Representing Music with Prefix Trees

Yan Han
University of Cambridge
United Kingdom
yh412@cam.ac.uk

Nada Amin
University of Cambridge
United Kingdom
na482@cam.ac.uk

Neel Krishnaswami
University of Cambridge
United Kingdom
nk480@cam.ac.uk

Abstract

Tonal music contains repeating or varying patterns that occur at various scales, exist at multiple locations, and embody diverse properties of musical notes. We define a language for representing music that expresses such patterns as musical transformations applied to multiple locations in a score. To concisely represent collections of patterns with shared structure, we organize them into prefix trees. We demonstrate the effectiveness of this approach by using it to recreate a complete piece of tonal music.

CCS Concepts • **Applied computing → Sound and music computing**.

Keywords Prefix tree, music representation, Haskell

ACM Reference Format:
Yan Han, Nada Amin, and Neel Krishnaswami. 2019. Representing Music with Prefix Trees. In *Proceedings of the 7th ACM SIGPLAN International Workshop on Functional Art, Music, Modeling, and Design (FARM '19), August 23, 2019, Berlin, Germany*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3331543.3342586

1 Introduction

Programming languages have been used in a variety of ways to represent and compose music. Many tools aim to maximize the space of music they are able to express; such languages typically provide score or signal level primitives, backed by a general purpose programming language. Others take more specialized approaches based on models such as constraint satisfaction [6] and formal grammars [5], all of which excel at representing different structural properties of music.

In this paper, we propose a language that excels at expressing shared patterns in tonal music. We define such patterns using two components: the locations they apply to, and the way they modify the musical events at those locations. We allow patterns to modify arbitrary dimensions of musical events (such as pitch, volume and duration), enabling a single note to be comprised of multiple, overlapping patterns. Since patterns often recur at diverse musical locations, we express them in such a way that similar locations can be compressed using a prefix tree. This allows us to organize a piece of music based on shared patterns between its sections, providing an expressive representation that complements its score.

Our approach is based primarily on analysis, rather than composition. We developed our model by using it to progressively recreate a single piece of music, Johannes Brahms's Waltz in A Flat Major Op. 39 No. 15. The functions and data structures we introduced were motivated by how well they could be used to model sections of the Waltz. Narrowing our focus to a single piece limited the generality of our techniques across different styles of music, but allowed us to examine all of the hierarchical levels within our chosen piece, from individual notes to periods comprising a quarter of the piece.

This paper will introduce our basic representation of music as events organized into a tree (section 2), describe some patterns exhibited by tonal music in this representation (section 3), specify the data structures we use to express these patterns (section 4), outline our recreation of the Waltz using these data structures (section 5), and discuss our approach in relation to other music representation languages (section 6).

2 Basic music representation

In this section, we introduce two fundamental components of our music representation: musical events, and a tree structure to organize them in a hierarchy. These are the final output types of the algorithms we describe throughout the paper.

We use Euterpea [2], a Haskell library for music composition, to play our music as MIDI. Our basic data structures are closely related to those used by Euterpea, with a few differences to account for our modeling requirements.

2.1 Musical events

2.1.1 Basic types

We can build a playable musical event with three basic pieces of data: absolute pitch, duration and volume. We use Euterpea's definitions for all three:

- An *absolute pitch* is an integer corresponding to MIDI note number. For example, 60 represents C4.
```
type AbsPitch = Int
```

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
FARM '19, August 23, 2019, Berlin, Germany
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6811-7/19/08…$15.00
https://doi.org/10.1145/3331543.3342586

- A *duration* is a positive fraction representing a number of beats. For example, 1/4 represents a quarter note.
  ```
  type Dur = Rational
  ```
- A *volume* is an integer from 0 (silent) to 127 (loudest) corresponding to MIDI note velocity.
  ```
  type Volume = Int
  ```

These three pieces of data suffice to represent notes for playback. However, we found it useful to replace `AbsPitch` with a more abstract type to represent additional tonal information.

2.1.2 Tonal types

We will define a collection of types that build up to the `ScalePitch`, a representation of an absolute pitch in terms of an underlying musical scale.

A `PC` (stands for "pitch class") represents an absolute pitch modulo 12. 0 represents C, 1 represents C♯/D♭, and so on. It does not distinguish between enharmonic pitch classes, and is implemented as a `newtype` wrapper around an absolute pitch.

```
newtype PC = PitchClass AbsPitch
```

A `Scale` is a nonempty set of pitch classes, where one pitch class is distinguished as the root. It can be represented as a root pitch class, and a list of distinct offsets that are added to the root to form the other pitch classes in the scale. We implement it in Haskell as an algebraic data type that represents the product of a `PC` and `[PC]`.

```
data Scale = Scale
  { _root :: PC
  , _offsets :: [PC]
  }
```

As an example, the E major scale is constructed as follows:

```
eMajor = Scale 4 [2, 4, 5, 7, 9, 11]
```

A `ScalePitch` is our tonal representation of an `AbsPitch`. It consists of a `Scale`, an octave, and a scale degree.

```
type Octave = Int
type Degree = Int

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

To represent the note A2 (pitch class A, octave 2), we could construct a `ScalePitch` as follows. This representation contains the additional information that our pitch belongs to the E major scale; it is the fourth scale degree of it. Note that scale degree is zero-indexed, so the fourth scale degree is represented by the integer 3.

```
a2 :: ScalePitch
a2 = ScalePitch eMajor 2 3
```

We provide a function that converts a `ScalePitch` to the pitch it represents:

```
getPitch :: ScalePitch -> AbsPitch

getPitch a2 == 45
```

2.1.3 Defining an event

We define a musical event using a `ScalePitch`, duration and volume. A volume of 0 means that the event is a rest; otherwise, it represents a note.

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

2.2 Hierarchical structure

Now that we can represent musical events, we need a way to organize them into a hierarchical structure. We will give an intuitive overview of a certain type of hierarchical structure, and define a data type that represents it.

Lerdahl and Jackendoff define hierarchical structure as an "organization composed of discrete elements or regions related in such a way that one element or region subsumes or contains other elements or regions [4]". They describe multiple instances of hierarchical structure in music, one of which is grouping structure. As seen in Figure 1, grouping structure is the recursive partitioning of a piece into contiguous groups of events. This roughly corresponds to our perception of a piece as composed of nested sections such as periods, phrases and measures.

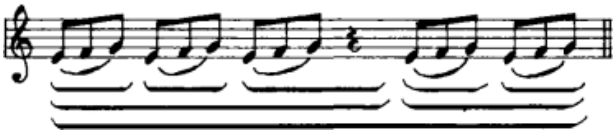


Figure 1. An example of grouping structure [4]

We extend the the notion of grouping structure by adding an orientation (horizontal or vertical) to each group. This allows us to capture common musical relationships such as the division of a piano piece into parallel left hand and right hand parts, and the partitioning of a chord into parallel notes. Figure 2 shows an oriented grouping structure on a snippet of the Waltz. It displays a single horizontal group containing four vertical groups, which each contain two notes.

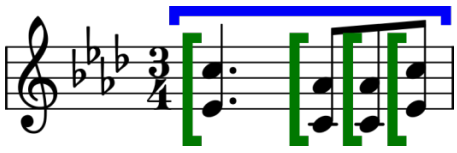


Figure 2. Grouping structure with orientation

This structure can be implemented in Haskell as a polymorphic tree whose branches have arbitrary sizes and are

labeled with their orientation. It is straightforwardly converted to Euterpea's tree structure, which recursively composes pairs of events in series or in parallel.

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

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

Using this data structure, we can represent Figure 2 as a tree of musical events. Figure 3 provides a visual representation of this tree.

```
exampleTree :: OrientedTree Event
exampleTree =
  Group H [
    Group V [  -- Chord 0
      Val c5', -- Voice 0
      Val ef4' -- Voice 1
    ],
    Group V [ Val af4, Val c4 ], -- Chord 1
    Group V [ Val af4, Val c4 ], -- Chord 2
    Group V [ Val c5, Val ef4 ] -- Chord 3
  ]
```

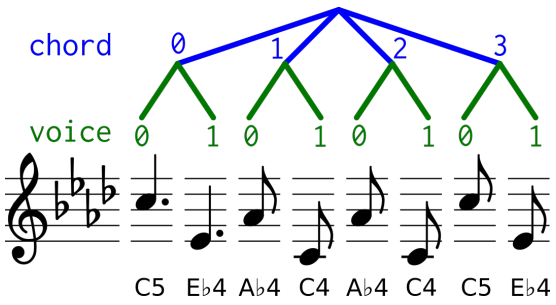


Figure 3. Figure 2 as a tree

3 Shared structure in music

We have defined music in terms of two types: a multidimensional type representing a single note, and a hierarchical tree structure representing nested groups of notes. We now consider some patterns in music represented in this format, which may allow a piece to be more concisely expressed than its naive transcription into a tree.

3.1 Shared hierarchical structure

Music often features repetition across multiple hierarchical levels. At the highest level, pieces are often structured as variations on a few sections. For example, a piece in binary form might be described as AA'BB': a section (A), a variation on that section (A'), a new section (B), and a variation on the new section (B'). Similar patterns occur at lower levels; a sequence of four measures might feature one measure repeated twice with minor changes, and a single measure might contain repeated instances of a few motifs (see Figure 4 for examples of both). In general, music features significant repetition, and the units of repetition frequently follow some grouping structure.

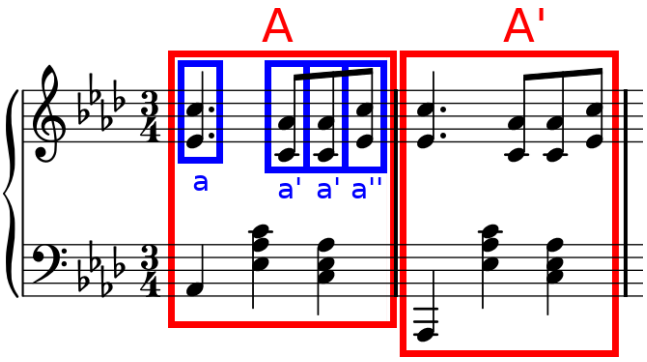


Figure 4. Repeated patterns at two different structural levels

3.2 Shared dimensional structure

To represent a note, we decompose it into multiple dimensions: duration, volume, scale, octave, and scale degree. The relationship between these dimensions varies considerably within a single piece of music. Figure 5 demonstrates this by displaying three dimensions of a four-measure section of the Waltz. The first two measures are identical in all three dimensions, and are completely distinct from the third measure. The final measure can be seen as a variation of the first two, since it differs from them in the pitch dimension only. This example shows that variations in musical patterns can be reduced to variations in their constituent dimensions. Therefore, it it important to able to flexibly define these dimensions.

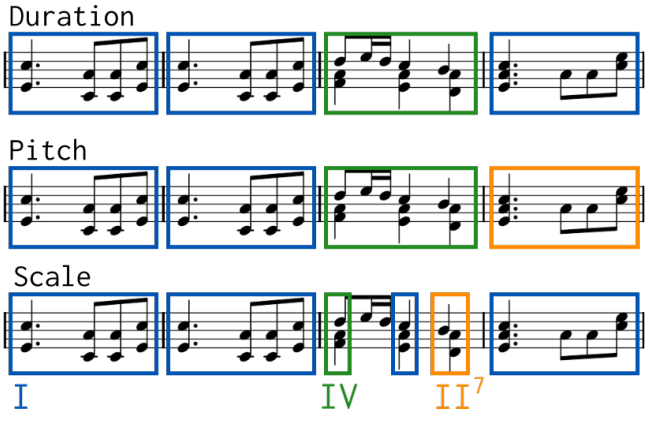


Figure 5. The duration, pitch and scale dimensions of a four-measure section of music

4 Modeling shared structure

This section describes a collection of data structures and operations that allow us to concisely represent musical trees with repeated hierarchical and dimensional structure. We will define a type representing multiple locations in a tree (section 4.1), a type representing a modification of events at multiple locations in a tree (section 4.2), and a way to organize these types into a prefix tree (section 4.3).

4.1 Representing locations in a tree

We can uniquely identify each leaf in a tree with a list of the indices of the branches taken to reach it. In the tree below, the leaf containing `z` is indexed by the integer list `[3, 0]`. We call this representation of a leaf's location a *path*.

