replicate them until the specified duration is reached. We can apply this to generate a 30Hz cosine wave (see Figure 9), which will have a period of $\frac{1}{30}$.

```
cosine :: Sound
cosine = fromFunction 44100 2 (Just $ 1/30) $
           \t -> [cos (2*pi*30*t)]
```

5.2.1 The period problem

6 Other sounds

Besides the most common waveforms, WĀVY has more exotic wave generators predefined. Below an exhaustive list of them.

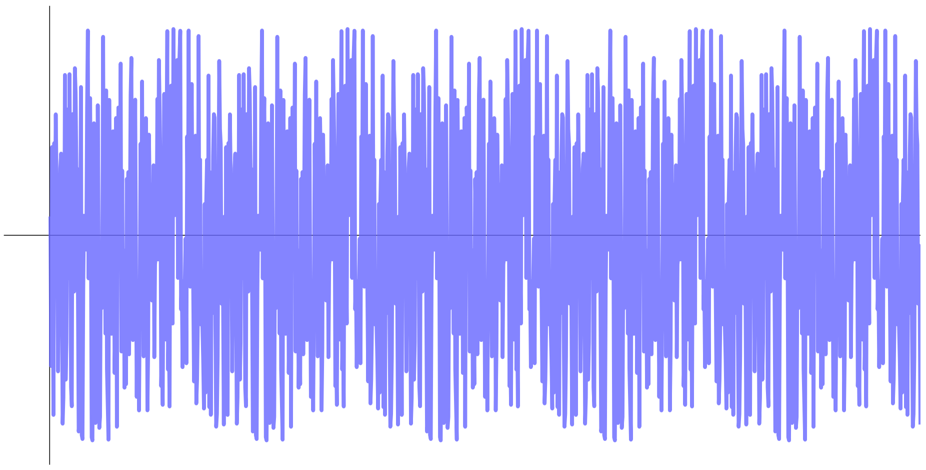

Figure 10: A sample output of `noise`.

`noise` This function produces a randomly generated signal that is repeated over time according to a given frequency. The seed for the random values must be provided. Different seeds will result in different sounds.

```
noise :: Time -> Double -> Time -> Int -> Sound
```

In order, the arguments are duration, amplitude, frequency and random seed. The frequency will fix the number of repeatitions of the noise per second. A sample is shown in Figure 10.

`karplus` An implementation of the Karplus-Strong string synthesis. It is based on the `noise` generator function, applying an exponential decay to its output.

```
karplus :: Time -> Double -> Time -> Double -> Int -> Sound
```

First three arguments are duration, amplitude and frequency as in `noise`. The fourth argument is a `Double` indicating the decay of the signal. It must be a number between 0 and 1, where 0 gives you instant decay (the signal becomes constantly 0) and 1 gives you no decay at all (the signal will be equivalent to `noise`). To produce a string-like sound it is recommended to provide small values for the decay. Last argument is the random seed that will be passed to `noise`. Figure 11 shows an example with decay 0.01 over one second. Note that, as in `noise`, different seeds will generate different signals.

Part IV
Manipulating sounds

If you have generated (Part III) some sounds and/or you have decoded some others (Part V) you may want to make operations with them, you may want to

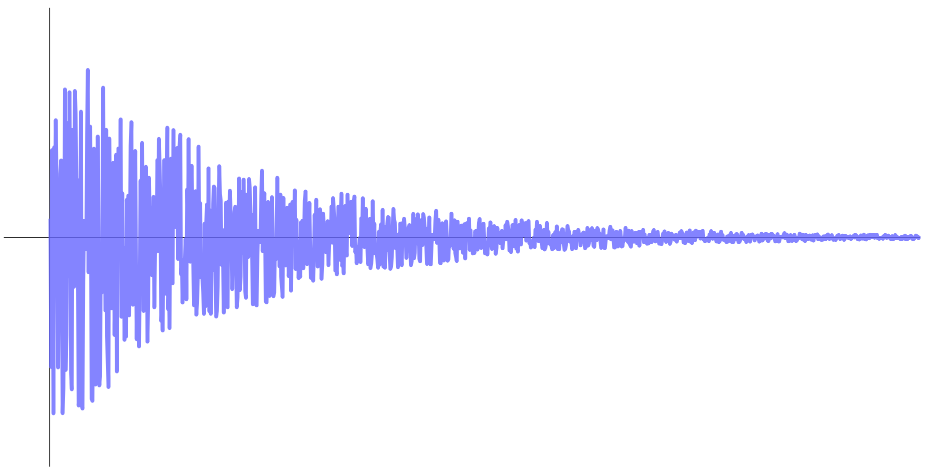

Figure 11: A sample output of `karplus` .

append them, sum them, modify their amplitude, move them throughout the channels, apply some decay or other effects like echo or reverb... Essentialy, you want to do something with the sounds you have in your hands, and this is exactly what this part is about.

