A Categorial Grammar for Music and Its Use in Automatic Melody Generation

Halley Young
University of Pennsylvania
United States
halleyy@seas.upenn.edu

Abstract

Like natural language, music can be described as being composed of various parts, which combine together to form a set-theoretic or logical entity. The conceptualized parts are more basic than the music seen on a page; they are the musical objects subject to music-theoretic analysis, and can be described using the language of functional programming and lambda calculus. This paper introduces the types of musical objects seen in tonal and modern music, as well as the combinators that allow them to combine to create other musical objects. We propose a method for automatically generating melodies by searching for combinations of musical objects which together produce a valid program corresponding to a melody or set of melodies.

CCS Concepts • **Theory of computation → Grammars and context-free languages**; • **Applied computing → Sound and music computing**;

Keywords Categorial grammar, music generation, automated music

ACM Reference format:
Halley Young. 2017. A Categorial Grammar for Music and Its Use in Automatic Melody Generation. In *Proceedings of 5th ACM SIGPLAN International Workshop on Functional Art, Music, Modeling and Design, Oxford, UK, September 9, 2017 (FARM'17),* 9 pages.
https://doi.org/10.1145/3122938.3122939

1 Introduction

Music theorists have developed rigorous systems for describing music in ways that go beyond the individual note values. Consider the beginning of Claude Debussy's piano prelude "The Girl with the Flaxen Hair", shown in figure 1. One can describe the "surface" of this short phrase of music - the actual notes seen on the page.[1] However, one can also discuss properties of the music that contribute to that surface: The phrase uses a pentatonic scale centered around the pitch-class D. The beginning contains two nearly repetitive motifs, with the first note being slightly shortened in the second repetition. Each of these motifs itself consists of a transposition and inversion operator applied to an even shorter snippet of music.[2] The phrase contains many [0.5, 0.25, 0.25] rhythmic patterns. Except for the end, it appears to be metrical.

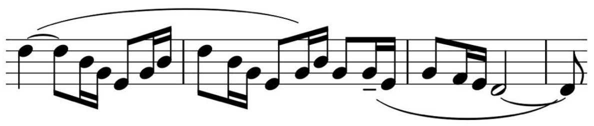

Figure 1. An excerpt from "The Girl with the Flaxen Hair"

All of the above properties are assumed to be important to understanding the piece as it is performed. However, simply listing the properties of a piece as above can make the properties described seem arbitrary and disconnected. This is patently not the case, as they clearly are all derived from the musical surface of the piece (the actual notes), and thus must all combine in some way to create those notes. In addition, some properties seem to be dependent on others - the prominence of the note D is related to the use of the D pentatonic scale. Unfortunately, without knowing the relationship between a property and other properties or the musical surface, it is difficult to understand how and why a given property contributes to the piece.

In this paper, we propose a formal method for characterizing musical properties based on the linguistic concept of categorial grammars. Categorial grammars describe how "objects" (computational expressions) of various types combine to form larger expressions. After a basic overview of the usage of categorial grammars in linguistics, we turn to their usage in music, showing that they can express the relationship between different musical properties, which can be thought of as computational objects. We show how larger pieces of music can be built from the interaction of musical objects with other objects representing shorter pieces of music, and discuss how this method relates to a model of the process of musical analysis.

While it is clearly possible to use categorial grammars for the purpose of musical analysis, this paper will focus on their application to music generation. Categorial grammars lend themselves to automatic generation of music. Combinators can be used to derive new musical objects, including melodies, from pre-existing musical objects. There are a set of valid musical objects and functions, and

[1] Here I am borrowing terminology from David Temperley, who distinguishes the "surface" of the music (what is seen on the page) from its underlying structure. [1]

[2] Transposition and inversion are two operators applied to musical motifs. Transposition moves the motif up or down on the musical staff, while inversion reverses the direction of the motif.

they can be put together in such a way as to result in an expression that is a melody. By automatically generating valid lambda expressions, we generate small musical pieces.