```
exampleTree :: OrientedTree Event
exampleTree =
  Group H [
      Group V [Val x, Val x]
    , Group V [Val x, Val x]
    , Group V [Val x, Val x]
    , Group V [Val z, Val x]
  ]
```

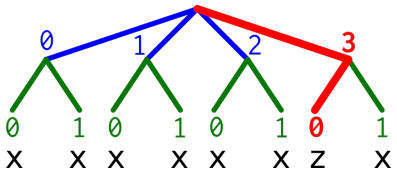


Figure 6. A single path in a tree

Sometimes it is convenient to represent many paths as one value. Consider the following four overlapping paths on a tree, and their corresponding representation in Haskell:

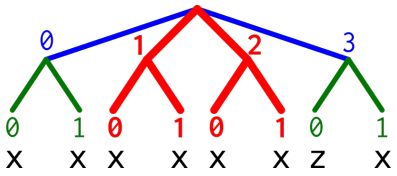


Figure 7. Multiple paths in a tree

```
fourPaths :: [[Int]]
fourPaths = [[1, 0], [1, 1], [2, 0], [2, 1]]
```

This collection of paths can also be summarized as a sequence of choices: take either branch 1 or 2, then take either branch 0 or 1. We can represent each choice as a list of integers, and the complete collection of paths as a list of choices. Thus, we can represent the list of paths above as follows:

```
twoChoices :: [[Int]]
twoChoices = [[1, 2], [0, 1]]
```

This collection of paths can be described another way: take either branch 1 or 2, then take any branch. It is sometimes useful to specify all branches at a level as a choice, without knowing precisely how many there are. Therefore, instead of using a raw list of integers to represent a choice of branches, we use a sum type:

```
data Choice = Some [Int] -- One or more branches
            | All        -- All branches
```

This allows us to rewrite `twoChoices` as follows. It now chooses all branches at the second level, regardless of the shape of the tree (see figure 8).

```
twoChoices' :: [Choice]
twoChoices' = [Some [1, 2], All]
```

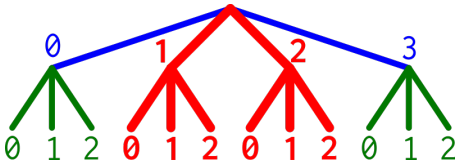


Figure 8. The paths selected by `twoChoices'` on a tree with a different shape

We call this representation of a collection of paths a *slice*.

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

4.2 Transforming multiple locations in a tree

A *tree modifier* represents a transformation on a set of musical events in a tree. It contains a slice to represent the locations of the events, and an event modifier to represent a transformation on those events. In other words, the slice represents hierarchical structure, and the event modifier represents dimensional structure.

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

We define a function that applies a `TreeModifier` to a tree of musical events:

```
applyModifier
  :: TreeModifier       -- Modifies part of a tree
  -> OrientedTree Event -- Input tree
  -> OrientedTree Event -- Output tree
```

The code below shows the application of a tree modifier to an oriented tree using `applyModifier`. The tree modifier represents the application of the function `f` to the leaves at the paths `[0, 0]` and `[1, 0]`. Figure 9 displays this transformation.

```
applyModifier
  (TreeModifier [Some [0, 1], Some [0]] f)
  (Group H [
    Group V [Val e, Val e],
    Group V [Val e, Val e]
  ])
  ==
  Group H [
    Group V [Val (f e), Val e],
    Group V [Val (f e), Val e]
  ]
```

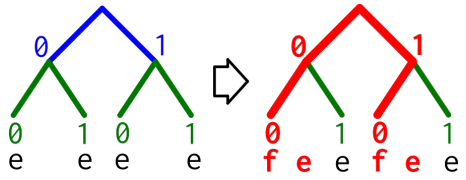


Figure 9. An application of a tree modifier

4.2.1 Constructing event modifiers

We define various ways to construct event modifiers (functions of type `Event -> Event`). For example, the following functions update a field inside an `Event`.

```
modifyDuration
  :: (Rational -> Rational) -> Event -> Event
modifyVolume :: (Int -> Int) -> Event -> Event
modifyScale :: (Scale -> Scale) -> Event -> Event
modifyOctave :: (Int -> Int) -> Event -> Event
modifyDegree :: (Int -> Int) -> Event -> Event

modifyDuration (+1)
  (Event 1 vol (ScalePitch scale octave degree))
== Event 2 vol (ScalePitch scale octave degree)

modifyOctave (+1)
  (Event dur vol (ScalePitch scale 4 degree))
== Event dur vol (ScalePitch scale 5 degree)
```

We also define functions that set a field to a constant value, instead of applying a function to it. For example, `setDuration` is a weaker version of `modifyDuration` that sets the duration of an event instead of applying a function to it.

```
setDuration :: Rational -> Event -> Event
setVolume :: Int -> Event -> Event
setScale :: Scale -> Event -> Event
setOctave :: Int -> Event -> Event
setDegree :: Int -> Event -> Event

setDuration 2
  (Event 1 vol (ScalePitch scale octave degree))
== Event 2 vol (ScalePitch scale octave degree)
```

These functions can be composed to create an event modifier that adjusts multiple fields of an `Event` at once.

```
( setDuration 1
. modifyVolume (+ 20)
. setScale (Scale 8 [4, 7])
$ Event dur vol (ScalePitch scale octave degree))
== Event 1
         (vol + 20)
         (ScalePitch (Scale 8 [4, 7]) octave degree)
```

4.2.2 Constructing slices

To build slices, we define ways to produce functions of type `Slice -> Slice`, which we call *slice modifiers*.

```
-- 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
```

Using these slice modifiers, we can change the collection of leaves that a slice selects:

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

We can also compose these functions to modify multiple choices in a slice. This method of constructing a slice is more verbose than simply using a list literal, but has properties that make it useful later on.

```
( atChords [0, 1]
. atVoices [2, 3]
$ [All, All])
== [Some [0, 1], Some [2, 3]]
```

4.2.3 Using tree modifiers

To demonstrate the usage of tree modifiers, we will construct the right hand part of a measure of the Waltz using them.

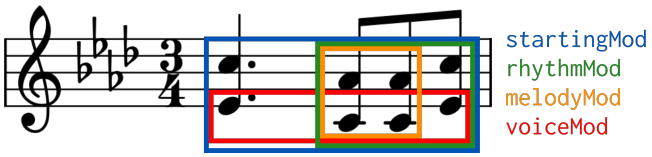


Figure 10. A measure of the Waltz as four tree modifiers

We first define a starting tree to apply tree modifiers to. The events in the starting tree are technically redundant, as they will be overwritten by the modifiers. They allow us to move all of the musical material into tree modifiers.

