Fractal Music with Functional Programming

Note in 2023: This was written in 2011 when I was an undergraduate. There are some inaccuracies or areas for improvement.

1 Introduction

This project is about generating music with computer programs. Our aim is to generate pleasant, natural music that sounds like the result of human improvisation or composition. This is not a new idea, but we are re-visiting it from several new angles and building on previous work.

The project revolves around the idea of generating music according to a set of rules, phrased as a so-called L-system. We shall use L-system rules to generate individual tracks. The challenge is to create music that consists of multiple tracks which go well together, both in terms of rhythm and harmony. For that reason, we write combinators which generate the L-system rules themselves.

We do this all in the declarative language Haskell. We chose Haskell because we wish our musical scripts to be short and simple. Furthermore a functional language like Haskell is ideally suited to the symbolic manipulations required for generating rules which themselves generate music.

It is difficult to communicate the evaluation of this work merely on paper – it has to be heard to fully appreciate the achievements of this project. Readers are therefore encouraged to visit `[]` to listen to examples and possibly play with the scripts themselves.

As a preview of what is to come, an example of a complex, multi-track piece of music can be found at `[]`, generated by the first example script in chapter six.

Overview

This dissertation is structured as follows:

We start by reviewing some preliminaries in chapter two, including the fundamentals of music theory. This is intended for Haskell programmers who are not familiar with music theory, but for those who are familiar it gives an indication of the approach used.

In chapter three we describe the basics of generating music in Haskell. The style of presentation here is that of a ‘literate script’, where Haskell code serves as the formal definitions which our prose expands upon.

In chapter four we discuss generative grammars and how they can be manipulated in Haskell. The notion of randomisation presents a small obstacle in Haskell which is overcome by the use of the `IO` monad. We also develop

some useful ‘utility’ functions before presenting Haskell definitions for L-systems.

In chapter five we use our Haskell definitions for L-systems to generate rules which themselves generate music. We start by manually specifying L-system rules, but find that this is too tedious. We develop a strategy for generating the rules themselves whilst ensuring that the tracks have coherent rhythm and harmony, by generating rules for multiple tracks according to the same pattern. We explore refinements for sharing harmonic structure and generating chord patterns.

Having completed our Haskell library for music generation, we present its evaluation in chapter six. The code itself was written as a collection of very small and simple Haskell functions which were tested separately. To evaluate the capability of the project we present some attractive scripts and discuss the music they generate. Each of these is available from the website cited above.

In chapter seven we discuss previous work relating both to other Haskell music libraries and computer generated music.

Finally, in chapter eight we present our conclusions, what was learnt from this project, and possible improvements or directions the work could be taken in the future.

Table of contents

2 What is music?

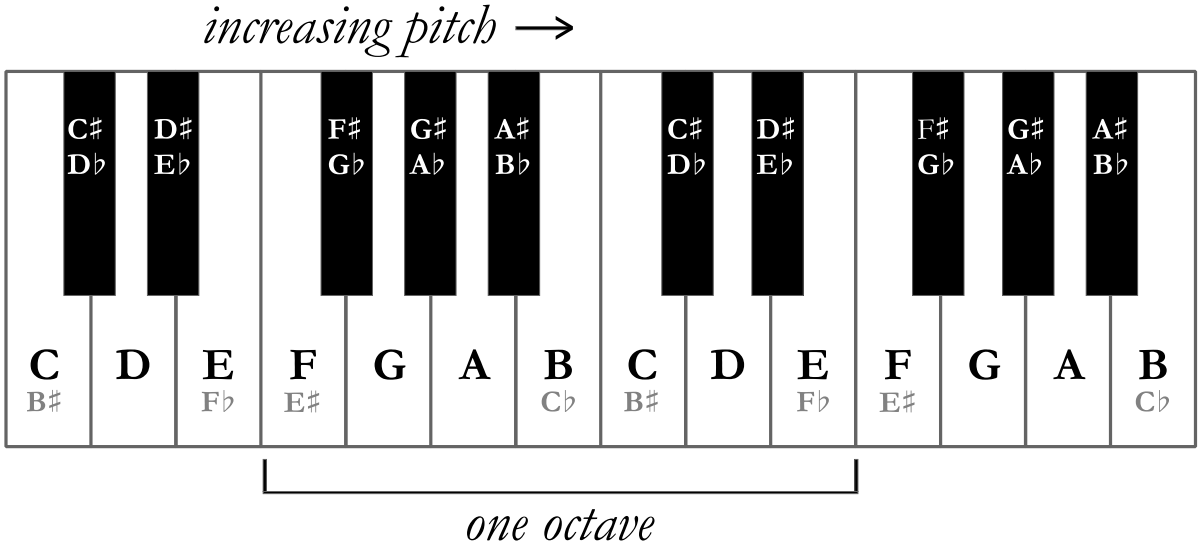
This section serves as an introduction to the musical theory used in this project, aimed at those who lack any musical knowledge at all. For those with some musical knowledge it serves as an indicator of the approach used in this project.

2.1 Some musical background

The definition of music is subjective. Whenever an object vibrates it creates a sound, and the musical term 'pitch' is used to describe how often the vibrations occur (the 'frequency' of the sound) [Apel, p. 8, 678]. The music in this project uses the standard Western twelve tone equal temperament tuning system (also known as equal division of the octave, and hereafter referred to as 'EDO') which is used in almost all modern music [Apel, p. 835].

When a musician plays what he or she may consider to be a only a single pitch, the resulting sound often contains many frequencies which resonate at the same time. The wave heard is the sum of all of these frequencies, and physicists call this 'wave superposition'. The lowest of these is the fundamental frequency, which determines the note, and the rest are higher frequencies, referred to as 'harmonics'. The frequencies of the harmonics are often integer multiples of the fundamental frequency (this can be shown by considering the wavelengths along a vibrating string, but is beyond the scope of this report) [Apel, p. 8, 370]. Frequencies sound more 'harmonious' with each other when the ratio of their respective frequencies is 'simple'. [Apel, p. 371].

In EDO, there are twelve named musical notes. The pitch of a note depends on which 'octave' [Apel, p. 589] it is in – the same note in two contiguous octaves will have exactly twice the frequency in the higher octave as it has in the lower octave. The smallest gap, or 'interval', between two notes is called a semi-tone. A note with frequency f will have another note one semitone above it with frequency $\sqrt[12]{2}\,f$. The notes are labelled with the letters A, B, C, D, E, F and G along with the modifiers ♯ (pronounced 'sharp') and ♭ (pronounced 'flat'), which increase or reduce the pitch of a note by one semi-tone respectively. The note 'middle A' is specified to have a frequency of exactly 440Hz. The diagram below shows a piano keyboard with pitches increasing from left to right - the highlighted octave includes the notes F, F♯ (or G♭), G, G♯ (or A♭), A, A♯ (or B♭), B, C, C♯ (or D♭), D, D♯ (or E♭) and E.



Notes with different names but in the same location on the keyboard (C♯ and D♭, for example) have the same frequency in the EDO tuning system (musicians would consider them to have different pitches). The reason for the colour and position of the notes on a piano keyboard is historical, and is arbitrary in the context of the EDO tuning system that is used today.

The ratios of frequencies between notes in the EDO tuning system are mostly irrational, but are quite close to the simpler ratios that they developed from in past tuning systems [Apel, p. 709] .

2.2 Time

A piece of music is not only defined by the notes that are played, but also when they are played and how long each note lasts for. The length of a note or a rest is usually some simple fraction of a reference length. This reference length is referred to as a ‘bar’ or ‘measure’ [Apel, p. 513]. A bar could be split into four beats of equal duration (quarter notes), which could be split in half (eighth notes) or into quarters or eighths (sixteenth notes and thirty-second notes respectively). Fractions consisting of reciprocals of powers of two and three are most common, often mixed together, but divisions of 5, 7 and 11 do exist in some pieces of music [Apel, p. 729].

2.3 MIDI

This project focuses on creating music without worrying about synthesising sounds. MIDI (short for Musical Instrument Digital Interface) is a protocol used by musical devices to communicate with each other[1]. Music is represented as a series of instructions such as ‘play this note on this instrument at this volume’.

[1] http://www.midi.org

MIDI signals can be communicated in real time between two devices via a MIDI cable, a typical example being a sequencer connected to a synthesiser, with the sequencer telling the synthesiser what sounds to generate. A collection of MIDI signals can also be stored in a file, and this project generates such files. A benefit of this approach is that MIDI files can be imported into most digital-audio workstation software (such as Reaper, Logic, Cubase, Pro-Tools, Samplitude, Live, etc) and edited, assigned to different electronic instruments, or mixed with live recorded audio.

This project includes some Haskell code which can generate multi-track MIDI files. Most modern operating systems have a media player (such as Microsoft Windows Media Player or Apple QuickTime Player) which can play MIDI files through a software synthesiser provided by the operating system.

3 Generating music in Haskell

To create any music at all, we first need the means to generate MIDI instructions from a Haskell program. The script for doing that (named `Note.hs`) is presented in this chapter. We first show how simple notes can be generated, and then how the representation of simple notes is converted to the format used in MIDI files. As described in the introduction to this report, an important design criterion for us is to ensure that the generated music has a natural feel. The last part of this chapter is dedicated to doing that by altering the timings in the generated music.

