Erlend Kilvik Hoff

Functionally Generating Music Structures with Prefix Trees

Master's thesis in Cybernetics and Robotics
Supervisor: Sverre Hendseth
June 2021

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

NTNU
Norwegian University of
Science and Technology

Erlend Kilvik Hoff

Functionally Generating Music Structures with Prefix Trees

Master's thesis in Cybernetics and Robotics
Supervisor: Sverre Hendseth
June 2021

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

NTNU
Norwegian University of
Science and Technology

Project Description

A piece of music is a complex structure, exhibiting shared patterns across multiple dimensions in a hierarchical manner. Any given note in a musical composition can be viewed in many contexts, such as melody, rhythm, and harmony. Successful music exhibits a form of coherent repetition in all these contexts. A suspicion arises that a great reason why most generated music today sounds *lacking*, is the inherent difficulty designing software which can represent and generate such complex structures. Expressing shared structure in both the long and short term, as well as non-hierarchical relationships between arbitrary leaves, is especially difficult.

When tackling such a difficult problem with no clear solution, it might be advantageous to look at it through the lens of a different programming paradigm. Functional programming is a paradigm which requires the programmer to think in a way that is quite different from the imperative mindset. Functional languages, such as Haskell, are notorious for being hard to learn and understand, but also tend to give very neat and well-formed software. Could using a functional language to design and implement a music generation software that addresses the structure-related problems described earlier, lead to a unique and possibly better solution?

In this project, the student shall:

- Do background research to learn the functional language Haskell sufficiently to carry out the project.
- Do background research on previous work on representing and generating music with a structural focus.
- Take this into account and design a system which can express and generate musical structures.
- Implement this design in Haskell.
- Evaluate the system's ability to generate well-structured music.
- Discuss the adequacy of the design and implementation for generating sound musical structures.
- Discuss the suitability and experience of using Haskell for this task, compared to an imperative language.

Abstract

In this project, a novel system for generating music with a focus on a structure is designed and implemented in Haskell. The system expresses and generates music with shared material across multiple hierarchical levels and musical dimensions with *generative prefix trees*, a structure designed in this project, based on previous work by Yan Han [1]. A generalized generation process is proposed. It consists of generating and applying a series of prefix trees, which each address a given musical aspect, like rhythm, harmony, or melody. The prefix trees are generated according to a meta-model structure called *generative plan*, in order to ensure cohesion and variety in the output music. A *generative example* is designed to demonstrate the systems ability to generate structured music. Here, structure is randomized by randomising the generation of the prefix trees. Chord progressions, rhythms and "playing patterns" are chosen from a small pool of hand-written candidates. Each generating piece exhibits a unique structure, with clear repetition on all hierarchical levels. This structural freedom is something that lacks in most music generations systems today, where a certain form is usually chosen a priori [2]. Many avenues for future work in improving the system ability to generate music are outlined. Since the generative system is so abstract, its general principle could also be applied to hierarchical structures in other areas, such as figures or text.

Sammendrag

I dette prosjektet ble et nytt system for å generere musikk med fokus på struktur designet og implementert i det funksjonelle språket Haskell. Systemet representerer og genererer musikk med delt struktur på tvers av flerer hierarkiske nivå og musikalske dimensioner. Dette gjøres ved hjelp av en prefix-trestruktur som ble designet i dette prosjektet basert på tidligere arbeider av Yan Han. En generalisert prosess for generering med slike prefix-trestrukturerer ble presentert. Denne består av å først generere en rekke prefixtrær som hver adressererer sin musikalske dimensjon, altså hvert sitt aspekt av musikken som kan anses individuelt, slik som rytme, harmoni og melodi. Disse genereres etter en meta-modell kalt "generative plan", for å sikre sammenheng og variasjon. For å demonstrere denne generelle generative metodens egnethet ble et snevrere generativt eksempel designet. Her ble den generelle metoden brukt til å generere akkordprogresjoner med strukturert variasjon i rytme og spillemønster. Den genererte musikk hadde en ny struktur hver gang, med tydelige repetisjoner på alle hierarkiske nivå. Denne strukturelle friheten mangler fra my av dagen musikk-genereringssystemer, der man i stedet gjerne velger form/struktur a priori, og lar algoritmens jobb heller være å fylle denne strukturen. Gjennom hele arbeidet har det vært et tydelig fokus på abstraksjon og generalitet. Det kan dermed tenkes at den generelle generative metoden også kan overføres til andre domener, som viser en lignende hierarkisk inndeling. Dette kan være generering av figurer, eller generering av tekst.

Preface

This master's thesis was written in spring 2021, under the supervision of Sverre Hendseth. I would like to thank him for letting me work on a field as interesting as music generation, and for turning me on to his ideas on the meta-model / model relationships that are present in so many hierarchical structures. We met almost every week and his influence has been invaluable to me, both to the project and life in general.

Furthermore I would like to thank my friends and family. I especially want to thank my girlfriend Inga and the friends I live with. Finally i would like to thank my mother and my friend Ole Peder Brandtzæg for proof-reading.

The work presented here builds upon previous work by Yan Han. I would like to thank him for the interesting work he did, and for kindly answering my questions.

Erlend Kilvik Hoff
Trondheim, 14/6 2021

Contents

Chapter 1

Introduction

Generating music is a notoriously difficult problem that has been tackled with great interest throughout history. Since the time of Pythagoras, musicians have used algorithmic procedures in their compositions. Famous examples include Bach's procedural counterpoint, Erik Satie's Vexations, Schoenberg's serialism and minimalist techniques by the likes of Steve Reich and John Cage.

With the advent of computer science, the field of computer generated music appeared. Today, this is a thriving field, populated by both academics and artists. This has resulted in great progress in different subtasks of composition, such as melody generation, harmonisation and rhythm generation. Still, today's music generation systems still struggle with generating music that exhibits reasonable long term structure, with reasonable repetitions of subpatterns, i.e. shared structure [2]. Solving this problem would be a major stepping stone towards better generated music, since it appears that hierarchical structure is inherent to human music perception [3].

Parts of these difficulties obviously arise from the fact that music composition is a *creative* pursuit, which is not exactly what computers are known for handling best. Still, the fact that certain composition tasks that are just as creative, such as melody generation, has to a much greater degree been successfully carried out by computers, leads one to think that there might also be a more *technical* reason why generating music with varied and cohesive long term structure has not been successfully done yet.

It is both the author's and the supervisor's suspicion that a great reason for why this problem is so difficult is that music exhibits such a complex web of relationships that representing and generating such data structures is incredibly difficult in itself. In music, every note belongs to many groups, such as chords, motifs, melodies, phrases, verses, choruses, etc. Each note is perceived in many contexts at the same time, and there are recurring patterns in all these groups. Additionally, one can decompose musical information into multiple dimensions, so that

patterns of repetition might occur in one dimension but not in the others. For instance a given rhythmic pattern might reoccur, but with different chords, or a given motif might reoccur, but in a transposed form. A need emerges for data structure which can capture the shared patterns across these musical dimensions, and thus be used to generate music that is *explicitly structured*.

1.1 Goals

In this project we want to design and implement a software that can express and generate music with shared structure in both the long and short term. This shared structure is in the form of repetitions of material in all musical dimensions, and at every hierarchical level. We wish to come upon a generalized method for generating music with such shared material.

Parameterised Stylistic Constraints A major part of this generalization is that the system should not inherently enforce any *stylistic constraints* on the music. We do need some constraints on generation process for the music to be cohesive, but these constraints should be passed as input arguments to the generative functions, not be embedded into the code itself.

Focus on Structure The focus of this project is to generate music with novel structures, i.e. music where the form is not chosen a priori, but is randomly generated. The line between *form* and *content* can be a bit fuzzy. In this project we will use the term *structure* to refer to the relationships between *notes and rests*. Especially how notes are grouped together, and how such groups are further grouped together, revealing patterns of relationships between different groups that are different for each musical aspect/dimension considered.

Other Aspects of Musical "Quality" not an Explicit Goal Since our focus is on generating structures, we are not primarily interested in the other musical aspects of the generated output. We will not measure our success on the quality or listenability of the music, but rather on the systems ability to enforce shared patterns. Thus, we will no go very deep into music theory, and keep the usage of terms from music theory to a minimum. The average reader will probably be familiar with most terms used, but when in doubt, the reader is encouraged to look at the music theory glossary that is provided in the appendix, where a short explanation of each used term will be provided.

Design and Implementation in Haskell The software is to be designed for and implemented in the functional language Haskell. In order to do so, the language first must be learned, which is quite a daunting task, as it is known for being a difficult language for beginners who are unfamiliar with functional concepts. Thus, in this project, learning Haskell can be considered a goal in itself.

1.2 Motivation

This project is mainly motivated by two factors:

1. It is interesting from a software design perspective, for the reasons described above.
2. It is interesting from an aesthetic perspective.

Point 2 is a major personal motivation for doing this project. With generated music, one cannot point to a sentient being as its direct creator. This has very interesting implications. Can such music be considered art, or at least appreciated for its beauty? Most people can appreciate the beauty of things that are generated by the laws of nature, like as a sunset, or a waterfall. This form of beauty can give an almost mystical feeling, since it doesn't involve the kind of communication between the creator and the user that is inherent to human-made products. It is the authors opinion that the same thing can be said for generated music, where a strange sort of beauty can appear from the void that is the lack of a sentient creator. Working with this project will involve confronting these ideas, which is endlessly interesting.

There is also another interesting aspect to generated music that has to do with the following question: Are there some limits to human creativity that constrains the pool of possible pieces that can be written? And does this mean that there is a possibility that music that is generated from an entirely different creative process can reveal new interesting ideas that a person, due to both cultural and possibly neurological restrictions is not able to conceive of? If this is the case, then computer generated music could serve a very interesting role in pushing the bounds of human creativity and thus push the field of music composition forward. It could serve to show us how music could *possibly* sound like, giving a joy that is similar to listening to a new genre for the first time, or discovering the music of an entirely different culture. While these ideas point to a future far ahead, if they are to be realized, then music generation systems must be freed from a priori chosen forms. It is of no use to simply automate the composition of corpus-bound compositions like one would automate any other industrial process. The goal of computer generated music should be exploration and experimentation, and for this to be the case, it is vital that also the *form* of a generated piece is itself generated.

1.2.1 Why Haskell?

Before starting this project I had no knowledge on how to program in Haskell. So why was it chosen as the implementation language of this project? This decision was motivated by a few reasons. The first of these was a personal interest in learning more about Haskell and functional programming in general. Haskell's reputation as a difficult but beautiful language made it intriguing to explore. Secondly, as functional programming requires a different mindset from the imperative way of thinking, the hope is that this will also give a different approach to the cent-

ral problem in this project. As this problem of structure-generation is more or less unsolved, looking at it from a different angle might be advantageous. Finally, some initial research has indicated that there is already a decently-sized milieu for handling music-related tasks in Haskell. Paul Hudak, one of Haskell's creators is also a musician, and has written a great book called *The Haskell School of Music*, where Haskell is taught in a music context through the library *Euterpea* [4]. There is also a conference on functional generation of art called FARM (Functional Art Modeling and Design), where music generation in Haskell is a common topic.

1.3 Report Outline

- **Background**
  - Haskell and functional programming in general
  - The Euterpea library for music composition in Haskell
  - Previous work on music structure
- **Specification**
  - Requirements put on the software's functionality, as well as its design and implementation
- **Design**
  - Presentation of the design for generating structured music
- **Implementation**
  - Implementation of the design in Haskell
- **Evaluation**
  - Evaluation of the system's ability to generate structured music
- **Discussion**
  - On the design
  - On the Implementation.
  - On using Haskell/functional programming for this project
- **Conclusion**
  - Conclusion on the main takeaways from the project

Chapter 2

Functional Programming in Haskell

This chapter serves as the necessary introduction to functional programming in Haskell needed to understand and appreciate the work done in the later chapters. As Haskell and functional programming in general is not well known to most imperative programmers, this chapter will start with the very fundamentals, and work its way to the more abstract features of the language. The final section will cover some guidelines for developing and designing Haskell software.

While the contents of this chapter is interesting for the unfamiliar reader, it is also uncontroversial and can more or less be found in any introductory book to Haskell. Thus the main source for learning the basics of Haskell was chosen based on how educational it was. In our case, this was a book called "Learn You a Haskell for Great Good" [5]. This is a famous and very popular beginners guide, recommended by the official Haskell website haskell.org, among others. Additional sources used were Serrano's book "Practical Haskell: A Real World Guide to Programming" [6] and "The Haskell School of Music" [4] by Hudak. Certain examples were taken from from the official Haskell Wiki. Since the material presented here is a general introduction, and Haskell is a community-driven language, the official Haskell Wiki deemed as an appropriate source, especially due to its reputation as well-maintained and "official".

2.1 The Haskell Approach to Functional Programming

This section is based on the chapter *Going Functional* from "Practical Haskell" [6]. It will explain what makes Haskell different from other programming languages, and the advantages that follow from these differences. The defining traits of Haskell can be summarized like this:

- Functions are mainly pure, data is immutable, side effects are isolated
- Evaluation is *lazy*

- Types are statically checked by the compiler
- The type system is polymorphic, based on parametricity and type classes.

The following four subsections will elaborate on these four characteristics.

2.1.1 Pure functions

Haskell is a *functional language*, and thus lends itself well to functional programming. In functional programming, functions are treated as data. They can be assigned to a variable, and be both input to and output from another function. This allows for a very high level of abstraction, and thus reuse of code.

Since Haskell is a purely functional language, its functions are pure expressions. This means that, like mathematical functions, they always give the same output when given the same input. Side effects are still possible, but they are isolated from the pure part of the code.

The main benefit of purity is that it is easier to reason about code and test it, since you can be sure that the outcome of a function depends only on its parameters. This is known as referential transparency. Just like in maths, a function cannot change anything outside itself. It can just give an output based on an input.

2.1.2 Laziness

In Haskell, the evaluation of pure expressions is not dependent on the order of their execution. In contrast to imperative languages, where the code acts as sequence of instructions to be carried out, in Haskell you tell the computer what things *are*. This is known as being *declarative*.

Since pure expressions don't need to be evaluated in a particular order, different *execution strategies* are possible. The default execution strategy in Haskell is *lazy evaluation*. With this form of evaluation, an expression is only evaluated when it is needed to evaluate another one. This allows the expressions of infinite data structures, such as the infinite list of positive integers `[0..]`.

Sometimes we really need something to be executed in a given order, depending on some sort of state. This can still be done functionally by chaining function calls utilizing a concept known as the *monad*. This is a concept that can be difficult to grasp, and will be expanded on later.

2.1.3 Strong static typing

Type systems are a way of categorising the values that could appear during execution. Types are used to *tag* values, normally to restrict the set of possible functions that can be applied on values with a given type. Such a system is clearly needed, as most functions only really makes sense when applied to a certain set of values, and

make no sense when applied to others. For instance, it would make sense to perform a division on doubles, but doing the same with strings is clearly meaningless.

In general, types can be checked at two times: At execution time, known as *dynamic typing*, or at compilation time, known as *static typing*. Haskell belongs to the latter category. Its type system is very *strong*, which means that the number of type errors that can be caught at compile time is very high. This gives the programmer greater confidence in the code once it has compiled, because it is unlikely that a type error will appear during execution. To quote a common saying in the Haskell community: "Once it compiles, it works".

2.1.4 Polymorphism

The type system in Haskell exhibits great polymorphism. This means that types can be abstract, defined to contain other types without specifying what this type should be. For instance, the list type can contain any type inside it, and you can define functions on lists that are independent on what the list actually contains. What type the list actually contains is a parameter to the list type. This is known as *parametric polymorphism*, and will be further elaborated on in section 2.4.

Another form of polymorphism in Haskell is the ability to group types by the actions that can be performed on them. This is known as *type classes*. For instance, it is clear that the function *map*, which applies a given function to all the elements in a list, could be abstracted to work on other data structures as well, such as trees, graphs and sets. This concept is very high-level, and allows for great abstraction and re-use of code.

2.2 Basic syntax

Since we now know the defining characteristics of Haskell, it is time to get to know the language syntax itself. In this section, the basic syntax of Haskell will be explained, as well as the basics of lists (the most fundamental data structure) and types. This will give the minimal introduction needed to understand the more in-depth features of Haskell covered in the succeeding sections. The main source for this section is chapters 2, 3 and 4 in *Learn You a Haskell for Great Good* [5].

2.2.1 Defining values and functions

Defining Values In Haskell, values are defined like this:

```
number = 1
name = "Johnny"
```

As data is immutable in Haskell, once values are defined, they can never be changed. All values must be lower case, as upper case is reserved for types.

Defining Functions Functions can be considered values with arguments. Thus they also must be lower case. A function's arguments are written after its name, separated by a space. what the function returns is written after the equals sign:

```
add x y = x + y
```

Functions can also contain other functions:

```
average x y = (add x y)/2
```

Infix and Prefix Functions Infix functions are written between their arguments. This is mostly done to make a function easier to read. To make a function written with letters infix, we need to surround it with backticks. For instance, the `div` function, which takes two integrals and does integral division between them, can be be written prefix and infix like this:

```
div 92 10
92 `div` 10
```

Functions that are written with symbols (known as operators), such as `*`, `+` and `/`, are by default infix. To make them prefix, we need to surround them by parentheses:

```
1 + 2
(+) 1 2
```

Let In and Where It is often useful to define expressions within the scope of a function. This can be done with *let in* notation:

```
average x y =
    let sum = add x y
    in sum / 2
```

The same thing can also be achieved with *where* notation:

```
average x y = sum / 2
    where sum = add x y
```

Pattern matching Functions can also be defined with pattern matching. This is one of the most useful features of Haskell. Pattern matching consists of defining a function with several different function bodies, where each function body defines what the function should do for a given pattern of input. For instance:

```
is7 7 = True
is7 _ = False
```

Here, we see that the function `is7` is defined with separate function bodies for different patterns, which makes for very clear and readable code. (the _ symbol is used to indicate that the argument can be anything). During execution, the compiler will check the patterns from top to bottom, and use the function body which corresponds to the first pattern the input conforms to. This means the order we specify the patterns in matters. If we instead defined our function like this:

```
is7 _ = False
is7 7 = True
```

The function would match with the first pattern, and thus return False for an input value of 7, which of course is incorrect.

Control Structures In Haskell, there are a few different ways to put conditions on on values. The first of these are *if* expressions:

```
greaterThan100 = if (>100) then "Great" else "Small"
```

In order to make if statements more readable when you have several conditions, we can use *guards*:

```
greaterThan n
    | n > 100 = "great"
    | n > 1000 = "very great"
    | otherwise = "small"
```

If the boolean expression within a guard evaluates to True, the corresponding function body is used. If it evaluates to False, the next guard will be checked in a similar fashion. The *otherwise* expression is simply defined as $otherwise = True$, and thus will always evaluate to True, serving as a sort of catch-all.

Finally, we can use *case expressions*. These basically work as pattern matching inside functions. The syntax for case expressions are pretty simple:

```
case expression of pattern -> result
                   pattern -> result
                   pattern -> result
                   ...
```

Thus, we can redefine our `is7` function like this:

```
is7 n = case n of 7 -> True
                  _ -> False
```

2.2.2 Defining Lists

Lists are by far the most common data structures in Haskell, and thus deserve their own subsection in this minimal introduction. Lists are defined like this:

```haskell
numbers = [1,2,3,4,5]
```

They can also be defined as ranges:

```haskell
numbers = [1..5]
```

Since Haskell is a lazy language, we can define infinite lists. This can be very useful when dealing with the concept of infinity:

```haskell
positiveInts = [1..]
```

The : operator tacks an element onto a list. The way `numbers` has been defined thus far is a special syntax for:

```haskell
numbers = 1:2:3:4:5:[]
```

Basic List Functions Functions operating on lists are abundant in Haskell. The more advanced ones will be covered in later sections. Some of the basics are:

- combining lists with the ++ operator:

```
>> numbers ++ [6,7,8,9,10]
[1,2,3,4,5,6,7,8,9,10]
```

- accessing elements:

```
>> head numbers -- first element
1
>> tail numbers -- list with head removed
[2,3,4,5]
>> last numbers -- last element
5
>> init numbers -- list with last removed
[1,2,3,4]
>> numbers !! 1 -- access by index
2
```

Many more basic functions exists, with self explanatory names, such as `reverse`, `sum` and `maximum`.

List Comprehensions List comprehensions in Haskell are similar to set comprehensions in mathematics. If we wanted to express the set of all squares of natural numbers that are smaller than 10, we would write the following set comprehension:

$$S = \{x^2 \mid x \in \mathbb{N},\ x \leq 10\}$$

If we wanted to express the same thing in haskell, we could use the following list comprehension:

```haskell
s = [x**2 | x <- [1..], x <= 10]
```

As we can see, the notation is very similar to that of set comprehensions. Here we also see laziness being put to good use, with the expression of all natural numbers as the infinite list `[1..]`.

2.2.3 Type Declarations and Typeclasses

While a more thorough explanation of types in Haskell will be given in section 2.4, this subsection will serve to give the minimal introduction to types needed to understand further sections.

Type Declaration of Values While the compiler can infer the type of most values, we can also give a value a *type declaration*. This is done to help the compiler in cases of ambiguity, and when we want to make the type explicit for the reader. Values can be given a type declaration with the `::` operator. Types are always written in upper case. For instance, the type declarations for `number` and `numbers` are as follows:

```
number :: Int
numbers :: [Int]
```

Type Declaration of Functions It is common to annotate a function with its type declaration, as this makes it clearer for the reader what the function does. Also, as previously stated, while types are inferred in Haskell, the compiler sometimes needs help in cases of ambiguity. The type declaration for `average` is as follows:

```
average :: Int -> Int -> Double
```

As usual, the :: operator is used to indicate that a type declaration follows. The arrows before the last type indicate that this is the type of the function's output. The preceding types are the types of the input. The reason why there are also arrows between these has to do with *currying*, and is explained in subsection 2.3.2

Type Variables Type variables are used in the type declaration of polymorphic functions. Let's examine the type declaration of the `head` function:

```
head :: [a] -> a
```

The type variable `a` means that the head function can operate on lists of any type. We can see that it is a type variable because it is lower case.

Typeclasses As previously stated, typeclasses are a key feature of Haskell that allows the grouping of types according to what functions are allowed to be performed on them. For instance, all types that can act like numbers, such as `Int`, `Double`, `Float` etc. belong to the `Num` typeclass. Thus, the type declaration of add can really be defined like this:

```
add :: Num a => a -> a -> a
```

Here we see that the => operator is used to indicate that the typeclass(es) apply to the type(s) that follow. If we tried to use this function on a type that does not belong to the `Num` typeclass, such as a `String`, the compiler would throw an error. Examples of other often-used typeclasses are `Eq` for types that support equality testing, `Ord` for types that can be orderedg and `Show` for types that can be shown as strings. More abstract concepts that will be covered later, such as Functors, Applicatives and Monads are also expressed in Haskell as typeclasses.

2.3 Working with Functions

With the minimal introduction out of the way, we can take a deeper look into how we can work with functions in Haskell. Common techniques for writing functions include pattern matching, recursion and building higher order functions. Higher order functions are functions that either return functions or take them as input parameters. The main source for this section is chapter 5 and 6 of *Learn You a Haskell for Great Good* [5].

2.3.1 Pattern matching and recursion

Pattern matching can be used to define functions recursively. This is a common way to define functions in Haskell. A classic example of recursion in Haskell is getting the nth number of the Fibonacci sequence:

```
fib 0 = 0
fib 1 = 1
fib n = fib(n-1) + fib(n-2)
```

On Lists Recursion is commonly used in Haskell to iterate through lists, or similar data structures. For example:

```
maximum [x] = x
maximum (x:xs) = max x (maximum xs)
```

This function gets the maximum of a list. The first function body says that if the list contains only one element then this must be the largest element. This is the edge case, needed for the function to terminate. The other function body says that the maximum of a list is the largest value of (the first value) and (the largest of the rest of the list). This is a common way of thinking when writing functions on lists recursively: First think of all edge-cases, for some minimal input where the answer is obvious, and then cover the general case by pattern matching to split the list into a head and a tail [5].

2.3.2 Higher order functions

Higher order functions are functions that either take a function as one of its arguments, return a function, or do both [5]. Higher order functions is a really powerful concept that is used a lot in Haskell.

Curried Functions

In Haskell, all functions take one parameter and return one value. So why can we define functions that seemingly take multiple arguments, like for instance the add function? The answer is that functions with multiple arguments really are *curried functions*. As previously stated, functions can also return functions. For instance, the type declaration of `add` can also be written like this:

```
add :: Num a => a -> (a -> a)
```

Here we can see that add is really a higher order function that takes a number and returns a *function*. This function takes a number as input, and finally returns a number. When evaluating `add 4 5`, what really happens is that first the function `add 4`, which adds 4 to its input is created, and then this function is applied to 5. This results in 9, the output of the entire function.

Currying allows the *partial application* of functions: we can create new functions by leaving out parameters to a given function. For instance:

```
addTo4 :: Num a => a -> a
addTo4 = add 4
```

Here we can see that applying 4 and 5 to add is equivalent to applying 5 to addTo4:

```
>> add 4 5
9
>> addTo4 5
9
```

Map

Higher-order functions that take functions as input are very common in Haskell. One of the most common of these, is `map`, which takes a function and a list as its arguments, and applies that function to all of the list's elements:

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

For instance, if we want to add 4 to every number in a list, we can do the following:

```
>> map (add 4) [1,2,3,4]
[5,6,7,8]
```

Here, we have used partial application to create the function `add 4`, which adds 4 to any number.