```
startingTree :: OrientedTree Event
startingTree =
  (Group H [
      Group V [Val dummy, Val dummy]
    , Group V [Val dummy, Val dummy]
    , Group V [Val dummy, Val dummy]
    , Group V [Val dummy, Val dummy]
  ])
```

The starting modifier sets all eight of the events in a tree to the same note: a dotted quarter note with volume 100 and

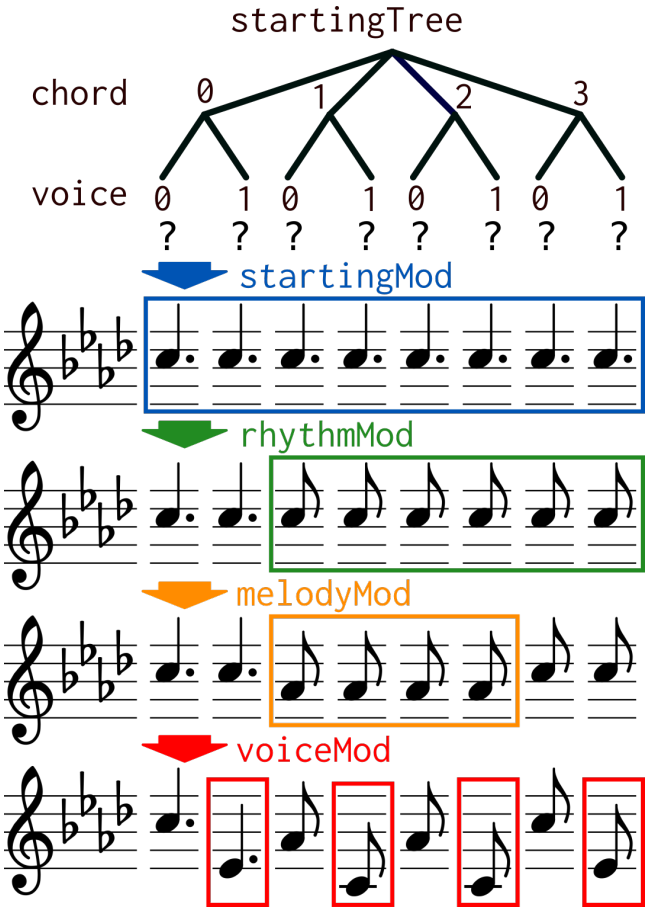


Figure 11. Reconstructing a measure with tree modifiers

pitch C5 (represented as degree 1 of octave 4 in the A flat major triad).

```
startingMod :: TreeModifier
startingMod = TreeModifier
  ( atChords [0, 1, 2, 3]
  . atVoices [0, 1] $ [All, All])
  ( setDuration (3 / 8)
  . setVolume 100
  . setScale (extractTriad 0 $ mkMajorScale 8)
  . setOctave 4
  . setDegree 1)
```

The second modifier replaces the dotted quarter notes in chords 1, 2 and 3 with eighth notes. It also reduces their volume slightly, since chords 1, 2, and 3 fall on weaker beats than chord 0.

```
rhythmMod :: TreeModifier
rhythmMod = TreeModifier
  ( atChords [1, 2, 3]
  . atVoices [0, 1] $ [All, All])
  ( setDuration (1 / 8)
  . modifyVolume (subtract 20))
```

The third modifier moves chords 1 and 2 down by one absolute scale degree. Since the scale underlying all of the events is the A♭ major triad, this moves each A♭ to the next highest E♭, each E♭ to the next highest C, and each C to the next highest A♭. This absolute degree transformation is performed by `modifyAbsDegree`, which is distinct from the previously defined `modifyDegree`.

```
melodyMod :: TreeModifier
melodyMod = TreeModifier
  ( atChords [1, 2]
  . atVoices [0, 1] $ [All, All])
  ( modifyAbsDegree (subtract 1))
```

The final modifier moves the bottom voice of each chord down by two absolute scale degrees. It also reduces their volume slightly, as they are of less melodic importance than the top notes.

```
voiceMod :: TreeModifier
voiceMod = TreeModifier
  ( atChords [0, 1, 2, 3]
  . atVoices [1] $ [All, All])
  ( modifyAbsDegree (subtract 2)
  . modifyVolume (subtract 20))
```

Applied in succession to the starting tree, these modifiers produce the complete section of music.

```
music :: OrientedTree Event
music =
    applyModifier voiceMod
  . applyModifier melodyMod
  . applyModifier rhythmMod
  . applyModifier startingMod
  $ startingTree
```

4.3 Compression using a prefix tree

We have demonstrated the usage of the tree modifier, a data structure that modifies a slice of a tree of musical events. This section outlines how multiple tree modifiers can be represented concisely using a prefix tree.

4.3.1 Prefix trees in Haskell

A *prefix tree* is a tree data structure that represents a map from keys to values. Each value is represented by a leaf, and each key is represented by a path from the root node to a leaf. This saves space when multiple keys have shared prefixes, as the shared material only needs to be specified once.

Figure 12 shows a prefix tree representing a map from strings to integers, where each key is constructed by concatenating all of the strings along a path. It maps "COT" to 1, "CAT" to 2, and "CANE" to 3.

We can express a prefix tree in Haskell using a tree data structure where the leaves are annotated with an additional value. The code below recreates the tree in Figure 12.

```
data PrefixTree k v =
    Leaf k v
  | Node k [PrefixTree k v]
```

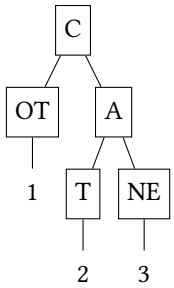
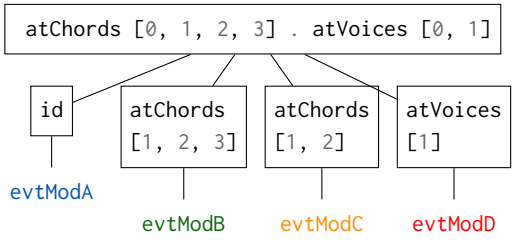


Figure 12. An example of a prefix tree

```
samplePrefixTree :: PrefixTree String Int
samplePrefixTree =
  Node "C" [
    Leaf "OT" 1,
    Node "A" [
      Leaf "T" 2,
      Leaf "NE" 3
    ]
  ]
```

4.3.2 Compressing tree modifiers using prefix trees

A musical prefix tree maps slices to event modifiers, producing a collection of tree modifiers. For example, the prefix tree below represents the four tree modifiers in section 4.2.3. Each slice is built by composing the slice modifiers along a path, and each event modifier is a leaf.