3.1 Data types for notes: a simple Haskell sequencer

We start with symbols for all the possible pitches within one octave:

```
data PitchName = Bs | C  | Cs | Db | D  | Ds | Eb | E  | Es | Fb | F  | Fs
               | Gb | G  | Gs | Ab | A  | As | Bb | B  | Cb
                      deriving (Eq, Show)
```

And then we can represent any pitch by also specifying its octave:

```
data Pitch = Pitch PitchName Octave deriving Show
type Octave = Int
```

A MIDI note can have one of 128 different pitches, numbered from 0 to 127. Pitch 60 must be ‘middle C’, which is the C immediately below ‘middle A’. Therefore, an `Octave` value must be in the range [0, 10]. The lowest note we can have is C in octave 0, and the highest note we can have is G in octave 10. The functions for converting between a `Pitch` and an `Int` representing a MIDI pitch number are as follows (note that `midiToPitch . pitchToMIDI` is not equal to `id`):

```
pitchToMIDI :: Pitch -> MIDIPitch
pitchToMIDI (Pitch n oc) = 12*oc + case n of
                                     Bs ->  0; C ->  0; Cs ->  1;
                                     Db ->  1; D ->  2; Ds ->  3;
                                     Eb ->  3; E ->  4; Es ->  5;
                                     Fb ->  4; F ->  5; Fs ->  6;
                                     Gb ->  6; G ->  7; Gs ->  8;
                                     Ab ->  8; A ->  9; As -> 10;
                                     Bb -> 10; B -> 11; Cb -> 11;

midiToPitch :: MIDIPitch -> Pitch
midiToPitch n = Pitch p oc
    where p = case n `mod` 12 of
                0  ->  C; 1  -> Cs; 2  ->  D; 3  -> Ds; 4  ->  E; 5  ->  F;
                6  -> Fs; 7  ->  G; 8  -> Gs; 9  ->  A; 10 -> As; 11 ->  B;
          oc = n `div` 12
```

A musical note is not just a pitch – it also has a volume (loudness) and a length of time for which the note is sustained:

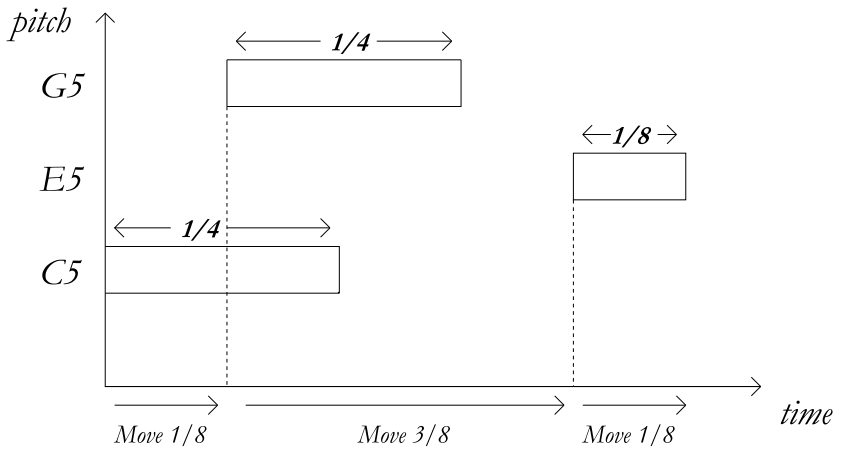
```
data Note = Note Pitch Volume Duration | Move Duration deriving (Eq, Show)
type Volume = Int
type Duration = Ratio Int
```

The volume must be in [0, 127], which is the range of volumes used by the MIDI protocol. The type `Duration` will be used throughout the project to denote musical lengths of time – it uses the type `Data.Ratio`, which takes an `Integral` type as a parameter to create an exact rational number type. This is better than using floating point numbers as they may not be accurate enough – we would not want to end up with notes that are slightly misaligned in time. We can construct a `Duration` type using the `Data.Ratio.%` constructor – the value ¼ can be constructed by giving `(1%4)`, for example. A `Duration` should always be non-negative.

The `Move` constructor is an instruction to advance the 'current' time. The convention used in the code is that a list of `Note` is interpreted as all notes starting at the same instant in time until a `Move` is encountered which advances the current time by its `Duration`. Consider the following fragment of music:

```
[Note (Pitch C 5) 80 (1%4), Move (1%8), Note (Pitch G 5) 80 (1%4), Move (3%8),
 Note (Pitch E 5) 80 (1%8), Move (1%8)]
```

On a graph of pitch against time with notes represented as blocks, it denotes the following:



This is a good intuitive representation of music, and is often used in MIDI sequencing software where the user is able to 'draw' notes as a method of composition.

To convert a list of notes into a piece of music, we need to know which MIDI channel we would like those notes to be heard through. The MIDI protocol has 16 channels, numbered 0 to 15. Each channel can be assigned to a different MIDI instrument, referred to as a 'program', at any time. However, note events on channel 9 map to specific percussion sounds regardless of what program is assigned for that channel. These percussion and drum sounds cannot be accessed on any other channel regardless of program setting. There are 128 named MIDI instruments, numbered from 0 to 127. The `Track` type is (initially) defined as:

```
data Track = Track Channel [Note]
type Channel = Int
type Program = Int
```

A `Channel` value must be in [0, 15] and a `Program` value must be in [0, 127]. The function `writeMIDIFile` is defined in `Note.hs`:

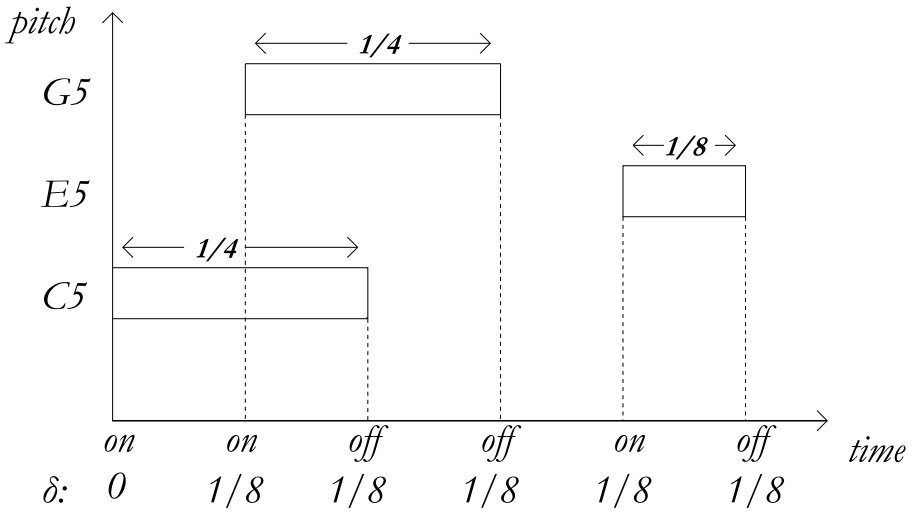
```
writeMIDIFile :: FilePath -> [Track] -> [(Channel, Program)] -> BPM -> IO ()
type BPM = Int
```

`FilePath` is a type synonym defined in the GHC Prelude as `String`, which is itself a type synonym for `[Char]`, and denotes the name of the MIDI file which will be created. `writeMIDIFile` also takes a `[Track]` and writes each `Track` to a separate track in the MIDI file. These tracks are played back simultaneously. Events assigning programs to channels are written to the file as given in `[(Channel, Program)]`. Finally, `BPM` specifies the global speed of the MIDI file in 'beats per minute'. A beat is a quarter note (with a `Duration` of `(1%4)`), so a

'beats per minute' value of 60 would mean that a note with a `Duration` value of `(1%4)` would last for exactly one second.

3.2 MIDI Data Representation

Representing music with the type `[Note]` is convenient for creating and generating music, but it is slightly different to the way MIDI events are stored in a file. A single musical note actually translates into two MIDI events – one for the start of the note, and one for the end. Each MIDI note has a 'delta time', which denotes how long to wait since the last event before this event should happen. The previous music example would look something like this when converted to MIDI events:



A MIDI event is represented by the `MIDIEvent` type:

```
data MIDIEvent = MIDIEvent Bool MIDIPitch Volume DeltaTime deriving Show
type MIDIPitch = Int
type DeltaTime = Int
```

The `Bool` value denotes whether this is a 'note on' or 'note off' event. The `MIDIPitch` and `Volume` values represent the pitch and volume of the event respectively, while the `DeltaTime` value represents the time to wait since the last event before this event should happen, as explained above. The `MIDIEvent` type does not have a value for its channel, as the code uses a channel value from the `Track` type when converting a `MIDIEvent` to its binary representation.

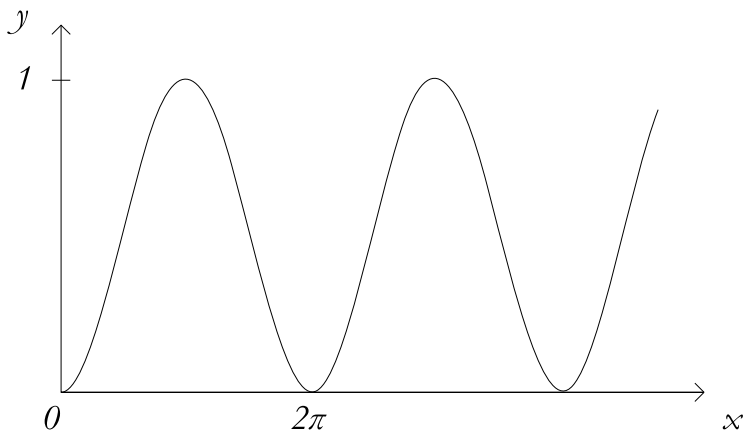
The process of converting `[Note]` to `[MIDIEvent]` works by generating `[(AbsoluteTime, MIDIEvent)]` with all `DeltaTime` values in `MIDIEvents` set to `0`,

and the `AbsoluteTime` value representing the absolute location (relative to the beginning of the track) of this event. This list is then sorted by the first element of the pairs, and then converted to `[MIDIEvent]` with correct `DeltaTime` values.

3.3 Human time feel

When the computer plays a MIDI file, the note timing is extremely accurate. Human timing is much less accurate, but with a good musician this inaccuracy is not random and contributes to the 'time feel' of the music. We can attempt to simulate this 'time feel' by delaying notes by varying amounts depending on their location in time. This is often referred to as adding 'swing' to a track.