2 Music and Language

For at least 2500 years, music has been described as "sounding number," or something inherently mathematical [2]. At the same time, at least since the Renaissance, music has also been seen as something akin to language, with syntax and semantics [3]. Recently, developments in the field of linguistics have shown that analysis of music as natural language and as mathematical system are not orthogonal to each other, but rather complementary; natural language can be explored in precise ways by using areas of math as diverse as category theory and statistics [4] [5]. Therefore, it is logical that pursuing the "music as language" metaphor could be useful not only for philosophical purposes, but also for the purpose of computational analysis and generation of music - assuming that some of the same formalisms used to describe natural language also apply to music. In fact, certain linguistic formalisms, particularly context free grammars, have already been applied to music generation [6]. Categorial grammars are another linguistic formalism that have proved useful to the study of music.

3 Categorial Grammars in Language

In 1935, semanticist Kazimierz Ajdukiewicz introduced the concept of categorial grammars in natural language - a concept which became well-known in the linguistics community after the publishing of Richard Montague's paper "Universal Grammar" in 1970 [7]. Categorial grammars are different than the Chomsky-style generative grammars more well-known to computer scientists in that they do not only describe the syntax of a sentence, but also how the meanings of the individual words combine to create the meaning of the entire sentence. The meaning of the sentence is assumed to be a predicate logic statement. This predicate logic statement is arrived at through reductions of an expression in a typed lambda calculus. The typed lambda calculus only has two primitive types: entity (typically used to describe nouns such as a person or an item), and truth value. A full expression in the typed lambda calculus corresponding to a sentence, which reduces to a predicate logic statement, is created by concatenating the computational meanings of each word in the sentence. Each word has an associated type (shown here in Haskell notation).

For example, "Kim walked and fed the dog" is composed of the following words:

$$\begin{gathered}
\text{Kim} = k :: entity \\
\text{walked} = \lambda x[Walked(x)] :: entity \rightarrow truth \\
\text{and} = \lambda x, y, z[x(z) \wedge y(z)] :: (entity \rightarrow truth) \rightarrow (entity \rightarrow \\
truth) \rightarrow entity \rightarrow truth \\
\text{fed} = \lambda x, y[Fed(y, x)] :: entity \rightarrow entity \rightarrow truth \\
\text{the} = \lambda x[x] :: entity \rightarrow entity \\
\text{dog} = d :: entity
\end{gathered}$$

Putting it all together, we get

$$\begin{gathered}
lambda\ x, y, z[x(z) \wedge y(z)](\lambda x[Walked(x)]) \\
(\lambda x, y[Fed(y, x)](\lambda x[x](d)))(k) = Walked(k) \wedge Fed(k, d)
\end{gathered}$$

Together, these words express a logical premise (that Kim walked and fed the dog), which can be true or false. Alternatively, one can interpret the words in the sentence as well as the resulting expression set-theoretically - it describes a set of situations in which Kim walked and fed the dog. This model is technically a grammar, as there are proof mechanisms to show a certain collection of tokens is a valid sentence [8], one can assign parts of speech according to the types of the words, and it can be used to show how individual parts make up a greater whole. However, it is more commonly studied by semanticists than syntacticians, and is the foundation of much research in natural language meaning.

4 Categorial Grammars in Music

4.1 Previous Research

In his 2013 thesis, Wilding develops a categorial grammar for musical harmony [9]. According to Wilding, harmonic progressions can be described in terms of series of functions. He refers to one common function, which takes a chord and outputs that chord connected with the chord transposed up by 7 semitones, as *leftonto* (he considers dominant motion as leftward motion). For example, he shows how a D-G-C harmony can be described as

$$(\lambda x.\text{leftonto}(x))(\lambda x.\text{leftonto}(x))(0)$$

where 0 represents the pitch-class C. He goes on to provide a way of parsing tonal harmonies and describing them as repeated applications of *leftonto* and *rightonto* operations, as well as a few more basic operations.

4.2 Current Research

In this paper, we suggest a method for using categorial grammars to describe not just the harmony but also the surface of a piece of music. This is accomplished by drawing a close analogy between words and sentences and all types of musical objects and the musical surface. One could argue that individual measures of music should be given the same status in the categorial grammar as words, and the piece as a whole (or a musical phrase within the piece) should be given the same status as a sentence. However, such an explanation doesn't allow for the sort of combination of words into a meaningful whole that we see in Montague's grammar for natural language, and thus renders the use of categorial grammars fairly meaningless. To describe the relationship between two measures of music computationally (and thus to benefit from the use of categorial grammars), it is necessary to look beneath the surface of a piece of music at the musical objects from which it is composed.