7 Basic operators

There are three operators that are considered the *basic operators*. They don't generate all possible operators by combining them, but they are used widely and perform elementary transformations. Also, the performance of the library depends strongly in how this operations are implemented, so they should be a focus of improvement. However, in this section we limit ourselves to their usage. Each operator has also some usage restrictions. For example, a common restriction is that you can't operate sounds with different sample rates. You should first change the sample rate of one of them to be equal to the other.

7.1 Seq: <.>

The *sequencer operator* joins two sounds in one which content is the first sound followed by the other. For example, in Figure 12 a sine wave is seq'd with a square wave. Characteristics of the *seq* operator:

- It is left-associative.
- Its abreviated name is *seq*. Don't confuse it with the Haskell `seq` operator!

The restrictions of the *seq* operator are:

- Both arguments must share the same *sample rate*.
- Both arguments must share the same *number of channels*. The reason is that the behaviour can't be uniquely determined. For example, suppose

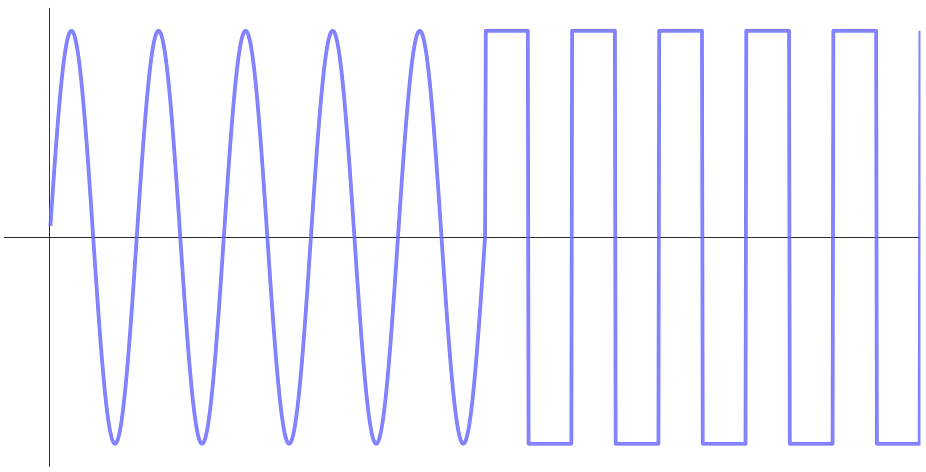
Figure 12: A sine wave followed by a square wave.

that s_1 is a mono sound and s_2 is a stereo sound. In which channel should live the sound s_1 once is seq'd with s_2. Left? Right? Center? Any of these answers is reasonable, so we choose none of them. If you want to seq sounds with different number of channels first use other combinators to make their channels match and determine in which way you want to seq them.

7.2 Add: <+>

The *addition operator*, abbreviated as *add*, adds two sounds together suming their waves. Look Figure 13 to see an example. The sum is done channel-by-channel so channel 1 of the first sound is added to channel 1 of the second, channel 2 of the first sound is added to channel 2 of the second, and so on. Therefore, when adding two sounds the number of channels of both of them must coincide. Otherwise, as with the *seq* operator, the behaviour can't be uniquely determined[4]. Thus, restrictions in *add* are the same as in *seq*:

- Both arguments must share the same *sample rate*.
- Both arguments must share the same *number of channels*.

Note that there is no restriction on both sounds to have the same *duration*. That would be a very strong restriction since having different durations in both is a very common situation. What *add* does is to leave the remainder of the longest sound unchanged.

7.3 Par: <|>

The *parallel operator*, abbreviated as *par*, place two sounds together in different channels. For example, given mono sounds s_1 and s_2, the result will be an

[4]However, it is reasonable to think that this function should work as a `zipWith (+)` for lists. We may take that approach by default, by I think that doing so the user can find in this an unexpected behaviour.