```
evtModA, evtModB, evtModC, evtModD :: Event -> Event

musicPrefixTree
  :: PrefixTree (Slice -> Slice) (Event -> Event)
musicPrefixTree = Node
  (atChords [0, 1, 2, 3] . atVoices [0, 1])
  [ Leaf id evtModA
  , Leaf (atChords [1, 2, 3]) evtModB
  , Leaf (atChords [1, 2]) evtModC
  , Leaf (atVoices [1]) evtModD
  ]
```

Consider the path from the root to the leftmost leaf. Its event modifier is the leaf `evtModA`, and its slice is equal to all of the slice modifiers along its path applied to the starting slice `[All, All]`. Therefore, its tree modifier is

```
TreeModifier
  ( id
  . (atChords [0, 1, 2, 3] . atVoices [0, 1])
  $ [All, All])
  evtModA
== TreeModifier [Some [0, 1, 2, 3], Some [0, 1]]
                evtModA
```

The second path from the left represents the tree modifier

```
TreeModifier
  ( atChords [1, 2, 3]
  . (atChords [0, 1, 2, 3] . atVoices [0, 1])
  $ [All, All])
  evtModB
== TreeModifier [Some [1, 2, 3], Some [0, 1]]
                evtModB
```

Similarly, the third and fourth paths represent the following tree modifiers:

```
TreeModifier [Some [1, 2], Some [0, 1]] evtModC
TreeModifier [Some [0, 1, 2, 3], Some [1]] evtModD
```

We have defined four `TreeModifier`s in one prefix tree by sharing some of the choices in their slices. Though this specific example is not much more concise than specifying all four modifiers separately, the principle it illustrates generalizes well to larger sections of music.

4.3.3 Converting a prefix tree to music

From here on, we will use the type synonym `MusicTree` to denote the type `PrefixTree (Slice -> Slice) (Event -> Event)`. We will outline the full process of converting a prefix tree (`MusicTree`) to a musical oriented tree (`OrientedTree Event`).

First, we obtain a list of tree modifiers from it using the function

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

This function follows the process outlined in the example: it collects a list of tree modifiers built from root-to-leaf paths from left to right. The code essentially performs an in-order tree traversal.

Next, we use the function `makeStartingTree` to generate a starting tree filled with dummy events. When there is insufficient information on the number of branches in a certain subtree, such as when all of the choices at that level are `All`, the function creates a single branch by default.

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

Finally, we apply the list of tree modifiers, in order, to the starting tree. This is implemented as a left fold:

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

5 Case study

We examine our recreation of Johannes Brahms's "Waltz in A Flat Major Op. 39 No. 15" using prefix trees. We split our analysis into lower level patterns relating to dimensional structure (section 5.2) and higher level patterns relating to hierarchical structure (section 5.3).

5.1 Hierarchical levels

Since our previous examples focused on small sections of music, we used trees with two levels: chord and voice. To model the complete Waltz, we expand our hierarchy to seven levels. They are as follows:

0. Hand (vertical): Right (index 0) or left (index 1) hand.
1. Period (horizontal): A group of phrases. The Waltz consists of four periods.
2. Phrase (horizontal): A group of measures. Contains 4-6 measures in the Waltz.
3. Measure (horizontal): Corresponds to a measure in the score. Usually contains 3-4 chords.
4. Chord (horizontal): Contains 1 or more voices.
5. Voice (vertical): Contains 1 or more notes.
6. Note (horizontal): A single note, along with optional passing tones.

This hierarchy requires adjustments to the existing code. We redefine the slice modifiers `atChord` and `atVoice` to modify the slice indices 4 and 5, respectively. We also define slice modifiers for each of the new levels: `atHand`, `atPeriod`, `atPhrase`, `atMeasure`, `atNote`. Finally, we use the default slice `[All, All, All, All, All, All, All]` when needed, matching the number of hierarchical levels.

5.2 Dimensional patterns

The flexibility of event modifiers allows us to specify events in many ways: we can construct a complete event using a single event modifier, build it incrementally by specifying each dimension separately, or anything in between.

The Waltz exhibits multiple trends in dimensional structure. First, scale can often be treated independently, since harmony is less granular than pitch. Second, duration and volume vary together, and can mostly be reduced to a handful of reusable, measure-sized trees. Finally, the pitch-related dimensions of octave and scale degree exhibited diverse patterns and required a variety of abstractions to model concisely. We will describe how each of these trends are expressed in the prefix tree.

5.2.1 Harmony

We found that the scale dimension of a `ScalePitch` frequently changed less often than the octave and scale degree components. As a result, it was often clearer to declare the scale of a section of music separately from the other dimensions. For example, figure 13 shows a five-measure section in the Waltz where each measure contains precisely one scale for both hands, while the other musical dimensions vary between both hands, and at the more granular chord and voice levels. We therefore separate the scale dimension of the section into its own tree.

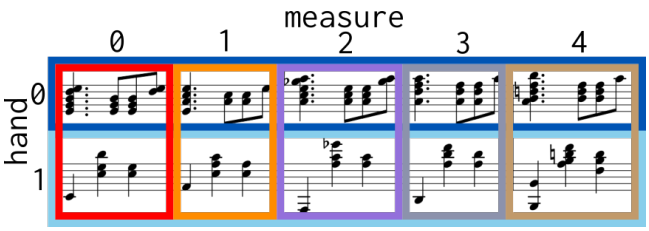


Figure 13. Harmony as an independent dimension