Filter

Another fundamental higher order function is the `filter` function. It takes a predicate (a boolean-valued function) and a list as its input, and outputs a list where all the elements that don't satisfy the predicate are filtered out:

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs)
    | p x       = x : filter p xs
    | otherwise = filter p xs
```

Some examples:

```
>> filter (>2) [1,2,3,4]
[3,4]
>> filter even [1,2,3,4]
[2,4]
```

Fold

The final fundamental higher order function covered here is `fold`. It takes a binary function as its first argument, a starting value (often referred to as an *accumulator*) as its second argument, and a list as its third argument. The fold function then applies the binary function to the accumulator and the first element of the list, which results in a new accumulator. The binary function is the applied to the new accumulator and the second value of the list, and so on until it has been through the entire list. A fold can work either from the left of a list (`foldl`) or from the right (`foldr`). For instance, a function that finds the sum of all the elements of a list can be implemented very succinctly with a fold:

```
sum' :: Num a => [a] -> a
sum' = foldl (+) 0
>> sum' [1,2,3,4]
10
```

Folding is a very powerful concept that can be used anywhere you want to traverse a list and return a value based on that traversion [5]. What is output by the fold needs to be of the same type as the accumulator, but other than that, it can be anything. We can for instance implement *map* with *foldr*, where the accumulator is a list:

```
map' :: (a -> b) -> [a] -> [b]
map' f xs = foldr (foldingFunc f) [] xs
foldingFunc f x acc = f x : acc
```

Lambda functions

We often want to define functions quickly in-line without giving them a name. This can be done in Haskell with lambda functions. They are defined with a \, which is meant to look like λ, the greek letter lambda. For instance:

```
sum = (\x y -> x + y)
```

Lambdas are often used as the argument to maps, filters and folds. For instance:

```
sum'' :: (Num a) => [a] -> a
sum'' = foldl (\x y -> x + y) 0

reverse' :: [a] -> [a]
reverse' = foldl (\acc x -> x : acc) []
```

This short but clear implementation of reverse' shows the power of lambdas combined with folds.

Application and Composition

In Haskell it is common to define functions as combinations of other functions. There are two operators that make this process easier: the application operator and the composition operator.

Application Normally, we need to use parenthesis to denote that a function is applied to the result of a larger computation. The application operator $ can make this notation simpler:

```
-- equivalent definitions of h:
h x = f (g x)
h x = f $ g x
```

In addition to saving parentheses, $ also has the advantage that it expresses the notion of function application *with a function*. Thus it can be used to for instance map the application to a value to a list of functions [5]:

```
map ($ 3) [(4+), (10*), (^2), sqrt]
[7.0,30.0,9.0,1.7320508075688772]
```

Composition In mathematics, function composition is defined like this:

$$f \circ g(x) = f(g(x)) \tag{2.1}$$

A special operator for composition is useful when chaining many functions together. That is why haskell has a composition operator ., which works just like in mathemathics:

```haskell
-- equivalent definitions of h:
h = \x -> f(g x)
h = f . g
```

The latter equation is much more succinct and readable.

2.4 Haskell's Type System

Now that an introduction to functions in Haskell has been given, it is time to look at *types*. Understanding the type system is a very important part of learning Haskell. This section will explain the type system more in-depth, covering topics such as algebraic datatypes, value constructors, type constructors, kinds and type classes. The main source for this section is chapter 3 and the first half of chapter 8 in *Learn You a Haskell* [5].

2.4.1 Algebraic datatypes

In Haskell, a type is defined by the all the values that it can have. For instance, `Bool` is defined like this:

```haskell
data Bool = False | True
```

The part before the "=" symbol specify the name of the type. The part after "=" consists of *value constructors*, which specify what values the type can have. `|` is read as "or".

Shapes In other to get to know algebraic datatypes better, we will look at a modified version of an example from the section "algebraic data types intro" from chapter 8 in *Learn You A Haskell* [5]: Let's say that we want to define a type `Shape`, that can either be a circle or a rectangle. A circle is defined by three numbers: its radius and the 2D-coordinates of its centre. A rectangle is defined by the 2D-coordinates of its two opposing corners:

```haskell
data Shape = Circle Float Float Float | Rectangle Float Float Float Float
```

Here `Circle` is the value constructor for a circle, and `Rectangle` is the value constructor for a rectangle. Now, if we wanted to express a circle with a radius of 1 and a center in (0,0), we would write:

```haskell
ball = Circle 1 0 0
```

Let's look at the types of `ball` and `Circle`:

```haskell
ball :: Shape
Circle :: Float -> Float -> Float -> Shape
```

From the type of `Circle`, we can see that a value constructor is a *function* that takes some parameters and returns a value of the type it belongs to.

Now, if we want to write a function that gets the surface of a shape, we can pattern match on value constructors:

```haskell
surface :: Shape -> Float
surface (Circle r _ _) = pi * r ^ 2
surface (Rectangle x1 y1 x2 y2) = (abs $ x2 - x1) * (abs $ y2 - y1)
```

Record syntax

Sometimes it can be confusing to define a value by using a value constructor and its parameters in sequence. For instance, when defining the `circle` above, it might be easy to forget which of the `Floats` represent the radius and which represent the point. This confusion can be cleared up with record syntax, which looks like this:

```haskell
data Shape' =
    Circle' {radius :: Float, x :: Float, y :: Float} |
    Rectangle' {x1 :: Float, y1 :: Float, x2 :: Float, y2 :: Float}
```

Now, a circle can be constructed by specifying each field by name:

```haskell
ball = Circle' {radius = 1, x = 0, y = 0}
```

2.4.2 Type constructors

Just like a value constructor can take values as parameters to create a new value, a type constructor can take *types* as parameters to create a new types. To further explain this concept, some common type constructors are covered in this subsection.

Maybe

The type constructor `Maybe` is used to represent a value that might or might not exist. It is implemented like this:

```haskell
data Maybe a = Nothing | Just a
```

Here we see that the type parameter `a` is a type variable, i.e. it can be any type. The `Maybe` type constructor can be used to express functions that we know will fail for certain values. An example of such a function, is `find`, from the `Data.List` module:

```
find :: Foldable t => (a -> Bool) -> t a -> Maybe a
```

From its type signature, we can see that `find` takes a predicate function as its first argument, and a value inside a type constructor of the `Foldable` typeclass as its second argument. Foldable types are types that can be folded over, such as lists. The function returns a type of `Maybe a`. It will return `Just` the first element from the left in the structure (commonly a list) which satisfies the predicate, or `Nothing` if no elements satisfy this predicate:

```
>> find (> 10) [0, 6, 12, 16]
Just 12
>> find (> 10) [0 .. 6]
Nothing
```

`Maybe` is useful because it clearly expresses the possibility of failure. If we see from the type declaration that the function returns a `Maybe a`, we know that the function might return nothing. This is a more graceful way of failing compared to throwing errors, as the function does what the type signature says it will do, even when no result can be given.

Lists and Trees

Data structures that contain data of a given type are created using type constructors. Lists, for instance, are defined this way. Another common container for data are binary trees:

```
data Tree a = EmptyTree | Node a (Tree a) (Tree a)
```

We see that a `Tree` can contain data of any type `a`. A binary tree is either empty, defined by the parameterless value constructor `EmptyTree`, or it is a node containing a value, and two other Trees. This allows us to recursively build trees.

2.4.3 Kinds

Just like a value has a type, a type has a kind. This might sound confusing, but it is actually rather simple. To get the kind of a type, we write ":k". Here are a few examples:

```
>> :k Int
Int :: *
>> :k Maybe
Maybe :: * -> *
<< :k []
[] :: * -> *
```

Concrete types, like `Int`, are denoted by a star *. Concrete types are types which don't take any type parameters. Values can only have concrete types. We see that the kind of `Maybe` is `* -> *`. This means that it takes a concrete value as a parameter, and returns a concrete value. It wouldn't make any sense for a value to be of type `Maybe`, but if we give `Maybe` a concrete type, like String, we get a concrete type `Maybe String`, which a value can have.

2.4.4 Defining type classes

As previously stated, a type class defines a one or more of functions that that all types belonging to it must have an implementation of. To make a type belong to a type class, we need to make it an *instance* of it. A type class is implemented as a set of functions that can be used on types that are instances of it. For example, the `Eq` type class, representing types that can be equated, is defined the following way:

```
class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
    x == y = not (x /= y)
    x /= y = not (x == y)
```

We see that the type class contains two functions, "==" and "/=", which both take two values of the same types as its arguments, and returns a Boolean value. Next we see that these functions are implemented recursively, by stating that two values are equal if the are not different, and different if they are not equal.

Deriving Instances To learn more about deriving instances, we will look at another example from *Learn You a Haskell* [5]: Let's say we have made our own data type, called `TrafficLight`:

```
data TrafficLight = Red | Yellow | Green
```

If we wanted to equate this datatype with the == and /= operator, we would need to make it an *instance* of the `Eq` type class. This can be done in two ways. The first is to let Haskell automatically *derive* it with the `derive` keyword:

```
data TrafficLight = Red | Yellow | Green deriving (Eq)
```

Using `derive` will make a type an instance of the given type class with a default implementation. In the case of `Eq`, values will be considered equal if the value constructor and all the parameters are equal. This means that the parameters also need to have an instance of the `Eq` type class.

Defining Instances The other way to make a datatype an instance of a type class, is to define it by hand:

```
instance Eq TrafficLight where
    Red == Red = True
    Green == Green = True
    Yellow == Yellow = True
    _ == _ = False
```

If we look back to the class declaration for the `Eq` type class, we see that == is defined in terms of /=, and vice versa, which is why we only need to overwrite one of them in the instance declaration for `TrafficLight`. This is what is called the *minimal complete* definition for the type class. It is the minimum of functions that need to be defined in the instance of a type for to be part of the given type class.

2.5 Some Higher Level Typeclasses

Haskell makes use of some quite abstract concepts, stemming from the branch of mathematics known as *category theory*. This is where we get terms such as functors and monads from [4]. These concepts are implemented in Haskell as type classes. The most essential of these, monoids, functors, applicate functors and monads, are covered here. While this is a very deep topic, we will restrict our scope to the parts that are relevant to the implementation of our system. The main source for this section is chapter 11 and 12 from *Learn You a Haskell* [5].

2.5.1 Monoids

A monoid consists of "an associative binary function and a value which acts as an identity with respect to that function" [5]. For something to act as an identity with respect to a function, this function called with a given value and this identity as its arguments will result in the given value. For instance, for numbers, the number *0* along with the function *+* is a monoid. The binary function *+* adds two numbers together producing a third number, and adding a given number with 0 will return that number. *1* and * are also a monoid, since * is a binary function on numbers that returns a number, and doing ** 1* on any given number will return this number. The monoid typeclass is implemented like this in Haskell:

```
class Monoid m where
    mempty :: m
    mappend :: m -> m -> m
    mconcat :: [m] -> m
    mconcat = foldr mappend mempty
```

Here, `mempty` is the identity, `mappend` is the binary function, and `mconcat` is a way of applying mappend sequentially to a list of monoid values to flatten the list into a single monoid value.

Lists as monoids

List are also an instance of the monoid class. It's implemented like this in Haskell:

```
instance Monoid [a] where
    mempty = []
    mappend = (++)
```

This makes sense, as "++" is a binary function that takes two lists and returns a single list, and adding an empty list to a list results in an unchanged list.

2.5.2 Functors

The typeclass `Functor` is for types that can be mapped over. For a type to be able to act like a functor, it needs to be a type constructor with one parameter, i.e be of the kind `* -> *`. A common analogy for such types, is that of a "box" or a context". Lists and trees for instance, can be viewed as a box or a context which contains values of a given type. The same goes for Maybe a, where Just a contains a value of type a.

Class Implementation with `fmap`

The Functor typeclass consists only of the function `fmap`, with no default implementation:

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

From the type signature of `fmap`, we can see that it takes a function from type a to b as its first argument, and type constructor f applied to a as its second argument. It then applies this function to the element(s) inside the functor, returning a type of f applied to b. In other words, it allows us to apply context-free functions to values wrapped in a context. This can be made very clear if we write fmaps type signature like this:

```
class Functor f where
    fmap :: (a -> b) -> (f a -> f b)
```

Here, it is made clear that fmap takes a function and returns a function with the variables wrapped in a context. How fmaps works is also illustrated in figure 2.1.

Examples of Types as Functors

What follows are a few well known examples of types acting as functors:

Lists Lists are implemented as functors in a quite simple way:

```
instance Functor [] where
    fmap = map
```

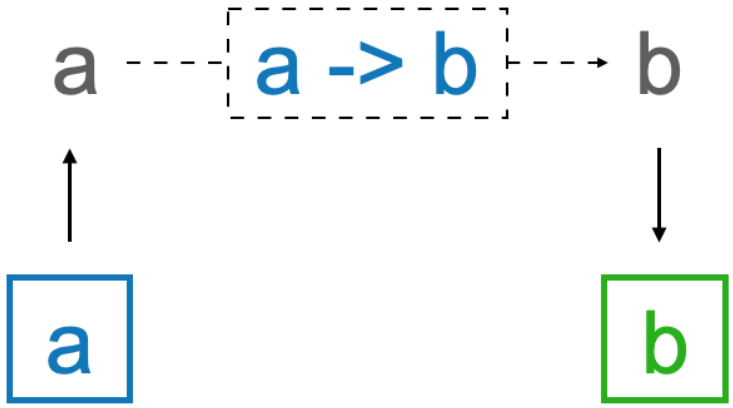


Figure 2.1: How `fmap` works for Functors. A solid box around a value represents a context. The inputs are displayed in blue and the output is displayed in green. This figure is based on a similar figure created by by Bhargava [7].

Maybe Maybe is implemented as functors like this:

```
instance Functor Maybe where
    fmap f (Just x) = Just (f x)
    fmap f Nothing = Nothing
```

It uses pattern matching on the value constructors. if it is an x is wrapped in a Just, we apply f to x inside a Just. If it is Nothing, we simply return Nothing, as we cannot apply a function to Nothing.

Binary Trees

```
instance Functor Tree where
    fmap f EmptyTree = EmptyTree
    fmap f (Node x left right) =
        Node (f x) (fmap f left) (fmap f right)
```

For binary trees, `fmap` is implemented recursively. Applying `fmap f` to an empty tree will naturally result in an empty tree. Applying `fmap f` to a Node will apply `f` to the node's value, and `fmap f` on both of its children.

2.5.3 Applicative Functors

Applicative Functors are Functors with extra functionality: They allow us to take functions *in a context*, and map it to a a value in a context:

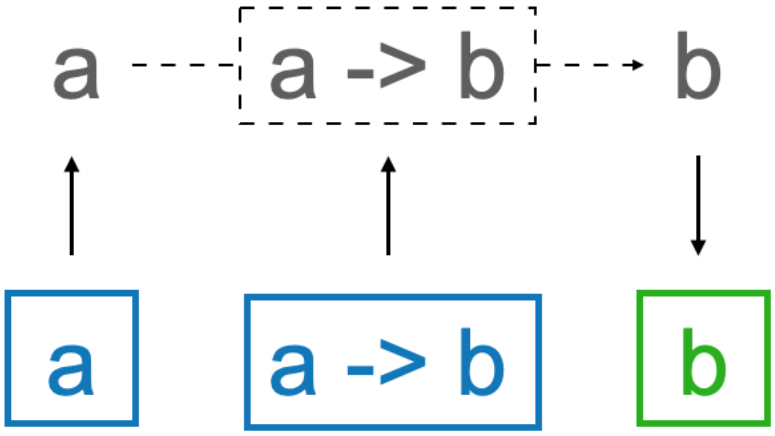


Figure 2.2: How <*> works for applicatives. A solid box around a value represents a context. The inputs are displayed in blue and the output is displayed in green. This figure is based on a similar figure created by by Bhargava [7].

```
class (Functor f) => Applicative f where
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

The first function, `pure`, takes a value of any type and returns an applicative functor wrapped around the value. This is referred to as putting the value in in a *default context*. The second function <*> looks very similar to `fmap`, except that the function `a -> b` is wrapped in an applicative `f`. The function works by first taking both `f (a -> b)` and `f a` out of their context. Then the function is applied to the value of type a, resulting in a value of type b, which is then wrapped in a context and returned. This process is illustrated in figure 2.2.

Maybe as an Applicative Functor

Maybe has an instance of the Applicative typeclass:

```
class (Functor f) => Applicative f where
instance Applicative Maybe where
    pure = Just
    Nothing <*> _ = Nothing
    (Just f) <*> something = fmap f something
```

The default context of `Maybe` is `Just`, which makes sense. We see that <*> is implemented by pattern matching. For `Nothing`, it always returns nothing. If we have a function wrapped in a Just, we extract this function and apply it to the other `Maybe` with `fmap`. This means we can do computations while keeping context:

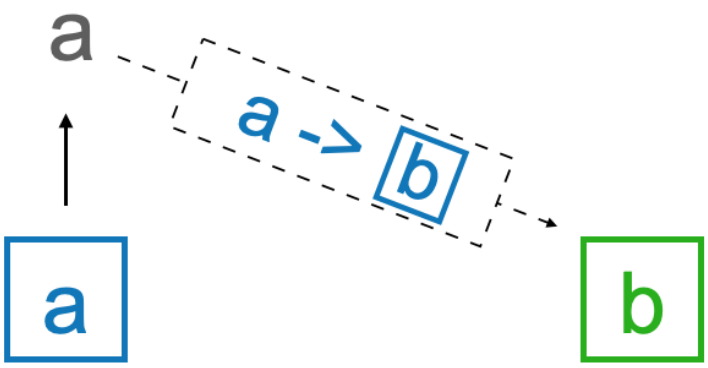


Figure 2.3: How bind (»=) works for Monads. A solid box around a value represents a context. The inputs are displayed in blue and the output is displayed in green. This figure is based on a similar figure created by by Bhargava [7].

```
Just (+2) <*> Just (2)
>> Just 4
Just (+2) <*> Nothing
>>Nothing
pure (+) <*> Just 3 <*> Just 5
>> Just 8
```

2.5.4 Monads

Monads are applicative functors with extra functionality. Let's dive right in and look at how its type class is implemented:

```
class Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
    (>>) :: m a -> m b -> m b
    x >> y = x >>= \_ -> y
    fail :: String -> m a
    fail msg = error msg
```

First, we see that the function `return` is just `pure` from the `Applicative` typeclass. It takes a value and puts in its default context. The next function, »= (pronounced "bind"), is the meat and bones of the Monad typeclass. It takes a value with a context as input, unpacks it from this context, and applies the appropriate function that returns a new value in the appropriate context [5]. This process is illustrated in figure 2.3. Implementing bind along with return is enough for a minimal complete definition of an instance of the Monad typeclass. The other two functions, »

and `fail` are provided a default implementation. These are less important to how monads function and are therefore ignored for now [8].

Maybe as a Monad

Monads allow us to combine functions that take context-free values and return values with context. This is done with the »= operator. This is an infix operator, with the value in a context on the left and the function on the right. Let´s see how Maybe is implemented as a Monad [5]:

```
class Monad m where
instance Monad Maybe where
    return x = Just x
    Nothing >>= f = Nothing
    Just x >>= f  = f x
    fail _ = Nothing
```

We see that the default context is `Just`. The unwrapping from context in the bind function is done with pattern matching. Binding any function to `Nothing` returns `Nothing`. Binding a function `f` to `Just x` applies this function to `x`.

Chaining Table Lookups The usefulness of Maybe as a monad is illustrated in a modified version of an example from *All About Monads* [8]: Let's say you need to perform a series of lookups on the form `Int -> Maybe Int`. Each lookup might either return `Nothing`, or it might be return a value in the form `Just Int`. So, how do we chain these? The problem is that the types don't match, since the output is wrapped in `Just`. Clearly we need a way to unwrap the value from its context, before applying the next lookup. This can be done with the bind operator:

```
lookupA :: Int -> Maybe Int
lookupB :: Int -> Maybe Int
lookupAB :: Int -> Maybe Int
lookupAB n = return n >>= lookupA >>= lookupB
```

We see that by composing the two look ups, (which each has the type signature `Int -> Maybe Int`), with the bind operator, we have created a new function `lookupAB`, that is also of type `Int -> Maybe Int`! If any of the lookups in `lookupAB` return `Nothing`, the entire function will return `Nothing`. What the monad is doing for us is abstracting out the concept of unpacking a value from a context and acting accordingly to this context to return a value in a new context.

State as a Monad

A purely functional language cannot update a global state, as it violates referential transparency. A common strategy to get around this problem is to "thread" a state parameter through a sequence of functions, letting the state be updated along the way [8]:

```
states (state0) =
    let (a, state1) = f1 state0
        (b, state2) = f2 state1
        (c, state3) = f3 state2
    in ([a,b,c], state3)
```

This approach works, but is difficult to maintain and generalise. Instead, we can abstract away the threading of the state with the bind operator of a `State` monad.

The State Type Let's take a look at how the `State` type is defined in Haskell [8]:

```
newtype State s a = State { runState :: (s -> (a,s)) }
```

We see that a State is defined as a *function* from an initial state to a tuple of a resulting value and an updated state. Such a function is often called a *stateful computation* [8]. It is wrapped in a type constructor `State`, which takes a state variable of type `s` and a value of type `a` as its parameters. If we look at how we defined our stateful computations by hand in `states`, we see that `f1`, `f2` and `f3` all must have the same type as `runState`, since they take a state, and return a tuple of an updated state and a returned value.

The State Instance of Monad Let's take a look at how the monad instance is defined for State [8]:

```
instance Monad (State s) where
    return a        = State $ \s -> (a,s)
    (State x) >>= f = State $ \s -> let (v,s') = x s in runState (f v) s'
```

We see that `return` simply takes a value and inserts it into a state function that leaves the state unchanged. The bind function takes a `State x` and a function `f`, and returns a new `State`. The stateful computation inside this returned `State` is a function that first runs the stateful computation `x` on the input state `s` to get a new state `s'` and a new value `v`. It then applies `f` to this new value `v` to get a new stateful computation, which then is applied to `s'`. In other words, we can see that the new stateful computation is a composite of the stateful computation `x` and `f` [5]. Now we can write our stateful function `state` with the `State` monad:

```
states' :: State s [a]
states' =
    state f1 >>= (\a ->
    state f2 >>= (\b ->
    state f3 >>= (\c ->
    return [a,b,c])))

states :: s -> (a,s)
states s0 = runState states' s0
```

We see that the *auxiliary* function `states'` takes care of the passing of the states by chaining the stateful computations with the bind operator. We use `state` to turn a stateful computation into a a `State` value. When providing the same `s0` as in our previous, manually state-passing definition of `states`, we will get the same result.

Do Notation

Another useful aspect of monads in Haskell, is that they support a simple notation that does away with the bind operators, known as *do notation*. With this notation, the passing of states can be written like this [8]:

```
states :: State s [a]
states = do
    a <- state f1
    b <- state f2
    c <- state f3
    return [a,b,c]
```

Every line written after `do` is a monadic value [5]. We use `<-` to unwrap the value from its context. When we compare this definition of `states` with the one that uses the bind operator, we can see how each line in the do block corresponds to one step in a chain of bind operations, and how this do notation is simply a lot cleaner to read in many cases.

2.6 Input/Output

So far we have not covered how Haskell handles Input/Output. After all, any function that contains an I/O-action is by definition impure, since its result is dependent on the outside world. A typical application of I/O in an imperative program is writing something to the terminal, and then waiting for a result. In Haskell, where expressions are evaluated lazily, how could we write such a program? The main source for this section is chapter 14 and 15 from *Haskell School of Music* [4].

2.6.1 The `IO` Monad

The answer is that in Haskell, a special type constructor known as `IO` is used to denote an *action* [4]. For instance, a value of type `IO Int` is an action that returns an `Int`. Actions that don't return any values, i.e. output actions, have a type `IO ()`, where the empty tuple, known as "no-op", is used to indicate that the returned value has no significance.

Writing and Reading Strings from Terminal

Let's take a look at a simple program that reads a string from the terminal and returns the string in all caps:

```haskell
main :: IO()
main = do
    s <- getLine
    putStrLn (map toUpper s)
```

We see that `main` is a function that returns a value of type `IO()`, i.e. an output I/O action. The function is a monadic computation, written with do notation. We first get an input string with the function `getLine`:

```haskell
getLine :: IO String
```

This value is then unwrapped from the `IO` context, and then used in `putStrLn`, which has the following type:

```haskell
putStrLn :: IO()
```

Thus, with `IO` as an instance of Monad, we can perform a series of I/O-actions in a single function.

Actions are Values

It is important to stress that while IO might seem a bit magical, `IO` is just a type constructor like any other, which means that IO-actions are *values* [4]. Thus we can for instance keep them in a list:

```haskell
actions :: IO String
actions = [putStrLn "hello", putStrLn "goodbye"]
```

We can then use `sequence` to convert these actions from type `[IO String]` to `IO [()]`, i.e. execute these actions.

2.6.2 Randomness

Functions that can generate pseudo-random numbers is a feature of almost all programming languages. Such functions are however by definition impure, as they give a different result each time they are called. Therefore, to get random numbers in Haskell, we need to use `IO` to get a random seed. This is done with the function `newStdGen :: IO StdGen`. (`StdGen` is the type for random seeds). Once we have a random seed, we can use the pure functions inside the base module `System.Random` to generate random numbers.

Generating Random Values with Pure Functions

The function `random` found in System.Random has the following type signature:

```haskell
random :: RandomGen g => g -> (a, g)
```

It takes a generator and returns a random value along with a new generator. The default type of the random value is `Int`. If we want another type we need to specify it in the type declaration. Since random also returns a new generator, we can use this to generate a new value and so on. Let's look at an example from *Learn You a Haskell*, where the flipping of three coins is simulated [5]:

```
threeCoins :: StdGen -> (Bool, Bool, Bool)
threeCoins gen =
    let (firstCoin, newGen) = random gen
        (secondCoin, newGen') = random newGen
        (thirdCoin, newGen'') = random newGen'
    in  (firstCoin, secondCoin, thirdCoin)
```

System.Random contains a lot of other useful random functions, such as randomR, which generates a random value within a given range:

```
randomR :: RandomGen g => (a, a) -> g -> (a, g)
```

Randomness with State As the reader might have noticed, `threeCoins` works by explicitly passing a state, the random generator, from function to function. Thus we can make use the `State` monad, to abstract away the passing of seeds [5]:

```
randomBoolSt :: State StdGen Bool
randomBoolSt = state (random :: (Bool, StdGen))
```

Now, we can easily define a function for tossing x amount of coins:

```
coins :: StdGen -> [Bool]
coins x gen = sequence . replicate x . randomBoolSt
```

The function `sequence` is part of Control.Monad and simply evaluates each monadic action in a traversable structure from left to right, and collects the results in a single monadic action.

2.7 Programming Guidelines

This section covers guidelines for developing software in Haskell. There are many such guides out there, often with quite overlapping advice, where the differences are mostly down to taste. It should also be mentioned that much of this advice is not specific to Haskell, but applicable to software development in general. This section is mainly based on the guidelines from the official Haskell wiki [9], *Learn You a Haskell* [5], and the chapter "Architecting your application" in Serranos book *Practical Haskell* [6].

2.7.1 Naming

As with all languages, Haskell has some conventions when it comes to naming:

Case Values, and thus functions, should be written in *lowerCaseCamel*. Types should be written in *UpperCaseCamel*. Internal code can be written in *snake_case* [9].

Names Names should be short and descriptive.Types and values should be named based on their semantic domain, not their representation. I.e. it is better to name a list of animals `animals`, than `animalList` [10].

2.7.2 Functions

The main guidelines for writing functions in Haskell are as follows:

Keep Functions Short and Sweet Functions should be "short and sweet". They should do *one thing and one thing only*, and be written succinctly, to be as readable as possible [9].

Maximize Code Reuse Splitting functions into smaller parts is also useful because it facilitates the reuse of functions in other functions. One should always strive for code reuse, as it's shorter and easier to maintain if a given functionality is only written once [6].