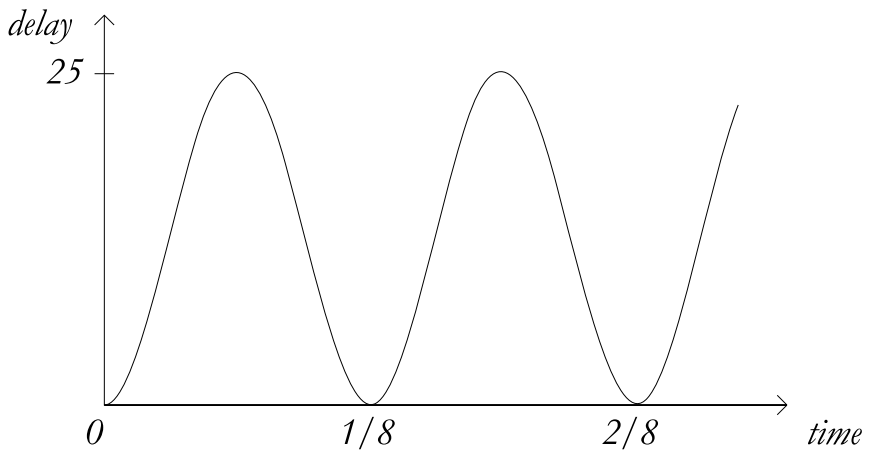
Consider a graph of the function $-\frac{1}{2}\cos x + \frac{1}{2}$:



We can interpret the *x* axis as representing time, and the *y* axis as representing how much to delay an event at that given time. Then we can scale the x axis for various different durations, and scale the y axis for the maximum permitted delay. This gives us the following formula:

$$maxDelay \times \left(-\frac{1}{2}\cos\left(\frac{2\pi \times (currentTime \bmod periodTicks)}{periodTicks}\right) + \frac{1}{2}\right)$$

maxDelay is the maximum amount that an event can be delayed by. *currentTime* is the current time in MIDI ticks, and *periodTicks* is the length in MIDI ticks of the swing period (which is calculated from a `Duration` value, using a globally defined constant for the number of MIDI ticks in a quarter note). For example, given a swing period of `(1%8)` and a *maxDelay* of 25 MIDI ticks, we would have the following graph:

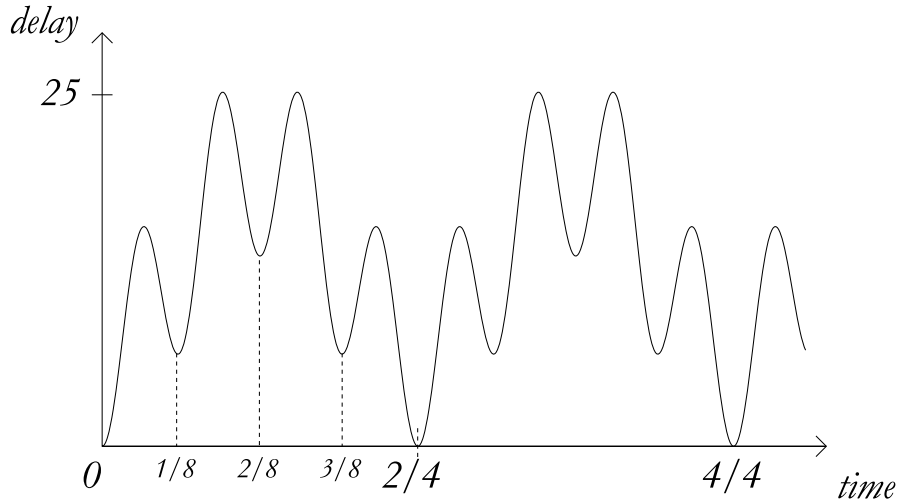


Each event on the *time* axis is delayed by the amount on the *delay* axis – the event's *delay* amount is simply added to its absolute location in time.

However, for a given track we may wish to add 'swing' for more than one swing period. In this case, we simply sum the above formula for each swing period, using its corresponding *maxDelay*:

```
type DelayProfile = [(Duration, DeltaTime)]
```

An example of a `DelayProfile` would be the list `[(2%4, 25), (1%8, 25)]`, which would result in the following delay graph:



The function `pocket` takes a `DelayProfile` and an `AbsoluteTime`, the latter representing the location in time of some event, and applies the `DelayProfile` to calculate a new location for that event. (The function is named 'pocket' due to the musical term 'playing in the pocket', which means to play with good 'time feel').

```
pocket :: DelayProfile -> AbsoluteTime -> AbsoluteTime
pocket [] currentTime = currentTime
pocket ((period, maxDelay):profiles) currentTime =
 (+ pocket profiles currentTime) $ r $ (*(f maxDelay :: Double)) $ (+0.5) $ (*(-0.5)) $
  cos $ (2*pi * f (currentTime `mod` periodTicks) :: Double) / (f periodTicks :: Double)
 where periodTicks = durationToMIDITicks period
       f = fromIntegral
       r = round
```

The code is made complicated by the explicit conversions between `AbsoluteTime` (which is a type synonym for `Int`) and `Double` (which is a 64 bit floating point number type). The Haskell functions `fromIntegral` and `round` are defined in the Prelude. `pocket` essentially just implements the above formula for each pair in the delay profile, and is called by the MIDI file generation.

Delay profiles are specified in the `Track` type, which we can expand:

```
data Track = Track Channel [Note] | DelayTrack DeltaTime DelayProfile Channel [Note]
             deriving Show
```

The first `DeltaTime` value of a `DelayTrack` is a constant delay added to the entire track after the `DelayProfile` has been applied.

4 Generative grammars in Haskell

This section discusses how to generate random numbers in Haskell, and then we present some utility functions before introducing an approach for using generative grammars in Haskell.

4.1 Haskell and random numbers

Suppose that there existed some Haskell library function

```
rand :: Int -> Int -> Int
```

generating a random number in a given range. For example, `rand 1 6` would give a random number in the range [1, 6]. Then consider the following example of that function being used:

```
randomSum = let r = rand 1 6 in r + r
```

Since Haskell is a lazily evaluated referentially transparent pure functional language, it is entirely unknown as to when a value of `r` will be computed and how many times it will be computed. It may or may not be the case that any particular compiler or interpreter pre-computes the value of `r` so that `randomSum` always gives an even number.

Similarly, if we changed the definition:

```
randomSum = (rand 1 6) + (rand 1 6)
```

We are still not sure whether or not the interpreter/compiler will apply some program transformation/optimisation so that one value of `rand 1 6` is calculated and used twice.

To get over this ambiguity, we have to thread a notion of state through our program, and this is achieved using the `IO`[1] type which is an instance of the `Monad` class[2]. The code is written using the '`do`' keyword, which is simply syntactic sugar – GHC re-writes this code into code using `>>=`, `>>` and `let` statements or anonymous functions, using some very simple rules[1]. Code which

[1] http://www.haskell.org/haskellwiki/IO_inside

[2] http://www.haskell.org/tutorial/monads.html
http://haskell.org/ghc/docs/latest/html/libraries/base/Control-Monad.html

uses 'do notation' may look like imperative code, but it is not and thinking of it in this way can lead to fundamental misunderstandings (and code that looks like it should work but does not even compile).

GHC provides a global random number generator in System.Random, which can be seeded and accessed from any part of the program. The functions seed and rand, defined in NoteUtil.hs, make this more convenient:

```
import System.Random

seed :: Int -> IO ()
seed x = setStdGen (mkStdGen x)

rand :: Int -> Int -> IO Int
rand x y = getStdRandom $ randomR (x, y)
```

If we wanted the randomSum function to return the sum of two different random numbers in [1, 6], we could re-write it as:

```
randomSum :: IO Int
randomSum = do r1 <- rand 1 6
               r2 <- rand 1 6
               return $ r1 + r2
```

or without using do notation:

```
randomSum = rand 1 6 >>= \r1 ->
            rand 1 6 >>= \r2 ->
            return $ r1 + r2
```

4.2 Some utility functions

There are some other common tasks for which we require useful functions (whose usage will be presented later) and many of these functions are in the source file NoteUtil.hs. One is shuffle which randomises a list by taking a random element from the list, and forming a new list with that element at the start followed by the rest of the elements shuffled in the recursive case:

[1] http://book.realworldhaskell.org/read/monads.html#monads.do

```
shuffle :: [a] -> IO [a]
shuffle [] = return []
shuffle xs = do r <- rand 0 (length xs - 1)
                rest <- shuffle $ take r xs ++ drop (r+1) xs
                return $ xs !! r : rest
```

Another useful function is `randElem`, which picks a random element from a list:

```
randElem :: [a] -> IO a
randElem xs = do r <- rand 0 (length xs - 1)
                 return $ xs !! r
```

`sometimesMap` takes `p` in [0, 10] and each element in the list has a `p`-in-10 chance of having `f` applied to it:

```
sometimesMap :: Int -> (a -> a) -> [a] -> IO [a]
sometimesMap _ _ [] = return []
sometimesMap p f (x:xs) = do r <- rand 1 10
                             rest <- sometimesMap p f xs
                             if r <= p then return $ (f x) : rest else return $ x : rest
```

Some useful functions for manipulating the pitch of a `Pitch`, `[Pitch]` and `[Note]` respectively (note that `shiftPitch (-n) . shiftPitch n` is not equal to `id`):

```
shiftPitch :: Int -> Pitch -> Pitch
shiftPitch n = midiToPitch . (+n) . pitchToMIDI

shiftPitchPitches :: Int -> [Pitch] -> [Pitch]
shiftPitchPitches n = map (shiftPitch n)

shiftPitchNotes :: Int -> [Note] -> [Note]
shiftPitchNotes n = map (\note -> case note of
                                    (Note pitch volume duration) ->
                                        Note (shiftPitch n pitch) volume duration
                                    (Move duration) -> Move duration )
```

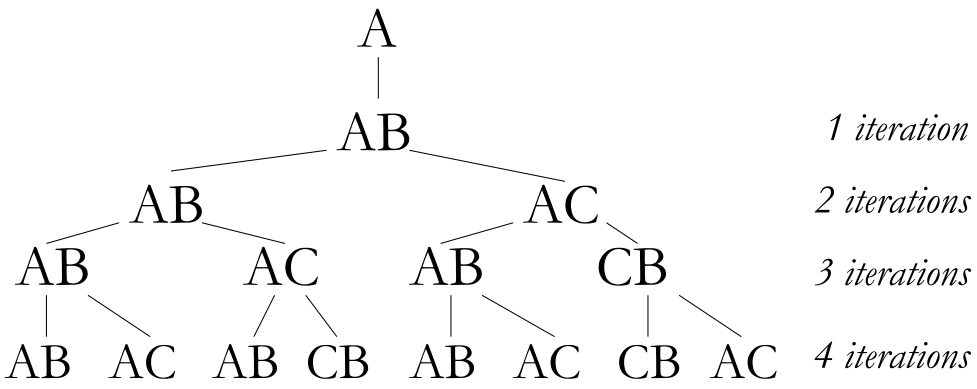
Rotating a list is an idea that is used often later on in the code for generating rules automatically:

```
rotate :: [a] -> [a]
rotate [] = []
rotate (x:xs) = xs ++ [x]

rotations :: [a] -> [[a]]
rotations xs = rts (length xs) xs
    where rts :: Int -> [a] -> [[a]]
          rts _ [] = []
          rts 0 _ = []
          rts n xs = let y = rotate xs in y : rts (n-1) y
```

4.3 Generative Grammars

The project uses ideas from generative grammars and L-systems to generate music. A tree with increasingly concrete abstract representations of music is built from the top down by rules which are automatically generated, and this tree may at some point split into multiple related tracks.

An L-system (short for Lindenmayer system, after Aristid Lindenmayer [TABoP]) consists of a set of symbols, a set of production rules and a starting string [TABop, p. 4]. For example, if we had the set of symbols {A, B, C}, production rules {A → AB, B → AC, C → CB} and starting string A, we can apply the rules as shown:



After four iterations we obtain the string ABACABCBABACCBAC.