```
harmonyExample :: MusicTree
harmonyExample =
  Node (atPhrases [2] . atMeasures [0, 1, 2, 3, 4])
  [ Node (atHands [0, 1])
    [ Leaf (atMeasures [0])
        (setScale (extractSeventh 4 (mkMajorScale 8)))
    , Leaf (atMeasures [1])
        (setScale (extractTriad 0 (mkMajorScale 8)))
    , Leaf (atMeasures [2])
        (setScale (extractSeventh 4 (mkMajorScale 1)))
    , Leaf (atMeasures [3])
        (setScale (extractTriad 0 (mkMajorScale 1)))
    , Leaf (atMeasures [4])
        (setScale (extractSeventh 4 (mkMajorScale 3)))
    ]
  , Node (atHands [0])
    [ {- Right hand duration, volume, pitch -} ]
  , Node (atHands [1])
    [ {- Left hand duration, volume, pitch -} ]
  ]
```

5.2.2 Rhythm and meter

In the Waltz, duration and volume often vary together, and follow a small collection of simple, measure-sized patterns. We define these patterns as reusable prefix trees, allowing them to be conveniently inserted where necessary. This often allows us to factor out the volume and duration dimensions of music.

A common rhythmic pattern in the right hand is to make chord 0 a dotted quarter note, and chords 1, 2, and 3 eighth notes. The section below shows an example of this patter, which we define in code as `unevenRhythmMeter`.

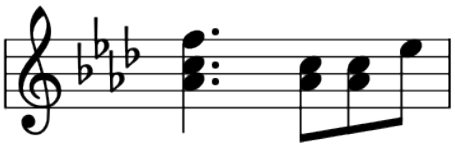


Figure 14. A right hand measure with the "uneven" rhythm

```
unevenRhythmMeter :: MusicTree
unevenRhythmMeter = Node id
  [ Leaf (atChords [0])
         (setDuration (3 / 8) . setVolume 100)
  , Leaf (atChords [1, 2, 3])
         (setDuration (1 / 8) . setVolume 90)
  ]
```

Another common rhythmic pattern in the piece consists of three quarter-note chords. This is used in every measure in the left hand. It is present less uniformly in the right hand, with slight variations due to shorter passing tones between chords. We define a two versions of this pattern with different volume settings, for the left and right hand.

```
evenRhythmMeterL :: MusicTree
evenRhythmMeterL = Leaf
  (atChords [0, 1, 2])
  (setDuration (1 / 4) . setVolume 60)

evenRhythmMeterR :: MusicTree
evenRhythmMeterR = Leaf
  (atChords [0, 1, 2])
  (setDuration (1 / 4) . setVolume 100)
```

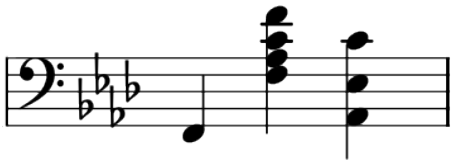


Figure 15. A left hand measure with an even rhythm

5.2.3 Voice stacks

A voice stack is a pitch-based pattern that is common enough in the Waltz to be abstracted into a helper function. It is a chord consisting of a root pitch, and a series of absolute degree offsets from that pitch.

For example, consider the A♭ major chord in figure 16. We can recreate its pitch content in code by setting all three voices to the pitch of voice 0 (the top voice), and decreasing the absolute degrees of voices 1 and 2.

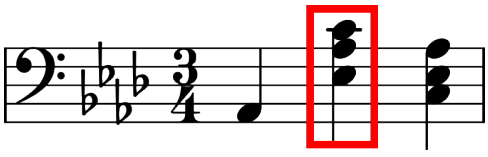


Figure 16. An example of a voice stack

```
exampleChord :: MusicTree
exampleChord = Node
  (atChords [1] . atVoices [0, 1, 2])
  [ Leaf id
      ( setScale (extractTriad 0 (mkMajorScale 8)
      . setOctave 3
      . setDegree 1)
  , Leaf (atVoices [1]) (modifyAbsDegree (subtract 1))
  , Leaf (atVoices [2]) (modifyAbsDegree (subtract 2))
  ]
```

To write this more concisely, we define a function that can generate the code above given a slice modifier, event modifier, and list of absolute degree offsets:

```
voiceStack
  :: (Slice -> Slice) -- Location of the voice stack
  -> (Event -> Event) -- Event modifier for voice 0
  -> [Int]            -- Absolute degree offsets
  -> MusicTree        -- Output tree
```

This allows `exampleChord` to be rewritten:

```
exampleChord' = voiceStack
  ( atChords [1])
  ( setScale (extractTriad 0 (mkMajorScale 8))
  . setOctave 3
  . setDegree 1)
  [-1, -2]
```

We use this pattern throughout the Waltz to simplify the creation of multi-voice chords. All of the left hand measures, along with a significant portion of the right hand measures, can be expressed as a series of voice stacks.

5.2.4 Voice leading

A few measures in the Waltz can be expressed in terms of voice leading between scales, using a collection of functions that modify all of the components of the `ScalePitch`.

```
roundDown :: Scale -> ScalePitch -> ScalePitch
stepDown :: Scale -> ScalePitch -> ScalePitch
roundUp :: Scale -> ScalePitch -> ScalePitch
stepUp :: Scale -> ScalePitch -> ScalePitch
```

The `roundDown` function takes a `Scale` and a `ScalePitch`. It changes the `ScalePitch` to use the new `Scale`, and sets its absolute pitch to the highest pitch in the new scale that is less than or equal to its old pitch. `stepDown` works similarly, except the new pitch is strictly lower than the old pitch. Finally, `roundUp` and `stepUp` are analogous to `roundDown` and `stepDown`, except they move the pitch up.

In the figure 17, the two passing notes belong to the A flat major scale, and the three chords each belong to different triads and sevenths taken from the A flat major scale. We model the measure as voice leading transformations on a single voice stack with some ad-hoc tree modifiers for the passing tones.

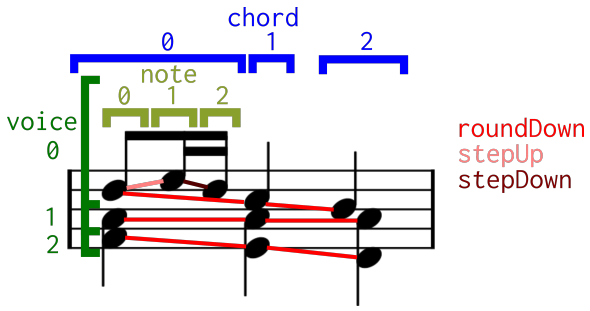


Figure 17. A measure modeled using voice leading functions