Use Well-Known Functions In most cases, it is best to build functions from well-known built-in functions. This makes them easier to read for others, and less likely to fail, as they are built from well-known and well-tested building blocks. For instance, when operating on lists, it is often better to use a combination of functions such as `map`, `filter` and `fold`, instead of list comprehensions [6].

Avoid Partial Functions Partial functions are functions which only work for a restricted set of possible inputs. One should try very hard to avoid such functions. A type declaration should not lie. If a function might fail, make sure this is made explicit in the type declaration by making it return `Maybe a`. If partial functions are necessary, one should document their preconditions (if not obvious) and make sure that they are only called when these conditions are guaranteed to be met [9].

2.7.3 Types

The main guidelines for writing types are as follows:

Use `Maybe` for Unsure Results If a unsure result need to be propagated, one should avoid implementing it directly into the datatype [9]:

```haskell
data Value a = Value a | Undefined
```

Instead, It is better to wrap it in a monad like Maybe:

```haskell
data Value a = Value a
Maybe (Value a)
```

This allows us handle failure more directly with the `Maybe` monad.

Use Type Signatures One should always write a type signature for every non-trivial function. This works as a sort of documentation for the reader, making it easier to understand what the function does [5].

In General: Make Illegal States Unrepresentable The goal of using types in Haskell is to make illegal states unrepresentable, that is, to make Haskell's strict type checking able to catch potential errors since they will be manifested as type errors [10] This sentiment is shared with Serrano [6].

2.7.4 General Guidelines for Software Design

In the chapter *Architecting Your Application* of *Practical Haskell*, Serrano gives a few general guidelines on software design in Haskell [6]. Much of the more specific advice has already been covered in previous subsections, so now we will look at some more general tips:

Seperate the Pure from the Impure The general rule is to keep the impure parts of the code separate from the pure functions. Pure functions are easier to test and develop, since one can always be sure that they only depend on their input. The core of the program should therefore consist of entirely pure functions. Thus one can build functionality by building functions from functions handling different sub-tasks. Each function can then be tested in isolation, without worrying about other parts of the code.

Strive for Polymorphicity Writing functions so that they are as polymorphic as possible is advantageous because it let's you use the same function in different ways. This will serve to maximize reuse. A high degree of polymorphism also makes it easier to refactor code: if one for example wants a function to map on trees instead of lists, this requires no extra work if it has been implemented to work on any Functor.

Use Type Classes Wisely Type classes are a powerful tool for abstracting the commonalities of data types. Almost every important programming concept in Haskell is implemented as a type class [6]. One should try and implement one's concepts as type classes if it makes sense. However, type classes should not be overused. Two cases where it is likely that the type class concept does not fit is if a type class has only one instance, or if one is using type classes as a direct mapping of classes from object-oriented programming, which is an entirely different concept.

Chapter 3

Euterpea

Euterpea is a Haskell library for computer music applications. It can handle note-level tasks such as music representation and algorithmic composition, as well as signal-level tasks such as low-level audio processing, sound synthesis, and virtual instrument design. Since this project will focus note-level music generation, we will only cover a selection of Euterpea's note-level functionality, namely it's system for representing music, as well as its MIDI-functionality. As a source for this chapter, we will use *Haskell School of Music*, written by Euterpea's creators, Donya Quick and Paul Hudak [4]. The latter was a famous musician and professor in computer science at Yale University, and one of Haskell's creators [11].

3.1 Representative Structures

In order to express musical concept, Euterpea employs a set of data structures, which are presented here, from lower level types such as pitch and duration, to the organisation of entire pieces of music.

3.1.1 Pitch and Duration

Pitch In Euterpea, Pitch is represented as a tuple of type `PitchClass, Octave`. PitchClass is represented as a datatype of pitch classes within a scale:

```
data PitchClass = C | Cs | Df | D | Ds ...
```

Here `Cs` is short for C sharp, and `Df` is Short for D flat, and so on. Octave is represented as an integer.

AbsPitch A pitch can also be represented by it's absolute value on a MIDI scale, i.e a number from 0 to 127, where each number corresponds to a key on a MIDI-keyboard:

```
type AbsPitch = Int
```

To convert a Pitch to an AbsPitch we can use Euterpea's built-in function `absPitch`, and vice versa with `pitch`.

Duration Duration is in Euterpea expressed with the type synonym `Dur`:

```
Dur = Rational
```

In Haskell a Rational is a rational number. For convenience, Euterpea has a set of predefined durations, for example:

```
wn, hn, qn :: Dur
wn = 1
hn = 1/2
qn = 1/4
```

Here, `wn` is short for whole note, `hn` short for half note and `qn` is short for quarter note.

3.1.2 Primitives

In Euterpea, *Primitives* are used to represent the notion of a basic musical element that can be either a Note or a Rest:

```
data Primitive a  =  Note Dur a | Rest Dur
```

As we can see, Primitives are implemented polymorphically. This allows us to specify what sort of Primitive we want. if we want just pitch information, we can use `Primitive Pitch`. If we also want to add volume information, we use `Primitive (Pitch, Volume)`.

3.1.3 Music

To represent a piece of music, Euterpea employs the data structure `Music`:

```
data Music a  =
        Prim (Primitive a)              --  primitive value
     |  Music a :+: Music a             --  sequential composition
     |  Music a :=: Music a             --  parallel composition
     |  Modify Control (Music a)        --  modifier
   deriving (Show, Eq, Ord)
```

Prim The `Prim` wrapper is used to transform a `Primitive` to a `Music` value.

Parallel and Sequential Composition The operator `:+:` is used for sequential composition, i.e. to place `Music` values after each other. This can also be done with `line`, to transform a list of `Music` values to a single `Music` value. For parallel composition , i.e. organising `Music` values so that they play simultaneously, we use the `:=:` operator. The function `chord`, will transform a list of `Music` values to a single `Music` value.

3.2 MIDI Functionality

What is MIDI? "MIDI is short for “Musical Instrument Digital Interface” and is a standard protocol for controlling electronic musical instruments" (page 125) [4]. It is fundamentally an interface for communicating *musical events*, such as note-on, note-off, for each key. It is also able to communicate musical meta-events, most often related to switching settings on a given synthesizer/virtual instrument. MIDI does, like Euterpea's Music type, only express abstract musical information, like when to start and stop pitches on one or more instruments. What instrument is actually used is up to the user.

MIDI in Euterpea Euterpea is able to stream `Music` values as MIDI with the function `play`. Due to Haskell laziness, we can define infinite pieces of music, and have them stream indefinitely with `play`. The only downside to lazily stream is that there might be some latency for Music values which are especially time-consuming to compute. If we want strict playback of finite pieces, we can use `playS`, which computes the music *before* it is streamed. To write the music to a MIDI-file we use `writeMIDI`. The process of converting a `Music` value, goes in multiple steps, but essentially consists of flattening the structure so that is a list of music events, captured in the data type `MEvent`:

```
data MEvent = MEvent {
     eTime    :: PTime, -- onset time
     eInst    :: InstrumentName, -- instrument
     ePitch   :: AbsPitch, -- pitch number
     eDur     :: DurT, -- note duration
     eVol     :: Volume, -- volume
     eParams  :: [Double]} -- optional other parameters
     deriving (Show,Eq,Ord)

type Performance = [MEvent]
```

This structure is then converted to a series of MIDI-messages or a MIDI file depending on if the user wants to stream midi or to simply save the piece as a MIDI file. This process is illustrated in figure 3.1.

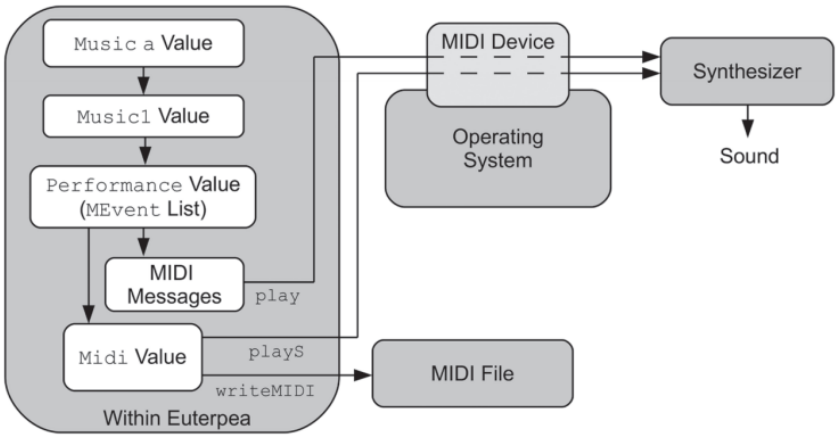


Figure 3.1: The process of converting a value of type `Music` to MIDI in Euterpea [4].

Chapter 4

Previous Work on Musical Structure

This chapter covers previous work on detecting, representing and generating musical structure. First, some papers which examine the inherence of hierarchical structure in music are presented. Then some papers on the state of the art of music generation are examined to find out how good current music generation systems are at generating well-structured music. Finally, some systems which address structure, both representative and generative, are presented. While we wish to design and implement our system in Haskell, the literature search was not restricted to systems implemented in this language.

4.1 Discovering Hierarchies Inherent in Music

In music, the notion of hierarchical structures is fundamental. Music is not merely seen as a sequence of notes, but rather as a sequence of *groups of notes*, such as motifs, melodies and chords, which further belong to larger groups, such as phrases, periods, verses and choruses. But is this hierarchical division simply a matter of convention, or is it fundamental to how humans compose and perceive music? As the two following subsections will show, both neurological research and statistical analysis seem to indicate the latter.

4.1.1 Automatic Analysis

Automatic analysis of popular music has revealed a tendency towards hierarchical structure. Da et al. has demonstrated this by devising a method of automatically detecting this hierarchical structure [12]. The system detects phrases by analysing repetitions of note sequences. A similar notion of repetition is used to further group these phrases into higher-level sections. Analysis revealed that most songs in the corpus had two or three sections, which each consisted of one to six phrases, where one to three of these very melodically distinct.

This research also showed that musical elements, such as harmony, melody and rhythm, all behaved differently at different hierarchical levels. This indicates that different levels in the musical hierarchy serve different roles, instead of just essentially being larger groupings of notes. For instance, half cadences were more often seen at the *end* of phrases, and in the *middle* of sections. This also is consistent with music theory, which states that a half cadence gives a sense of unreleased tension, while full cadences give a sense of relief.

4.1.2 Human Perception

The tension within a piece of music is central to how it is percieved by humans. While it is commonly known that tonal violations cause tension, research by Sun et al. has indicated that an increase in tension is also associated with *structural* violations, i.e. deviations from the listeners expectations for how the structure of a piece of music should be [3]. Both a behavioral rating and EEG analysis showed that structural violations elicited increasing and accumulating tension as the music unfolded. Additionally, and maybe more importantly, violations at higher hierarchical levels, such as periods, resulted in a larger and longer lasting tension response than violations at the lower levels such as phrases. This suggests that musical tension is processed hierarchically. Similar conclusions on humans ability to generate and detect hierarchies when listening to music has been indicated by Martins et al. [13].

4.2 State of the Art of Music Structure Generation

To examine how far the field of music generation has come in generating structurally sound music, three major, newer *state of the art* articles were consulted.

4.2.1 An Open Challenge

The first of the consulted papers, *A functional Taxonomy of music generation* [14], indicated that while research in music generation has achieved great progress in generating different aspects of music such as melody, harmony, rhythm and timbre, most systems struggle in creating structurally sound music:

> Why then are we not using music generation systems in our day to day lives? The above survey shows that an important overarching challenge remains: that of creating music with long-term structure. Long-term structure, which often takes the form of recurring themes, motivs, and patterns, is an essential part of any music listening experience. (page 69:24) [14]

This sentiment is also displayed by Carnovalini et al. in their paper on the state of the art of music generation systems [2]. In the subsection "Structure and Mapping" in the section "Open Challenges for Music Generation Systems" they claim

that the challenge of generating longer pieces containing reasonable repetitions and subdivisions is both difficult and often not considered.

A review paper on deep learning methods for music generation by Briot et al., seems to suggest that the problem of structure is especially prevalent in this field:

> The music lacks some structure and appears to wander without some higher organization, as opposed to human composed music which usually exhibits some global organization (usually named a form) and identified components [...] (page 989) [15]

4.2.2 Some Previous Works

While the task of generating music with reasonable global structure remains an open challenge, there are works out there, that address this problem. This section will cover five of these, which each use a different technique to represent and/or generate musical structures. The first three were pointed out in the state of the art papers.

GEDMAS: Musical Structure Generated with Markov Chains

GEDMAS, short for Generative Elecronic Dance Music Algorithmic System is a music generation system that generates full electronic dance music compositions, developed by Anderson et al. [16]. It uses a first order Markov chain model to first generate an overall *form*, and then fill in this form with chord progressions, melodies and rhythms. An nth-order Markov chain determines the probability of a given event occurring based on what has happened n steps in the past. Thus, a first order Markov chain is essentially a look-up table of transition probabilities. Figure 4.1 shows GEDMAS's transition table for different sections in the overall form of a piece. Each numbered capital letter refers to an 8-bar section of a piece. Sections with the same letter are grouped together on a macro level. Within a given macro section the numbers must be from 1 to x in rising order. The probabilites of each Markov model are learned from a corpus.

Implementations of *a Generative Theory of Tonal Music*

A generative theory of tonal music (GTTM) is a well-known work of music theory presented by music theorist Fred Lerdahl and linguist Ray Jackendoff in 1983 [17]. It is described as a search for the "grammar of music", inspired by the theories of Noam Chomsky. An implementation of this system for analysis purposes was proposed by Hamanaka et al. [18]. Here, a short explanation of GTTM 4 subtheories is found. These are: grouping-structure analysis, metrical-structure analysis, time-span reduction and prolongational reduction. The grouping-analysis hierarchically organizes a series of notes into motives, phrases, periods and higher musical structures. The metrical-structure analysis is concerned with a metric hierarchy of quarter notes, half notes, whole notes, measures etc. It seeks to identify

	8 bar Sections and Probability of Next Section								
Possible Next Sections		A1	A2	A3	A4	A5	A6	B1	B2
	A1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	A2	87.5	0.0	0.0	0.0	0.0	0.0	0.0	0.0
	A3	0.0	25.0	0.0	0.0	0.0	0.0	0.0	0.0
	A4	0.0	0.0	88.9	0.0	0.0	0.0	0.0	0.0
	A5	0.0	0.0	0.0	4.2	0.0	0.0	0.0	0.0
	A6	0.0	0.0	0.0	0.0	100.	0.0	0.0	0.0
	B1	12.5	75.0	11.1	25.	0.0	100.	0.0	0.0
	B2	0.0	0.0	0.0	0.0	0.0	0.0	91.7	0.0
	…								

Figure 4.1: Transition table for generating overall form in GEDMAS [16].

strong and weak beats. The time-span reduction assigns a structural importance of parts of a melody. The prolongation reduction ascribes the tone pitches a hierarchical importance in terms of their “tension,” “relaxation” or “duration". Figure 4.2 shows a simple melody with its corresponding time-span tree, grouping-structure and metrical-structure.

GTTM For Generation In his paper *Towards the Generation of Melodic Structure*, Groves utilises a modified gttm-representation that is based on intervals, for his melody generation system [19].

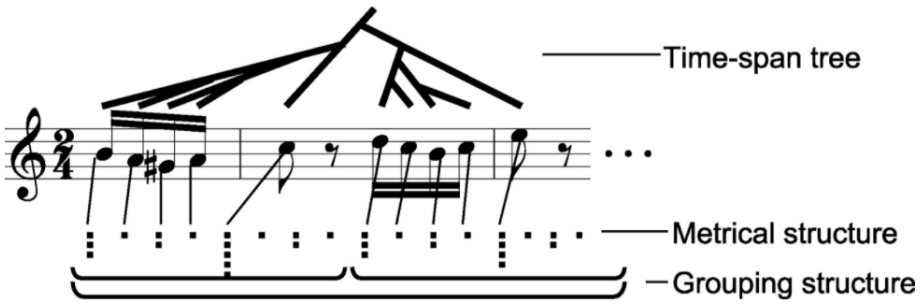


Figure 4.2: A simple melody, with corresponding time-span tree, grouping-structure and metrical-structure [18]

MorpheuS: constraining recurrent patterns

MorpheuS is a music generation system by Herremans et al. that seeks to tackle the problem of generating music with long-term structure [20]. It is able to generate polyphonic pieces that both adhere to a given tension profile and feature repeated patterns in both the long- and short-term. The tension profile describes how the tension in a given piece should develop during its run time, and is expressed via a mathematical model for tonal tension. Repeated patterns are extracted from template pieces with a pattern detection algorithm.

The system's generative algorithm takes a given tension profile and pattern structure as input, and then select pitches that best fit this profile with an optimization algorithm that constrains the long-term structure that is given by the pattern structure. The system is quite successful, as music it has generated has been performed live in concerts. Still, as the generation system generates music based on an input piece, the generated output is highly constrained to the style of this input.

Kimon: Object-Oriented Hierarchies

In *Formalising Form: An Alternative Approach To Algorithmic Composition*, Hedelin proposes *Kimon*, an object-oriented programming environment for computer-aided composition [21]. Hedelin explores the relationship between the generative algorithm and the resulting musical structure. He is critical of algorithms in which musical form is not explicitly handled, but rather appears as a by-product of the algorithm. He points out that context is at the core of how music is perceived, where the listener "tends to structure the music in terms of parts, wholes and form-structural functions" (page 251) [21]. In Kimon, his proposed system, music is created as a hierarchy of musical objects, closer to how it is perceived, as groups containing groups, shown in figure 4.3.

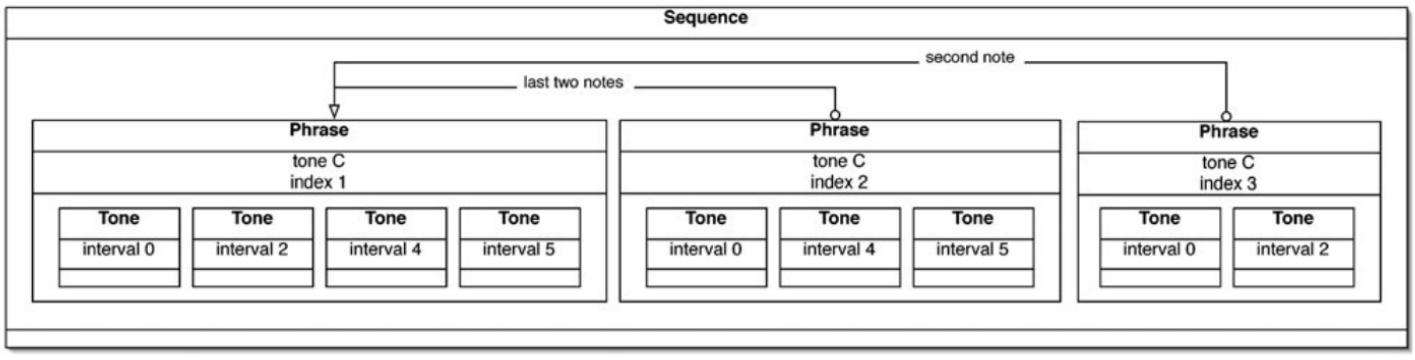


Figure 4.3: A hierarchy of musical objects denoting a small melodic sequence [21]

Representing Music with Prefix Trees

The final existing system presented here is by Yan Han, presented at the FARM conference [1]. It details a novel approach to representing shared patterns across different hierarchical levels of a piece of music. This work serves as a basis for the work done in this project, so it will therefore be covered in its own section 4.3.

4.3 Representing Music With Prefix Trees

Representing Music With Prefix Trees is a paper by Yan Han et. al, presented at the conference *Functional Art, Music, Modeling and Design* [1]. It introduces the use of a tree structure know as a prefix tree to concisely represent shared patterns across the hierarchical structure of a given piece of music. The system is implemented in Haskell. This section will cover the main ideas this paper introduces.

4.3.1 Basic types

Pitch To represent pitch, Han uses a datatype containing scale, octave and degree information:

```
data ScalePitch = ScalePitch
{ _scale :: Scale
, _octave :: Octave
, _degree :: Degree
}
```

By defining pitch in terms of an underlying scale it is easier to carry a notion of scale into the musical structure in later stages. One could simply use an integer from 0-127 to represent pitch, each corresponding to one key on a MIDI keyboard. However, this would leave out a lot of information that is contained in the scalePitch structure.

Musical Events The primitive musical element in Han's design is that of the *event*. This is the smallest playable musical element, i.e. either a note with a certain pitch or a rest. A musical event is defined with ScalePitch, duration and volume. Rests are represented as musical events with a volume of 0.

```
data Event = Event
{ _duration :: Dur
, _volume :: Volume
, _scalePitch :: ScalePitch
}
```

4.3.2 Oriented Trees

A datastructure known as an *oriented tree* is used to represent the grouping structure of a piece of music as it unfolds in time. It is implemented like this in Haskell:

```
data Orientation = H | V -- Horizontal | Vertical

data OrientedTree a = Val a | Group Orientation [OrientedTree a]
```

As we can see, an oriented tree can have two kinds of nodes:

- Value nodes, which contain a Value of a given type
- Group nodes, which consist of a list of nodes, and an associated orientation, vertical or horiziontal.

Horizontal groups are interpreted as unfolding sequentially in the given dimension, while in Vertical groups, its members are interpreted as unfolding in parallel to each other. Oriented trees are thus able to express two-dimensional data structures. How an oriented tree is interpreted with regards to time is shown in figure 4.4 and 4.5.

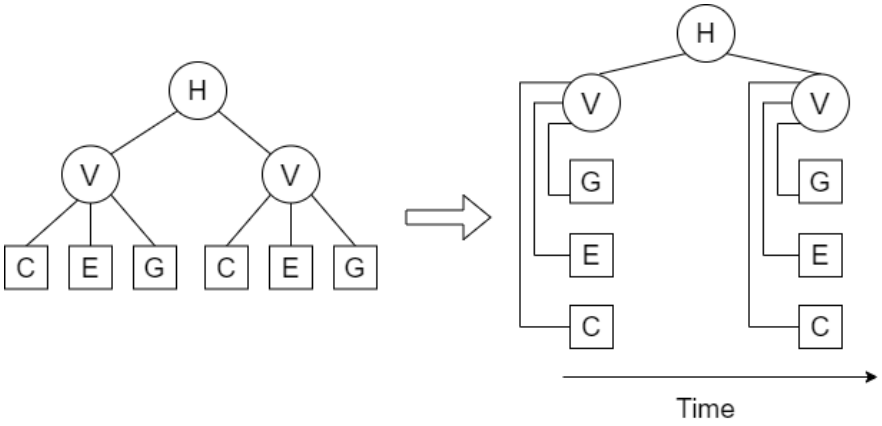


Figure 4.4: How a horizontal group of vertical groups is interpreted with regards to time. Circular nodes refer to groups and squares refer to values.

Expressing Music as a Hierarchy

Oriented trees are good for representing music, as they can be used to both capture musical information existing simultaneously and sequentially in time. For instance, the structure in figure 4.4 can be used to represent a sequence of two chords followed by a single note. Similarly, the structure in figure 4.5 can represent three musical lines starting simultaneously: Two containing three notes, and one just a single note. If we consider music on the note level to consist of a number of notes either occurring in parallel or sequentially in time, we see that this oriented tree structure is sufficient to express an arbitrarily complex piece of music.

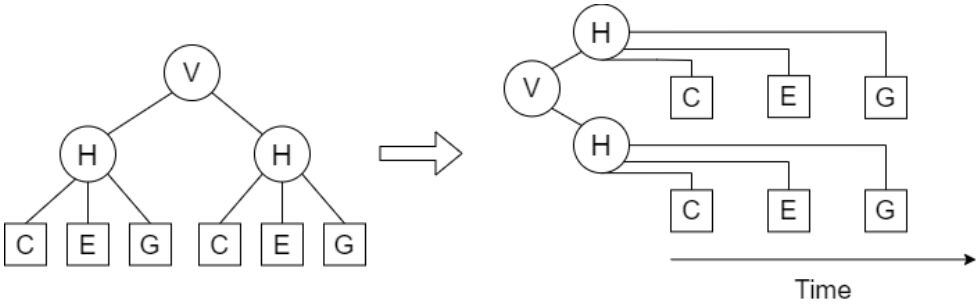


Figure 4.5: How a vertical group of horizontal groups is interpreted with regards to time.

Han also points out that this structure is advantageous because it is very similar to how music is expressed in Euterpea. This makes it trivial to convert a piece of music represented with this structure to Euterpea's internal represenational structure, so that it can be played as MIDI with Euterpea's built in functionality.

4.3.3 Representing Locations in a Tree

To uniquely identify a given leaf in a tree, all we need is its *path*, the indices of each branch taken to reach it. In music, where multiple locations in an oriented tree contain the same information, or information that needs to be modified in the same way, it is convenient to represent many paths in one value. To access multiple elements in a tree, Han introduces two datatypes: Choice, and Slice.

Choice For a given group node, a *Choice* represents a selection of its children. It is a datatype that can either be `Some [Int]`, a list of the indexes of the elements to be selected, or `All`, denoting every element:

```
data Choice = Some [Int] | All
```

Slice A slice is a list of choices, where the choice at a given index in the slice represents the members to be selected at this depth level:

```
type Slice = [Choice]
```

How slices work is demonstrated in figure 4.6. We see that the left slice denotes the path to the first member of all members of the top node, and that the right slice denotes the path to all members of the first member of the top node. This ability to represent multiple paths as one slice value is very useful. Slices express succinctly a grouping of elements that are not explicitly grouped but still are related. If, for instance, the first note of two successive motives are the same or related in any other way, they can be grouped by the slice to the left in figure 4.6.

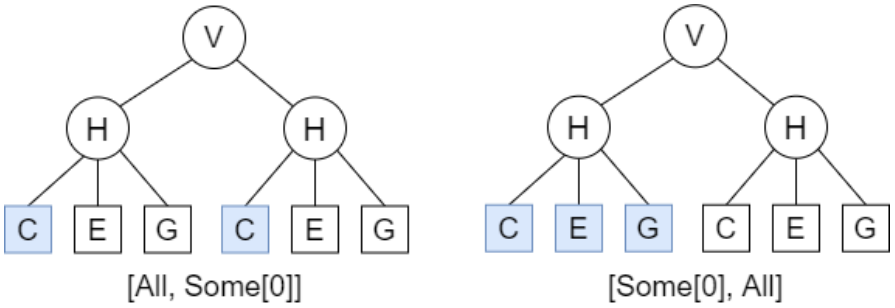


Figure 4.6: How multiple elements in a tree can be accessed with slicing

Constructing slices

To construct slices, Han introduces the concept of *slice modifiers*. These are functions that produce functions of type `Slice -> Slice`:

```
-- Sets the choice in a slice at index 0
atChords :: [Int] -> Slice -> Slice
-- Sets the choice in a slice at index 1
atVoices :: [Int] -> Slice -> Slice
```

This allows us to change the choice at a given level:

```
atChords [0, 1] [All, All] == [Some [0, 1], All]
atVoices [2, 3] [All, All] == [All, Some [2, 3]]
```

Slices can then be constructed by applying functions of type `Slice -> Slice` to a default slice which contains the entire tree i.e. `[All,All, ...]`. This is useful as it allows us to create slices *by composition*.