5 Musical Objects

There are a rich variety of musical semantic objects, or properties that contribute to the identity of the music, that may be discussed when describing a piece of music. Some examples include the following:

- The basic parameters of music: duration (a decimal number), octave (an integer), pitch class (an integer), and timbre (which can be simply described as an enum of instruments)
- Product types: Examples include a pitch, which can be described as a tuple of (octave, pitch class). For musical analyses where pitch including octave is important, see [10].
- Lists of musical objects, which include

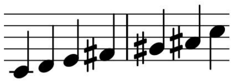

Figure 2. A Whole-Tone Scale

1. Unordered lists, such as an unordered list of pitch classes which is then permuted in various ways
2. Ordered lists, such as a rhythm, or ordered list of durations
3. Lists where every element of the list is unique, such as a set class
4. Lists where elements can be repeated, such as the pitch classes in a chord with unison intervals

- Predicates which may be true of musical objects (such as the predicate *is_symmetric_rhythm* which is true of a rhythm whose reverse is the same as itself)
- Functions which transform musical objects, such as a function that takes a melody and an interval and moves the melody up or down that interval
- Polymorphic functions, including
  1. Higher order functions, such as a function which takes a list, a function which transforms members of that list, and a number n, and transforms every n^{th} element of the list
  2. Non-higher order functions, such as functions which change the order or number of times a given element is seen

All of these objects are often discussed even in non-mathematical musical analyses. However, the use of some of the musical objects tends to be localized in music of a certain repertoire - for instance, set classes (unordered unique lists of pitch classes) are found more often in atonal music than in tonal music, while scale degrees (integers corresponding to indices of scales) are not commonly found in atonal music. Thus, the type of objects being used in an analysis or in a generation procedure are indicative of the style of the piece in question.

The musical objects can be part of a categorial grammar in that they combine together to form set-theoretic entities or characteristic functions. To build a whole-tone scale (a common structure in Impressionist music), we only need three musical objects: a starting pitch, a *transpose* function, which takes a pitch and an interval n and shifts the pitch up or down by n, *repeatApply* that computes

$$[x, f(x), f(f(x))...f^n(x)]$$

The lambda-expression below computes a whole-tone scale starting on the pitch C5, shown in figure 2.

$$(\lambda i, j.\text{repeatApply}(i, j, 6))((\lambda n, x.\text{transpose}(x, n)))(2))$$
$$(((\text{octave}, 5), (\text{pitch class}, 0)))$$

For a more involved example, consider three musical objects: the rhythm [0.5,0.5,1.0], the starting pitch (5,0) (where 5 is the octave, and 0 is the pitch class) and a contour [1,3,2] (a contour describes the shape of a melody - it consists of a list of numbers ranging from 1 to n, where $n <=$ the number of notes in the melody, such that the i^{th} pitch is lower than the k^{th} note if the i^{th} element of the contour is smaller than the k^{th} element of the contour, is equal to the k^{th} note if the i^{th} element of the contour is equal to the k^{th} element of the contour, and is higher than the k^{th} pitch if the i^{th} element of the contour is greater than the k^{th} element of the contour).[3] There is a set of objects in the musical universe which has all of these properties. Thus the semantics of

$$\lambda x, y, z.\text{combine}(x, y, z)$$
$$(\text{rhythm}, [0.5, 0.5, 1.0])(\text{start\_pit},(5, 0)),$$
$$(\text{contour}, [1, 3, 2])$$

is the set of all musical objects (which in this case have to be melodies) which possess those properties (some of which are shown in figure 3).