```
voiceLeadingMeasure :: MusicTree
voiceLeadingMeasure =
  Node (atHands [0])
  [ evenRhythmMeterR -- Rhythm and meter
  , voiceStack -- Base voice stack
      (atChords [0, 1, 2])
      ( modifyScale (extractTriad 3 (mkMajorScale 8)))
      . setOctave 5
      . setDegree 0)
      [-1, -2]
    -- Passing tones
  , Node (atChords [0] . atVoices [0])
    [ Leaf (atNotes [0]) (setDuration (1 / 8)
    , Leaf (atNotes [1, 2])
        ( setDuration (1 / 16)
        . modifyScalePitch (stepUp (mkMajorScale 8)))
    , Leaf (atNotes [2])
        (modifyScalePitch (stepDown (mkMajorScale 8)))
    ]
  , Leaf (atChords [1, 2]) -- First roundDown stack
        (modifyScalePitch (roundDown
          (extractTriad 0 (mkMajorScale 8))))
  , Leaf (atChords [2]) -- Second roundDown stack
        (modifyScalePitch (roundDown
          (extractSeventh 1 (mkMajorScale 8))))
  ]
```

5.3 Hierarchical patterns

Slice modifiers allow us to specify the choices in a slice in any order, separating the physical structure of an oriented tree from the organization of its prefix tree. At a high level, we structure the prefix tree to minimize repetition by recursively partitioning it into slices with shared structure. We provide examples of this technique at multiple scales, starting with measures and ending with the complete piece.

5.3.1 Organizing measures

Many instances of shared material in the Waltz manifest at the measure level. As a simple example, the two measures in figure 18 differ in precisely the first note in the left hand. To model this, we use a prefix tree that specifies both measures together, except for a single branch at chord 0 in the left hand.

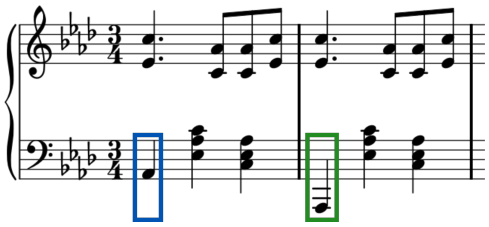


Figure 18. Two measures that differ slightly

```
phraseAMeasures01 :: MusicTree
phraseAMeasures01 =
  Node (atMeasures [0, 1]) -- Both measures
  [ Node (atHands [0]) [ ... ]
  , Node (atHands [1]) -- Left hand
    [ ...
    , Node (atChords [0]) -- Chord 0
      [ Leaf (atMeasures [0]) -- Measure 0
             (setOctave 2 . setDegree 0)
      , Leaf (atMeasures [1]) -- Measure 1
             (setOctave 1 . setDegree 0)
      ]
    ]
  ]
```

Shared patterns in the Waltz are usually more complex. Consider the complete first phrase, which consists of the first four measures in the piece. Figure 19 uses colors to show the high level organization of the phrase as nested slices in the prefix tree, corresponding to the code below.

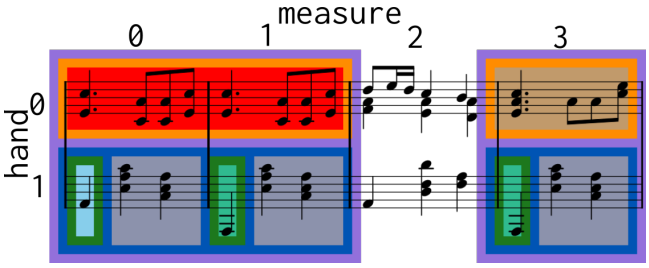


Figure 19. The high level slice structure of phrase A

We place measures [0, 1, 3] in their own branch because they have a significant amount of shared material. Most notably, they all use notes from the same scale: the root triad of the A♭ major scale. As we will see, their right and left hand parts also contain a significant amount of repeated structure.

We partition measures [0, 1, 3] into the right and left hand. The right hand measures have the same rhythm and meter, but measure 3 differs in pitch content from measures [0, 1]. This is modeled by specifying the rhythm for the entire slice, and then creating a branch between the two groups of measures. The measures in the left hand are identical except for the first note, which is modeled by a partition between measure 0 and measures [1, 3].

```
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]) [ ... ]
  ]
```

This outline of Phrase A exemplifies our organizational philosophy for the Waltz. At each subtree in the prefix tree, we partition it along the hierarchical dimension that creates the most similar groups. This allows us to group related sections together at every level, providing a conceptual organization of the piece that complements its physical organization as a score.

5.3.2 Organizing phrases

To outline the entire Waltz, we divide its material into three phrases, labeled A B and C. The piece then consists of four consecutive periods composed of variations on these phrases: AB AB'C AB''C A'B'''. The most complex phrase in the piece is Phrase B, which appears in four different variations. Using lowercase letters to denote measures, these variations are as follows:

- B (period 0): aabc
- B'(period 1): aabc'
- B''(period 2): aab'c''
- B'''(period 3): a'a'b''c'''

Figure 20 displays the variations of Phrase B with the first few levels of its prefix tree. As with smaller sections of the piece, we partition the tree to maximize the amount of shared structure within its slices. In this instance, we first partition into periods [0, 1, 2] and period 3, since period 3 shares little material with the others. We then partition into measure 2 and measures [0, 1, 3] due the rhythmic and harmonic differences in measure 2. We then partition by period again to specify the different variations of measures b and c.

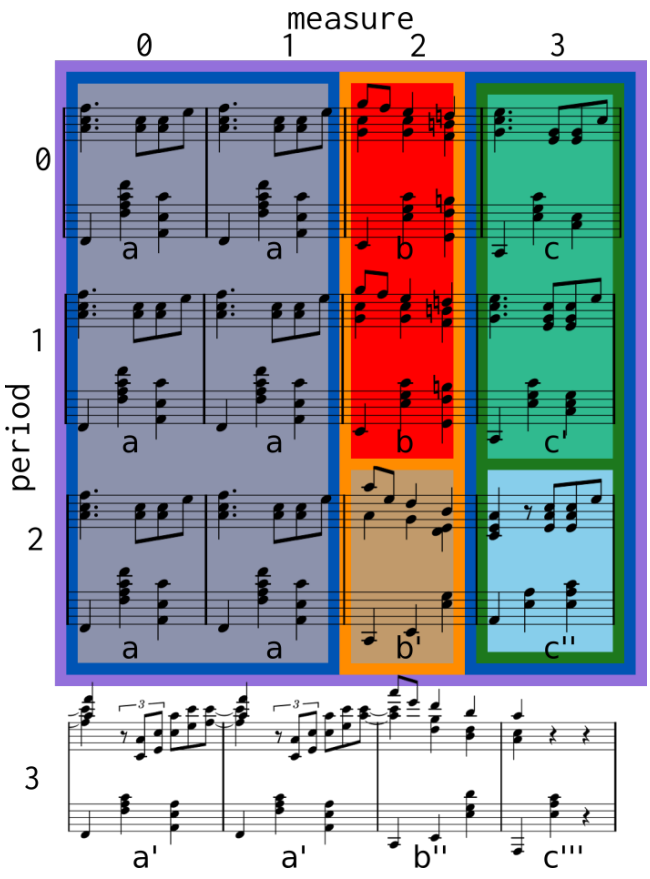


Figure 20. The slice structure of Phrase B across all periods