L-systems naturally tend to produce strings with repeating patterns, or repeating patterns with small modifications, and that is what I conjecture is one of the properties of music that humans find pleasant (repeating patterns provide a structure to follow along and engage with, and modifications provide surprise and interest). If you were to interpret the string above as a list of musical notes to play in order, you may be able to hear the patterns of repetition and modification.

In Haskell we can represent an L-system as a list, but I have used a more general type class:

```
class LSystem s where
    grow :: (Applyable a b) => [Rule a b] -> s a -> IO (s b)
    leaves :: s a -> [a]
```

Before this makes sense, we need to introduce the `Applyable` type class and the `Rule` type. The `Applyable` type class is simply a mechanism for providing default rules:

```
class (Eq a) => Applyable a b where
    defaultRule :: a -> [b]
```

`defaultRule` is usually similar to the identity function. For example:

```
import Debug.Trace

instance Applyable Int Int where
    defaultRule = trace "defaultRule Int Int called" (:[])
```

`defaultRule` has type `Int -> [Int]` in this instance. The type `Rule` is defined as:

```
data (Applyable a b, Eq a) => Rule a b = Rule a (WeightedList [b]) deriving Show
```

A `Rule a b` will match against its `a` value (using `==` from the `Eq` class) and we choose a `[b]` from the `WeightedList [b]`. (The `WeightedList` type synonym is discussed below). Rules are applied using the `applyRules` function, which takes a list of `Rules` `rs` and finds the first one which matches against `x` (or uses `defaultRule`) to give a list of `b` (`IO [b]`):

```
applyRules :: (Applyable a b) => [Rule a b] -> a -> IO [b]
applyRules rs x =
    case findRule x rs of
        Nothing -> return $ defaultRule x
        Just r -> chooseRule r
        where findRule :: (Applyable a b) => a -> [Rule a b] -> Maybe (Rule a b)
              findRule _ [] = Nothing
              findRule n ((Rule r p):rs) = if r == n then Just (Rule r p) else findRule n rs
              chooseRule :: (Applyable a b) => Rule a b -> IO [b]
              chooseRule (Rule n xs) = weightedRandElem xs
```

If we choose to use lists as the basis for our L-system, then we need the following instance:

```
instance LSystem [] where
    grow rs = fmap concat . unwrap . map (applyRules rs)
    leaves = id
```

`grow` simply applies one iteration of the rules to an entire list. (The `unwrap` function is discussed below). It is convenient to write code that can use L-systems which have different underlying structures. An example for trees:

```
data Tree a = Node [Tree a] | Leaf a deriving Show

instance LSystem Tree where
    grow rs (Node xs) = fmap Node $ unwrap $ map (grow rs) xs
    grow rs (Leaf x) = do children <- applyRules rs x
                          return $ Node $ map Leaf children
    leaves (Node xs) = concatMap leaves xs
    leaves (Leaf x) = [x]
```

We then have `growN`, which can apply multiple iterations of rules:

```
growN :: (LSystem s, Applyable a a) => Int -> [Rule a a] -> s a -> IO (s a)
growN 0 _  xs = return xs
growN n rs xs = do growOnce <- grow rs xs
                   growN (n-1) rs growOnce
```

Much of the interesting parts of this project simply involve combinators which generate `Rule`s which are then handed to `grow` or `growN`.

The `Unwrap` class is a convenient class for 'joining' together the `IO` types in some structure:

```
class Unwrap u where
    unwrap :: u (IO a) -> IO (u a)

instance Unwrap [] where
    unwrap [] = return []
    unwrap (x:xs) = do first <- x
                       rest  <- unwrap xs
                       return $ first : rest
```

It is quite a common case in the code that we have some type `[IO a]` when in fact we would like to have `IO [a]`.

The type synonym `WeightedList` is used throughout the code for representing the frequency that some item in a list should be chosen. For example, in the list `[(0, 5), (1, 10), (2, 20)]`, the value `20` is chosen twice as often as the value

`10` by `weightedRandElem`. The function `expandWeighted` would expand that list as `[10, 20, 20]`, and the function `expandUnWeighted` would expand that list as `[5, 10, 20]`. Note that `expandUnWeighted` still includes the value 5 in its expansion of the list and this behaviour is relied upon in the code. The definitions of `WeightedList` and its associated functions are as follows:

```
type WeightedList a = [(Int, a)]

expandWeighted :: WeightedList a -> [a]
expandWeighted [] = []
expandWeighted ((n, x):xs) = take n (repeat x) ++ expandWeighted xs

expandUnWeighted :: WeightedList a -> [a]
expandUnWeighted = map snd

weightedRandElem :: WeightedList a -> IO a
weightedRandElem = randElem . expandWeighted
```

5 Generating Fractal Music

In this chapter, we start by generating very simple music with L-systems, and then progress on to creating complex multi-track pieces of music with coherent structure and harmony.

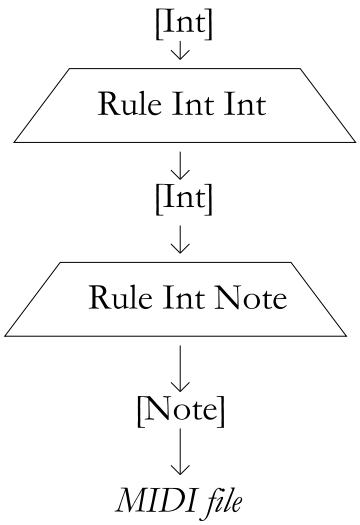
5.1 Using numbers to represent abstract notes

Often, a piece of music does not use all twelve musical pitches, but is restricted to some subset of them. This selection is called a ‘scale’ [Apel, p. 753]. The chromatic scale consists of all twelve musical pitches. The C Major scale consists of the pitches C, D, E, F, G, A and B.

A scale (or any piece of music) can be ‘transposed’ [Apel, p. 860] by shifting each of its pitches by a given number of semitones, as in the three `shiftPitch` functions above.

The concept of a ‘key’ [Apel, p. 450] is similar to that of a scale, except that a key has a ‘root’ [Apel, p. 740] note, and each note in the scale has an associated ‘chord’ [Apel, p. 162]. (A chord is simply two or more notes played at the same time, and is discussed later). The A (natural) minor scale has the same notes as the C Major scale, but the root note is A instead of C. If you play the notes C, D, E, F, G, A and B in order, it will sound quite different to playing the notes A, B, C, D, E, F and G in order. Some pieces of music stay in one key and use only one scale. Some other pieces of music change their key or scale occasionally, and some other pieces of music have no concept of key or scale, or use many different scales.

A simple way to generate music with an L-system is to choose a scale, and then use natural numbers as the set of symbols in the L-system, and then map them to musical notes. For example, we could use the integers in [0, 6] to map to the notes in C Major. The process would look something like this:



A simple example follows:

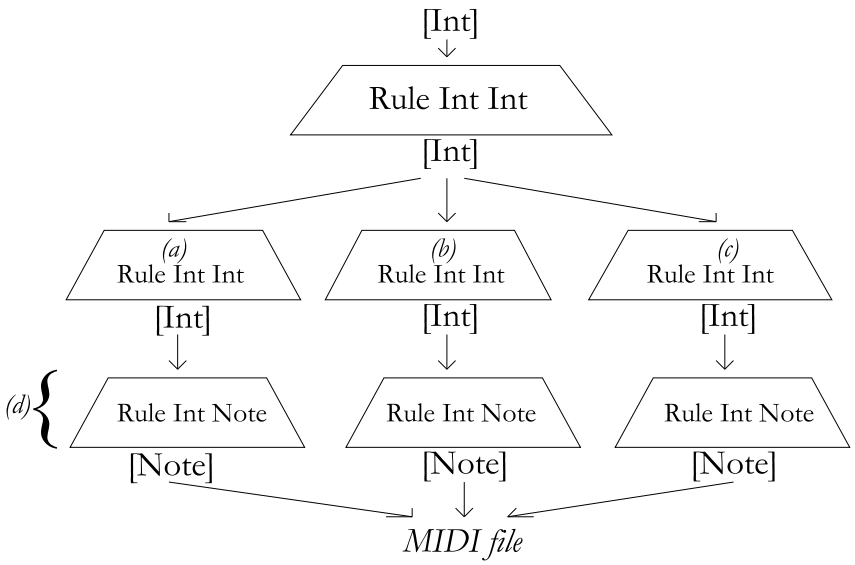
```
intRules :: [Rule Int Int]
intRules = [Rule 0 [(1, [0, 2, 4])],
            Rule 1 [(1, [1, 3, 2])],
            Rule 2 [(1, [4, 5, 6])],
            Rule 3 [(1, [5, 4, 3])],
            Rule 4 [(1, [0, 2, 1])],
            Rule 5 [(1, [6, 6, 6])],
            Rule 6 [(1, [4, 3, 2])] ]

noteRules :: [Rule Int Note]
noteRules = [Rule 0 [(1, [Note (Pitch C 5) 80 (1%8), Move (1%8)])],
             Rule 1 [(1, [Note (Pitch D 5) 80 (1%8), Move (1%8)])],
             Rule 2 [(1, [Note (Pitch E 5) 80 (1%8), Move (1%8)])],
             Rule 3 [(1, [Note (Pitch F 5) 80 (1%8), Move (1%8)])],
             Rule 4 [(1, [Note (Pitch G 5) 80 (1%8), Move (1%8)])],
             Rule 5 [(1, [Note (Pitch A 5) 80 (1%8), Move (1%8)])],
             Rule 6 [(1, [Note (Pitch B 5) 80 (1%8), Move (1%8)])] ]

manualTest = do ints <- growN 4 intRules [0]
                notes <- grow noteRules ints
                writeMIDIFile "oldTest1.mid" [Track 0 notes] [] 140
```

This generates a simple (and relatively uninteresting) piano melody in the file `oldTest1.mid`. By the end of this chapter, we will have developed music that is much nicer.