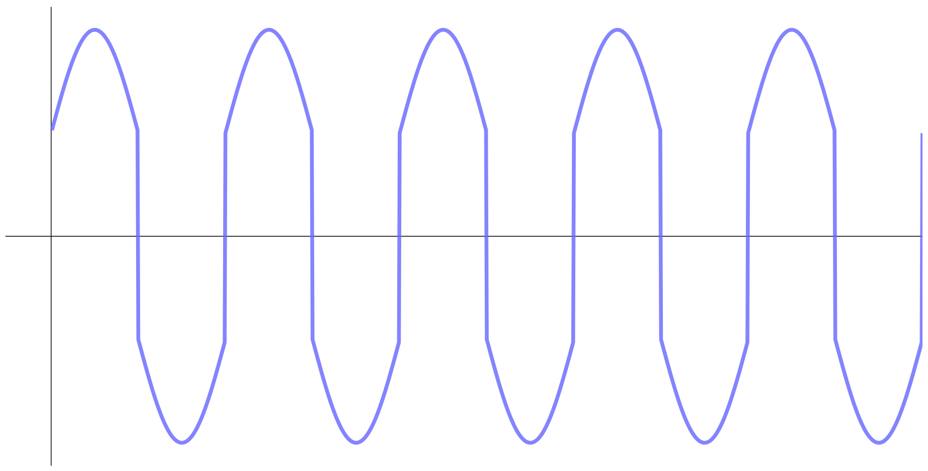

Figure 13: Addition of a sine wave and a square wave.

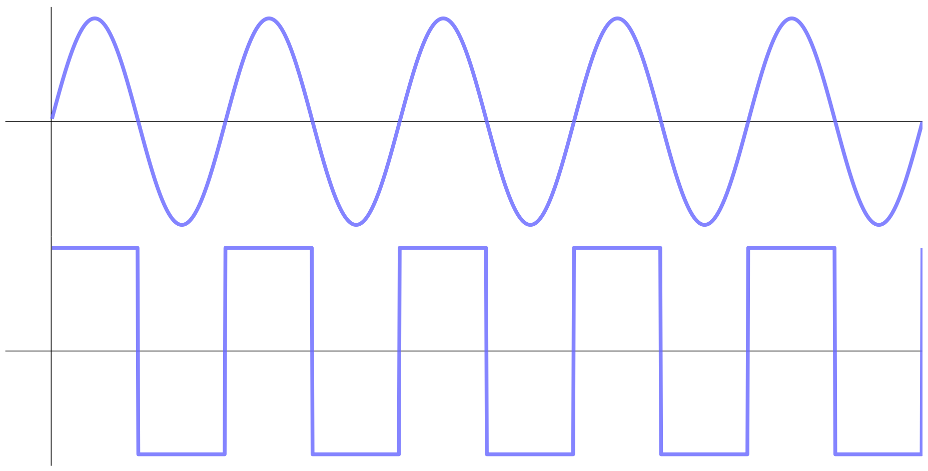

Figure 14: Result of applying *par* to a sine wave with a square wave.

stereo sound with s_1 in the left channel and s_2 in the right channel, as shown in Figure 14. The only restriction of this operator is the usual:

- Both arguments must share the same *sample rate*.

Part V
Sound encodings

Values of type `Sound` can be encoded and stored in files for later use. Different encodings are implemented but their interfaces are very similar. They *at least* implement:

- A datatype for the encoded value.

- A `fromSound` function which takes a `Sound` and translates it to the encoding type.
- A `toSound` function that translates a value of the encoding type to a `Sound`.
- A function `encode` that perform the encoding to binary data as a `ByteString`. Also a variant `encodeFile` that writes the output `ByteString` to a file.
- A function `decode` that perform the decoding from binary data to the encoding type. Also a variant `decodeFile` that reads the input `ByteString` from a file.

Encoders and decoders are lazy, so don't expect a big memory usage from using this functions. However, if your function requires the entire data strictly you can't avoid to have the whole wave data in memory. Be careful.

8 PCM WAVE

One of the encodings implemented in WĂVY is PCM-WAVE [1]. Reading a WAVE file let you access to the following information *before* reading the whole file:

- Number of channels with `numChannels`.
- Sample rate with `sampleRate`.
- Bit depth with `bitsPerSample`.
- Size of the data with `dataSize`.

Part VI
Wavy-Draw

WĂVY-Draw is an extension to WĂVY that allows you to draw sounds from `Sound` values using Cairo [3]. Thus, you can use every Cairo backend to print a sound wave.

WĂVY-Draw was initially designed to serve as a graphic tool to visualize different sound waves along this text. However, I thought that it could be an useful tool by itself, so I made a separated package for it. The interface is simple and we dedicate a small section to it here. Import the `Data.Sound.Draw` module of the WĂVY-Draw package to use it.

9 Usage

Once you have a sound wave the easiest way is to apply `renderFileSound` to it. This will create a PDF file at the specified `FilePath`.

```
drawSine :: IO ()
drawSine = renderFileSound "sine.pdf" $ sine 1 0.9 5 0
```

However, you can customize how the wave is rendered. To do so, the alternative function `renderFileSoundWith` receives an extra argument of type `RenderConfig`. This argument determines how the wave is rendered. A default configuration is provided by `defaultConfig`. You can create your own configuration from the default using record syntax.

```
myConfig :: RenderConfig
myConfig = defaultConfig { signalWidth = 3 }
```

These are the fields of `RenderConfig`:

`rcwidth :: Int`	Width in pixels of the image.
`rcheight :: Int`	Height in pixels of the image.
`rcsteps :: Int`	Distance in pixels between samples. Minimal value: 1.
`signalColor :: RGBA`	Color of the wave line.
`signalWidth :: Double`	Width of the wave line.
`bgColor :: RGBA`	Background color of the image.
`axisColor :: RGBA`	Color for the axis.
`leftFactor :: Double`	Percentage of the image that will be asigned for space at the left of the image.