On first inspection, the above formula doesn't seem nearly as interesting as the corresponding categorial descriptions of linguistic phrases. The benefit of linguistic categorial grammars is that functions and objects fit together in interesting ways to produce a predicate-logic or set-theoretic entity, and in the above formula it appears as though the individual musical objects are unrelated to each other. In fact, this is not quite the case. The *combine* function has a specific meaning, which references the properties of the objects it is combining. In particular, this *combine* function works as follows:

1. Take the cartesian power of all possible pitches with $n = (length(contour) - 1)$.
2. For every member of the cartesian power set, prepend the pitch start_pitch to that member.
3. Filter out members of the derived set which do not have the specified contour - that is, members of the derived set (which are themselves lists) where it is not true for every n and i that the n^{th} element of the set is larger than the i^{th} element of the set if $contour[n] > contour[i]$, the n^{th} element of the set is smaller than the i^{th} element of the set if $contour[n] < contour[i]$, and the n^{th} element of the set equal to the i^{th} element of the set if $contour[n] == contour[i]$.
4. Every member of this derived set is a list. For each of these lists, pair up the elements of the list with the elements of the rhythm, such that you achieve a list $[(x[0], rhythm[0]), (x[1], rhythm[1])...(x[n], rhythm[n])]$ for every element x of the previously derived set. The result is the set containing melodies which fit the given constraints, namely having the correct pitch, contour, and rhythm.

What is interesting about this *combine* function is that it contains an implicit description of the semantics of each of the arguments in terms of their relationship to each other: The start pitch and the contour are related in that both describe properties of the pitches, and the rhythm is related in that the list product type of pitches and rhythm is a melody. If one was starting instead with a set of start pitches, a set of contours, and a set of rhythms, the same effect could be achieved by applying the above algorithm to the cartesian

[3]Thus, the contour [1,3,2] implies a list of 3 pitches such that the first is the lowest, the second is the highest, and the third is between the first two.

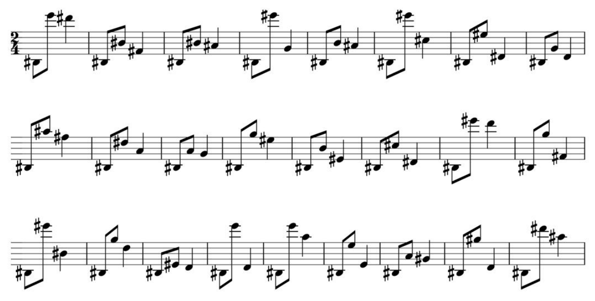

Figure 3. A subset of the set of melodies with rhythm [0.5,0.5,1.0], start_pitch (5,0), and contour [1,3,2]

product of all combinations of start pitches, contours, and rhythms being described.

It may seem tempting to, instead of giving procedures for a *combine* function, treat the *combine* function as simply a logical AND. In such a case, the *combine* function would take all possible melodies, and filter out ones which do not solve each of the properties - namely, having the given contour, the given starting pitch, and the given rhythm. In this case, the contour, pitch, and rhythm variables would have to be fed to functions which returned logical predicates, which is not a problem, and which in fact may be more similar to linguistic categorial methods than the previously proposed approach. The problem with this approach is that it would require considering all possible melodies and filtering out only those which satisfy the given conditions. As a thought experiment, consider only melodies made up of 4 beats of sixteenth notes, which only span the range of an octave. There are an enormous 12^{16} possible measures that satisfy these constraints. Now consider all melodies of any reasonable length (which may be conservatively estimated as 800 beats) of any rhythm, and without the range constraint. There are clearly far more possible melodies than there are atoms in the universe, and so it would not be feasible to produce an algorithm that operates only by filtering the set of all possible pieces of music using logical predicates. Whether this problem seems merely practical or "real" depends on one's thoughts about the philosophical implications of complexity theory, but it is safe to say that if one wants a computationally tractable grammar, one should use a combinator function that redefines the relationship between various musical objects in a computationally tractable way.

As there are many types of musical objects, there are many times of combinator functions. Not all combinator functions necessarily describe sets of melodies; one can imagine a combinator which takes a total duration and a number of durations, and returns the set of all rhythms with the correct total duration and number of individual durations. In addition, there are combinator functions which only describe a single musical object, rather than a set of musical objects. For instance, the combinator that takes a list of pitches and rhythms, and returns a list of notes, only returns a single melody.