5.2 Maintaining time across multiple tracks

We could expand this method to generate multiple tracks which are related to each other by specifying some rules of type `Rule Int Int` to generate a long starting string, and then specify different sets of rules to generate multiple

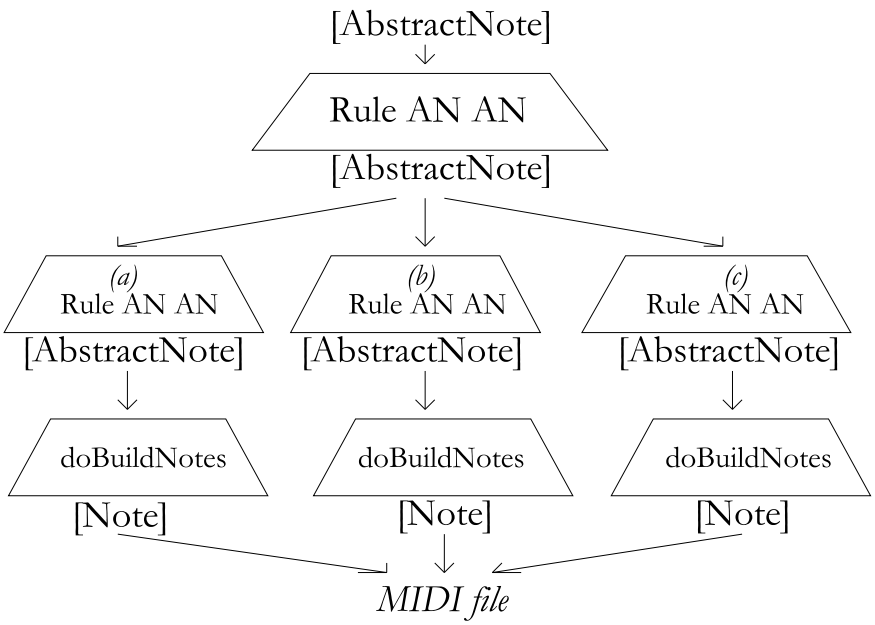
tracks with the same starting string. The tracks are then played back at the same time. This might look something like this:



However, we would have to be very careful when specifying rules for (a), (b) and (c) to ensure that the differing tracks were of the same length and maintained their 'relatedness' to each other. Specifying all of these rules manually is tedious and it would be better to do it automatically. To do this, we need a better way to represent an abstract note:

```
type AbstractNote = (Int, Duration)
```

Instead of using numbers to represent notes, and then deciding later on what these numbers actually mean, we use numbers to represent pitches and a `Duration` value to represent the note's length. The process then looks like this:



The benefit of using a type of `(Int, Duration)` is that we can generate rules which, on each iteration, multiply the length of the track by a given multiplier m by replacing each abstract note with length d by one or more abstract notes whose lengths sum to m × d (see [KJ89]). If the rules used in (a), (b) and (c) have the same multiplier and the same number of iterations, then not only will the tracks be the same length, but they will be extremely 'related' to each other, in the sense that the same combinations of musical parts will occur together repeatedly across the separate tracks. This can also be thought of as having separate tracks with the same musical structure.

Rule generation is achieved with the `doBuildRules` function, which can generate such rules automatically:

```
buildRules :: Duration -> WeightedList Int -> WeightedList Duration
               -> IO [Rule AbstractNote AbstractNote]
buildRules multiplier notes durations =
 do noteTable   <- fmap (map (\xs -> (head xs, cycle $ rotate xs))) $
                     fmap rotations $ shuffle $ expandWeighted notes
    lengthTable <- buildLengthTable multiplier durations
    return [rule n d noteTable lengthTable |
            n <- expandUnWeighted notes, d <- expandUnWeighted durations]
      where rule :: Int -> Duration -> [(Int, [Int])] -> [(Duration, [Duration])]
                    -> (Rule AbstractNote AbstractNote)
            rule n d noteTable lengthTable = let ns = unJust $ lookup n noteTable
                                                 ds = unJust $ lookup d lengthTable
                                             in Rule (n, d) [(1, zip ns ds)]
```

`buildRules` takes a multiplier, a weighted list of `Int` and a weighted list of `Duration`. The multiplier denotes the proportional increase in the sum of

`Durations` of the list of `AbstractNote` after one iteration of the (deterministic) rules that `buildRules` generates, as explained above.

The weighted list of `Int` must include every integer that appears in the list of `AbstractNote` that the rules generated by `buildRules` are applied to, otherwise `defaultRule` may be used. The weighted list of `Int` represents the pitches that the generated rules should match against and substitute for, and the weightings mean that we can specify that we much prefer some pitches to others. The list may include some values with a weighting of zero, meaning rules will be generated matching that value but no rules will be generated substituting that value.

The strategy for generating rules using the weighted list of `Int` is to apply `expandWeighted` and then randomise the order using `shuffle`, and then create a lookup table (`noteTable`) based on an infinite cyclical list. Consider the list `[(1, 0), (1, 1), (2, 2), (1, 3)]`. After `expandWeighted` is applied, we will have the list `[0, 1, 2, 2, 3]`. After `shuffle` is applied, we might have the list `[1, 0, 2, 3, 2]`. From this, our lookup table conceptually contains the rules {1 → 0232102321... , 0 → 232... , 2 → 321... , 3 → 210... , 2 → 102...}. (Note that, in fact, two different production rules matching 2 will be generated which is slightly wasteful, but only one will ever be used since `applyRules` finds the first rule which matches). The production rules substitute infinite strings, which Haskell makes easy.

The next step is to use the weighted list of `Duration` to create a lookup table (`lengthTable`) of production rules for `Duration` values. Again, the weighted list of `Duration` must include every Duration that appears in the list of `AbstractNote` that the rules generated by `buildRules` are applied to, otherwise `defaultRule` may be used. The weighted list of `Duration` must also include a `Duration` d with a non-zero weighting which, for all other `Durations` e in the list with a non-zero weighting there must exist a natural number m such that m×d = e. The strategy for creating `lengthTable` is used in many other functions throughout the code, and so has its own function `buildLengthTable`:

```
buildLengths :: WeightedList Duration -> Duration -> IO [Duration]
buildLengths _ 0 = return []
buildLengths durations d = do first <- randElem $ filter (<=d) $ expandWeighted durations
                              rest  <- buildLengths durations (d-first)
                              return $ first : rest

buildLengthTable :: Duration -> WeightedList Duration -> IO [(Duration, [Duration])]
buildLengthTable multiplier durations =
 unwrap $ map (\d -> do { ls <- buildLengths durations (d*multiplier); return (d, ls) })
  $ expandUnWeighted durations
```

The easiest way to explain this is with an example. Consider the weighted list of `Durations` `[(1, 1%4), (2, 1%8), (1, 1%16)]` with a multiplier of `2`. After `expandWeighted` is applied, we have the list `[1%4, 1%8, 1%8, 1%16]`. We might end up with production rules which conceptually look like {`1%4` → `1%8` `1%8` `1%4`, `1%8` → `1%8` `1%16` `1%16`, `1%16` → `1%8`}. Each production rule will have been generated by `buildLengths`, which uses the weighted list of `Duration` to make the target `d` recursively, which is why it is important that the smallest `Duration` (`1%16` in this case) must be able to 'make' all the others.

Once `buildRules` has a lookup table for both `Int` and `Duration`, we can finally combine them to create a list of `Rule AbstractNote AbstractNote` by using a list comprehension and the rule function to create a rule for every `Int` and `Duration` pair (`AbstractNote`) that we can encounter.

`buildRules` generates rules, but does not apply them to an L-system. For convenience, I have written a function `doBuildRules`:

```
doBuildRules :: (LSystem s) => Duration -> WeightedList Int -> WeightedList Duration
                               -> Int -> s AbstractNote -> IO (s AbstractNote)
doBuildRules multiplier notes durations n lsystem =
    do rules <- buildRules multiplier notes durations
       growN n rules lsystem
```

An example of this:

```
aTestIntProfile = [(1, 0), (1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (1, 6)]
aTestDurationProfile = [(1, 1%4), (1, 1%8), (1, 1%16)]
aTestMelodyPitches =
    [Pitch C 5, Pitch D 5, Pitch E 5, Pitch F 5, Pitch G 5, Pitch A 5, Pitch B 5]
aTestBassPitches = shiftPitchPitches (-24) aTestMelodyPitches

aTest x =
 do seed x
    structure <- doBuildRules 2 aTestIntProfile aTestDurationProfile 2 [(0, 1%4)]
    abstractMelody <- doBuildRules 2 aTestIntProfile aTestDurationProfile 2 structure
    abstractBass <- doBuildRules 2 aTestIntProfile aTestDurationProfile 2 structure
    writeMIDIFile "oldTest2.mid" [Track 0 (doBuildNotesSimple aTestMelodyPitches abstractMelody),
                                  Track 1 (doBuildNotesSimple aTestBassPitches abstractBass)]
        [(1, 33)] 120
```

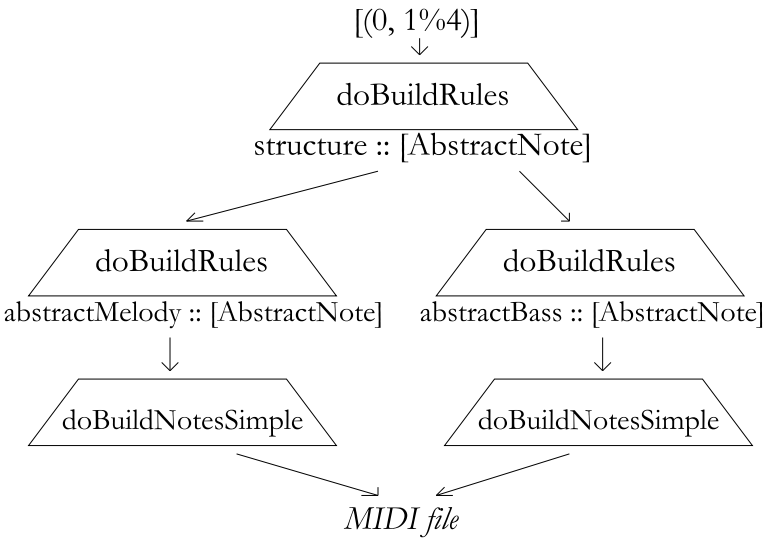
Here, we generate a two track piece of music with a bass line and melody in the file `oldTest2.mid`. The value `[(1, 33)]` sets channel one to the sound of an electric bass. The seed value `x` enables us to generate lots of examples and recall previous examples by giving the same seed value. However, this may not be stable across small code changes or different versions of GHC. The `doBuildNotesSimple` function is very simple and uses a list of `Pitch` to convert a list of `AbstractNote` into real `Note` values:

```
doBuildNotesSimple :: [Pitch] -> [AbstractNote] -> [Note]
doBuildNotesSimple pitches =
    concat . map (\(n, d) -> [Note (pitches !! n) defaultMIDIVolume d, Move d])
```

`defaultMIDIVolume` is defined in `Note.hs`:

```
defaultMIDIVolume = 80 :: Volume
```

A visualisation of what is going on here:



We end up with a melody and bass line which seem somewhat related to each other, and are certainly the same length. The tracks do not clash harmonically because only the pitches in the C Major scale are used.