4.3.4 Transforming Multiple Locations in a Tree

To transform the events at multiple locations in a tree, Han introduces the concept of *tree modifiers*:

```
data TreeModifier = TreeModifier
{ _slice :: Slice
, _modifier :: Event -> Event
}
```

It contains the slice denoting the locations of the events to be modified and the function that is to be applied to these events. These modifiers can change any of the elements of the given element, such as duration, scale, octave or scale degree. In this way one can address on or more dimensions of a piece of music at multiple locations, and thus represent shared patterns across multiple locations of the tree.

4.3.5 Expressing Shared Structure with Prefix Trees

Multiple tree modifiers can together represent a complex piece of music. Han demonstrates how these tree modifiers can be represented concisely with a prefix tree.

What are prefix trees?

"A prefix tree is a tree structure that represents a map from keys to values" [1]. The values are represented by a leaf, and the keys are represented by the nodes along the path from root node to leaf. If there is shared material in these keys, then prefix trees will serve as a form of compression. The classic example of a prefix tree is when each key is a word, as shown in figure 4.7. Here we can see that "CAR" is mapped to 1, "CAT" is mapped to 2, "COOL" is mapped to 3, and "COP" is mapped to 4.

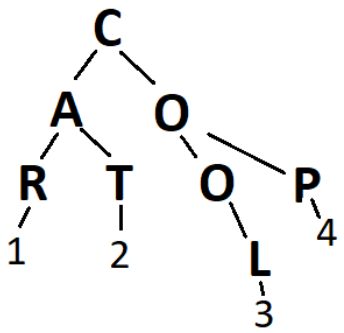


Figure 4.7: A prefix tree containing some words and their associated key

Musical Prefix Trees

Han's general polymorphic implementation of prefix trees is as follows:

```
data PrefixTree v k = Leaf k v | Node k [PrefixTree v k] deriving (Show)
```

To use prefix trees to represent a piece of music, Han introduces `MusicTree`, which maps slices to event modifiers:

```
type MusicTree = PrefixTree (Event -> Event) (Slice -> Slice)
```

Each slice is built by composing the slice modifiers along a path, leading to the leaf containing the event modifier in question. A visual demonstration of how a prefix tree can represent a piece of music, is shown in figure 4.8.

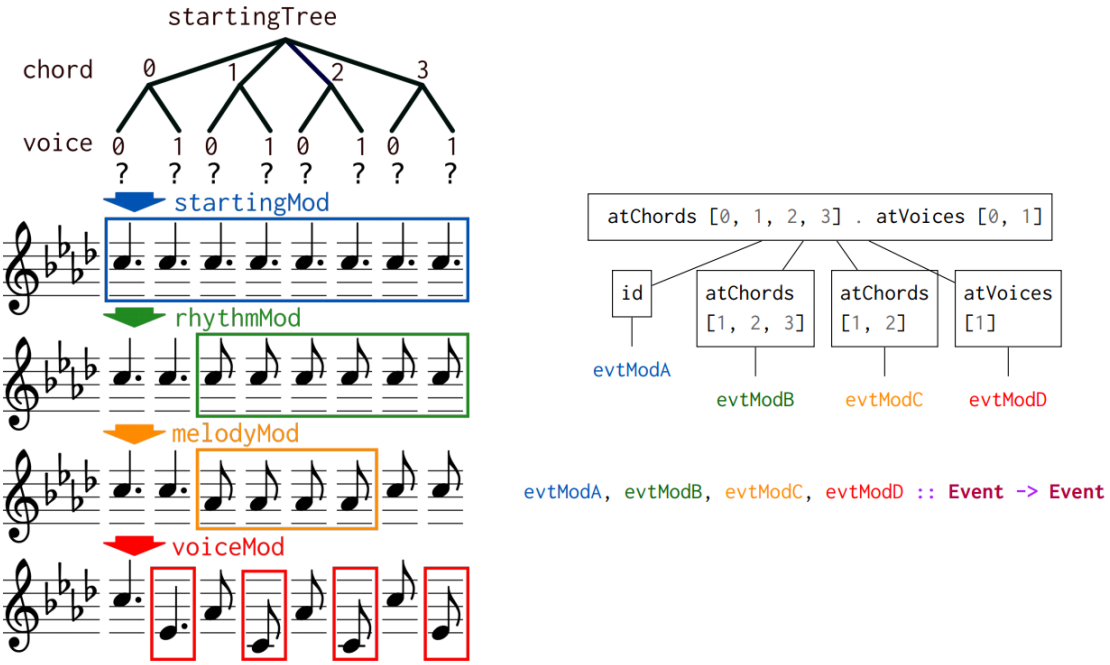


Figure 4.8: How a piece of music, defined as the series of modifications on a starting tree shown to the left is captured by the prefix tree on the right. Event modifiers on the left correspond to event modifiers on the right of the same color. [1]

Phrase Structure Musical Prefix Trees can also represent larger structures, such as *phrases*. Figure 4.9 shows the slice structure of a phrase of Johannes Brahms's Waltz in A Flat Major Op. 39 No. 15. Each colored slice modifier in the code to the left corresponds to the highlighted area of the same color in the sheet music to the right. We see that each pattern is only specified once, even though it occurs in multiple spaces.

Converting to Musical Oriented Trees

Han outlines a process for converting a Musical Prefix Tree to a Musical Oriented Tree. The first step of this process is to get a list of tree modifiers from the prefix tree from left to right:

```
toTreeModifiers :: MusicTree -> [TreeModifier]
```

The second step is to generate a starting tree filled with dummy events, based on on the prefixtree:

```
makeStartingTree :: MusicTree -> OrientedTree Event
```

```haskell
phraseA :: MusicTree
phraseA =
  Node
  ( atPhrases [0]
  . atMeasures [0, 1, 2, 3])
  [ Node (atMeasures [0, 1, 3])
    [ -- Scale
      Leaf id
        (setScale (extractTriad 0 (mkMajorScale 8)))
    , Node (atHands [0] . atChords [0, 1, 2, 3])
      [ unevenRhythmMeter -- Rhythm and meter
      , Node (atMeasures [0, 1]) [ ... ]
      , Node (atMeasures [3]) [ ... ]
      ]
    , Node
      (atHands [1] . atChords [0, 1, 2])
      [ evenRhythmMeterL -- Rhythm and meter
      , Node (atChords [0])
        [ Leaf (atMeasures [0]) ( ... )
        , Leaf (atMeasures [1, 3]) ( ... )
        ]
      , Node (atChords [1, 2]) [ ... ]
      ]
    ]
  , Node (atMeasures [2]) [ ... ]
  ]
```

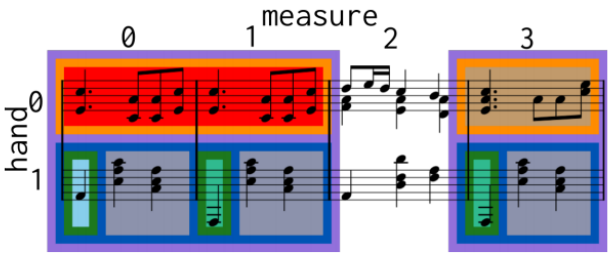


Figure 4.9: The slice structure of a phrase of music. [1]

The third and final step is to apply the list of tree modifiers to the starting tree, implemented as a left fold:

```haskell
toOrientedTree :: [TreeModifier] -> OrientedTree Event
toOrientedTree modifiers =
    foldl' (flip applyModifier) (makeStartingTree modifier) modifiers
```

Since the process for converting from prefix tree to oriented tree and later to MIDI is fully defined, a piece of music can be fully represented as a prefix tree.

Chapter 5

Specification

Our background research revealed that existing music generation systems struggle with generating music which has shared material, i.e. repetitions, across multiple hierarchical levels and multiple musical dimensions. What follows is a short elaboration on what is meant by hierarchies and dimensions in a musical context, and why these are important to consider in music generation systems:

Hierarchies Our research covered in section 4.1 affirmed our belief that music is inherently a *hierarchical phenomena*. Therefore, some sort of tree representation is unavoidable if the goal is to create well-structured music. Inspired by the sentiments of Hedelin (subsection 4.2.2), we think it is better to explicitly handle this tree-like nature of music than to make such a structure appear indirectly in our generated pieces through some sort of algorithm.

Dimensions The term *musical dimensions* is used by Yan Han to refer to aspects of music which can be separated out and viewed on their own. Examples of such dimensions on the note level are duration and pitch (pitch can further be decomposed into a scale degree and octave for a given scale). We can also consider musical dimensions on a group level, such as scale, chord type, melody, rhythms, different forms of arpeggiation. We can for instance play a given melody in many different scales and rhythms, and still recognize it as the same melody. The advantages of decomposing our representation system into multiple dimensions is that it enables us to express different structures of repetition for each dimension. For instance, we can let a given rhythmic pattern reappear throughout a piece, but for different chords.

The overarching goal of this project is to work towards the generation of music with these structural traits. In order to reach this goal, a music generation software is to be designed and implemented in Haskell. In order to refine and clearly define our goals, we must specify our requirements with regards to the software's functionality, design and implementation.

5.1 Functional Requirements

We put the following requirements on the functionality of the system, with regards to user input and generative output:

Intended Use and Users The intended use of the software is for experimental music generation and as a groundwork for further research. Therefore the intended user of the software is a person who is reasonably competent in programming, ideally with a decent understanding of Haskell.

User Input From the user's perspective, the system should not require any input to generate music. Internally, the system should generate a random seed and use this as input for the systems generative function, which should return the generated output. In order to shape the output's sample space, i.e. the generation process' possible outcomes, the user should be able to modify *parameters* within the source code itself.

Output Music The music should be generated on a note level, i.e. in MIDI format. The system should stream out a piece of randomly generated music each time it is run, across a MIDI channel that can be set by the user in `main`. The generated music should consist of chords that display a the structured repetition described at the start of this chapter, in both *rhythm* and *playing patterns*. By playing pattern we mean they way a chord is broken, for instance by arpeggiation. The output music should be cohesive and varied in its *structure*. The focus is on structural novelty, not on maximising musical "quality" or listenability.

5.2 Design Requirements

In our design specification, we will separate our design into two aspects: *representation*, i.e. the ability to express musical ideas, and *generation*, the ability to generate music in the chosen representation.

5.2.1 Representation

In order to be able to generate music with shared structure across multiple hierarchical levels and dimensions, we need a representative system that can express this:

Representing Shared Material Representing shared material across hierarchies is a fundamental goal of our design. This should be handled by a representation based on Han's prefix trees, presented in section 4.3, since they tackle the problem of representing tree structures with shared material head on. The fact that Han's

code is written in Haskell is of course a plus, and that it is a new, novel representation system, that has not been used for generation before. Additionally it is very generalized, in the sense that is does not directly or indirectly enforce any styles or aesthetic preferences.

Group-level Representations The system should be able to represent scales and chords. To capture music's hierarchical aspect, a full piece of music should be represented as a tree-like structure which can arbitrarily group notes into groups either unfolding parallelly or sequentially in time.

Primitives The *primitives* of our system should express the notion of notes and rest, i.e. the fundamental musical events that can occur in a piece. The dimensions of music are expressed by the the primitives, and thus they should be composed of pitch and duration, where each can be accessed and modified separately.

5.2.2 Generation

In order to generate music with shared material, we want to design a system which makes use of Han's prefix tree representation, modified for generation.

Generalized Procedure We wish to design a generalized pipeline for generating music with shared structure that is as general and stylistically unbound as possible. This will serve as a basis that can be further improved upon in the future. Additionally it will serve to demonstrate the generality of generating music with prefix trees. The generalized procedure should include a parameterised way to generate prefix trees, and a way to chain multiple prefix trees together to generate a full piece of music.

Generative Example In order to get some musical output of the system, we need to make some stylistic constraints. We want to design a system which makes use of the generalized procedure to generate music with shared structure. It is the generative example that the user will interface with via `main`. Thus, its design should work to generate music with the characteristics specified in the section on functional requirements on the output music.

5.3 Implementation Requirements

The design should be implemented in Haskell, by use of Euterpea's MIDI functionality.

Code Quality We want well-formed software that is readable and as easy as possible to understand and work with. The main mechanism towards achieving this is by adhering to the programming guidelines featured in section 2.7.

Extensibility The program should be extensible. It should be easy for someone who knows Haskell to look at the software and understand what's going on. This will be achieved with adequate code quality and documentation in the form of useful comments.

Performance and Stability Performance is not a main factor. What is important is displaying the validity of the design's high-level concepts. Still, performance should be good enough that we can lazily stream the music with Euterpea without noticeable timing delays. We also require the implementation to be stable; it should never fail to generate a piece of music.

Chapter 6

Design

This chapter covers the design of the system that was created in this project, based on the specification presented in the previous chapter. First, the representational structures will be presented, starting with the very basics and ending up with a presentation of the systems musical trees and modified prefix tree. After that, we will focus on how this representation system can be used in a generalized generative procedure, followed by a narrowed example of generation of structured variations of chords progressions.

6.1 Basic Representational Structures

To generate structured music, we need to be able to express musical ideas with data structures. This section will cover how basic musical elements are expressed in this design, in bottom-up order, going from pitch and duration to notes and rests, finally ending up at scales and chords.

6.1.1 Pitch and Duration

Pitch and duration are the very building blocks of of music: A piece of music can fundamentally be considered as a set of pitches and durations structured in time. Since we are using Euterpea's MIDI functionality in this project, it makes sense to also make use of its built-in Pitch and Duration data types. These are described in the background chapter on Euterpea, in section 3.1.1. Another advantage to using Euterpea's built-in representations is that it allows us to use its many handy functions, such as `absPitch` and `pitch` to convert back and forth between absolute pitch and pitch.

6.1.2 Primitives: Notes and Rests

We will also use Euterpea for expressing *Primitives*, a common type for both Rests and Notes. This is also for simplicity's sake, for similar reasons as with pitch and duration. The fact that `Primitive` is a polymorphic type is very convenient.

For now, we will operate with `Primitive Pitch` as our primitive. Another option would be to include volume information, for more natural and expressive performances, with `Primitive (Pitch, Volume)`. This however was deemed to cumbersome, especially since volume information is easy to add later, due to Primitive's polymorphism. This would simply be a matter of making our tree structure an instance of `Functor` and mapping a function that adds volume information to the tree.

Alternative: Expressing Pitch More Abstractly

An alternative to the approach chosen for this project, is to express pitch as a composite of scale, scale degree and octave, like in Han's design. This would yield the same information as Euterpea's `Pitch` type, but with greater dimensional separation, since all notes within a scale are related, and the same goes for notes of the same scale degree, or notes of the same octave. Still, this was decided against, for a few reasons:

1. Our representation is simpler and thus easier to work with.
2. Working "within the box" of Euterpea's type system is always easier.
3. Our main goal with this design is more related to *tree structure* than the primitives themselves. There is no reason to add more complexity than necessary if a simpler primitive will still demonstrate the structure-generation abilities of the system.
4. We can still work with the notion of scales, scale degrees and so on at the generative stage.

Still, the benefits of decomposing a pitch into more dimensions are not to be understated. At a later stage, abstracting pitch to a similar datatype as Han's might be benefitial.

6.1.3 Scales

A notion of scale is more or less unavoidable in order to generate music that sounds reasonable. There are two notions of scale that we want the system to have:

Scale as a Series of Pitches One way to define the C major scale is as the following pitch classes: *C, D, E, F, G, A, B*, where any note within that list belongs to this scale. This notion of scale as the list of all notes that belong to it is a necessary part of representing a Scale: If one were to select a random note from the C major scale, one would need to know what the candidate notes are.

Scale as Root and Mode A more abstract notion of scale includes information on what the root of the scale is, and the idea of a mode. For instance C major and C minor are two different scales with the same root and different modes. Specifying

a scale as a root and a mode is more abstract, shorter and more readable than as a series of all allowed pitches.

Combining the Two with Scale Constructors

In order to be able to operate with the notion of root and mode, while still working with scales as a list of allowed pitches, we use *scale constructors*, functions from root and mode to a list of allowed pitches. The list of absolute pitches serves analogously as a list of all the allowable keys to be played on a MIDI keyboard within a given scale. An advantage of this as our main representation of scale is that it is in principle possible to represent any scale. In our design we wish to focus on the diatonic scale for now, i.e major, minor, and the rest of the modes. These are by far the most popular scales in western music. Still, even though this means that the scale constructors in this system only construct diatonic scales, we can (at a later stage) construct and use other scales, such the chromatic scale, the pentatonic scale, or any custom scale.

Including a Notion of Scale Degrees

The notion of scale degrees is fundamental in music generation, as it denotes the *role* or *function* of a note within a given scale. With our notion of the notes in a scale as all the absPitches it contains in rising order, the scale degree of a note simply becomes its index modulo the zero-indexed amount of notes within the scale. For diatonic scales, this number is 7. To handle conversion from a scale degree within a give scale to an absolute pitch, and vice versa, two functions were created, one for each direction.

6.1.4 Chords

A notion of grouping notes together as *chords* is fundamental to most music, and it is required of our to design to create a type to represent it. Similarly to scales, chords can both be defined by the series of notes they are made of, or by their root, and their quality, e.g. major, minor. Thus, a similar solution to the one for scales was decided for representing chords. Chords are constructed by their root and quality, and represented as a series of pitches.

6.1.5 Musical Oriented Tree

A full piece of music is defined as a *musical oriented tree*: An oriented tree, similar to the one defined by Yan Han, with time as the dimension the tree unfolds in and the leaves as the musical primitives we detailed in section 6.1.2. Some advantages of such a representation was detailed by Han, and will be restated and expanded upon here:

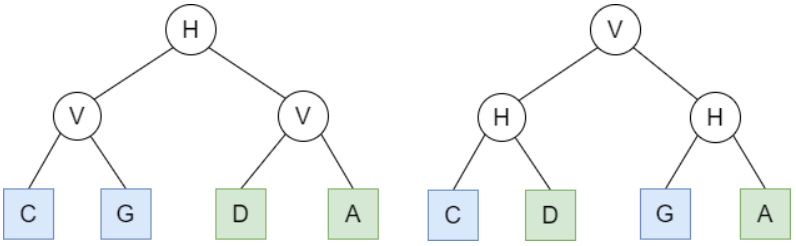


Figure 6.1: Two different oriented tree representations of identical sounding music. Notes that occur at the same time (given that all notes have the same duration) have the same color.

1. They succinctly and flexibly express how a piece of music unfolds in two dimensions: notes can be grouped to occur either simultaneously or sequentially.
2. They are not bound by any stylistic constraints, which was one of the specified requirements.
3. Their explicit tree structure allows us perform operations on multiple leaves of trees and group them arbitrarily as needed.
4. They are easily converted to Euterpea's Music Structure and thus Euterpea's MIDI playback.

An important aspect to consider with musical oriented trees is that they can express identical-sounding music in many different ways, as shown in figure 6.1. This will make it more complex to determine how the music will turn out, especially which notes will be heard at the same time. If we for instance restricted the representation system so that horizontal groups could contain vertical groups but not the other way around, it would be much easier to determine which notes are heard at the same time (essential information when trying to avoid dissonance). Still, there *is* a fundamental difference in the structural interpretations of the concrete music that each of the trees represent. The left tree considers the music a sequence of two chords, while the right tree considers the music two parallel melodies. These are two very fundamentally different ways of viewing the music, and this richness of expression is worth keeping.

6.2 Slicing Oriented Trees

The ability to access multiple nodes of a tree with slices is kept from Han's design. This is an essential part of the representation system, as the ability to group many selections of leaves into one value is very useful when we want shared material in multiple locations. Examples of such transformations that can be captured in a slice are *the last note of every other motif in a period*, or *the first chord of every chord progression in the second chorus*.

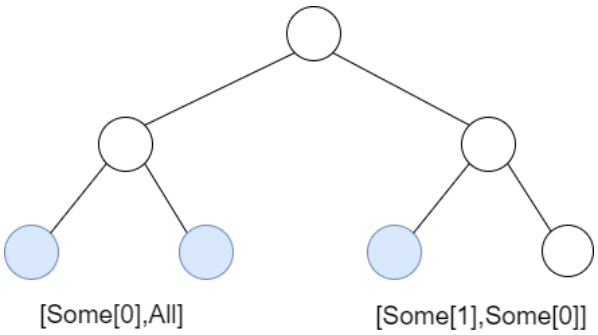


Figure 6.2: A selection that cannot be represented as a single slice, but as composite of two slices.

Expressiveness

Slices do have some limitations in their expressiveness: At each hierarchical level, the same choice must apply to all nodes. Therefore some selections are not possible to represent in a single slice, as exemplified in figure 6.2. Still, this restricted expressiveness also has some advantages: As exemplified in the introduction to this subsection, the multiple paths expressed by slices usually map well to areas where one would want the same musical transformations to be carried out. Thus, the usage of slices will be of benefit for a generation algorithm, because they tend to group musically related locations together by their nature. We can still apply the same transformations to every location marked in blue in figure 6.2, but we would need to use two slices.

Constructing Slices

Since Han's system is created for the specific purpose of recreating a single piece of music, its slice constructors are hardcoded, with functions such as `atChords` and `atPhrases`. To accommodate generation, slice construction was generalized from Han's design. Now, choices at given hierarchical depths are accessed by a function that is parameterised by the depth we want to access. Thus, we can generate pieces with an arbitrary amount of levels, and still be able to slice the nodes at each level.

6.3 Generative Prefix Trees

The *Generative prefix tree* is an essential part of the music representation system in this design. This section will cover some aspects of Han's original version that were changed, and why these changes were made. It will also explain how musical prefix trees are applied to musical oriented trees, and how they can be directly converted to a musical oriented tree.

6.3.1 Problems with Yan Han's Prefix Tree

We contacted Yan Han, author of *Representing Music with Prefix Trees*, and asked if he thought it was feasible to use his prefix tree representation for generative purposes. He replied that he thought it could be a "useful intermediate representation", but also described some weaknesses with the system:

> It had a lot of weaknesses though - the main problem was that any kind of non-hierarchical relationship (e.g. to between adjacent pitches, or between arbitrary unrelated leaves on the tree) was difficult or messy to express. In the end, the music sounded complex but pretty weird. (Quoted with permission from Han)

Expressing relationships between arbitrary leaves (even adjacent ones), is, as stated in the project description/introduction, a prerequisite for generating music, as relationships between notes is fundamental to music. It is in these grouping relationships that the repetition we seek occurs. Getting this right is therefore crucial for the success of our generation system. A modification to Hans prefix tree was proposed to allow for this.

6.3.2 Modifications for Generation

It is our purpose to modify Han's prefix tree representation to better accommodate generation, and to be able to more easily express relationships between the leaves of the tree. This subsection will walk through the different aspects of this modification.

The Problem with Leaves as Event Modifications

A way to solve our problem of expressing relationships between nodes is to use transformations which work on multiple nodes. In fact, a lot of transformations that we wish to perform make most sense when performed on a *group of notes*. For instance, many transformations from classical music theory, such as inversion and retrograde, only make sense when applied in a context of a sequence of notes. Similarly, for chords, transformations such as inversions, only make sense when considering a group of notes. In Han's design, the leaves of the prefix tree are functions of type `Event -> Event`, i.e. functions that operate on the note level. This means that music is represented as a series of modifications on a starter tree, note-by-note. One *could* envision a form of generation that featured functions on groups of notes, while keeping the leaves functions that operate on the note level. This would require a function that created a series of note-level functions, that when applied in succession on a sequence of notes would result in a group modification. Still, one wishes for a less indirect solution that is simpler to implement.

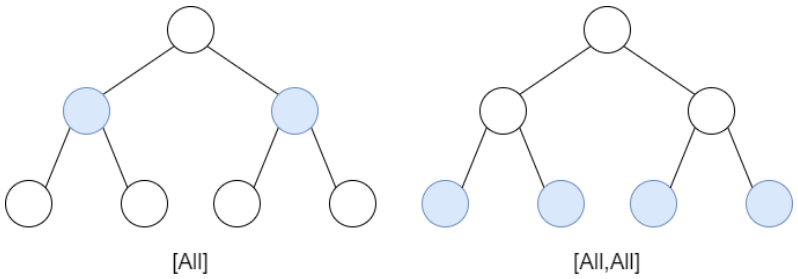


Figure 6.3: The difference between trees selected with the slice [All] and the slice [All, All]

Introducing Tree Transformations

A simpler solution, that allows the direct transformation of *groups of nodes*, is to make our prefix tree's leaves functions of type `OrientedTree -> OrientedTree`. Such a function will from now on be referred to as a *tree transformation*, or *TT* for short. Due to how `OrientedTree` is defined algebraically and recursively, tree transformations can work on both single notes, i.e. values wrapped in `Val`, and groups of notes, wrapped in `Group`. In addition to allowing the modification of both single notes and groups of notes, tree transformations also have a generative advantage. Since we are able to modify any location in a tree, with a function that outputs a tree, we can in principle use our new prefix tree to *build trees*. The functions at the leaves of the tree can change the structure of the tree itself! Thus it is clear that prefix trees with tree transformations as leaves allow for a very general way of generating music.

Applying Tree Transformations

With our new prefix tree, oriented trees are transformed by applying tree transformations to the subtrees that are selected by the given slice. This requires a function that is fundamentally different to a function that applies event modifications to each leaf in a slice. For such a function, a slice will always have a length that is equal to the depth of tree, since leaves are the only values which can be modified. In our case however, slices can refer to any location within a tree. Thus we want the following functionality: If the slice consists of more than one choice, the function should recursively apply itself to the trees in the first choice, with the rest of the slice as an argument. If the input slice consists of just a single choice, we have reached the end of the slice, and the tree transformation should be applied to each subtree in the choice. This form of slicing means that we can for instance differentiate between the selection of All trees at a certain level from All trees at another level. The difference between [All] and [All, All] is displayed in figure 6.3.

6.3.3 Conversion from Prefix Tree to Oriented Tree

Prefix Trees essentially express a structured set of functions that can be applied to an oriented tree. When specifying a piece of music entirely from a prefix tree, we need some sort of default blank tree to apply this prefix tree to. The generation of such a starting tree is handled well by Yan Han, and is therefore only slightly changed in this design to accommodate our new prefix tree. The rest of the conversion process, however, is a bit different from Han's design, due to our new, generative prefix tree.