Table 1. A Partial List of Combinator Functions

Argument Types	Return Type	Returns Set of Characteristic Values or a Single Value
Number of Notes	Interval Vector	Set
Interval Vector	Intervals	Set
Length	Contour	Set
Pitch, Duration	Note	Single
N-tuplets, Length	Rhythm	Single
Rhythm, Pitch List, Vertical Interval List	Melody	Single
Rhythm, Start Pitch, Horizontal Interval List, Vertical Intervals List	Melody	Single
Harmony, Rhythm	Melody	Set

6 Describing/Generating Larger Musical Objects

It is certainly possible to describe a 32-measure piece of music as consisting of a single contour, rhythm, and starting pitch. However, it is potentially more useful to describe musical objects which span large durations as being composed of several smaller objects that stand in a given relationship to each other. By considering functional musical objects, it is possible to describe a large musical object as consisting of various transformations of smaller objects. One example is the function *transpose*, which takes a melody and an interval and returns that melody transposed by that interval. Thus,

$$\lambda x, n.[x, \text{transpose}(x, n)](\lambda x, y, z.\text{combine}(x, y, z)$$
$$(\text{rhythm}, [0.5, 0.5, 1.0])(\text{start\_pit}, (5, 0)),$$
$$(\text{contour}, [1, 3, 2]))(3)$$

results in the set of melodies such that the first half is a member of the set of melodies with rhythm [0.5,0.5,1.0], starting pitch (5,0), and contour [1,3,2], and the second half is that first calculated melody transposed up 2 semitones. One can arbitrarily nest transforms, so

$$\lambda x, d.[x, \text{augment}(x, d)]$$
$$\lambda x, n.[x, \text{transpose}(x, n)](\lambda x, y, z.\text{combine}(x, y, z)$$
$$(\text{rhythm}, [0.5, 0.5, 1.0])(\text{start\_pit}, (5, 0)),$$
$$(\text{contour}, [1, 3, 2]))(3))(2.0)$$

returns the set of melodies described above where each element is then transposed by 3, and the result is augmented in rhythm by a factor of 2.0. The construction of large musical objects from small musical objects corresponds more directly to traditional notions of musical syntax.

7 Equivalent Descriptions and Normal Form

One property of any system which describes the structure of music as a program involving basic musical objects is that it may allow for two different descriptions of the same musical object. For example, one could derive the rhythm [0.25, 0.5, 1.0, 1.0, 0.5, 0.25] by applying

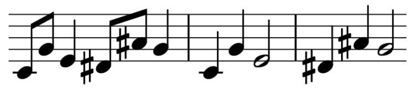

Figure 4. One element of the set $\lambda x, d.[x, \text{augment}(x, d)]$ $\lambda x, n.[x, \text{transpose}(x, n)](\lambda x, y, z.\text{combine}(x, y, z)$ $(\text{rhythm}, [0.5, 0.5, 1.0])(\text{start\_pit}, (5, 0)),$ $(\text{contour}, [1, 3, 2]))(3))(2.0)$

the "multiplicatively augment rhythm by 2" operation twice and then appending the reverse of the result to itself, or by looking at the list of rhythms which have total duration 3.5, have six notes, and are symmetric. In some instances this is a good thing; different musical analyses typically highlight different facets of the piece, and composers may be paying attention to one particular property of a musical object while considering whether to use that object. However, in other instances one can achieve a redundancy of descriptions which obscures the ideas behind the musical objects. For example, given the function *fourOf* which returns the subdominant of a chord and the function *fiveOf* which returns the dominant of a chord, the sequence of pitch-classes [[7,11,2], [0,4,7]], [0,4,7], [5,9,0]] can be generated by either of the following two expressions:

$$\lambda x.[x, \text{fourOf}(x)](\lambda y.[y, \text{fourOf}(y)]([7, 11, 2]))$$

$$\lambda x.[\text{fiveOf}(x), x](\lambda y.[\text{fiveOf}(y), y]([5, 9, 0]))$$

Similarly, the rhythm [[0.5,0.25], [1.0,0.5]] can be generated by taking either of the following two expressions:

$$\lambda x.[x, \text{augment}(x, 2)]([0.5, 0.25])$$

$$\lambda x.[\text{diminish}(x, 2), x]([1.0, 0.5])$$

One way of reducing the number of descriptions like these is to impose a rule that, when composing objects using an array of functions, the first function has to be the id function, or $\lambda x.x$. Similarly, one can say that transformation functions are not used except in list-application scenarios; there is no reason to describe the chord [0,4,7] as

$$\text{fiveOf}(\text{fiveOf}(\text{fiveOf}([3, 7, 11])))$$

unless it is being placed next to another chord.