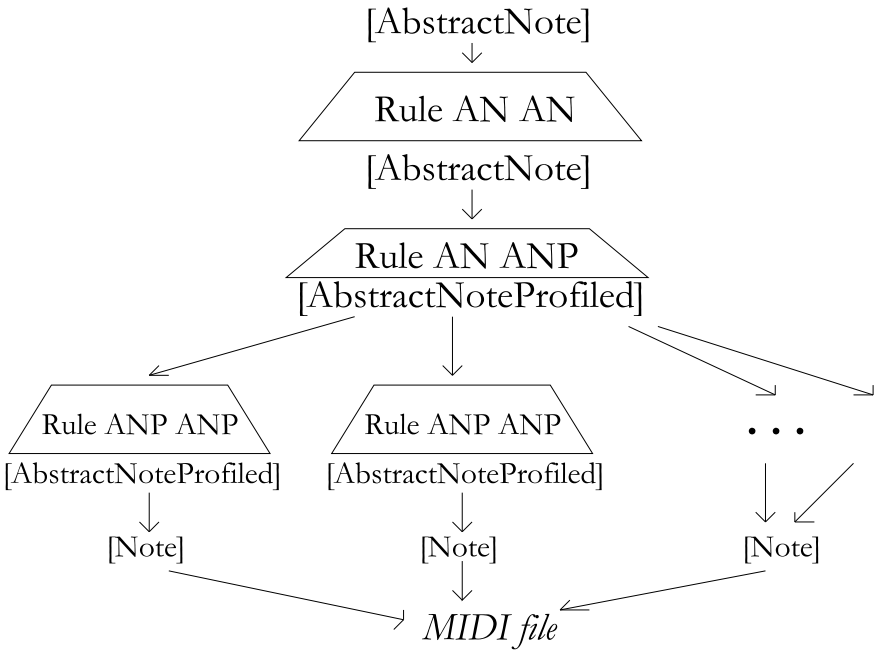
5.3 Interpreting the structure as a chord pattern

The music generated does not exhibit a lot of harmonic coherence. In most music, the structure is based around a 'chord pattern'. A chord is two or more notes which are played at once [Apel, p. 162]. Often a piece of music has short periods of time where some notes are much more important than others. The melody may be based around those notes and some instrument may simply play the chord notes when they change.

To do this, we generate a structure and then interpret it as a chord pattern, after introducing a new type synonym:

```
type AbstractNoteProfiled = (AbstractNote, WeightedList Int)
```

Here, the weighted list of `Int` denotes which pitch numbers are more or less likely to be used in production rules. Our strategy for generating music will then look like this:



Here, the structure is a list of `AbstractNoteProfiled`. When the structure is expanded into multiple tracks, the production rules generated by `buildRulesProfiled` (described below) use the profile on each iteration, so that sections of music across multiple tracks use similar notes.

An L-system of type `AbstractNote` is converted into one of type `AbstractNoteProfiled` using rules from the `addProfiles` function:

```
addProfiles :: Int -> WeightedList Duration -> WeightedList (WeightedList Int)
                -> IO [Rule AbstractNote AbstractNoteProfiled]
addProfiles notesRange durations profiles =
 unwrap [rule n d | n <- [0..(notesRange-1)], d <- expandUnWeighted durations]
 where rule :: Int -> Duration -> IO (Rule AbstractNote AbstractNoteProfiled)
       rule n d = do chordProfile <- randElem $ expandWeighted profiles
                     return
                      $ Rule (n, d)
                         [(1, [((n, d), transposeProfile n notesRange chordProfile)])]

transposeProfile :: Int -> Int -> WeightedList Int -> WeightedList Int
transposeProfile places notesRange profile =
    map (\(x, y) -> (x, (y+places) `mod` notesRange)) profile
```

The `profiles` argument of type `WeightedList (WeightedList Int)` is the key here. Each `WeightedList Int` is a chord profile, and `addProfiles` builds a lookup table for each `Int` we might encounter, by randomly picking a chord profile and then transposing it for that value. For example, if we had a range of note values in [0, 6] (the argument `notesRange` would have a value of 7 in this case) and the chord profiles were passed as `[(1, [(1, 0), (1, 2), (1, 4)])]`, then for the value 3 we would pick the chord profile `[(1, 0), (1, 2), (1, 4)]` and then transpose it to `[(1, 3), (1, 5), (1, 0)]` (since `notesRange` is 7, the last pair is (1, 0) as 7 mod 7 is 0).

`transposeProfile` simply makes sure that, when transposing a chord profile, we stay within the range of `Int` values for pitches that are allowed. `addProfiles` has a simple convenience function:

```
doAddProfiles :: (LSystem s) => Int -> WeightedList Duration -> WeightedList (WeightedList Int)
                              -> s AbstractNote -> IO (s AbstractNoteProfiled)
doAddProfiles notesRange durations profiles lsystem =
    do rules <- addProfiles notesRange durations profiles
       grow rules lsystem
```

Once we have a structure with chord profiles, each individual track can be developed using `buildRulesProfiled`:

```
buildRulesProfiled :: Duration -> Int -> Int -> WeightedList Duration
                       -> WeightedList (WeightedList Int)
                        -> IO [Rule AbstractNoteProfiled AbstractNoteProfiled]
buildRulesProfiled multiplier inNotesRange outNotesRange durations profiles =
 do lengthTable <- buildLengthTable multiplier durations
    unwrap [rule n d p lengthTable | n <- [0..(inNotesRange-1)],
            d <- expandUnWeighted durations, p <- allPossibleProfiles inNotesRange profiles]
 where rule :: Int -> Duration -> WeightedList Int -> [(Duration, [Duration])]
                -> IO (Rule AbstractNoteProfiled AbstractNoteProfiled)
       rule n d p lt = do replNotes <- fmap cycle $ shuffle $ map (`mod` outNotesRange)
                           $ expandWeighted p
                          return $ Rule ((n, d), p)
                                    [(1, zip (zip replNotes (unJust $ lookup d lt)) (repeat p))]
```

This is similar to the function `buildRules`, except that we build rules for each possible combination of `Duration`, `Int` and `WeightedList Int` (chord profile). For example, with a `Duration` multiplier value of 2, the production rules might specify that `((3, 1%4), [(1, 3), (1, 5), (1, 0)])` be replaced with `[((3, 1%4), [(1, 3), (1, 5), (1, 0)]), ((5, 1%4), [(1, 3), (1, 5), (1, 0)])]` on one iteration.

`buildRulesProfiled` has a corresponding convenience function:

```
doBuildRulesProfiled :: (LSystem s) => Duration -> Int -> Int -> WeightedList Duration
                                       -> WeightedList (WeightedList Int) -> Int
                                       -> s AbstractNoteProfiled -> IO (s AbstractNoteProfiled)
doBuildRulesProfiled multiplier inNotesRange outNotesRange durations profiles n lsystem =
 do rules <- buildRulesProfiled multiplier inNotesRange outNotesRange durations profiles
    growN n rules lsystem
```

Let us now see how all of these definitions can be put to work. The starting point is to define some basic parameters:

```
clDurations = [(7, (1%4)), (4, (1%8)), (9, (1%16))]
clProfiles = [(1,
              [(5, 0), (1, 1), (5, 2), (1, 3), (5, 4), (1, 5), (5, 6), (1, 7), (5, 8), (1, 9),
               (5, 10), (1, 11), (1, 12), (1, 13)]
              )]

cls = [Pitch C 5, Pitch D 5, Pitch E 5, Pitch F 5, Pitch G 5, Pitch A 6, Pitch B 6]
clScale = cls ++ shiftPitchPitches 12 cls
clScaleLow = shiftPitchPitches (-24) clScale
```

Here we have `clDurations` as a weighted list of preferred note lengths, `clProfiles` as a single profile of preferred notes (relative to some root note), and `clScale`, which contains two octaves of the C Major scale. `clScaleLow` is the same as `clScale`, but two octaves lower.

Note how we frequently use constant literals here. In a fully developed product based on this project, one would enter such literals via appropriate GUI elements. For illustrative purposes, however, these literals in Haskell will do.

Now that we have these constants, we are ready to define the structure for our music:

```
chordlessTest x =
 do seed x
    structure <- generateStart 4 7 2 clDurations 2
    structureProfiled <- doAddProfiles 14 clDurations clProfiles structure
```

Note that this code runs in the `IO` monad, to cope with the generation of random numbers. The parameter `x` allows us to use the same seed and generate identical music, or use a different seed to generate different pieces from the same template. The function `generateStart` is trivial and defined in `Fractal.hs`, and provides a random starting list of `AbstractNote`.

Given this structure, we can use it to generate a bassline:

```
    basslineAbstract <- doBuildRulesProfiled 2 14 14 clDurations clProfiles 1 structureProfiled
    bassline <- return $ doBuildNotes clScaleLow basslineAbstract
```

The large number of parameters (seven, in the above invocation of `doBuildRulesProfiled`) may appear somewhat daunting, but in practice this has not been found to be a problem, and in the future assistance can be given to functional composers via a GUI. Here, `doBuildRulesProfiled` provides abstract notes which are then interpreted as actual notes by `doBuildNotes`.

Finally, we can define the melody itself, in much the same way that we defined the bassline:

```
    melodyAbstract <- doBuildRulesProfiled 2 14 7 clDurations clProfiles 1 structureProfiled
    melody <- return $ doBuildNotes clScale melodyAbstract
```

This completes this example. All that remains is to write the result to the file `chordless.mid` using the Haskell MIDI sequencer (presented in chapter two):

```
    writeMIDIFile "chordless.mid" [Track 0 bassline, Track 1 melody] [(0, 33)] 120
```

The `[(0, 33)]` parameter specifies that the bassline will be played back using the MIDI 'Electric Bass (finger)'[1] instrument.

The result is that we have a coherent bassline and melody in the key of C Major.

The key to this example is the value of `clProfiles`. We only have one chord profile (although it is transposed for multiple starting notes) which denotes which notes from the C Major scale are more or less likely to be used, but transposed along the scale for the current root note.

5.4 Adding chords

So far, the tracks we have generated are just collections of melodies. The melodies share a structure between them, and the structure denotes sections where some notes are more likely to occur than others. Quite often a piece of music is accompanied by chords [Apel, p. 162] which mark the important notes in any given section.