Combining Slice Transformations

Like in Han's design, the conversion of a branch of slice transformations to a single slice transformation is done by composing all slice transformations along the branch, from top to bottom. This slice modifier is then applied to a default slice, of type [All, All, ..]. The difference is that in our design, this slice has *length equal to the depth of the slice modifier*, i.e. the deepest level that the slice modifier addresses. This is because we need to be able to access arbitrarily deep levels of the tree. As shown in figure 6.3, a slice will refer to nodes in a tree at a depth equal to its length. Thus, if we applied the slice modifier to a default slice that was of a fixed length, like the height of the tree, we could only access its leaves, i.e `Val`'s. Additionally, the need for a slice transformation to be able to access arbitrarily deep levels is furthered by the fact that the height of the oriented tree might change from tree transformation to tree transformation within a prefix tree.

Transformative Instructions

The process of converting a prefix tree to an oriented tree makes use of an intermediate data structure that we call a *transformative instruction*, similar to tree modifiers in Han's design. Such an intermediate data structure is useful for keeping track of which tree transformations should be applied to which slice. How a transformative instruction is created from a given branch of a prefix tree is shown in figure 6.4.

List of Transformative Instructions The overall process of converting a prefix tree to a list of transformative instructions is the same as in Han's design. While the implementation is different, the main process still consists of going over each branch from left to right to get a final list of tree transformations.

6.4 Generative Principle

Now that we have a music representation system that works well with generation, it is time to formalise a general procedure for generating music with this representation system. This section will present the design for such a procedure. First

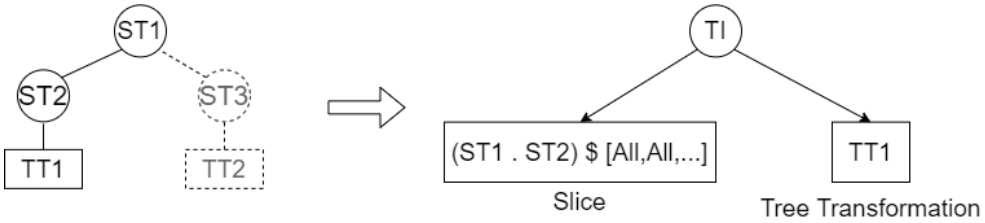


Figure 6.4: Tranformative instruction created from left branch of prefix tree. We see that the slice is constructed by composing the slice modifiers along the branch, and applying it to a default slice of length equal the depth of the total slice modifier.

we will give a short overview of our generative scheme, then go into detail on how this scheme will be realized, by first covering how we generate prefix trees, and then how we sequence them to generate a full piece.

6.4.1 Overview

The underlying idea of our generative principle is to parameterise the generation, so that the generation process starts with abstract parameters and works its way down to more concrete musical elements as it unfolds. This is pictured in figure 6.5. The generation process we envision essentially consists of generating prefix trees that transform a given musical dimension in multiple ways in multiple locations, and then applying these prefix trees to an *initial tree* containing "musical raw material". This raw material can be things such as chord progressions indicating the overall harmonic development of the piece, or motifs that indicate the motific core of a given melody. The initial tree can be obtained by conversion from a generated prefix tree. This is because we have modified our prefix tree to be able to do tree transformations, which includes growing the tree by *insertions* of other trees, such as chords or motifs, or even larger structures of so desired.

Shared Material The reason we want to *generate* prefix trees is that this would allow us to generate music with completely novel structure, where shared patterns across different dimensions is baked in by default in the data structure.

Cohesion Within Each Dimension The idea is that each prefix tree will specialize in transforming a given musical aspect, such as rhythm, harmony etc. This makes for a very clear generation process, where we can ensure that there is cohesion *within* each dimension.

Cohesion Between Each Dimension In order to make sure that there is cohesion *between each prefix tree*, and thus between each musical dimension, we want to employ a *generative plan* to our generation process. This is a meta-model structure that instructs and constrains the generation of each prefix tree. In order to

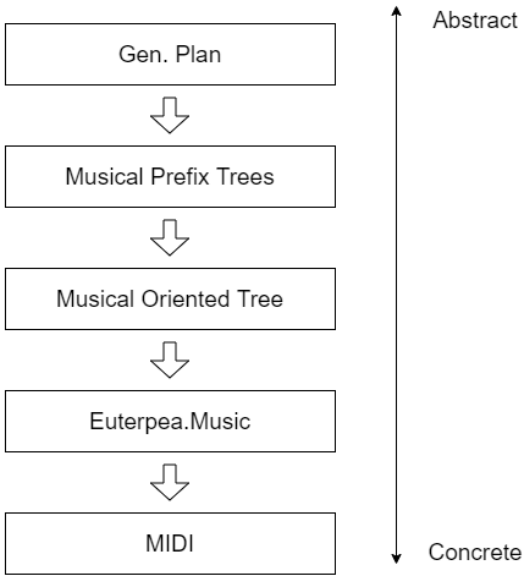


Figure 6.5: The general scheme we envision for generating music with prefix trees

make sure that the output music is cohesive across all dimensions we need to make sure that the plans fit well together. These plans can be written by hand, or one could envision a generation function that generates a set of cohesive generative plans. For now, due to time constraints, we specify them by hand, in our final generative example.

6.4.2 Generating Prefix Trees

The generative principle conceived in this design is based on the generation of prefix trees. This chapter will cover the different aspects of the function we designed.

Inputs to the Generative Function

We clearly need to enforce some constraints on the function that generates prefix trees. After all, there is an endless amount of possible prefix trees that can be generated, but only a certain selection of these will result in cohesive music. Additionally, since generative prefix trees refer to concrete locations on the oriented tree it is meant to transform, the set of possible "legal" prefix trees is further restricted. Thus it is clear that some of the constraints on the generation process will be determined by the oriented tree the prefix tree is to be applied to, and some will determined by stylistic preferences. As stated in the specification, we wish not to embed these constraints into the generative function itself, but rather to pass them as input arguments. These considerations lead to our generative function taking three inputs:

1. The seed for the randomness aspect of the generation
2. A *generative plan*, containing aesthetic choices
3. The music oriented tree that is to be transformed

Generative Plan

The generative plan, (introduced in section 6.4.1), is a data structure that contains all the parameters that dictate how the prefix tree should turn out that are not directly related to oriented tree that is to be transformed. In the future, this structure can be expanded to gain further explicit control of how the generated prefix tree turns out. For now, it contains the following three parameters:

1. A list of all tree transformations that the prefix tree can contain
2. A function that gives the depth level in the given oriented tree that the transformations should be transformed at
3. A skeletal tree denoting the shape of the prefix tree to be generated

List of Possible Tree Transformations The first parameter is very useful, because it is a general way of constraining the type of prefix tree the function should generate. As stated in section 6.4.1, we want each prefix tree to only alter one dimension. For instance, if we wanted to create a prefix tree that transforms the rhythm of a piece of music, this would be done by only include functions that transform rhythm. To ensure rhythmic cohesion, we would only include functions that generate rhythms that work well together. Alternatively, having for instance a list of possible rhythms as an explicit field in the plan would make for a less general and more messy generative plan.

Depth Function The reason why we include a certain depth that the slice transformations should refer to, is that we often wish to address a certain hierarchical level with our tree transformations. Some might only make sense at a give hierarchical level, most often levels where we have groups of notes or a groups of groups of notes. This is the case for the tree transformations in our generative example, which will be further described in section 6.5.1. It seems to be a general trait for tree transformations to only work as intended at certain locations in the tree. One might be able to think of a case where it would be necessary for the prefix tree to operate on multiple hierarchical depths, but this would be for very avant-garde music. If deemed necessary, one could just apply more prefix trees, each addressing a given depth.

Prefix Tree Shape A skeletal prefix tree denoting the *shape* of the prefix tree that is to be generated, is included as a parameter to the generative functions. This will allow us to control how many nodes and leaves the prefix tree will contain, and how these are to be connected. The reason this is desired is twofold: First of all, the shape of the prefix tree says something about the *complexity* of the

generated music. If there are many leaves, then we know that there are many different functions applied to the tree. If there are many nodes, then we know that there is a lot of shared structure in the resulting music. Secondly, prefix trees of the same shape can have wildly different musical outcomes. It all depends of the actual slice transformations and trees transformations contained within the trees. Separating out the shape of the tree is thus natural. This also allows us to keep a given fixed shape of the prefix tree for now, while still exploring the generative variety of the system.

The Generation Function

The process of generating random prefix trees is as follows:

1. Get the skeletal prefix tree from the generative plan (skeletal prefix trees are defined so that all its functions are identity functions, which simply return the input it is given as its output)
2. Generate list of random depth-fixed slice transformations based on the depth parameter in the generative plan and the oriented tree that is to be transformed
3. Generate list of Tree Transformations based on generative plan
4. Insert list of random Slice transformations
5. Insert list of random Tree transformations

Generating Slice Transformations

The locations in which the generated prefix tree will modify the input oriented tree is entirely determined by the slice transformations it contains. To ensure that no slices in the prefix tree refer to locations that do not exist in the oriented tree, their generation is constrained by the oriented tree that is to be transformed. The general method for generating a slice transformation is:

1. Select a random depth from a range of allowable depths within the oriented tree. The allowable depth is found in the generative plan.
2. For this depth, get the range of allowable children to access, determined by the smallest amount of children that a node at this depth has.
3. Input the depth and allowable child range to the slice transformation constructor described in section 6.2.

Figure 6.6 shows what the terms *depth* and *amount of children* refer to in a tree. The reason we get the *smallest* amount of children of the nodes at the given depth, is one or more of the nodes might have less children than the others. A slice with indexes larger than the amount of children for a given node would lead to an error. In most musical cases, we want each node at a given depth to have the same amount of children (often numbers like 2, 4 and 8), so this is not very restricting. Overcoming this restriction would require each slice transformation to be generated based on its location in the skeletal prefix tree, and what other slice

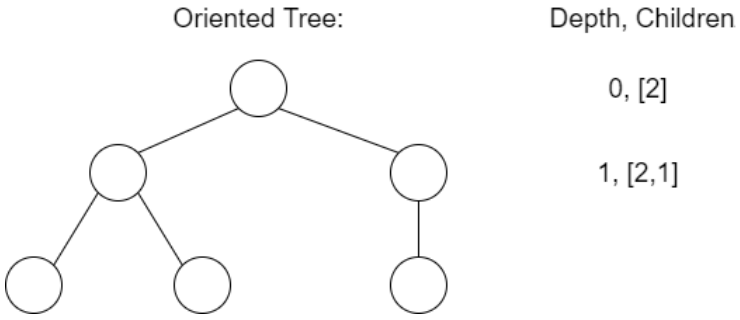


Figure 6.6: The amount of children at each depth in a tree.

transformations are on its branch. This would make random generation of slice transformations a lot more complex, and is therefore avoided for now.

Ensuring that the Prefix Tree only Access a Fixed Depth To make sure every slice in a depth-fixed prefix tree refers to the depth given by the generative plan, the slice transformation in the root node needs to transform the choice at this depth. Since every path will contain the root, every transformative instruction in the prefix tree will then transform the nodes at this depth. The rest of the slice transformations need to refer to a depth that is equal to or smaller than the one in the root node.

Insertion of List of ST's and TT's Into Prefix Tree

In order to insert the elements of a list into a prefix tree, we need a way to specify where each of the lists elements should go. This problem was solved with an enumeration function, that simply enumerates the elements of the prefix tree in pre-order. Nodes and Leaves are enumerated separately (as they contain different types). Figure 6.7 shows how a prefix tree is enumerated. It is important to stress that the enumeration function keeps the information on what value was inside the node before enumeration. Thus a function acting on the enumerated tree can decide to either keep the original value or perform a function on it corresponding to the index of a list of functions. The advantages of such a enumeration process is that it allows the sequential application of a list functions to a tree, where each function is to applied to a different node in the tree. In order to enumerate our nodes, we need a state-like variable that keeps track of how many nodes are under each node. Further details on how this is done will be given in section 7.4.5 in the implementation chapter.

Insertion of Slice transformations Once we have an enumerated prefix tree, we can insert a list of slice transformations into the tree, by for each node inserting the element of the list that has an index equal to the node's number. An advantage of such a list insertion is that we know that first element of the list will be the root

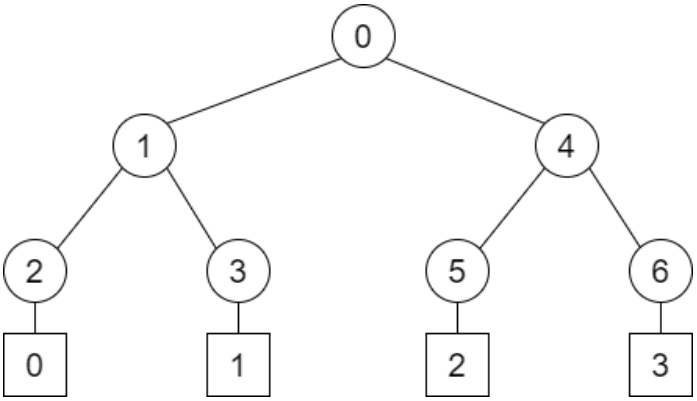


Figure 6.7: An enumerated prefix tree. The Nodes (circles) and leaves (squares) are enumerated seperately, both by pre-order traversal

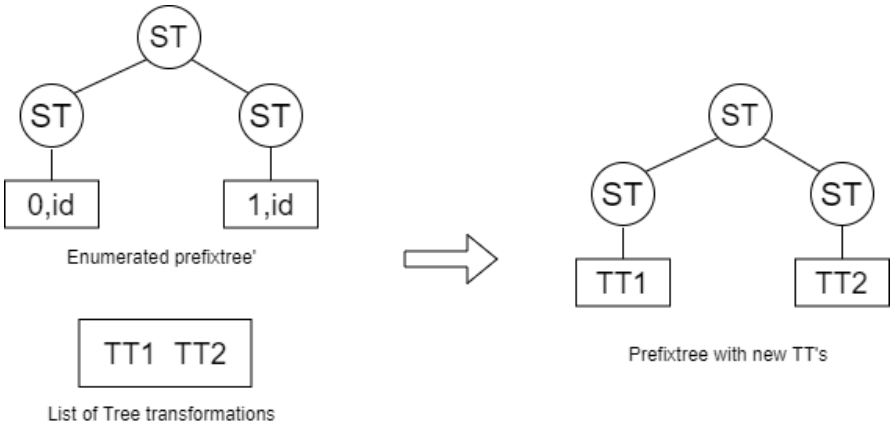


Figure 6.8: How a list is inserted into the leaves of the prefix tree

element of the prefix tree. Thus we can easily specify the depth of the prefix tree by first generating a depth-slice transformation, and then appending a random list of n - 1 slice transformations.

Insertion of Tree Transformations The tree transformations in the prefix tree are applied from left to right, i.e. in the same order as the leaves are enumerated. Thus the list of TT's that are added to the prefix tree are also in the same order as the order they will be executed. This gives the generative function better control of the outcome of the application of its resulting prefix tree, as opposed to just random insertion. The insertion process is illustrated in figure 6.8.

6.4.3 Sequencing Generative Plans

The list of generative plans serves as a minimal model of the final piece. Once the requested amount of generative plans have been generated, they need to be

sequenced together to obtain the oriented tree containing the final piece of music. Since the generative plans serve as meta-models instructing and constraining the generation of each prefix tree, there is really no need to pass state around between each tree, except for the updated seed value. The necessary amount of cohesiveness can be ensured through the generative plans themselves. Additionally, since the generative function for each prefix tree also measures its target oriented tree to ensure that it will fit, there is also no need to pass around size information on the oriented tree, etc. The sequencing basically comes down to a folding function, which will be described in detail in section 7.8.3 in the implementation chapter.

6.5 A Generative Example

In the previous section we have laid out the general principle of generating music with prefix trees. Now it is time for a more concrete example, that clearly demonstrates that this method can be very successful at generating music with novel structures with shared material. As per the specification, the goal is to generate chord-based music, each displaying a different structure of shared patterns in rhythm and playing style. Since there is no time to create our own chord progression generator, rhythm generator, melody generator etc, (all these tasks could be a masters degree in themselves), we will "fake" this for now, by making a small list of acceptable candidates and randomly choosing from it. The next sections will cover the tree transformation functions we have created and the plans generating a full piece of music.

6.5.1 Tree Transformations

To recapitulate, tree transformations are functions of type `MusicOT -> MusicOT`. In other words, they take a musical oriented tree as input and return a transformed musical oriented tree as output. This means that they can operate on values, groups of values, groups of groups of values, etc. This subsection will cover the design of the tree transformations that were implemented as part of this project.

Insertion

The simplest tree transformation is that of the insertion. This is simply the act of ignoring whatever is in the location it works on, and inserting its argument there. This is useful for generating the starting tree, by inserting a selection of chords, motifs, etc.

Rhythm

The rhythm tree transformation takes as input a Rhythm (a list of durations always summing up to 1) and the tree in question. It then copies this tree as many times

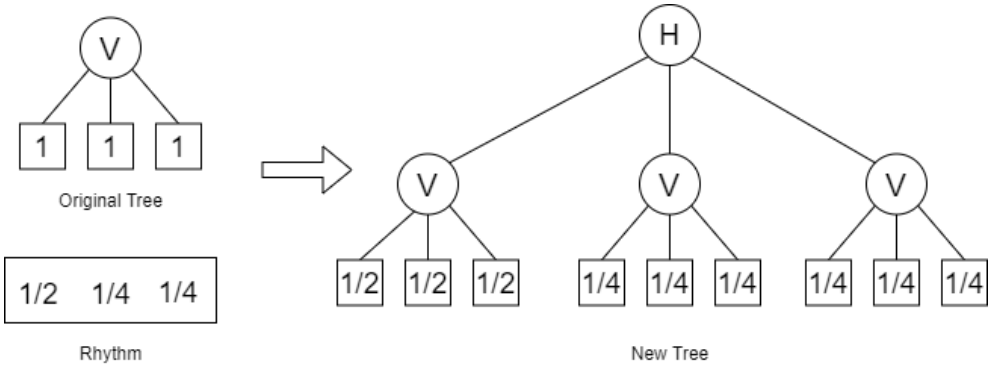


Figure 6.9: How the rhythm function takes a tree and a Rhythm, and creates a new tree with the same total duration but different rhythm. The figure only shows the durations of each Val, the pitch information remains the same.

as there are durations in the Rhythm, and sequentially maps the rhythm durations divided by the original tree's total duration on each tree in the group of trees. The division by the original total duration ensures that total duration of the new tree is the same as in the original tree, since the total duration of a Rhythm is always 1. The rhythm function allows us to insert rhythmic patterns into a piece of music while retaining the pitch information. This process is depicted in figure 6.9.

Pattern

While the rhythm function adds information to a tree, the *pattern* function removes it. It takes as input a list of lists of integers known as a Pattern, as well as the tree to be transformed. A Pattern encodes which notes should remain in a group of groups of notes. It encodes a sort of directional movement within the notes of a given tree. Together with Rhythm, a Pattern denotes how a chord is to be played. Using patterns along with slicing allows us to break chords up into their components, in multiple locations in the tree. By first using rhythm to copy the chord in a rhythmic manner, we can then break it up with patterns, to for instance arpeggiate it. We can for instance give a certain arpeggiation to multiple chords at different locations, helping us achieve a cohesive playing style throughout the piece. How the function works is shown in figure 6.11.

Depth Since the pattern function is meant to operate on a group of group of notes, it will only work as intended at certain locations in the tree. In this design, we assume that the music tree has an even height, and thus we can specify that the function should be applied at a certain depth relative to this height. As figure 6.10 illustrates, this depth is the height of the tree minus 2.

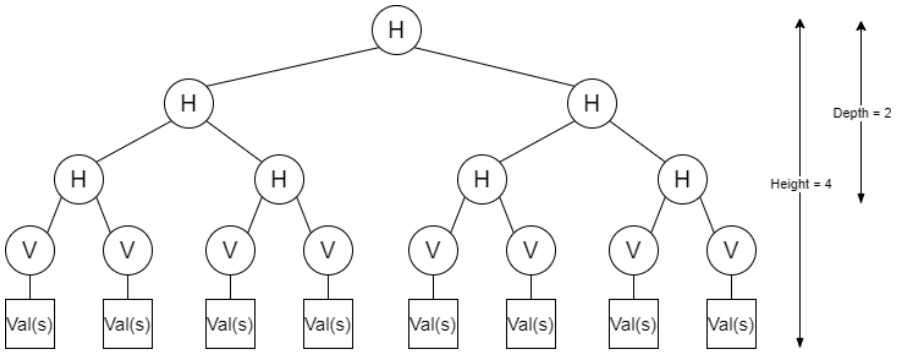


Figure 6.10: The depth at which a pattern should be applied, relative to the height of the tree: depth = height - 2

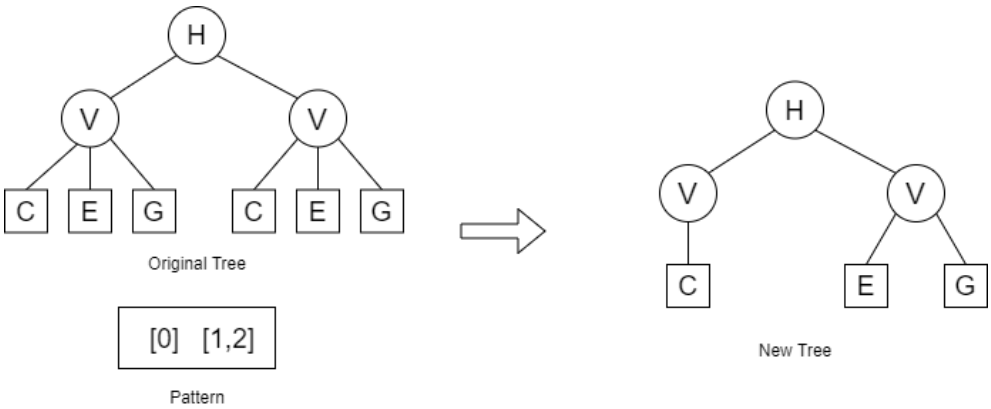


Figure 6.11: How the pattern function takes a tree and a Pattern, and creates a new tree displaying the pattern. The figure only shows the Pitch of each Val, the rest of the information remains the same.

6.5.2 Generating a Piece

In order to generate a piece with the generative scheme and tree transformations presented here, we need to specify certain things by hand: A set of possible chord progressions and the plans for each prefix tree. In the future, the process of selecting these could be automated, resulting in a fully generative system. Still, leaving these to be specified by hand still results in music that is quite structurally varied. As this is the goal of this generative example, we are satisfied with leaving this part of the music hand-written for now.

Two plans

The idea is to create two plans, one for rhythms and one for patterns. The rhythm function mapped to a list of pre-programmed Rhyhtms is used as the pool of tree transformations for the rhythm plan, and the pattern function mapped to a list of pre-programmed Patterns is used for the pattern plan. The reason we use the rhythm plan first, is that it adds information, which is then "pruned" by the pattern function.

What rhythms and patterns the actual plans should contain is up to the user. In the implementation a set of candidates is implemented by the author. These are then used to generate the music that is evaluated in chapter 8.

Chapter 7

Implementation

This chapter will explain how the design was implemented in Haskell. First, an overview of each module and their interactions will be presented, before we go into detail on the most important functions and data structures within each module. The modules are presented in bottom-up order, ending up at `Example.hs` and the function `genPiece`, which generates a random piece of structured music. The function `makestartingtree` and the functions it depends on were taken from Han's implementation. In the few cases where material is identical or similar to Han's implementation, it is specified.

7.1 Overview

Modules An overview of the modules and their dependencies is shown in figure 7.1. Except for Euterpea the system only uses modules from Haskell's base library.

The Composition of `main` A Haskell program is essentially one nested function, i.e. one function which calls other functions, which then call other functions, and so on. Figure 7.2 shows the composition of the most important functions of the system, and the modules they belong to. For clarity's sake, this figure leaves out the Random.hs module and the less important functions. The system is run from `main`, the only impure function of the system, see figure 7.3. It calls the `IO` function `getStdGen` to get a random seed and then calls `Example.genPiece` to generate a piece from this seed:

```
main = do
  gen <- newStdGen
  playDev 6 $ Example.genPiece gen
  -- |    ^ the midi channel you want to stream through
```

To generate a random piece, the user must run GHC's interactive environment with the command `ghci`, open `Main.hs` with the command `:l Main`, and then run `main`.

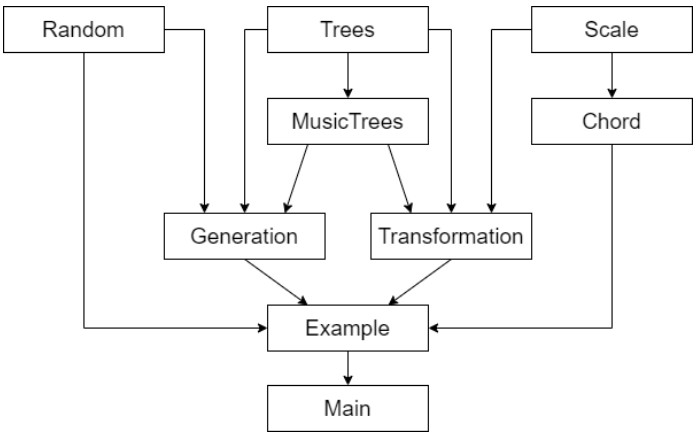


Figure 7.1: Overview of the modules in this implementation and their dependencies. The module an arrow points to imports the module this arrow points from.

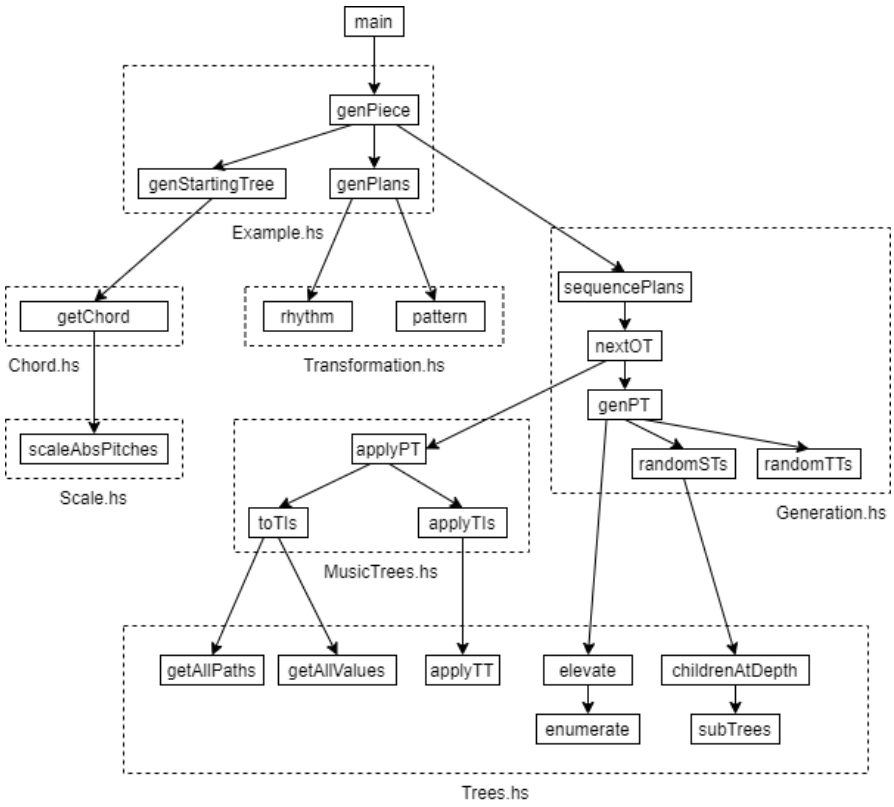


Figure 7.2: A figure showing the tree of the most important function calls from main, and what modules they belong to. An arrow pointing from a function f to function g indicates that f calls g.