8 Using Categorial Grammars to Generate Music

Categorial music analyses are generative in nature; one analyzes a piece by constructing lambda-expressions that describe that piece (and perhaps other, related pieces as well). It is natural to extend the use of categorial grammars to generate original music, as opposed to generating music modeled off a pre-existing piece. As categorial grammars describe music in terms of programs, generating music through categorial grammars requires generating valid programs, whose final output is a melody. In our implementation, the lambda-expression is treated as a graph corresponding roughly to an Abstract Syntax Tree, and is generated according to a graph-expansion process.

It is important to note that, unlike in other program synthesis (automatic generation of programs) tasks where every function is generated, in this task several basic musical objects were predefined [12]. Predefined functions included *combine* functions, musical transformation functions such as *transpose*, higher-order functions such as *repeatApply*, and musical predicates such as *is_symmetric_rhythm*, whose usages are described in more detail below. Predefining the basic musical objects to be used, as well as imposing limits on the number of transformations of a given type that could be performed in a single lambda expression, made the problem much more restricted and tractable than other program synthesis problems.

9 Implementation Details

The implementation of this categorial expression generator is done in Python. While Python is not typically considered a standard functional language, it does treat functions as first-class citizens, and includes anonymous functions. To fit Python to the desired purposes, metaprogramming and function attributes were used. Functions were enhanced with a publicly visible set of input and output arguments. Metaprogramming allowed for the following capabilities:

9.1 Mapping Over Lists

As seen above, the power of the categorial approach is to use functions to generate large objects from small objects, sometimes in a nested fashion. These hierarchical structures are described using nested lists. Thus, functions should automatically support mapping over nested lists of arguments. For instance, the function

$$f(a\_int, b\_int) = a\_int + b\_int,$$

when applied to

$$a = [a1, a2, a3], b = [b1, b2]$$
,

yields

$$[[a1 + b1, a1 + b2], [a2 + b1, a2 + b2], [a3 + b1, a3 + b2]].$$

Note that the mapped function is not commutative:

$$f([b1, b2], [a1, a2, a3])$$

yields

$$[[b1 + a1, b1 + a2, b1 + a3], [b2 + a1, b2 + a2, b2 + a3]].$$

In addition, the functions must map over set members. For instance, if the function f received a set (not a list) of three numbers $\{a1, a2, a3\}$, it should return the set that contains the results of $f(a1, b)$, $f(a2, b)$ and $f(a3, b)$. Mapping over lists programmatically does require knowing the type of the object, as, for example, a chord is a list and shouldn't necessarily be mapped. This is accomplished by supplying the name of the argument into the function.

9.2 Function Currying

Python doesn't automatically support function currying, but it is easy to implement with decorators. However, because of the approach to mapping described above (in particular that the order of the arguments affects the output), a somewhat unusual form of currying is needed - currying without regards to the order of the arguments. Thus, if the function $transpose$ takes arguments $melody$ and $interval$, calling

$$\text{transpose}([(\text{melody}, x])$$

should produce a function that is waiting for an interval, while calling

$$\text{transpose}([(\text{interval}, 3)])$$

should produce a function that is waiting for a melody.

10 The Type Graph

In order to build valid lambda expressions, the relationships between musical types must be known. This information can be described using a directed graph. In this graph, every musical type is a node, and edges represent ways of getting from one node to another. Edges can be broadly broken down into several types - edges for combine functions, edges for transformation functions, and edges for lifting single values to list values. When a given node $n1$ is one of (potentially several) types that are necessary for a combine function to produce type $n2$, an edge will be drawn from $n1$ to $n2$. Transformation functions, or functions where one of the inputs is the same type as the output, result in self-loop edges. There is another type of edge which describes the relationship between a value, and a list of that value. The graph on the final page of this paper (which does not show transformation edges or polymorphic functions) describes much of the type relations that produce the "form" of music. While there are many definitions of the semantics of music, one such definition could be rooted in the formal definitions and connections between different types of musical objects. The edges of such a graph define the style of the music it generates. Notably absent from this graph are the musical objects corresponding to Hindustani musical structures such as *ragas* and *talas*, and there is no edge corresponding to a transformation from first-species to second-species counterpoint. Thus, this system does not generate Hindustani music or music in the style of Palestrina. Categorial grammars are defined by the objects and transformations they permit, and one could imagine having many different categorial grammars which each produce a different style of music.