`buildSimpleChords` is a function which takes a structure and chooses the most important notes from each chord profile to generate a chord track:

```
buildSimpleChords :: [Pitch] -> Duration -> Int -> WeightedList Duration
                     -> WeightedList (WeightedList Int) -> IO [Rule AbstractNoteProfiled Note]
```

It works by generating a lookup table for every abstract note and chord profile combination:

```
buildSimpleChords pitches multiplier notesRange durations profiles =
 do profileLookupTable <- unwrap $ map (\x -> do { p <- choosePitches x ; return (x, p) } )
                          $ allPossible
    return [rule n d p profileLookupTable
            | n <- notes, d <- expandUnWeighted durations, p <- allPossible]
```

`rule` creates a rule for any given abstract note and chord profile combination:

---

[1] http://www.midi.org/techspecs/gm1sound.php#instrument – note that this table starts numbering instruments from one, whereas this project numbers instruments from zero, so 'Electric Bass (finger)' is given as instrument number 34.

```
 where rule :: Int -> Duration -> WeightedList Int -> [(WeightedList Int, [Pitch])]
                -> Rule AbstractNoteProfiled Note
       rule n d p table =
        Rule ((n, d), p) [(1,
                           (map (\x -> Note x defaultMIDIVolume (d*multiplier))
                              $ unJust $ lookup p table) ++ [Move (d*multiplier)])]
```

The crucial part of `buildSimpleChords` is `choosePitches`, which takes the highest weighted notes from a chord profile:

```
       choosePitches :: WeightedList Int -> IO [Pitch]
       choosePitches xs = do r <- rand 3 5
                             -- Take the 5 highest weighted pitches, and randomly choose r from
                             -- them.
                             fmap (map (\x -> unJust $ lookup x pitchMap)) $ fmap (take r)
                              $ shuffle $ map snd $ take 5
                                $ sortBy (\(x, y) (x', y') -> compare x' x) xs
```

`pitchMap` and `notes` are defined simply for convenience:

```
       pitchMap = zip notes pitches
       notes = [0..(notesRange-1)]
```

`allPossible` uses the (trivial) function `allPossibleProfiles`, defined in `Fractal.hs`, to generate all possible transposed chord profiles given a range of (`Int`) note values:

```
       allPossible = allPossibleProfiles notesRange profiles
```

Finally, a corresponding convenience function:

```
doBuildSimpleChords :: (LSystem s) => [Pitch] -> Duration -> Int -> WeightedList Duration
                                      -> WeightedList (WeightedList Int)
                                      -> s AbstractNoteProfiled -> IO (s Note)
doBuildSimpleChords pitches multiplier notesRange durations profiles lsystem =
    do rules <- buildSimpleChords pitches multiplier notesRange durations profiles
       grow rules lsystem
```

The chord tracks generated by `doBuildSimpleChords` are very simple, and simply play prominent notes from each part of the structure for the duration of that part. We can use rotated chords to make tracks more interesting. To rotate a chord, the highest pitched note's pitch is lowered by an octave, or the lowest pitched note's pitch is raised by an octave. However, simply taking any note from the chord and incrementing or decrementing its octave is good enough,

and in `buildRotatingChords` we replace each section of the structure with several rotated chords instead of one simple one:

```
buildRotatingChords :: [Pitch] -> Duration -> Int -> WeightedList Duration
                       -> WeightedList (WeightedList Int) -> IO [Rule AbstractNoteProfiled Note]
```

The basic idea is the same, with building a lookup table for each abstract note and chord profile combination:

```
buildRotatingChords pitches multiplier notesRange durations profiles =
 do lengthTable <- buildLengthTable multiplier durations
    unwrap $ [rule n d p lengthTable
              | n <- notes, d <- expandUnWeighted durations, p <- allPossible]
```

`rule` gives a replacement rule for any given abstract note and chord profile combination, using the `chordRotations` function discussed below:

```
 where rule :: Int -> Duration -> WeightedList Int -> [(Duration, [Duration])]
                -> IO (Rule AbstractNoteProfiled Note)
       rule n d p lt = do ps <- choosePitches p
                          chords <- chordRotations ps
                          notes <- return $ concat $
                           zipWith (\d ps ->
                                     map (\p -> Note p defaultMIDIVolume d) ps ++ [Move d])
                            lengths (cycle chords)
                          return $ Rule ((n, d), p) [(1, notes)]
                                where lengths = unJust $ lookup d lt
```

The rest of the code is much the same as `buildSimpleChords`:

```
       choosePitches :: WeightedList Int -> IO [Pitch]
       choosePitches xs = do r <- rand 3 5
                             fmap (map (\x -> unJust $ lookup x pitchMap)) $ fmap (take r)
                              $ shuffle $ map snd $ take 5
                               $ sortBy (\(x, y) (x', y') -> compare x' x) xs
       pitchMap = zip notes pitches
       notes = [0..(notesRange-1)]
       allPossible = allPossibleProfiles notesRange profiles
```

`buildRotatingChords` uses the `chordRotations` function, which takes a list of `Pitch` representing a chord, and provides all the possible chord rotations of that chord:

```
chordRotations :: [Pitch] -> IO [[Pitch]]
chordRotations pitches = shuffle $ map (shiftPitchPitches (-12)) $ cr (length pitches) pitches
 where cr 0 pitches = [pitches]
       cr n (p:ps) = let result = ps ++ shiftPitchPitches 12 [p] in [result] ++ cr (n-1) result

doBuildRotatingChords pitches multiplier notesRange durations profiles lsystem =
    do rules <- buildRotatingChords pitches multiplier notesRange durations profiles
       grow rules lsystem
```

Now that we have a method for adding nice rotating chords, we can modify the previous example:

```
rotatingTest x =
 do seed x
    structure <- generateStart 4 7 2 clDurations 2
    structureProfiled <- doAddProfiles 14 clDurations clProfiles structure

    basslineAbstract <- doBuildRulesProfiled 2 14 14 clDurations clProfiles 1 structureProfiled
    bassline <- return $ doBuildNotes clScaleLow basslineAbstract

    melodyAbstract <- doBuildRulesProfiled 2 14 7 clDurations clProfiles 1 structureProfiled
    melody <- return $ doBuildNotes clScale melodyAbstract

    chords <- doBuildRotatingChords clScale 2 14 clDurations' clProfiles structureProfiled

    writeMIDIFile "rotating1.mid" [Track 0 bassline, Track 1 melody, Track 2 chords]
                  [(0, 33)] 120
```

`rotatingTest` is almost identical to `chordlessTest` (and uses the same previously defined constants). The difference is that we have a third track with rotating chords, written to the file `rotating1.mid`.

6 Evaluation

6.1 More example scripts

The following example includes a bassline, melody and piano chords, with several different chord profiles in the file `test.mid`:

```
s = [Pitch C 5, Pitch Cs 5, Pitch D 5, Pitch Ds 5, Pitch E 5, Pitch F 5, Pitch Fs 5, Pitch G 5,
     Pitch Gs 5, Pitch A 5, Pitch As 5, Pitch B 5]
scale = s ++ shiftPitchPitches 12 s
scaleHigh = shiftPitchPitches 12 scale
scaleLow = shiftPitchPitches (-24) scale

profiles = [(1,
               [(5, 0), (5, 3), (5, 7), (5, 10), (5, 14), (5, 17)]
               ),
            (1,
               [(5, 0), (5, 4), (5, 8), (5, 10), (5, 14), (5, 17)]
               ),
            (1,
               [(5, 0), (5, 4), (5, 9), (5, 14), (5, 16)]
               )
           ]

durations = [(0, (4%4)), (0, (2%4)), (7, (1%4)), (4, (1%8)), (9, (1%16))]
structDurations = [(2, (4%4)), (2, (2%4))]
chordDurations = [(0, (4%4)), (1, (2%4)), (7, (1%4)), (6, (1%8)), (1, (1%16))]

test x =
 do seed x
    structure <- generateStart 4 12 2 structDurations 2
    structureProfiled <- doAddProfiles 24 durations profiles structure

    basslineAbstract <- doBuildRulesProfiled 2 24 24 durations profiles 1 structureProfiled
    bassline <- return $ doBuildNotes scaleLow basslineAbstract

    melodyAbstract <- doBuildRulesProfiled 2 24 12 durations profiles 1 structureProfiled
    melody <- return $ doBuildNotes scale melodyAbstract

    chords <- doBuildRotatingChords scale 2 24 chordDurations profiles structureProfiled

    writeMIDIFile "test.mid" [Track 0 chords, Track 1 bassline, Track 2 melody] [(1, 33)] 120
```

This example is not restricted to any particular key – on each chord change, the key effectively changes, and notes are picked which fit with the current chord. This is similar to the approach taken by some modern jazz. A modification of the above follows:

```
structDurations2 = [(1, (4%4)), (1, (2%4)), (1, (1%4)), (1, (1%8))]
bassDurations2 = [(0, (4%4)), (5, 2%4), (3, 1%4), (3, 1%8), (5, 1%16)]
melodyDurations2 = [(5, 4%4), (0, 2%4), (3, 1%4), (3, 1%8), (5, 1%16)]

profiles2 = [(1, [
                  (5, 0), (5, 4), (5, 7), (5, 11)])]

test2 x =
 do seed x
    structure <- generateStart 4 12 2 structDurations2 2
    structureProfiled <- doAddProfiles 24 structDurations2 profiles2 structure

    basslineAbstract <- doBuildRulesProfiled 2 24 24 bassDurations2 profiles2 1 structureProfiled
    bassline <- return $ doBuildNotes scaleLow basslineAbstract

    melodyAbstract <- doBuildRulesProfiled 2 24 12 melodyDurations2 profiles2 1 structureProfiled
    melody <- return $ doBuildNotes scale melodyAbstract

    chords <- doBuildSimpleChords scale 2 24 structDurations2 profiles2 structureProfiled

    writeMIDIFile "test2.mid" [Track 0 chords, Track 1 bassline, Track 2 melody] [(1, 33)] 80
```

Here, we only have one profile, but still have a bassline, melody and simple chords, written to the file `test2.mid`. A slightly more chaotic piece with two melodies follows (written to test3.mid):

```
structDurations3 = [(1, (4%4)), (1, (2%4)), (1, (1%4))]
bassDurations3 = [(0, (4%4)), (5, 2%4), (3, 1%4), (3, 1%12), (5, 2%12)]
melodyDurations3 = [(5, 4%4), (0, 2%4), (3, 1%4), (3, 1%12), (5, 2%12)]

profiles3 = [(1, [
                  (5, 0), (3, 2), (5, 3), (3, 7), (5, 11), (5, 17)]),
             (1, [(5, 0), (4, 4), (5, 7), (5, 10), (5, 14)
                  ])]

test3 x =
 do seed x
    structure <- generateStart 4 12 2 structDurations3 2
    structureProfiled <- doAddProfiles 24 structDurations3 profiles3 structure

    basslineAbstract <- doBuildRulesProfiled 2 24 24 bassDurations3 profiles3 1 structureProfiled
    bassline <- return $ doBuildNotes scaleLow basslineAbstract

    melodyAbstract <- doBuildRulesProfiled 2 24 12 melodyDurations3 profiles3 1 structureProfiled
    melody <- return $ doBuildNotes scale melodyAbstract

    melodyAbstract' <- doBuildRulesProfiled 2 24 12 melodyDurations3 profiles3 1
structureProfiled
    melody' <- return $ doBuildNotes scale melodyAbstract'

    chords <- doBuildRotatingChords scaleHigh 2 24 bassDurations3 profiles3 structureProfiled

    writeMIDIFile "test3.mid" [Track 0 chords, Track 1 bassline, Track 2 melody, Track 3 melody']
                  [(1, 33)] 145
```

7 Previous work

In this chapter we discuss previous work relating to Haskell and computer generated music.