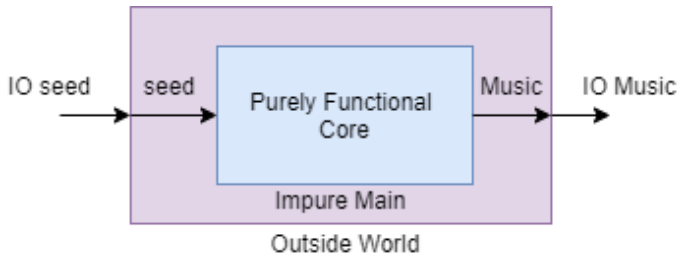


Figure 7.3: How our system is divided into an impure shell and a purely functional core.

7.2 Scale.hs

The module `Scale.hs` is used for generating scales. A scale is represented as the datatype `Scale`:

```
type Root = PitchClass
data Scale = Scale Root Mode
```

To get the list of absolute pitches that a given `Scale` consists of in a given octave, we use `scaleAbsPitches`:

```
scaleAbsPitches :: Scale -> Octave -> [AbsPitch]
scaleAbsPitches (Scale root mode) octave =
  let rootAbs = absPitch (root, octave)
      ff acc x = acc ++ [last acc + x]
  in  foldl ff [rootAbs] (intervals mode)
```

This function uses `foldl` to build a scale by adding each interval of the mode to the last `AbsPitch` in the accumulated scale, starting with the root. We use the function `intervals` to get the intervals of a given mode:

```
intervals :: Mode -> [Int]
intervals mode =
  let majorIntervals = [2, 2, 1, 2, 2, 2, 1]
      mIdx = modeIdx mode
  in drop mIdx majorIntervals ++ take mIdx majorIntervals

modeIdx :: Mode -> Int
modeIdx mode =
  let modes = [Major, Dorian, Phrygian, Lydian, Mixolydian, Minor, Locrian]
  in fromJust $ elemIndex mode modes
```

The function uses `modeIdx` to get the index of a given mode in the order specified by the list `modes`. Now, getting the intervals of a given mode is as simple as taking each interval after `mIdx` and appending the intervals before `mIdx`.

7.3 Chord.hs

The module `Chord.hs` is another small module, used for generating chords based on a `Pitch` and a `Mode`. A chord is represented in the datatype `Chord`:

```
data Chord = Chord { root :: PitchClass
                   , pitches :: [Pitch]
                   } deriving (Show)
```

To get a given chord, we use `getChord`:

```
getChord :: Pitch -> Mode -> [Int] -> Chord
getChord r@(rootPC, octave) mode intervals =
  let scale = map pitch $ scaleAbsPitches (Scale rootPC mode) octave
  in Chord {  root = rootPC
           ,  pitches = r : (map (scale !!) intervals)
           }
```

This function takes the root of the chord, the mode, and the intervals from the root (in terms of scale degrees), the rest of the chord notes should have. To get a triad, we use `getTriad`:

```
getTriad :: Pitch -> Mode -> Chord
getTriad pitch mode = getChord pitch mode [2,4]
```

7.4 Trees.hs

The `Trees.hs` module contains all the tree data structures that are used by the system, and the basic functions that operate on them. In this module, the trees are implemented polymorphically: they are defined independently of the type of the values they contain. Both `OrientedTree` and `PrefixTree` are implemented identically to Yan Han's implementation, shown in section 4.3.

7.4.1 Constructing Slice Transformations

A slice is the primary data structure used to access the nodes of the tree structures in this system. Just like in Han's design, a slice is a value containing multiple paths to nodes at a certain depth corresponding to the index of the last choice in the slice. Their implementation is identical Han's version, seen in section 4.3. To construct a slice transformation that transforms the slice at a given depth, we use `atDepth`:

```
atDepth :: Int -> [Int] -> (Slice -> Slice)
atDepth depth selection slice =
  let (a, b) = splitAt depth slice
  in a [Some selection] ++ tail b
```

This function simply splits the slice at the index given by `depth`, and inserts a choice selecting the given `selection`, overwriting whatever choice was previously there.

7.4.2 Accessing Subtrees of an Oriented Tree

To get the subtrees in a given `Choice`, we use the auxiliary function `subTrees'`:

```
subTrees' :: Choice -> OrientedTree a -> Maybe [OrientedTree a]
subTrees' _ (Val x) = Nothing
subTrees' All (Group _ ts) = Just ts
subTrees' (Some idxs) (Group _ ts) =
  if maximum idxs > length ts then Nothing else Just $ map (ts !!) idxs
```

This function returns a list of type `Maybe [Oriented Tree a]`, since it can fail for certain inputs. The function will return `Nothing` if one tries to get the subtrees of a `Val` (there are none by definition), and if one tries to get a subtree by an index that is larger than the amount of children in the given Group.

Access Subtrees by Slice To access the subtrees of an oriented tree given by a `Slice`, we use the function `subTrees`:

```
subTrees :: Slice -> OrientedTree a -> Maybe [OrientedTree a]
subTrees _ (Val x) = Nothing
subTrees [] tree = Just [tree]
subTrees (c : cs) tree =
  let flatten = fmap concat . join
  in flatten $ (sequence . map (subTrees cs)) <$> (subTrees' c tree)
```

This is a recursive function. The terminal condition is when the slice is empty, in which case we will simply return `Just` the tree inside a list. When we have a slice consisting of at least one choice, we recursively apply `subTrees`, by mapping it to all the subtrees in the first choice, given by `subTrees'`. Since both `subTrees'` and `subTrees` take a tree and return a list of trees in the `Maybe` context, we need to make use of `Maybe`'s instance of `Monad` and `Functor` to combine these functions. The meat and bones of `subTrees` is in it's last line, which is quite dense. We will go through this line step by step, and examine how the type changes:

1. **`(subTrees' c tree)`:** We apply `subTrees'` to the first choice in the slice, to get a value of type `Maybe [OrientedTree a]`

2. **`(sequence . map (subTrees cs))`:** This function is applied inside the `Maybe` with `<$>`. It first maps `subTrees cs` (`cs` is the rest of the choices), onto each tree in the list. This will result in a value of type `[Maybe [OrientedTree a]]`. We then apply the monad function `sequence`, which will give a value of type `Maybe [[OrientedTree a]]`).

3. **`flatten`:** Function 2 applied to function 1 with `<$>`, will result in a value of type `Maybe (Maybe [[OrientedTree a]])`. We then apply `flatten` to this value, which will first use the monad function `join` to flatten the two `Maybe`'s into one, and then use `fmap concat` to concatinate the double list inside the maybe, resulting in a final value of type `Maybe [OrientedTree a]`.

`Maybe`'s instance of monad is implemented so that `sequence` returns `Nothing` if any of the elements in the list is `Nothing`. If There are only `Just`'s in the list, we get `Just` the list. The other monad function we use, `join`, works in the same way. Thus, if any of the `subTrees` calls return a `Nothing`, the entire function will return `Nothing`, which is what we want.

7.4.3 Applying Tree Transformations

A fundamental part of this system is the ability to apply a tree transformation to an `OrientedTree` in the multiple locations contained in a `Slice`. This is done with the function `applyTT`:

```
applyTT :: Slice -> TT a -> OrientedTree a -> Maybe (OrientedTree a)
applyTT _ tt (Val x) = Nothing
applyTT slice tt tree@(Group o ts) =
  if length slice > height tree
    then Nothing
    else Just $ applyTT' slice tt tree
```

Since tree transformations can only be applied to Groups, we have the function return `Nothing` if applied to a `Val`. If the length of the slice is not greater than the height of the tree, we use the auxiliary function `applyTT'`:

```
applyTT' :: Slice -> TT a -> OrientedTree a -> OrientedTree a
applyTT' [c] tt (Group o ts) = Group o $ (applyTo c) tt ts
applyTT' (c:cs) tt (Group o ts) = Group o $ (applyTo c) (applyTT' cs tt) ts
```

Since `applyTT'` is only called from `applyTT` when pattern matched on `Group`, we don't need to include a pattern for `Val` for this function. We see that when the function is called with a slice consisting of a single choice, we apply the tree transformation to the subtrees in the choice. When the function is called with a slice consisting of more than one choice, we recursively apply `applyTT'` to each subtree in the choice. We use `applyTo` to get a function that applies the TT to the appropriate subtrees, depending on the choice:

```
applyTo :: Choice -> ( (a -> a) -> [a] -> [a] )
applyTo c = case c of
              All -> map
              Some idxs -> zipSome idxs
```

If the choice is `All`, we simply map the tree transformation to each subtree. If the choice is `Some idxs` we use zipSome to only apply the TT to the subtrees in the choice, leaving the rest intact:

```
zipSome idxs f ts =
   zipWith (\t idx -> if idx `elem` idxs then f t else t) ts [0..]
```

The function `zipSome` is also present in Han's implmementation under a different name.

7.4.4 Shape Functions

Our design requires us to get different measurements from oriented trees. These are the height of a given tree and the amount of children of a given node. The function `height` is implemented recursively:

```
height :: OrientedTree a -> Int
height (Val a) = 0
height (Group o trees) = 1 + maximum (map height trees)

children :: OrientedTree a -> Int
children (Val a) = 0
children (Group o trees) = length (trees)
```

Children at Depth We also need to get a list of the amount of children of each node at a given depth:

```
childrenAtDepth :: OrientedTree a -> Int -> Maybe [Int]
childrenAtDepth tree depth =
    map children <$> subTrees (replicate depth All) tree
```

This function uses `subTrees` to get a list of subtrees at the given depth of the tree. The function `children` is then applied to each of these to get a list of the amount of children each node has at this depth. We need to use functor aplication with `<$>` to apply `map children` inside the result of subTrees, as it is of type `Maybe`.

7.4.5 Elevation and Enumeration

An important part of this design is the ability to insert the elements of a list onto the nodes of different tree structures. An example of what we mean by this shown in figure 6.8 in the design chapter. This functionality is implemented with the `elevate` function:

```
elevate :: (Functor f, Enumerable f) => [a] -> f a -> f a
elevate list = fmap ff . enumerate where
  ff (idx, x) = if idx < length list then list !! idx else x
```

As we can see, it is implemented as a specialization of a general way to map a function to the elements in a tree, where the function is depending on a given index. To give a node in the tree a corresponding index, we enumerate them in the order of a pre order traversal. The neccessity of both mapping and enumeration is

made clear in `elevate`'s type signature, where `a` must be both a `Functor`, and an *Enumerable*, a custom type class we have implemented for enumeration. The next sections will show how we make our trees an instance of both these type classes.

Our Trees as Functors

To be able to use `fmap` on our trees, we need to make it an instance of Functor.

OrientedTree The functor instance of OrientedTree is defined like this:

```
instance Functor (OrientedTree) where
  fmap f (Val a) = Val (f a)
  fmap f (Group o ts) = Group o (map (fmap f) ts)
```

Since functions only can be applied to the values inside the `OrientedTree`'s leaves, when `fmap` is applied to a group it simply recursively applies `fmap` to each member of the group.

PrefixTree Since `PrefixTree` is a type constructor that takes two types, we need to partially apply the first type to make it an instance of Functor. When we want to map on the *keys* of a prefix tree, we define its instance of Functor like this:

```
instance Functor (PrefixTree v) where
  fmap f (Leaf k v) = Leaf (f k) v
  fmap f (Node k ts) = Node (f k) (map (fmap f) ts)
```

When we want to use fmap on the *values* of a prefix tree, we need to define a `newtype`, with the arguments reversed:

```
newtype PrefixTree' k v = PT' {pt :: PrefixTree v k}
```

Now we can make this reversed prefix tree an instance of Functor:

```
instance Functor (PrefixTree' k) where
  fmap f (PT' (Leaf k v)) = PT' $ Leaf k (f v)
  fmap f (PT' (Node k ts)) = PT' $ Node k (map (pt . fmap f . PT') ts)
```

Our Trees as Enumerables

We have defined our own typeclass `Enumerable`, for things that can be enumerated. It is defined like this:

```
class Enumerable e where
  enumerate' :: Int -> e a -> (Int, e (Int, a))
```

The `enumerate'` function is an auxiliary function, called by `enumerate`, which is defined like this:

```
enumerate :: Functor f => Enumerable f => f a -> f (Int, a)
enumerate = snd . enumerate' 0
```

As we can see, `enumerate'` takes an integer, which is the value that the first node of the enumerable should take on, along with the enumerable itself, and returns a tuple containing the number we should add to number of the current node to get the number of the next node, and the enumerated enumerable. "The number we should add to number of the current node to get the number of the next node" is the amount of enumerated nodes in the previous subtree, hereby denoted as "size". This function is used recursively to enumerate a tree.

OrientedTree as enumerable Let's take a look at how `enumerate'` is defined for `OrientedTree`:

```
instance Enumerable (OrientedTree) where
  enumerate' n (Val x) = (1, Val (n, x))
  enumerate' n (Group o ts) = (sum sizes, Group o enodes)
    where (sizes, enodes) = enumerateSubTrees n ts
```

As we can see, a `Val` is enumerated by inserting n into Val and tagging along a size of 1. A group of trees is enumerated by returning its size (the sum of the sizes of each child), along with the new enumerated group. As only Vals are enumerated, the size of a group is synonymous with the amount of Vals in it.

To enumerate each subtree in the group, we use the function `enumerateSubTrees`:

```
enumerateSubTrees :: Enumerable e => Int -> [e x] -> ([Int], [e (Int,x)])
enumerateSubTrees startSize (x:xs) =
  let totalsize = (sum . map fst)
      ff prev x = prev ++ [enumerate' (startSize + totalsize prev) x]
  in unzip $ foldl ff [enumerate' startSize x] xs
```

We see that is a folding function that takes a list of non-enumerated sub-trees and returns a list of enumerated sub-trees. It takes the accumulator (list of enumerated nodes thus far) and appends the next enumerated node. It always updates the current number by adding the total size of the previous subtrees to the starting size.

PrefixTree as enumerable The instance of enumerable for a normal prefix tree (i.e. when we want to enumerate the keys of the tree) is defined like this:

```
instance Enumerable (PrefixTree v) where
  enumerate' n (Leaf k v) = (1, Leaf (n, k) v)
  enumerate' n (Node k ts) = ((sum sizes) + 1, Node (n, k) enodes)
    where (sizes, enodes) = enumerateSubTrees (n + 1) ts
```

We see that it is very similar to OrientedTree's instance, except for the fact that since the keys are enumerated, we need to add 1 to the size of a Node, and to the starting number when enumerating the sub trees.

PrefixTree' as enumerable The instance of enumerable for a reversed prefix tree (i.e. when we want to enumerate the values of the tree) is defined like this:

```
instance Enumerable (PrefixTree' k) where
  enumerate' n (PT' (Leaf k v)) = (1, PT' $ Leaf k (n,v))
  enumerate' n (PT' (Node k ts)) = (sum sizes, PT' $ Node k (map pt enodes))
    where (sizes, enodes) = enumerateSubTrees n (map PT' ts)
```

Since we here are only interested in enumerating the values of the prefix tree, i.e. the leaves, this implementation is essentially the same as that of OrientedTree. Since we are using the newtype wrapper `PT'` to reverse the arguments, we need to add it to pattern match on and return values of type `PrefixTree'`.

7.5 MusicTrees.hs

The `MusicTrees.hs` module contains the *Musical Trees*, i.e. the polymorphic trees implemented in `Structure.hs`, with music types. It also contains functions to convert between these structures.

7.5.1 Musical Oriented Trees

Musical oriented trees are expressed as the type `MusicOT`:

```
type MusicOT = OrientedTree (Primitive Pitch)
```

Conversion to Euterpea's Music type

A value of type `MusicOT` can be converted into Euterpea's `Music` type (so we can play it as MIDI with `play`), with the function `toMusic`:

```
toMusic :: MusicOT -> Music (Pitch, Volume)
toMusic (Val x) = valToMusic (Val x)
toMusic (Group H trees) = line $ map toMusic trees
toMusic (Group V trees) = chord $ map toMusic trees
```

This function is implemented almost identically in Han's implementation. The functions `line` and `chord` are from `Euterpea`, and combine a list of musical values into a melodic line (i.e. sequentially) and a chord (i.e. parallelly) respectively. Converting a `Val` to music is done with the function `valToMusic`:

```
valToMusic :: MusicOT -> Music (Pitch, Volume)
valToMusic (Val (Note dur p)) = Prim ((Note dur (p, 75)))
valToMusic (Val (Rest dur)) = Prim (Rest dur)
```

This is to be able to reduce the volume of our output music (100 is a little harsh).

7.5.2 Musical Prefix Trees

A musical prefix tree is implemented by the type `MusicPT`:

```
type MusicPT = PrefixTree (MusicOT -> MusicOT) (Slice -> Slice)
```

Its leaves are the left paramater, which tree transformations, and its nodes are the right parameter, which are slice transformations.

Conversion to MusicOT

The conversion process from `MusicPT` to `MusicOT` makes use of an intermediate data type `TI`, short for *Transformative Instruction*:

```
data TI = TI { slc :: Slice, tt :: (MusicOT -> MusicOT)}
```

A musical prefix tree can be considered a compressed version of a list of transformative instructions. Converting a `MusicPT` to a list of TI's is done with `toTIs`:

```
toTIs :: MusicPT -> [TI]
toTIs pt =
  let stss = getAllPaths pt
      tts = getAllValues pt
  in zipWith toTI stss tts
```

We see that the function first uses `getAllPaths` from the `Trees.hs` module to get a list of all paths in the prefix tree, where each path is a list of keys from root to leaf. This results in the value `stss` of type `[[Slice -> Slice]]`. Similarly, to get `tts`, a list of all tree transformations, we use `getAllValues`. To get a list of all TI's, we use zipWith, to zip the two lists into one list with the binary function `toTI`:

```
toTI :: [Slice -> Slice] -> (MusicOT -> MusicOT) -> TI
toTI sts ttrans =
  let slice = foldl (\slc st -> st slc) (smallestDefault sts) sts
  in TI { slc = slice, tt = ttrans}
```

This function uses `foldl` to apply all slice transformations from left to right to a `default slice`. This default slice is the smallest possible slice of only `All`, of depth equal to the deepest slice transformation in `sts`. This default slice is found by `smallestDefault`:

```
smallestDefault :: [Slice -> Slice] -> Slice
smallestDefault sts = replicate ((getDeepest sts) + 1) All

getDeepest :: [Slice -> Slice] -> Int
getDeepest sts = maximum $ map getDepth sts

getDepth :: (Slice -> Slice) -> Int
getDepth st = fromJust . findIndex (isSome) . st $ repeat All
```

Here we see that we find the smallest default slice by finding the deepest slice transformation in `sts`. This is done with `getDepth`. Since a slice transformation is a function, we need to apply it to some default slice to find its depth. This slice is an infinite list of `All`'s. Slice transformations are defined to only transform one depth, which means that its depth will equal the index of the first `Some` in the resulting slice from its application to a default slice. Since all slice transformations are generated to modify one depth, `findIndex` should never fail, thus `fromJust` is applied to extract the index from its `Maybe` context.

Applying TI's Once we have converted a `MusicPT` to a list of type `[TI]`, we use `applyTIs` to apply these to a `MusicOT`:

```
applyTIs :: [TI] -> MusicOT -> MusicOT
applyTIs tis tree = foldl applyTI tree tis

applyTI :: MusicOT -> TI -> MusicOT
applyTI tree (TI slice tt) = fromJust $ applyTT slice tt tree
```

We see that this function uses `foldl` to apply each TI from left to right to the tree. `applyTI` uses `applyTT` from the `Structure.hs` module, which returns a value of type `Maybe (OrientedTree a)`. Thus we need to apply `fromJust` to the result. This assumes that `applyTT` returns a value of type `Just (MusicOT)`.

Applying MusicPT's Using transformative instructions and their functionality, applying a `MusicPT` to a `MusicOT` is really simple:

```
applyPT :: MusicPT -> MusicOT -> MusicOT
applyPT pt = applyTIs (toTIs pt)
```

To directly convert a musical prefix tree to a musical oriented tree, we use the function `toMT`:

```
toMT :: MusicPT -> MusicOT
toMT pt = applyPT pt (makeStartingTree (toTIs pt))
```

This function uses Han's function `makeStartingTree` to generate an empty starting tree based on the prefix tree. The prefix tree is then applied to the starting tree.

7.6 Transform.hs

The module `Transform.hs` contains *tree transformations*, which are functions of type `MusicOT -> MusicOT`. The two fundamental tree transformations used in our system is `rhythm` and `pattern`.

7.6.1 Rhythm

Rhythms are expressed as a list of Durations:

```
type Rhythm = [Dur]
```

Rhythm transformations are performed with the function `rhythm`:

```
rhythm :: Rhythm -> MusicOT -> MusicOT
rhythm rm tree =
  let td = totDur tree
      ts = (replicate (length rm) tree)
  in Group H $ zipWith (\d t -> fmap (giveDuration (d/td)) t) rm ts
```

This function simply replicates the input tree once for each duration in the input rhythm, and then returns a horizontal group of these trees, with the durations of the rhythm inserted with `zipWith`. Each duration in the rhythm is scaled by the total duration of the original tree, as to return a tree with identical total duration as the input tree. To get the total duration of a given tree, we use `totDur`:

```
totDur :: MusicOT -> Dur
totDur (Group H trees) = sum $ map totDur trees
totDur (Group V trees) = maximum $ map totDur trees
totDur (Val x) = getDur x
```

The total duration of a horizontal group is the sum of the total duration of all its subtrees, while the total duration of a vertical group is simply its longest duration. To insert a given duration into a node of tree, we use `fmap giveDuration` to apply `giveDuration` inside the tree:

```
giveDuration :: Dur -> Primitive Pitch -> Primitive Pitch
giveDuration dur (Note d p) = Note dur p
```

7.6.2 Pattern

Patterns denote a selection of notes within a group of groups (see design chapter section 6.5.1 for more info). They are expressed by the type `Pattern`:

```
type Pattern = [[Int]]
```

A pattern transformation is performed with the function `pattern`:

```
pattern :: Pattern -> MusicOT -> MusicOT
pattern p (Val x) = Val x
pattern p (Group o groups) =
  Group H $ zipWith extract (concat $ repeat p) groups
```

Patterns are infinite structures, which is achieved with `concat $ repeat`. We zip together the pattern with the groups with the function `extract`:

```
extract :: [Int] -> MusicOT -> MusicOT
extract xs (Val x) = Val x
extract xs (Group o ts) =
  let sel = sort xs
  in if length ts > maximum sel then Group V $ map (ts !!) sel
     else extract (unique $ init sel ++ [last sel - 1]) (Group o ts)
```

We want to be generous in our implementation of extract, i.e. we do not want it to be able to fail. If none of the selections in `xs` are larger than the amount of subtrees inside the `Group o ts`, we can simply get the selected subtrees. If not, we modify our list of selections to accommodate the tree. We recursively call `extract`, with a `sel` that has its maximum selection reduced by 1, until the maximum is smaller than the length of `ts`.

7.7 Random.hs

We use the module `Random.hs` to generate random values, and to thread random seeds through a series of n random functions. To get a random element from a list of candidates, we use the function `getRandom`:

```
getRandom :: (RandomGen g) => [a] -> g -> (a, g)
getRandom list seed =
  let randomIdx = randomR (0,(length list) -1) seed
  in (list !! fst randomIdx, snd randomIdx)
```

This function uses `randomR` from the the `System.Random` module to get a random index from `list` and then returns the element at this index along with the new seed. To get `n` random values from a list, we use `getRandoms`:

```
getRandoms :: Int -> [a] -> State StdGen [a]
getRandoms n = sequence . replicate n . state . getRandom
```

This function uses the `State` monad from `Control.Monad.State`. It first creates a list of n State monads containg the stateful computation `getRandom`. The Monad function `sequence` is then used to convert a list of State monads to a State monad containing a list, i.e. going from a type of `[State StdGen a]` to a type of `State StdGen [a]`. Now, generating an arbitrary amounts of random numbers from a list is simple: For instance if we want to generate 3 random numbers, we call `runState (getRandoms 3) seed`, which will return a tuple of the list of random numbers and the new seed. When we want to get one random number from each list in *a list of lists*, we use `getRandomss`:

```
getRandomss ::  [[a]] -> State StdGen [a]
getRandomss =  sequence . map (state . getRandom)
```

The function name ending with "ss" imply a *list of list*, just like a single "s" implies a list.

7.8 Generation.hs

The `Generation.hs` module contains functions for generating musical prefix trees. It also contains functions for chaining together prefix tree generations and the meta-model data structures used in this process.

7.8.1 Generative Plans

Generative plans are implemented as types with record syntax:

```
data Plan = Plan { _ttPool :: [MusicOT -> MusicOT]
                 , _ttDepth :: MusicOT -> Int
                 , _skeleton :: MusicPT
                 }
```

7.8.2 Generating Prefix Trees

The process of randomly generating a `MusicPT` based on a seed, generative plan, and the `MusicOT` it is to be applied to, is implemented in the function `genPT`:

```
genPT :: StdGen -> Plan -> MusicOT -> (MusicPT, StdGen)
genPT gen plan ot =
  let skel = _skeleton plan
      (sts, gen2) = randomDFSTs gen (keysAmt skel) ot (_ttDepth plan ot)
      (tts, gen3) = randomTTs gen2 (valuesAmt skel) (_ttPool plan)
  in  (elevateValues tts $ elevate sts $ skel, gen3)
```

As we can see, the seed is manually threaded through (and updated by) each random generation function, and finally output at the end. We use the `elevate` function to insert the lists of slice transformations and tree transformations into a default tree that is generated based on the general shape of the prefix tree.