11 Building the Lambda Expression

Building the lambda expression requires traversing the type-graph for multi-paths (called that because they typically involve multiple source nodes moving towards the same target node) that leads to a melody. There are certain "base types" which have ranges of values - for instance, a scale-type is either diatonic, whole-tone, or octatonic, while a pitch class is any number from 0 to 11. A valid multi-path starts exclusively on these base types, and moves through functional edges to "higher" types; for instance, the types *scale_type*, and *key* combine to produce a *scale*, which in turn can combine with a *sign*, *degree*, and *chord_type* (all base types) to produce a chord, which can in turn be used. Along the way, one can apply transformations, but these are chosen to be applied only in the context of the higher-order function *applyAllTo*, where the first argument is a list whose first element is the function *id* and the following elements are transformations. In addition, a limitation is set on how many times $applyAllTo$ can be applied to a given argument, to make the search space smaller. Other polymorphic functions, which are classified as to whether they can only be used on list types, can also be applied; polymorphic functions being considered are ones that act on a list, such as those that apply a function only to certain elements of a list or those that change the order of a list, and those that apply a function to an element n times.

The relationship between two types $t1$ and $t2$ where $t1$ is a list of $t2$'s can be exploited. For instance, a rhythm is a list of durations. While there are many ways of generating a rhythm, one way is to repeatedly apply functions to a single duration — for instance,

$$\begin{aligned}\text{repeatApply}(f &= \text{augmentDuration}([(\text{augment\_by}, 2.0)]),\\ x &= (\text{dur}, 0.5), n = 4)\end{aligned}$$

will generate the list [0.5, 1.0, 2.0, 4.0] — equal to [0.5, 0.5*2, (0.5*2)*2, ((0.5*2)*2)*2]. However, there are several lists which should not be compiled in this way; for instance, a scale should not be generated from a list of pitch classes, but rather exclusively from the appropriate combine function.

Filters are also applied to certain expressions which are sets of possibilities. For instance, an expression which generates a set of possible interval vectors may be filtered with the *is_consonant* predicate (meaning none of the resulting intervals sound harsh). Filters are common in human-composed music; a composer might know they want one of many set classes, and select one that is inversionally symmetric, or whose pitches are all in the same diatonic key. In some cases, the filters return an empty list, meaning that no elements of the set fit the constraints described. When this happens, the lambda expression will not yield a result, and another expression must be generated.

One significant problem with the current approach is that often, two arguments should be in a certain relationship to each other; most commonly, that they should be the same length. For example, melodies are created from durations and pitches by pairing each duration[i] with the i^{th} pitch, so a list of durations and a list of pitches being passed to the *combine* function should both be the same length. There are several possible ways of dealing with this. The simplest approach is to modify the combine function to take the minimum of the number of pitches and durations. This approach is not very satisfactory, as lambda expressions can have meanings not obvious from their statement. Another option is to return an empty set whenever a combinator receives mismatching arguments, so the expression-generating system is forced to try a new expression. While this is not too expensive for small expressions, it can be extremely inefficient for larger expressions.