RGBA colors are specified given the Red, Green, Blue and Alpha components respectively.

```
data RGBA = RGBA { redc   :: !Double
                 , greenc :: !Double
                 , bluec  :: !Double
                 , alphac :: !Double }
```

The alpha component is the opacity factor, where 1 means opaque and 0 transparent.

If instead of a PDF output you are interested in any other type, use `renderSound` or `renderSoundWith`, they return a `Render ()` status that you can use with the Cairo library to render the wave in other formats.

Part VII
Implementation

10 Chunks

The type `Chunks` is the core structure of WĂVY. Every sound data is stored in this form and almost every operation over sounds depends on this type. The type itself is defined at the `Data.Sound.Container.Chunks` module.

A value of type `Chunks` is a possibly empty sequence of chunks of data, each chunk dominated by the constant $\kappa \in \mathbb{N}$[5]. One and only one of the following statements is true for a *valid* value `c` of type `Chunks`:

- `c` is `Empty`.
- `c` consists in an array of at most κ samples followed by `Empty`.
- `c` consists in an array of *exactly* κ samples followed by any *valid* value of type `Chunks` different from `Empty`.

In other words, every chunk in the sequence has κ samples, except the last one which is free to have any number of samples less or equal to κ. A chunk with κ samples is said to be *full*. Every function in WĂVY performing operations over chunks must assume these properties for its arguments and preserve these properties in its results.

The aim of dividing the sound data in chunks is to control the space usage of the library. Tipically, dividing the work in chunks will up end in constant space usage, depending on which operation you are performing over the data. For example, if your operation depends on information contained in several chunks at the same time you cannot avoid to have them in memory, but if your operation can be made *chunk-by-chunk* then it should run in constant space. Also, this approach speeds up some computations like searching.

Each chunk contains a strict array of samples. The size of these arrays is at most κ so we can perform operations strictly within a single chunk without worrying about space usage.

The datatype declaration of `Chunks` is as follows:

```
type Array = A.Vector Sample

data Chunks =
   Empty
 | Chunk {-# UNPACK #-} !Array
         {-# UNPACK #-} !Word32
                         Chunks
```

[5] The value of κ is defined in the `Data.Sound.Container.Chunks` module by `chunkSize`.

Here, `A.Vector` corresponds to a `Vector` of the Vector library [2], and the `Word32` stores the length of the array to avoid its computation. The type declaration does not ensure the properties listed above, so any time a new function is declared for chunks, it must be checked to follow them and preserve them.

From the definition, you can clearly notice a similarity the ByteString library [4]. In fact, the approach is very similar, but WĂVY uses vectors instead of pointers and contains samples instead of bytes.

11 Sounds

The type `Sound` is basically a wrapper of the `Chunks` type with some extra information attached. It is defined in the `Data.Sound.Internal` module. This extra information includes:

(Sample rate) Without the sample rate is impossible to interpret adequately the information stored in the chunks. It also allows quick matching of sample rates between different sounds.

(Number of samples) The total number of samples in the chunks is stored here to avoid its computation. From the moment of its creation, each sound will have attached the correct number, and accesing to the chunks to compute it is unnecessary.

(Number of channels) The number of channels is also stored here to avoid accesing the chunks to compute it.

(Chunks) Chunks containing the sound samples.

Therefore, the datatype definition for `Sound` is as follows:

```
data Sound = S { rate     :: !Word32
               , nSamples :: !Word32
               , channels :: !Int
               , schunks  ::  Chunks
                 }
```

Part VIII
Open issues

Bibliography

[1] *Multimedia Programming Interface and Data Specifications 1.0.* IBM Corporation and Microsoft Corporation, 1991. URL `http://www-mmsp.ece.mcgill.ca/documents/AudioFormats/WAVE/Docs/riffmci.pdf`.

[2] Roman Leshchinskiy. *Vector library*. URL `http://hackage.haskell.org/package/vector`.

[3] Axel Simon and Duncan Coutts. *Cairo library*. URL `http://hackage.haskell.org/package/cairo`.

[4] Don Stewart and Duncan Coutts. *ByteString library*. URL `http://hackage.haskell.org/package/bytestring`.