```
phraseBAllPeriods :: MusicTree
phraseBAllPeriods =
  Node
  ( atPeriods [0, 1, 2, 3]
  . atPhrases [1]
  . atMeasures [0, 1, 2, 3)
  [ Node (atPeriods [0, 1, 2])
    [ Node (atMeasures [2])
       [ Node (atPeriods [0, 1]) [ {- b -} ]
       , Node (atPeriods [2]) [ {- b' -}  ]
       ]
    , Node (atMeasures [0, 1, 3])
      [ Node (atMeasures [0, 1]) [ {- a -} ]
      , Node (atMeasures [3])
        [ Node (atPeriods [0, 1]) [ {- c, c' -} ]
        , Node (atPeriods [2]) [ {- c'' -} ]
        ]
      ]
    ]
  , Node (atPeriods [3]) [ {- a', b'', c''' -} ]
  ]
```

5.3.3 Organizing the piece

For completeness, the overall structure of the Waltz is given by the code below. It reflects our partitioning of the piece

into three phrases, each of which contains its own variations across multiple periods.

```
waltz :: MusicTree
waltz =
  Node (atPeriods [0, 1, 2, 3])
  [ Node (atPhrases [0]) [ {- Phrase A -} ]
  , Node (atPhrases [1]) [ {- Phrase B -} ]
  , Node (atPhrases [2] . atPeriods [1, 2])
         [ {- Phrase C -} ]
  ]
```

6 Discussion

The distinguishing feature of our language is its prefix tree structure, which simplifies the encoding of patterns that vary in multiple dimensions and occur at multiple locations. It allows music to be organized according to the amount of shared structure, which complements the sequential view offered by a musical score. In our experience, it was cleaner to represent the Waltz using a prefix tree than using naive Haskell functions and variables. This is likely due to the interrelated nature of the patterns in the Waltz, which often conflicts with the clean, delineated nature of functions and variables. The flexibility offered by the prefix tree is, to the best of our knowledge, unique among music representation languages. Nonetheless, many existing languages have features that make them more powerful or ergonomic, providing avenues of improvement for our language.

6.1 Related work

Many music representation languages offer graphic representations of their code, which aid the composition process by making the output music easier to visualize. Abjad [7], a music notation format implemented in Python, achieves this by directly mapping its internal music representation to an output score. OpenMusic [1] and PWGL [3] provide more programmatic visualizations of music; both frameworks use a display of interconnected boxes to illustrate the musical outputs of functions and chart the flow of data through them. Our language would benefit from either a score-based or code-based visualization. The prefix tree could be visually tied to areas in a score, in a manner similar to the visualizations used in this paper. It could also be displayed more literally, as a graphical tree that contains visual representations of event modifiers at the leaves. Either visualization would make the code much more transparent, improving the usability of the language.

The breadth and power of our musical abstractions could be improved, as they were guided by the recreation of a single piece. This issue could be addressed by progressively analyzing and composing more pieces using the language, and implementing features as they are needed. It could also be addressed by directly incorporating useful features in existing systems. A notable candidate for this is constraint satisfaction, which has been implemented in Strasheela [6]. This would allow pieces of music to be specified nondeterministically, which better models situations in composition where multiple musical patterns can be used to achieve the same goal. In general, it is worth exploring the interaction between our prefix tree structure and other models of music representation to see how they might augment each another.

6.2 Future work

In addition to the ergonomic and expressive enhancements mentioned in the previous section, the implementation of our language could be improved in many ways. Our should also be generalized and decoupled from the Waltz, as it currently assumes a fixed number of hierarchical levels and orientations. Additionally, it is difficult to verify that all dimensions of the notes in a prefix tree are initialized, and to predict how overlapping modifiers in different sections of the tree interact. This could be addressed by encoding the dimensions that each tree modifier affects in its type, though this would be tricky to implement concisely. Alternatively, changing the language to use a deep embedding would allow its terms to be directly inspected, allowing such checks to be implemented as run-time algorithms.

7 Conclusion

Each piece of music is a complex web of relationships that exhibits vastly different structure when viewed from different perspectives. In this paper, we have presented a single perspective of a single piece; we have recreated Brahms's Waltz in A Flat Major by deconstructing it into a collection of musical patterns, and organizing these patterns hierarchically into a prefix tree. We find that this format excels at representing shared and varied content in the piece, illuminating a unique conceptual organization of the Waltz.

References

[1] Jean Bresson, Carlos Agon, and Gérard Assayag. 2011. OpenMusic: Visual Programming Environment for Music Composition, Analysis and Research. In *Proceedings of the 19th ACM international conference on Multimedia - MM '11*. ACM Press, Scottsdale, Arizona, USA, 743.

[2] Paul Hudak and Donya Quick. 2018. *The Haskell School of Music: From Signals to Symphonies* (1st ed.). Cambridge University Press.

[3] Mikael Laurson, Mika Kuuskankare, and Vesa Norilo. 2009. An Overview of PWGL, a Visual Programming Environment for Music. *Computer Music Journal* 33, 1 (2009), 19–31.

[4] Fred Lerdahl and Ray Jackendoff. 1983. *A Generative Theory of Tonal Music*. MIT Press.

[5] Donya Quick and Paul Hudak. 2013. Grammar-based automated music composition in Haskell. In *Proceedings of the first ACM SIGPLAN workshop on Functional art, music, modeling & design - FARM '13*. ACM Press, Boston, Massachusetts, USA, 59.

[6] Anders Torsten. 2007. *Composing Music by Composing Rules: Design and Usage of a Generic Music Constraint System*. Ph.D. Dissertation. Queen's University Belfast.

[7] Bača Trevor, Josiah Wolf Oberholtzer, Jeffrey Treviño, and Víctor Adán. 2015. Abjad: An Open-source Software System for Formalized Score Control.. In *Proceedings of The First International Conference on Technologies for Music Notation and Representation.*