Generating a Random List of Slice Transformations

The function used by `genPT` to generate a random list of depth-fixed slice transformations is called `randomDFSTs`, which is a specialised version of the general function `randomSTs`:

```
randomSTs :: StdGen -> Int -> OrientedTree a -> [Int] -> ([Slice -> Slice], StdGen)
randomSTs gen n ot depthRange =
  let (depths, gen2) = randomDepths gen n depthRange
      (children, gen3) = randomChildren gen depths ot
  in (zipWith st depths children, gen3)
```

This function generates a random list of depths in the range specified by `depthRange`. This list is passed as argument to `randomChildren` to get a random list `children`, where each child corresponds to the depth at the same index in `depths`:

```
randomChildren :: StdGen -> [Int] -> OrientedTree a -> ([Int], StdGen)
randomChildren gen depths ot =
  let childRanges = map (childRange ot) depths
  in runState (R.getRandomss childRanges) gen
```

The function `childRange` is used to determine the list of possible children within each tree and depth:

```
childRange :: OrientedTree a -> Int -> [Int]
childRange tree depth =
  let cad = childrenAtDepth tree depth
  in case cad of
    Just children -> [0 .. (minimum $ children) - 1] -- 0-index
    Nothing -> [0]
```

`childrenAtDepth` returns a type wrapped in `Maybe`, which is handled by returning a default range of `[0]` if `childrenAtDepth` returns `Nothing`. (This is further discussed in the implementation discussion in section 9.2.2). To ensure that each slice transformation only refers to children that actually exist in the OrientedTree, we use `minimum` to get the amount of children of the node at this depth with the lowest amount of children. Once we have all depths and corresponding children, we zip with `st` to generate our slice transformations:

```
st :: Int -> Int -> (Slice -> Slice)
st depth child = atDepth depth [0..child]
```

Random Depth-Fixed Slice Transformations To generate random depth-fixed slice transformations, we use `randomDFSTs`:

```
randomDFSTs
    :: StdGen -> Int -> OrientedTree a -> Int -> ([Slice -> Slice], StdGen)
randomDFSTs gen n ot depth =
  let (dfst, gen2) = randomDFST gen ot depth
      (sts, gen3) = randomSTs gen2 (n - 1) ot [0..depth]
  in (dfst : sts, gen3)
```

This function first generates a random depth-fixed slice transformation, i.e. a slice transformation that is fixed to transform the slice at a given depth:

```
randomDFST :: StdGen -> OrientedTree a -> Int -> (Slice -> Slice, StdGen)
randomDFST gen tree depth =
  let ([child], gen2) = randomChildren gen [depth] tree
  in (atDepth depth [0..child], gen2)
```

When we generate the rest of the slice transformations, we need to make sure that they cannot have depths that are deeper than `dfst`, by passing a depth range of `[0..depth]` to `randomSTs`. Finally we place `dfst` first in our exported result, to make this the root of the generated `MusicPT`.

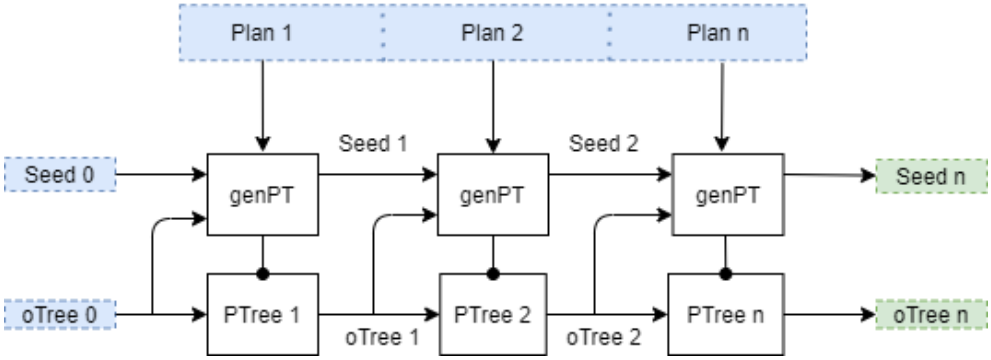


Figure 7.4: A block diagram displaying how the function `sequencePlans` works to generate a piece of music from a list of plans and a starting oriented tree and a seed. Input values are shown in blue and output values in green.

7.8.3 Sequencing Generative Plans

A list of generative plans can be sequentially applied to a `MusicOT` with the function `SequencePlans`:

```
sequencePlans :: StdGen -> [Plan] -> MusicOT -> (MusicOT, StdGen)
sequencePlans gen plans startTree =
  foldl (\(tree, gen) plan -> nextOT gen plan tree) (startTree, gen) plans
```

This function has a tuple of the starting tree and starting seed as its accumulator. We fold over this accumulator with the list of plans, using the function `nextOT` to get the resulting `MusicOT` from the seed, generative plan, and current `MusicOT`:

```
nextOT :: StdGen -> Plan -> MusicOT -> (MusicOT, StdGen)
nextOT gen ptPlan otree =
  let (newPT, gen2) = genPT gen ptPlan otree
  in (applyPT newPT otree, gen2)
```

We see that this function first generates a `MusicPT`, and then returns the pt-transformed `MusicOT` along with the updated seed. A block diagram displaying how `sequencePlans` works is shown in figure 7.4.

7.9 Example.hs

The module `Example.hs` contains the code for how a piece is generated in our generative example. The module only exports one function, `genPiece`, which generates a piece of music:

```
genPiece :: StdGen -> Euterpea.Music (Pitch, Volume)
genPiece gen =
  let (cps,gen2) = genChordProgs 2 gen
      chordsTree = genStartingTree cps
```

```
      (final, gen3) = sequencePlans gen2 ptPlans chordsTree
  in toMusic final

ptPlans = [rhythmPlan, patternPlan]

rhythmPlan = Plan { _ttPool = map rhythm' [ronettes, n, evn 4]
                  , _ttDepth = \tree -> measureDepth tree + 1
                  , _skeleton = defaultPT'
                  }

patternPlan = Plan { _ttPool = map pattern [full, waltz, sadwaltz, falling]
                  ,  _ttDepth = \tree -> measureDepth tree
                  ,  _skeleton = defaultPT''
                  }
```

Here `ronettes`, `n`, `evn 4` are rhythms, and `full`, `waltz`, `sadwaltz`, and `falling` are patterns implemented in `Example.hs`. Some examples of their implementation:

```
full = [[0,1,2]]
falling = [[2], [1], [0]]
n = [(qn + en), (qn + en), qn]
ronettes = [(qn + en), en, hn]
```

Chapter 8

Evaluation

After designing and implementing our music generation system, is it time to evaluate its generated output. The music will be evaluated on the grounds of the specification made in the paragraph *Output Music* in the functional requirements section 5.1. The requirements are quickly summarized here:

1. The generated music should display hierarchical structure, and there should be shared material both within and across hierarchical levels.
2. The structure of the output music should be both cohesive and varied.
3. The focus is mainly on structural novelty, not on quality. Additionally, we will not focus on the quality of the other aspects of the music.

8.1 Structure Visualization

To get a clearer picture of how good the system is at generating music with repeating structure across and within multiple hierarchical levels, we will look at a visualization of the MIDI files of three representative pieces. In these figures, time is along the horizontal axis, and MIDI-notes are on the vertical axis. Wave files of the pieces are included in the zip file, and links to listen to each piece will be provided in the figure texts. The reader is strongly encouraged to listen along to get a clearer view of the structure of each piece.

All three pieces are generated from a starting tree with the same initial structure, where there are 2 possible 2-chord progressions that can be inserted, chosen from a pool of the the following candidates: "CMaj7 FMaj7", "C6 G6", and "CMaj7 C6". These chord progressions where chosen from personal taste, as it is the *structural variation* of rhythms and playing patterns that is interesting to evaluate, not the actual chords themselves.

8.1.1 First piece

The first MIDI file to be inspected is generated with seed 1492526188 532080812. Figure 8.1a shows an overview of the entire piece, which indicates that it has a ABA'B structure, where A' is a modified version of A. Figure 8.1b shows a closer comparison of A and A'. We see that all subsections of A and A', i.e A1, A1' and A2 contain the same chords. What makes them different from each other is the way these chords are played. A2 is identical in both sections, but A1 and A1' are different in the first three notes of each chord. In A1 these notes are played in a descending arpeggio, while in A1' these notes are played with the full chord. Listening to the piece reveals that while the playing pattern varies, all chords in both A1 and A1' are played with the same underlying rhythm.

8.1.2 Second piece

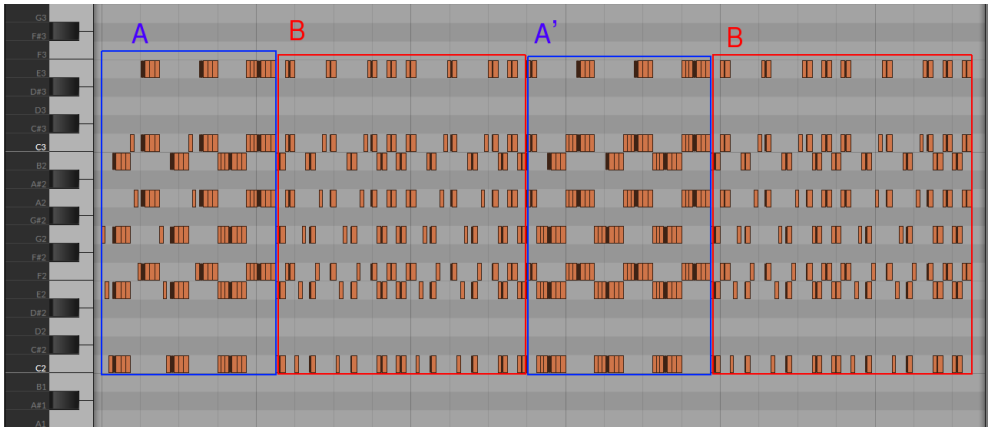
The second piece we will inspect is generated with seed 1622411450 299661201. This is piece with a lot of repetion, on all hierarchical levels. We see from figure 8.2a, That on the highest level, this piece can be divided into two sections, A and A'. It clear that A' is very similar to A, as they both contain three identical subsections, A1 and A1' respectively, and that each of these subsections indicate the same harmonic outline. Figure 8.2b gives a closer comparison of A1 and A1, revealing that they both end with the same section A13. They rest of the hierarchical division is similar, but each subsection is different in the playing patterns. This is made clear by figure 8.2c, which shows that both A11 and A11' share the same "skeleton" indicated in white, but that A11' has a playing patern with much fuller chords. Additionally, the two chords of A11' both end with an identical rhythm and pattern, that is different from the three preceding ones, indicated in yellow. In A11, this is not the case.

8.1.3 Third piece

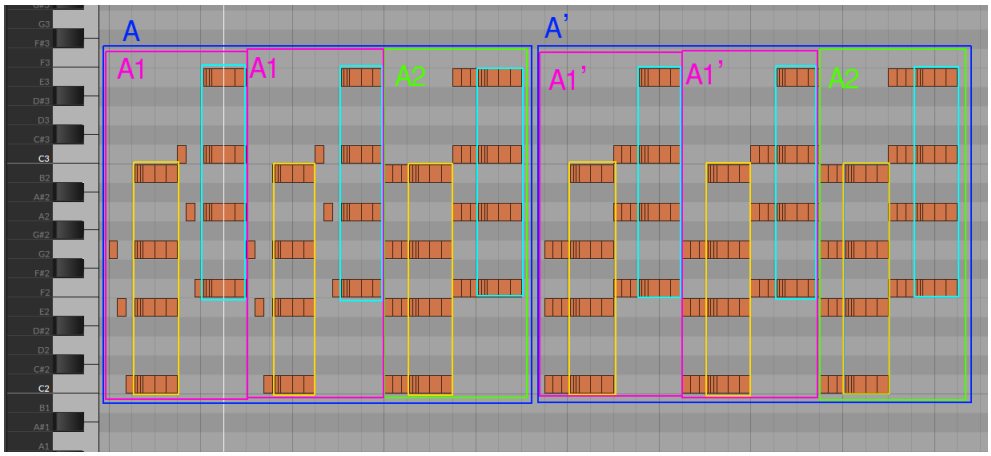
The third piece is generated from seed 1492566202 504813877 and is displayed in figure 8.3. This piece is divided into more unique high-level sections that the previous two, but has far fewer notes. Thus one figure suffices to highlight the most important shared material. We can see that the piece has a ABCA structure. The chords in section A, B1 and C are the same, but played differently. The notes inside the yellow boxes display the same playing pattern. Thus we can see shared material between sections. This pattern is played at half speed in section A, and returns in section C for the first 3 chords.

8.2 Discussion on Musical Output

As the visual inspection in the previous section demonstrated, the generated music has varied structure, with clear repetitions of rhythms and playing styles. Repetition occurs on all hierarchical levels, and patterns are shared across different

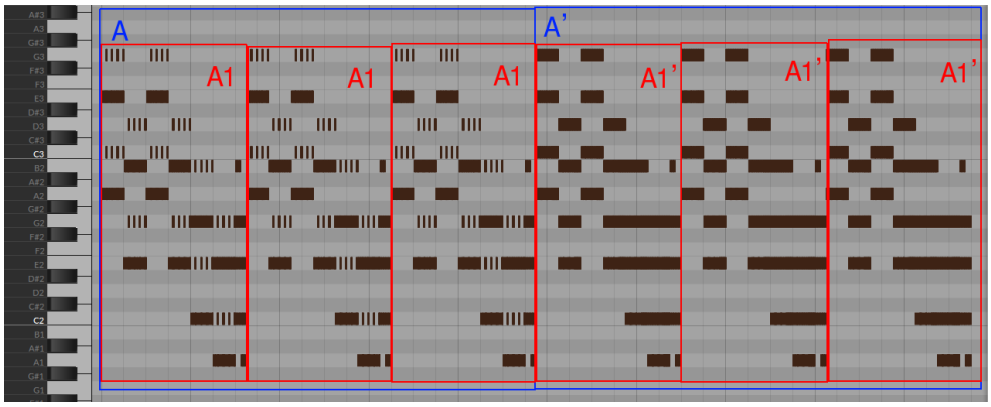


(a) Structural overview: We see that the piece has an ABA'B Structure, where A' is a modified version of A.

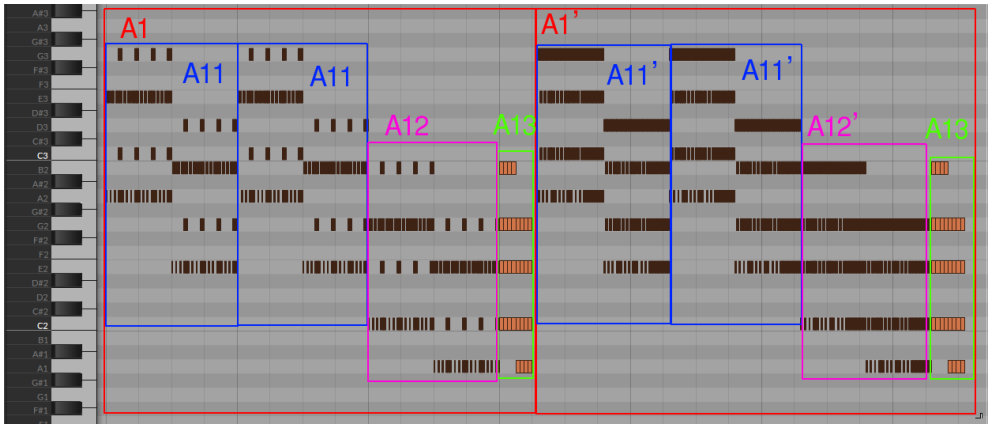


(b) Closer comparison between A and A'. The yellow and light blue rectangles indicate repeated material.

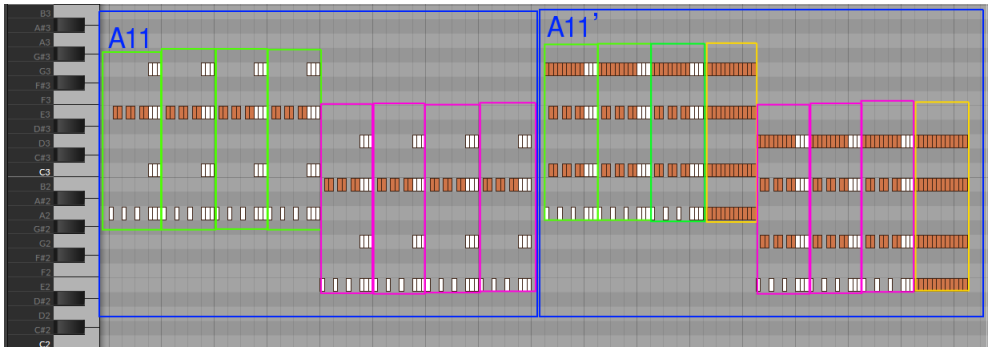
Figure 8.1: The structure of the first generated piece. Figure 8.1a gives an overview of the entire piece, while figure 8.1b gives a closer comparison of section A and section A'. It can be listened to here.



(a) Structural overview: We see that the piece has an AA' structure, where both A and A' contain three identical subsections.



(b) Closer comparison of section A1 and A1'.



(c) Closer comparison of section A11 and A11'. Colored boxes indicate shared material. The notes highlighted in white show an underlying pattern in both sections.

Figure 8.2: The structure of the second generated piece. Figure 8.2a gives an overview of the entire piece, while figure 8.2b gives a closer comparison of section A1 and section A1'. Figure 8.2c takes an even closer look, comparing A11 and A11'. The piece can be listened to here

.

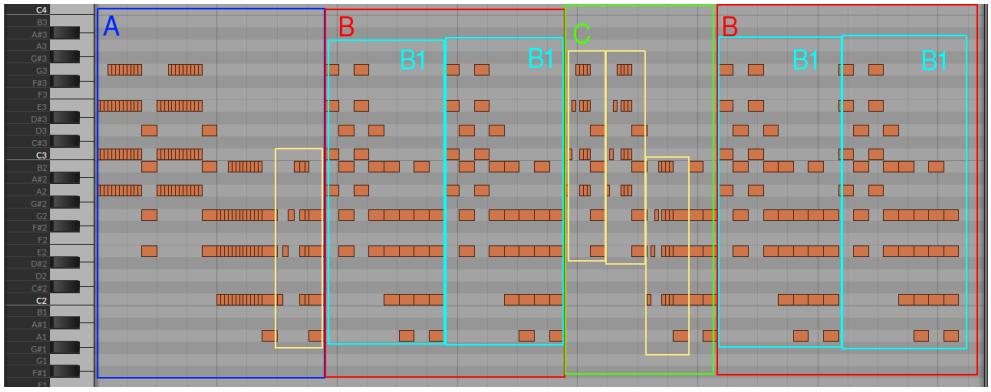


Figure 8.3: The third generated piece. Since it contains far fewer notes that the first and second piece, a single figure suffices to highlight it's structure. The yellow boxes indicated shared material. The piece can be listened to here

.

levels. Most of the generated music sounds fairly "musical", even if its structure is sometimes a bit strange. This is to be expected, since the structures are randomly generated. Still, the structural strangeness is *consistent*. A weird little quirk in a given composition will usually return later in a modified for later in the piece.

Variety and Cohesion We see that a clear structural variety both within and between pieces. Additionally, the degree of repetition varies for each piece: The second generated piece contains a lot of repetition, while the first and third is more varied. Since the same rhythmic patterns and playing styles consistently reoccur throughout a piece, the output music also sounds quite cohesive.

Some Weaknesses

Even though the output music successfully lives up to our goals with regards to structue, it also has clear weaknesses. These weaknesses are mostly in musical aspects that were not within the scope of this project. Thus, the following criticisms of the output music serve as a guide for possible future work:

- There are no rests in the music, every note lasts until the next note in its respective group. (Even though our system was able to represent rests, the generative example did not generate them.)
- Every note has the same volume, which makes the music sound a bit mechanical.
- Many songs feature a little section where the chords are played plainly: one full chord per bar. This is where no transformations have been applied.
- The music essentially only consists of chords. It would be greatly improved if there were melodic lines on top of these chords.
- The structures could be of greater "musical quality".
- The emotional development/narrative of a given piece can be improved.

Chapter 9

Discussion

The evaluation of the generated output revealed that the system designed and implemented in this project was largely successful at meeting the functional specification. Still, in this project, the *design* of the system is just as important as its functionality. The same goes for the implementation. Thus both of these can be considered important results in themselves, and therefore deserve their own extensive discussion in light of their specification requirements. After a discussion on both the design and implementation, this chapter will feature a discussion on using Haskell for this project. Haskell was chosen as the implementation language, with a hope that it would be advantageous to view the problem from the different viewpoint that this language provides. We will discuss whether this turned to be true or not, and comment on the general experience of developing software in Haskell.

9.1 On the Design

In this section, the design of the system is evaluated with regards to its requirements specified in section 5.2. Just like in the design chapter, we will start with covering representation, before moving on to generation.

9.1.1 Representation Structures

The representation system used in this design consisted of a time-based representation in the form of musical primitives structured in an oriented tree, as well as a compressed representation of transformations on these oriented trees, in the form of prefix trees. In this section we will discuss how well our representative system performed, especially focusing on the tweaks we made to Han's representation.

Musical Oriented Trees and Primitives

The structure used for representing a piece of music as it unfolds in time was a musical oriented tree, an oriented tree where the leaves were of type `Primitive`

`Pitch`. Working explicitly with trees gave us full control of the structure of the resulting music. The oriented tree was very expressive, but still manageable to work with. In the design chapter we mentioned that their potential for complexity might make it difficult to understand how certain oriented trees will actually translate to music. For smaller structures, such as little melodies or chord progressions, this was not the case. However, it turned out to be true for larger pieces, especially generated ones. When inspecting a generated piece of music in the form of an oriented tree, it was often quite difficult to see how it would sound. One would need to listen to the MIDI file to get a clear picture. Still, this was a minor problem, as diagnosing a generated piece was just as easy to do by inspecting and listening to the MIDI file using external software.

Greater dimensional decomposition The primitives that made up the leaves of the oriented tree were a data structure that could either be a note or a rest. A note was decomposed into a pitch and an octave. This was expressive enough for the purposes of our generative example, but when/if this system is expanded for more general generation, one should aim to decompose the primivites in even more dimensions. As mentioned in the design chapter, it would likely be useful to define our musical structures *in terms of scale*. Thus we could decompose pitch into a scale, scale degree and octave, and represent chords in terms of their degrees within the scale. Additionally one could also include a volume dimension to be able to generate more natural sounding performances.

Representing Shared Material with Prefix Trees

The structure we chose for representing shared material was the prefix tree. As previously stated, a prefix tree is essentially a compressed version of a map, where the shared elements of the keys are expressed only once. Our musical prefix tree expressed the same thing as a list of transformative instructions, with the additional ability to explicitly express the shared choices within each TI's slice. This allowed us to control the amount of shared material within a given piece of music, which was essential to our design. Thus, one the whole, the use of prefix trees for music generation can be deemed a success.

Challenges Still, due to their abstract nature, prefix trees do have some disadvantages. First of all, they can be difficult to work with. It takes some time to get an intuition for what the result of applying a given prefix tree to an oriented tree will be. Additionally, one could argue that while prefix trees are effective, they might be a somewhat "unnatural" tool for composing music, as they are unknown to most musicians, with no real analogous term in music theory. However, when used for generation, this is not really a problem from the user's perspective as they are only handled internally by the software. Additionally, since part of our motivation for generating music was the possibility of music output which is "creatively unbound" from human thinking, the fact that the generative system's

composition process is not close to what a human would do is actually more of an advantage. Anyway, their unique expressiveness when it comes to representing shared patterns, makes handling their somewhat non-intuitive nature worth the trouble.

9.1.2 Tree Transformations

In this design, we modified Yan Han's prefix tree by changing the leaves from *event modifications*, which were functions that modified single notes within the leaves of an oriented tree, to a type of function we called *tree transformations*. These were functions that modified the oriented tree itself. This was advantageous, as it allowed us to apply the functions in the prefix tree to *groups of notes* and *groups of groups of notes*, which was a prerequisite for the two main transformations we designed, *rhythm* and *pattern*. It also allowed us to build out the structure of a piece, by using functions such as `insert`.

Challenges with Tree Transformations

Still, as expected, this increased expressiveness came with some challenges with designing a well-formed system: Tree transformations can in principle be applied anywhere in an oriented tree, but from a musical context, each tree transformation usually only makes sense to apply in certain locations. For instance, the tree transformation `pattern` was designed to work on *groups of groups of notes*, while `rhythm` makes most sense to apply to a *group of notes*. Thus it is clear that a way to control where tree transformations can be applied is needed. In our design, we solved this problem by generating prefix trees so that each tree transformation would be applied at the same, predefined depth, specified in relation to the height of the oriented tree. As our evaluation showed, this worked well from a functional perspective. Internally however, it would sometimes lead to problems if the oriented tree had a non-uniform height, which would often be the case. This was because `rhythm` increases the height of the subtree it is applied to by 1, which will lead to a non-uniform height if this function is not applied to every subtree at its required depth in the tree. This problem was solved in the implementation stage, where the tree transformations were implemented so that they would not crash when applied to input that they are not designed for. For instance, the application of `pattern` to a single Val will simply return the Val unaltered. Thus application of tree transformations where they make no sense will be equivalent to not applying a tree transformation at all. This will result in the amount of tree transformations that were specified by the generative plan being reduced, which is not ideal.

Possible Solutions

It is clear that the application of tree transformations needs to be carefully controlled due to the sheer variety of possible outputs and inputs they can have, and

that the depth-based control mechanism in our design is not sufficient to handle this task. Possible solutions to this problem are to either disallow tree transformations that increase or decrease tree height, force such tree transformations to be applied so they will result in a tree with uniform height, or change the way we apply tree transformations so that they are not fixed to a given depth, but rather fixed to a given type of input tree. The latter solution is the most general, but probably the one that is most difficult to design. It would require that the way we generate prefix trees is further restricted by the oriented tree it is to be applied to, so that each slice in a given branch only points to subtrees which the tree transformation is suited for. This could be a function that inspected each branch in the prefix tree and adjusted them so that it they only refer to the correct type of subtree. The type of tree that a tree transformation could be applied to could be part of an expanded tree transformation structure with added metadata.

9.1.3 Generative Principle

The generative principle in this design was to generate a series of prefix trees where each addresses a given dimension in the music, and sequentially apply these to a starting tree that contained the raw musical information that the piece should be based upon. In order to ensure cohesiveness, each prefix tree was generated based on a *generative plan*, as well as the oriented tree that the prefix tree was to be applied to. To examine how well this systems worked to achieve the goals of our specification, we will look at each goal individually.

Aesthetic Parameterisation

One of the requirements for this design was that the generation process should be parameterised, to be able to control of the generated output and enforce cohesion and variety. The parameters for how a prefix tree was to be generated were contained within a generative plan, which contained general aesthetic choices and a pool of possible tree transformations that prefix tree could contain. The idea of a generative plan as a sort of meta-model worked well to ensure cohesiveness while maintaining a generalized generative pipeline. Possible improvements could be to add more parameters to it, to further control the generated output. Generative plans were in charge of ensuring cohesion both within and across dimensions. For now they are written by hand by the user, but in the future one could implement a function which generates a set of plans based on a set of input parameters. In other words, one would need to design a meta-plan to oversee the whole generation process.