7.1 Existing Haskell music libraries

There already exist some libraries for specifying and generating music in Haskell. The most notable of these is Haskore[1] which is a Haskell library for creating and analysing music. I have not used this library, although the `PitchName`, `Pitch` and `Note` types are inspired by Haskore's base types [DIMH]. Haskore uses parallel and serial composition of musical notes, effectively resulting in a lazy two dimensional structure to each track, whereas I have chosen to use lists of note types which map more simply to MIDI events, and make polyphonic tracks less confusing to deal with. The result is a very small library capable of creating MIDI files which only depends on the GHC base libraries. This project also takes a more pragmatic and less traditional approach to music theory, and so many of the interesting analytical features of Haskore would not be used. Writing my own tiny MIDI library allowed me to experience how Haskell deals with binary data, and I was able to compare the code to a similar simplistic MIDI library I wrote in Java some time ago. I was also able to do some post processing over MIDI events (as described in chapter three).

7.2 Fractal music

The idea of using computers or algorithmic processes to generate music is not new. Many people have also used fractals or generative grammars to produce music.

The website Pawfal.org[2] ('poor artists working for a living') is a loose community of artists and programmers. One particular page on their website[3] (author unknown) describes a method of using L-systems to describe and generate music, and then using genetic processes to audition, choose and improve the music and the process. This is all seen in the context of 'live coding', which is a broad term describing the act of writing and editing code as part of a performance process, which is interpreted in real time as (often) a

[1] http://www.haskell.org/haskellwiki/Haskore
[2] http://www.pawfal.org
[3] http://www.pawfal.org/index.php?page=LsystemComposition

combination of sound and animation. A prominent example of doing this with Haskell is work by Alex McLean[1].

P. Prusinkiewicz (author in [TABoP]) describes a method for interpreting L-systems musically in [PP86], whereby an L-system is interpreted visually (as in [TABoP]), and then "the resulting figure is interpreted musically, as a sequence of notes with pitch and duration determined by the coordinates of the figure segments". The resulting music exhibits interesting structure and patterns and the example given is monophonic. The paper does suggest the idea of using the branches of an L-System to create polyphonic tracks, (an idea used in this project), but the idea is not developed further in the paper.

In [LRB09] the idea of 'breeding' L-systems with genetic algorithms to obtain music is discussed, and a Python[2] program is presented which uses this method, which is different from the approach taken by this project.

[MS94] provides a comprehensive analysis of methods for interpreting L-systems as melodies and transforming them, and also examines music composed by Bach[3] to speculate how the self-similarity it exhibits could be derived from musical interpretations of L-systems.

[KJ89] discusses "Space Grammars", which relates to L-systems in the sense that over each iteration of the rules there is a decision to either split a geometric shape into two smaller shapes which occupy the same area/volume (in a graphical interpretation) or split one note into two smaller ones which occupy the same duration in time. My algorithm differs in that a multiplier is used, but these ideas are central to maintaining rhythmic coherence over multiple tracks.

The idea of hierarchical grammars is presented in [JM96], and the paper describes how "complex shifting themes" can be propagated in a polyphonic context, with the idea of context-sensitivity. This is similar in aims to what this project achieves, but this project achieves it by expanding the L-system alphabet to explicitly state the current context in the form of chord profiles.

A good example of the expressiveness of Haskell's algebraic datatypes is [MH2011], which presents some types representing harmonic structure, and

---

[1] http://yaxu.org/haskellmusic/
http://yaxu.org/category/livecoding/
[2] http://python.org/
[3] Johann Sebastian Bach, 1685 – 1750, notable German composer and musician

some methods for comparing the harmonies and harmonic structure of pieces of music. However, the model of harmony used in this project is significantly simpler and does not draw from traditional musical theory, as this paper does.

8 Conclusion

The aim of this project was to create a library of functions for generating music with a natural feel. This aim has been accomplished, both through appropriate use of randomisation and the use of L-systems. In this final chapter, I reflect on what I have learnt from this project, and consider some ways in which the work could be extended in future.

8.1 What I have learnt

I have made much use of Haskell's laziness, infinite lists, algebraic data types and pattern matching. Haskell is very expressive and although the complexity of the program is not hidden, the code is simpler and short.

The use of randomisation was crucial to this project, and that in turn forced me to write most of the code using monads. While this is not a drawback in terms of capability, it does make the code look somewhat less attractive than it might have been in a language that does not force every effect to be explicit.

To achieve the musical sophistication I aimed for, it turned out to be necessary to write functions with an unusually high number of parameters – sometimes as many as seven or eight. This makes the library functions hard to use for novices, or for users whose primary interest is music and not programming. The use of an object-oriented language might have made it easier to provide default values for parameters.

8.2 Future work

There are many additions and extensions that could be implemented.

The software could be augmented with a GUI so that users are able to able to draw in trees and graphically describe what sort of music they would like, much like in the diagrams in chapter five.

A graphical visualizer could be added, which would re-interpret the structure and notes of the music as an animation. This would be much like visualizers in media software such as Apple's iTunes or Microsoft's Windows Media Player, but unlike these visualizers, which generate animations using the actual waveform of the sound, a visualizer for this computer generated music would know about the structure and detail of the notes in the music, and so the animations would be more tightly coupled.

A simpler version of the software could include some preset scripts and graphical sliders, or some other UI element, to adjust properties of the music

generated, such as speed, harmony, rhythmic style etc. This could be accompanied by a visualization of the type described above as a simple and fun application to generate music. The nature of the music generated with this software includes specifying a random number seed, so that for any given test script there is a very large number of similar sounding pieces of music, and this could be exploited to create an (almost) infinitely long piece of music from any given script in real time to be used in the above application.

With more complicated test examples, the number of rules generated becomes very high, as rules are generated for every possible abstract note value that could be encountered, and the program could be modified to only generate rules as they are needed, and store them in a lookup table. This would mean that the space and time complexity would grow logarithmically with the length of the music to be generated, but would require some re-structuring of the program (but is easily achievable).

Finally, the combinators used in `Fractal.hs` could be incorporated in to a library which uses Arrows[1] (which are a generalisation of monads) to create a domain specific language, which would be points-free. The nature of the way that music is generated in this software would lend itself well to dataflow programming, which is encapsulated well within arrows. I expect this would make the code look more attractive, encouraging more musical programmers to work directly with Haskell.

[1] http://www.haskell.org/arrows/

Appendix A – Running the examples

The entire source code of this project can be found at `[]`. There are four code files, which should be placed in the same directory. These are `Main.hs`, `Fractal.hs`, `NoteUtil.hs` and `Note.hs`. In addition, MIDI files of all of the examples can be found at the same address.

This project was developed and tested on Microsoft Windows 7 with GHC[1] version 6.12.3. However, it does not use any platform dependent libraries and so should run on any platform and with later versions of GHC. The code uses some GHC extensions, and so may not run on other Haskell compilers or interpreters. The easiest way to install GHC is to install the Haskell platform[2].

The code can be loaded by typing the following into your shell (assuming the current directory contains all four source files):

```
$ ghci –fglasgow-exts Main.hs
```

`Main.hs` loads all of the necessary libraries and contains all the examples presented in chapters five and six. GHCi[3] is an interactive Haskell shell, and examples can be run by typing the name of their respective functions with any parameters, which will typically write a MIDI file to the current directory. After one has invoked the above command, GHCi may then display something similar to this:

```
GHCi, version 6.12.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Loading package ffi-1.0 ... linking ... done.
[1 of 4] Compiling Note             ( Note.hs, interpreted )
[2 of 4] Compiling NoteUtil         ( NoteUtil.hs, interpreted )
[3 of 4] Compiling Fractal          ( Fractal.hs, interpreted )
[4 of 4] Compiling Main             ( Main.hs, interpreted )
Ok, modules loaded: Main, Fractal, NoteUtil, Note.
*Main>
```

As discussed in earlier chapters, `Note.hs` contains the code for a simple MIDI sequencer, `NoteUtil.hs` contains code for utility functions over notes, and

[1] http://www.haskell.org/ghc/
[2] http://hackage.haskell.org/platform/
[3] http://www.haskell.org/ghc/docs/7.0.3/html/users_guide/ghci.html

`Fractal.hs` contains combinator functions for generating music. One might run one of the examples:

```
*Main> test 7
```

This will run the test `test` from chapter six with a seed of 7 and write a MIDI file in the current directory with the name of `test.mid`.

Appendix B - References

[Apel] Willi Apel. Harvard Dictionary of Music, 2nd Ed.

[TABoP] Przemyslaw Prusinkiewicz, Aristid Lindenmayer. The Algorithmic Beauty of Plants. See http://algorithmicbotany.org/papers/#abop

[PP86] Przemyslaw Prusinkiewicz. Score generation with L−systems. Proceedings of the 1986 International Computer Music Conference, pages 455−457.

[LRB09] Bruno F. Lourenço, José C. L. Ralha, Márcio C. P. Brandão. L-systems, scores and evolutionary techniques, 2009

[MS94] Stephanie Mason, Michael Saffle. L-Systems, Melodies and Musical Structure, 1994

[KJ89] Kevin Jones. Generative models in computer assisted musical composition, 1989

[JM96] Jon McCormack. Grammar Based Music Composition, 1996

[MH11] José Pedro Magalhães, W. Bas de Haas. Experience Report: Functional Modelling of Musical Harmony, 2011

[DIMH] Paul Hudak. Describing and Interpreting Music in Haskell, 2003. From the book The Fun of Programming. Jeremy Gibbons, Oege de Moor, 2003