Generating Random Structures

A fundamental requirement of this design was for the system to generate novel musical structures. As prefix trees represent the type of structures we were after, the most logical way to handle generating such structures was thus to generate

prefix trees. The design for this generation process was based on randomising the slice transformations and tree transformations of a "skeletal prefix tree", which was given as part of the generative plan. The skeletal prefix tree denoted the shape of the prefix tree, which allowed us to control the complexity of the generated structure from the generative plan. The process of random-generating slice transformation was constrained so that each slice key only addressed subtrees at a certain depth given in the generative plan.

Generation of Slice Transformations From a functional point of view, our method for generating slice transformations worked well. In the future, it could be improved by giving the generative plan more explicit control of how the slice transformations should turn out. In our design, each slice transformation in a prefix tree addressed the nodes from 0 to a random number within the range of children in the oriented tree for the depth in question. A better solution would be to define a pool of possible indexes in the generative plan, and having the algorithm choose from this, while keeping them within the range of possibility given by the oriented tree it is to be applied to. A natural example of such a pool of indexes would be the numbers defined by $n^2 - 1$ where n is a natural number, i.e. 0, 1, 3, 7, etc. This will result a sectional division in 2,4,8 and so on, which is common to western music.

Prefix Tree Skeletons Separating out the shape of the prefix tree from its content was advantageous in order to control complexity. Since we have not controlled the generation of slice transformations so that they do not overlap, this was especially useful. The amount of leaves in the prefix tree determines the amount of transformations. We noticed that more than three leaves for a prefix tree lead to a high chance of overly complex music. If there is much overlap here, some sections can for instance have very complex and fast rhythms while others are very slow. This problem could be solved by a variable in the generative plan that controls the probability that a randomly generated slice transformation is unique from the ones that are already generated.

Generating Music with Shared Patterns

A major goal of our design was for the system to generate music with shared patterns across multiple hierarchical levels and dimensions. Our system handled this well by generating a dedicated prefix tree for the shared material in each dimension, which gave clear and explicit control from the generative plan.

Co-variaton of Patterns With our random prefix tree generation algorithm, the shared material of each dimension were treated independently. In most music, these might co-variate, for instance a certain rhythm might tend to go well with a certain chord change and a certain playing pattern. One could define a parameter for the amount of co-variation of patterns in the generative plan. This would be

the probability that a given slice transformation is equal to the corresponding one in the previous plan's list of slice transformations. What is important to consider here is that the more dimensions are separated out, the more one likely has to enforce co-variation of patterns to avoid chaotic music. One could also envision a system where the function that generates prefix trees also has access to previous prefix trees in the generative pipeline, so that it would know which tree transformations are applied where, which would be useful information to inform which tree transformations should be applied on the basis of which have already been applied.

9.2 On Implementation

The design was implemented in Haskell. The requirements specified for the implementation were mostly met:

- Every impure I/O-action is handled by the outer layer defined in `main`, which made the generative core purely functional.
- The system seemed quite stable. While this was not exhaustively tested, its final iteration was run for around 100 times and never crashed for any seeds, which is about the level of stability we required of our system.
- Its performance was also adequate: The generation seemed to happen instantly and could be streamed lazily with no stutters or timing issues.
- The fact that the code was clearly divided into modules, with a great degree of polymorphism, made it quite extensible.

In other words, the implementation can mostly be deemed a success. Still, an important requirement for our implementation was that it should follow the programming guidelines given in section 2.7 in the background chapter on Haskell. It is in adhering to these that there are some areas where there is room for improvement.

9.2.1 Polymorphism

The implementation is mostly polymorphic. This is especially true in case of the tree structures, since all general tree operations are located in `Trees.hs` and all operations specialised to music are in `MusicTree.hs`. A possible improvement is to create a generalized tree type that both `OrientedTree` and `Prefix Tree` are specializations of. This would allow the use of generalized functions for operations that apply to both, such as a height function. In our implementation, this was however not necessary, as the only common functionality needed was enumeration. Here, each type of tree required a different form of enumeration, which is why the best solution was to create an `Enumerable` typeclass that each tree was made an instance of.

9.2.2 Functions

The majority of the functions implemented adhered to the programming guidelines. They were defined with type signatures, and were kept "short and sweet". We strove to use well-known higher order functions such as map and fold, instead of recursion and list comprehension, which can be harder to read. An exception to this was when working with trees. Although we did implement both trees as an instance of `Functor`, which allowed the extensive use of `fmap`, most other operations relied on recursion. In a future implementation, one could have implemented the trees as instances of typeclasses like `Foldable` and `Traversible`, to allow greater use of generalized access functions. The greatest sin towards the programming guidelines was in our use of partial functions.

Partial Functions

While we did our best to avoid partial functions, i.e. functions that only work for a certain subset of its legally defined input, there are certain functions in the implementation that are in fact partial:

atDepth The function `atDepth` is used to create slice transformations. This is a partial function that should ideally return a value wrapped in `Maybe`: If one tries to call it with a depth that is larger than the length of the slice, the function will crash. The reason we accepted this was that if `atDepth` returned a function of type `Slice -> Maybe Slice`, this would also change the type of our prefix tree to contain slice transformations of this type. This was deemed too cumbersome, because it would mean that one would need to use the `Maybe` monad to compose these functions. While such a solution might be more technically correct, it would involve a lot of complexity to solve a problem which is actually quite trivial. Our solution, which was to make sure the function which calls `atDepth` only does it on slices with an appropriate length, works just as well in practice.

getDepth The function `getDepth` is used to get the depth at which a given slice transformation will transform its input slice. This is done by applying it to an infinite default slice, consisting only of the choice `All`. The index of the first (and only) `Some` will correspond to the depth of the slice transformation. This works as long as a slice transformation only modifies one level, which in our case it does, but will give the wrong answer otherwise. Thus the function only works for certain input, but this is not captured in the type signature. This could be solved by embedding in the type of slice transformations that they only can transform one level. Here one could make slice transformations a datatype which also contains the depth of of the slice expressed as an Integer. This would remove the need for `getDepth` entirely.

childRange Another function that is problematic due to its partial nature, is `childRange`. Its job is to get the range of indexes of the children for the nodes at a given depth. This is done with `childrenAtDepth`, which returns `Nothing` if one tries to get the children at the depth of the leaves of the tree. Due to the problem with applying tree transformations at certain depths discussed in 9.1.2, this will sometimes happen. This is solved in `childRange` by simply returning `[0]` in the case of `childrenAtDepth` returning Nothing. Since `pattern` is implemented so that it does not fail even when called on `Val`'s, this is an acceptable solution to avoid failure, but it is highly suboptimal, and should be improved in future iterations. This is a problem created in the design that is solved in the implementation. It would not occur if a better control mechanism for the application of tree transformations was designed.

9.2.3 Typeclasses

Typeclasses were mostly used to great effect in this implementation. The `State` monad was used for random number generation, the `Maybe` monad was used for handling failure, and our own typeclass `Enumerable` was implemented for things that can be enumerated. As previously mentioned, we could have used typeclasses more extensively when dealing with trees. The implementation of `enumerate'` is a stateful computation, so the `State` monad could have been used here. Additionally we could possibly have avoided recursion had we implemented a general way to traverse the trees as an instance of `Traversible` or a similar typeclass.

9.3 On using Haskell vs. an Imperative Language

In order to do this project, I had to learn Haskell. Many people say that learning Haskell is like learning to program all over again, and there might be some truth to that statement. So the question remains, how was it different to design and implement a system with our requirements in Haskell versus how it would have been with an imperative language like Python? The comparison will be informed by a similar project done in Python, as well as a decent amount of experience with imperative languages in general. First, we will discuss how using Haskell shaped our work on generating music structures, and then we will discuss the general of experience of software development in Haskell.

9.3.1 Generating Structures

It is clear that the system was designed with a functional mindset. The generative process required the application of tree transformations, i.e. functions, in a structured way. This was captured in the generative prefix tree, where both the nodes and the leaves were *functions*. This was possible because in functional programming, functions *are values*. Thus, having functions as the elements of trees, lists, or any other data structure, is just as natural as any other type of value, like an

integer. While it is technically possible to implement our design for a generative prefix tree in an imperative language, this form of thinking is not encouraged. Without a functional mindset, it is not a solution one would think of, and if one did it would most likely be so difficult to implement that one would design another type of solution instead.

In general, we observe that the ability to treat functions like values is extensively used in our design: One of the most fundamental functions of our system `applyTT` takes a tree transfomation, a function, as its argument. Similarly, functions of type `Slice -> Slice` are passed around and composed very easily.

Still, Haskell's lack of imperativeness also gives rises to some challenges. This is mostly when one quickly wants to get an idea down into code. In a language like Python, it is very easy to translate the functionality that one desires into code. In Haskell, however, a different way of thinking is required, which, at least for a beginner, makes it more time-consuming to create the functionality one desires. One could argue that there is an inherent imperativity to composition, as it clearly can be considered a process of first do this then that etc. Haskell still allows us to use monads to write imperative code, but it needs to be more well thought out and it's pipeline more explicit. In the long run this is an advantage, but one does lose the free, creative, instant results that an imperative language might offer. This could however improve the more one gets used to Haskell and functional programming in general. A paper on harmony generation in Haskell by Magalhaes, speaks to the advantages of music generation in Haskell: "Functional Composition without explicit mutable state provides a composable way for defining a pipeline of independent processes, making the global algorithm easier to understand and adapt." [22]. This ended up ringing through in the case of the generative pipeline in this project as well, where the generative pipeline was defined in a very clear and composable way.

9.3.2 Haskell Software Development in General

The process of developing software in Haskell has overall been a positive experience. The fact that we are forced to work with pure functions means that it is relatively easy to manage the complexity of the system. Every function can be tested in isolation, and if it works as desired for all input, that is all that is required. This also makes refactoring easier, because changing the way a function works internally will not affect anything else. The disadvantage to such a pure system is that one cannot print out the result of computation within a function to see its result. This makes it harder to debug longer and more complex functions. This however is no disadvantage if you follow the principle of "short and sweet" and divide larger s into smaller sub-function, which is better anyway. This makes testing larger functions a matter of testing its components.

Strict Type Checking and the Compiler In general developing with Haskell requires one to work *with* the compiler, to check for type errors etc. The strictness of the type system proved to be really useful when developing, as code that actually compiled very rarely failed. The only case where the compiler comes short is when there are problems with pattern matching. An incomplete pattern can not be caught in the compiler, it will only be discovered at run time. The same is true for partial functions. Some care must therefore be taken to make sure that all possible inputs are handled with pattern matching. This is especially hard when recursion is involved.

9.4 Future Work

An important aspect of the requirements for both the design and implementation was *extensibility*. We wanted to design and implement a software which could be used as a base for future work. As our discussion has revealed, the work done in this project on generating music structures lays down a solid foundation for future work.

9.4.1 Generate better structures

One possible avenue for future work is to survey music theory sources and transfer theories on good musical structure to an improved system for prefix tree generation. Our project displayed how one can use prefix trees to generate music with random, novel structures. The question now remains how one can further constrain this generation process to generate music with more pop-friendly, sensible structures. Possible questions are how to include a beginning and end, how to improve *transitions between sections*, how to more clearly enforce the idea of a measure, and how to parameterise the generation process so that we can further control structural style.

9.4.2 Integrating Other Generative Algorithms

In our generative example, we "faked" random generation of chords, rhythms and playing patterns by randomly selecting from a list of pre-written candidates. In the future one could try and integrate existing generative algorithms for this purpose, or write new ones. As the generative principle outline here takes care of the musical structure, i.e. form, a greater variety of randomly generated musical raw material, i.e. content, would be integral for our system to generate varied music.

9.4.3 Meta-Plan

In our design, the plans for each prefix tree needed to be specified by hand. A possible next step could be to create a system which contained a meta-plan, i.e a list of higher-level parameters such as "key/mood", "structural complexity", "pattern

co-variation", etc., and a way to convert such a meta-plan to a list of generative plans. This could serve as a generative interface for the type of music the user wants to create, or these parameters could simply be randomised.

9.4.4 Linear Tension Narrative

Most music has a sense of narrative to it, where *musical tension* ebbs and flows to build and release emotional responses in the listener. Per now, our system does not consider this explicitly. The musical material inserted by the first prefix tree, (chords in the case of our generative example), does outline the musical development of the piece. Still, the way the prefix trees are generated and applied will simply create structured variations on these chords, without considering *the order* in which the musical events will occur. The generative algorithm will essentially transform the given music as if it was a static tree phenomena. An avenue for future research could be to integrate a notion of time in the generative prefix trees.

Two ideas are presented here: The first is to make the generative process' final prefix tree modify the "tension" dimension of the piece. This would require a more abstract, scale-parameterised representation as we have previously mentioned in our discussion on primitive representation. We know that the given tension of a piece is highly bound to the notion of cadence (see nomenclature). Thus we could let the last prefix tree transform certain locations, such as the end of higher hierarchical levels, to either end in a cadence (i.e. release tension) or a half-cadence (i.e. build tension). Alternatively one could look into how Herremans et al. uses optimization techniques to constrain a given piece to a given tension profile [20].

9.4.5 Multiple voices

An important aspect of music, which our design does not cover, is *polyphony*, i.e. multiple musical voices occurring at the same time. The generative example *does* produce polyphonic music in the sense that there are chords, but the chords are directly inserted, and not stacked on top of each other. In other words, the generative system works by building music sequentially, but does not combine its inserted elements in parallel. This was a deliberate decision because parallel composition adds much complexity, as one would have to make sure each voice fits both sequentially and in relation to the other parallel voices. In order to do this, one will often first compose one voice as it unfolds in time, say a melody, and then adapt the other material, like chords, so that they fit the first voice.

In order for such a system to work, it is essential to keep track of what notes are playing at the same time. As mentioned in our discussion on prefix trees, their great expressibility come with the disadvantage that this is not trivial to determine. This is because a vertical group can contain horizontal groups, so that when viewed only in its group, a given node can appear to sound on its own, but when viewed in a greater context it is clear that the other horizontal group that is played

in parallel contains a note which happens to land on the same beat. This is illustrated in figure 6.1.

Possible methods A simple solution to this problem would be to only allow vertical groups to contain notes, so that we would be certain that the only way two notes could occur simultaneously is if their in the same vertical group. This is not a bad idea, and it is still possible to express all music with this approach. Still, if we for instance have a melody line playing over a set of chords, it would make much more sense to group these as two separate vertical voices. This would make it a lot easier to keep track of what is actually the melody and what are the chords. This would require a system for summing up durations along the nodes of each vertical groups and keep track of what notes will occur at the same time as a given note. One could envision a function which would take the path of a given node, as well as the tree the node is in, and then give out a list of all other nodes/notes that occur simultaneously.

9.4.6 Using Prefix Trees to Generate Structure in Other Fields

Due to our great emphasis on abstraction and polymorphism, we have come upon a generalized process for generating 2D tree structures with shared material. The two musical dimensions in our musical oriented could easily be replaced with two dimensions from another domain entirely. The same goes for the prefix tree, which essentially encodes the shared locations in this tree where it should be transformed. Thus one can easily envision that our generalized system can be modified to work on domains unrelated to music. Maybe the two dimensions in the oriented tree can be dimensions along a 2D plane, so that our prefix tree system can be used to generate two dimensional pictures with shared content across both different sections of the 2D-plane and across different colors? A similar argument can be made for any structure which exhibits a tree-like structure, and in which there is significant shared material by hierarchically unrelated nodes. Maybe the generation of text is a possible application area? Generative plans, the meta-model mechanism for ensuring cohesion when generating prefix trees could also be ported to the given domain in order to ensure cohesion. It is clear that this is a very interesting avenue for future work.

Chapter 10

Conclusion

The goal of this project was to design and implement a music generation system in Haskell. This system was to generate music with hierarchical structure and shared material across and within the levels of this hierarchy. Background research indicated that such a structure is inherent to most music, and that current music generations system are not adequately capable of generating music which exhibits it.

The prefix tree designed by Yan Han worked well to represent music with shared structure. In order to accommodate generation, they were modified in our design by changing the type of the leaves to *tree transformations*. While this increased expressiveness came with some challenges, it was also highly useful, as it allowed us to directly express how any group within an oriented tree should be transformed in multiple locations. The general music generation scheme we designed, was based on sequentially applying a set of these prefix trees each randomly generated to address one musical dimension. The meta-model structure *generative plan* worked well to ensure cohesion. The generative system was quite successful at generating music which exhibited the structure we were after.

Learning and working with Haskell did constitute a challenge, but had some major advantages. Haskell's functional features were integral to our design, since it depended on being able to treat functions like values; both the nodes and the leaves of the prefix trees were functions.

To our knowledge, the generative system designed in this project is entirely novel. It outlines a general technique for generating structures, and thus much effort was put on making the system as extensible as possible, to facilitate future work. As revealed in our discussion, there are many avenues for improving this system's music generation abilities. Since generation system is so generalized, it could also be applied in entirely different areas which also exhibit a similar tree-like structure with patterns of shared material. Possible candidates for such future work are figures and texts.

Bibliography

[1] Y. Han, N. Amin and N. Krishnaswami, 'Representing music with prefix trees,' in *Proceedings of the 7th ACM SIGPLAN International Workshop on Functional Art, Music, Modeling, and Design*, ser. FARM 2019, New York, NY, USA: Association for Computing Machinery, Aug. 2019, pp. 83–94, ISBN: 978-1-4503-6811-7. DOI: 10.1145/3331543.3342586. [Online]. Available: https://doi.org/10.1145/3331543.3342586 (visited on 02/02/2021).

[2] F. Carnovalini and A. Roda, 'Computational Creativity and Music Generation Systems: An Introduction to the State of the Art,' *Frontiers in Artificial Intelligence*, vol. 3, Apr. 2020. DOI: 10.3389/frai.2020.00014.

[3] L. Sun, L. Hu, G. Ren and Y. Yang, 'Musical Tension Associated With Violations of Hierarchical Structure,' English, *Frontiers in Human Neuroscience*, Sep. 2020, Place: Lausanne, Switzerland Publisher: Frontiers Research Foundation Section: Original Research ARTICLE. DOI: http://dx.doi.org/10.3389/fnhum.2020.578112. [Online]. Available: https://search.proquest.com/docview/2443956203/abstract/50B289C051E94DCFPQ/1 (visited on 25/01/2021).

[4] P. Hudak and D. Quick, *The Haskell School of Music: From Signals to Symphonies*, en, ISBN: 9781108241861 9781108416757 Publisher: Cambridge University Press, Sep. 2018. DOI: 10.1017/9781108241861. [Online]. Available: https://www.cambridge.org/core/books/haskell-school-of-music/6B377BCD40386E9D27EB93FC2F3B13FB (visited on 12/01/2021).

[5] M. Lipovača, *Learn You a Haskell for Great Good!* en. Jun. 2010. [Online]. Available: http://learnyouahaskell.com/ (visited on 13/04/2021).

[6] A. Serrano Mena, *Practical Haskell: A Real World Guide to Programming*, eng, 2nd ed. Berkeley, CA: Apress LP, Apress, 2019, ISBN: 978-1-4842-4479-1.

[7] A. Bhargava, *Functors, Applicatives, And Monads In Pictures - adit.io*. [Online]. Available: https://adit.io/posts/2013-04-17-functors,_applicatives,_and_monads_in_pictures.html (visited on 10/06/2021).

[8] *All About Monads - HaskellWiki*. [Online]. Available: https://wiki.haskell.org/All_About_Monads (visited on 11/06/2021).

[9] *Programming guidelines - HaskellWiki*. [Online]. Available: https://wiki.haskell.org/Programming_guidelines (visited on 10/04/2021).

[10] *Haskell Style Guide*. [Online]. Available: https://www.cse.unsw.edu.au/~cs3161/15s2/StyleGuide.html (visited on 10/04/2021).

[11] P. Hudak, J. Hughes, S. Peyton Jones and P. Wadler, 'A history of haskell: Being lazy with class,' in *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, ser. HOPL III, San Diego, California: Association for Computing Machinery, 2007, 12–1–12–55, ISBN: 9781595937667. DOI: 10.1145/1238844.1238856. [Online]. Available: https://doi.org/10.1145/1238844.1238856.

[12] S. Dai, H. Zhang and R. B. Dannenberg, 'Automatic Analysis and Influence of Hierarchical Structure on Melody, Rhythm and Harmony in Popular Music,' *arXiv:2010.07518 [cs, eess]*, Oct. 2020, arXiv: 2010.07518. [Online]. Available: http://arxiv.org/abs/2010.07518 (visited on 25/01/2021).

[13] M. J. D. Martins, F. P. S. Fischmeister, B. Gingras, R. Bianco, E. Puig-Waldmueller, A. Villringer, W. T. Fitch and R. Beisteiner, 'Recursive music elucidates neural mechanisms supporting the generation and detection of melodic hierarchies,' en, *Brain Structure and Function*, vol. 225, no. 7, pp. 1997–2015, Sep. 2020, ISSN: 1863-2661. DOI: 10.1007/s00429-020-02105-7. [Online]. Available: https://doi.org/10.1007/s00429-020-02105-7 (visited on 25/01/2021).

[14] D. Herremans, C.-H. Chuan and E. Chew, 'A Functional Taxonomy of Music Generation Systems,' *ACM Computing Surveys*, vol. 50, no. 5, pp. 1–30, Nov. 2017, arXiv: 1812.04186, ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3108242. [Online]. Available: http://arxiv.org/abs/1812.04186 (visited on 06/01/2021).

[15] J.-P. Briot and F. Pachet, 'Deep learning for music generation: Challenges and directions,' en, *Neural Computing and Applications*, vol. 32, no. 4, pp. 981–993, Feb. 2020, ISSN: 1433-3058. DOI: 10.1007/s00521-018-3813-6. [Online]. Available: https://doi.org/10.1007/s00521-018-3813-6 (visited on 25/01/2021).

[16] C. Anderson, A. Eigenfeldt and P. Pasquier, 'The generative electronic dance music algorithmic system (gedmas),' *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 9, no. 1, Nov. 2013. [Online]. Available: https://ojs.aaai.org/index.php/AIIDE/article/view/12649.

[17] F. Lerdahl, *A generative theory of tonal music*, eng, Cambridge, Mass, 1996.

[18] K. H. " Masatoshi Hamanaka and S. Tojo", "'implementing ä generative theory of tonal music",' "*Journal of New Music Research*", vol. "35", 2006. DOI: "10.1080/09298210701563238".

[19] groves. ryan, 'Towards the generation of melodic structure,' 2016.

[20] D. Herremans and E. Chew, 'Morpheus: Generating structured music with constrained patterns and tension,' *IEEE Transactions on Affective Computing*, vol. 10, no. 4, pp. 510–523, Oct. 2019, ISSN: 2371-9850. DOI: 10.1109/taffc.2017.2737984. [Online]. Available: http://dx.doi.org/10.1109/TAFFC.2017.2737984.

[21] F. Hedelin, 'Formalising Form: An Alternative Approach To Algorithmic Composition,' en, *Organised Sound*, vol. 13, no. 3, pp. 249–257, Dec. 2008, Publisher: Cambridge University Press, ISSN: 1469-8153, 1355-7718. DOI: 10.1017/S1355771808000344. [Online]. Available: https://www.cambridge.org/core/journals/organised-sound/article/formalising-form-an-alternative-approach-to-algorithmic-composition/BCC43C87136157A059C0499A5432D23B (visited on 25/01/2021).

[22] J. P. Magalhães and H. V. Koops, 'Functional generation of harmony and melody,' en, in *Proceedings of the 2nd ACM SIGPLAN international workshop on Functional art, music, modeling & design - FARM '14*, Gothenburg, Sweden: ACM Press, 2014, pp. 11–21, ISBN: 978-1-4503-3039-8. DOI: 10.1145/2633638.2633645. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2633638.2633645 (visited on 12/01/2021).

[23] A. Latham, *The Oxford Companion to Music*. Oxford University Press, 2011.

[24] A. Blatter, *Revisiting Music Theory: Basic Principles*. Routledge, 2016.

Music Theory Glossary

This glossary gives a short introduction to the most fundamental terms in music theory. The source for this glossary is the *Oxford Companion to Music* [23]. For further details, the reader is encouraged to look at the online version, which can be found here. Another useful introductory resource is *Revisiting Music Theory* [24].

Pitch A fundamental dimension of a musical sound. A pitch is heard as either high or low, depending on its frequency, where pitches with higher frequency are perceived as higher, and vice versa.

Note Represents the pitch and duration of a musical sound.

Interval The difference between two notes in terms of pitch. Acoustically speaking an interval is defined in terms of frequency ratios, but more commonly it is defined as the amount of steps, or *semitones*, between two notes on the chromatic scale.

Octave An octave is the most consonant interval, which gives the impression of duplicating a note at a higher pitch.

Scale A scale consists of all the notes used or usable in the music of a particular period, culture or work, sorted by ascending or descending pitch. The basic function of a scale is to define and regulate the pitches in a performance or a composition.

Chromatic Scale A scale which divides the octave into 12 notes with equal intervals, corresponding to the twelve keys in an octave on the piano keyboard.

Diatonic Scale A diatonic scale is a set of seven notes, in which the pitch intervals add up to an octave. It is considered the basic scale of western art music. The two most common diatonic scales are the major and minor scales. a given diatonic scale is defined by it's intervals, and may begin on any root note. For instance C major and D major contain the same intervals, but starts on different notes.

Chord Two or more notes played together.

Triad A chord consisting of a root and two other notes. If the interval between the two first notes is 3 semitones and the interval between the second and third are 2 semitones, then it is a major chord. If it the intervals are 2 and 3 semitones respectively, it is a minor chord.

Cadence A melodic or harmonic motion that is assosiated with the ending of a musical group, such as a section, phrase, or an entire composition.

Full Cadence Gives a sense of harmonic or melodic closure by arrival in the root note or chord of the scale.

Half Cadence Gives a sense of harmonic or melodic tension by ending on the fifth scale degree or fifth chord of a given scale.

Consonance A quality to an interval or chord which in itself seems complete and stable.

Dissonance The opposite of consonance: A quality to an interval or chord which in itself gives a sense of tension and a need for resolution to consonance.

Scale Degree Indicates a note by its position in its scale sorted by ascending order. In diatonic scales each scale degree is associated with a given function. Two examples of these are the first scale degree, the root, which is called the tonic and is associated with stability, and the fifth scale degree, which is called the dominant and is associated with tension.

Broken Chord A chord where the notes are played individually, i.e. not at the same time.

Inversion An inverted melody is a mirrored version of its original form. Where the original melody ascends, the inverted melody descends and the other way around.

Retrograde A reversion of the original melody, i.e. read form right-to-left instead of left-to-right.

Erlend K Hoff

Functionally Generating Music with Prefix Trees

NTNU
Norwegian University of
Science and Technology