12 Results

12.1 Example Output

Below is one result of the lambda-expression generator:

```
mel = ( (lambda i1, j1: i1(j1))
((lambda i2, j2: i2(j2))(
applyAllTo , [id, augDimRepeatMelody,
addAppogiaturasMelody ,
chromaticInvertMelody] ) ,
(lambda i2, j2: i2(j2))
(combine10 ,
([("pcs_list", (lambda i4, j4: i4(j4))
(combine11 , ([("chord_list",
(lambda i6, j6: i6(j6))
((lambda i7, j7: i7(j7))
(applyAllTo ,
[ id , fourOf , fiveOf , ] ) ,
(lambda i7, j7: i7(j7))
(combine15 , ([("degree", -1 ),
("scale", (lambda i9, j9: i9(j9))
(combine17 , ([("scale_type", "diatonic"),
("pc", [6,11,5] ), ]) ) ),
("sign", 0 ),
("chord_type",
["triad","ninth", "seventh","eleventh"])
,])))),]))),
("rhythm", (lambda i4, j4: i4(j4))
(combine7 , ([("length", 2.0 ),
("n_length", 3),]))),
 ("octave",4),]))))[1]
writeScore(mel)
```

This program does the following: Chords are created by combining diatonic scales starting on different keys with chord types, namely triads, ninth, seventh, and eleventh chords. Each chord X is then made into a sequence of the chords X IV/X V/X (see [13] for a review of chord functions). The resulting chords are combined with a rhythmic figure with 3 notes and a total length of 2 beats. The resulting melody is manipulated in several ways, namely diminution with repetition, the addition of an appogiatura, and inversion. The resulting melody is then saved as a midi file. The output is shown as a musical score in figure 6 (rendered using the Musescore software). Below is a tree representation of this program's type structure. As you can see, the piece is "composed" of several different musical objects. As one would expect, the music has a given register, rhythm, and pitch content; each of these parameters was determined by other musical objects (in the case of rhythm, by the length and duration of each phrase; in the case of pitch, by the chords used, which were in turn determined by scale, scale degree, and sign).

12.2 Further Examples

For many more example outputs in midi format, please go to https://github.com/HalleyYoung/CategorialMusic.

13 Future Directions

The implementation presented here is very primitive. Only a few musical objects and ways of deriving and transforming them are proposed, and thus the style of the resulting music is rather narrow. In addition, the generated musical expressions are currently extremely simple — they can be described in lambda calculus without any recursive combinators. In the future, more interesting functions will be explored. In addition, programs can be written in a less cumbersome way than as lambda expressions, using variable assignment, without losing the clarity of semantics that lambda calculus affords. As mentioned above, there is currently no way of implementing dependencies of arguments upon each other, except by proceeding through trial and error through search space to find a good combination. Future versions should include ways of determining whether two arguments fit each other without repeatedly recreating entire expressions.

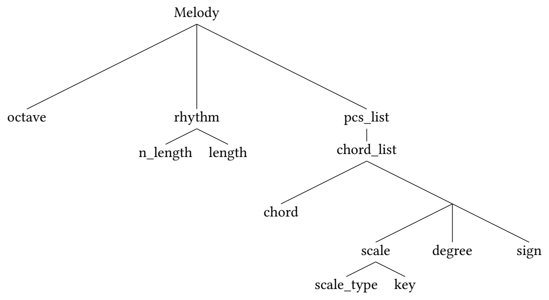

Figure 5. A diagram of the different types of the objects instantiated in the sample generated program; a node's children represent the types of all arguments used in the function that generated an object of the type represented by the node

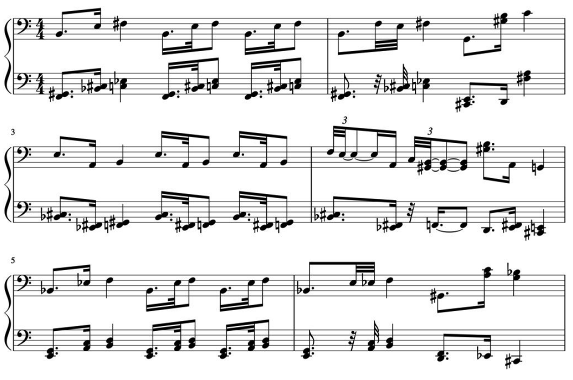

Figure 6. A Generated Score

14 Conclusions

There is no universal theory of music; there is no Western equivalent of the Hindustani raga, or an Indian equivalent of functional harmony. However, using categorial grammars, it's possible to construct a meta-theory of how musical objects of a given style interact with each other to produce the surface we hear as a piece of music or see as a musical score. When dealing with a very restricted set and usage of musical objects, one can even automatically compose objects to generate a melody. This is accomplished by treating the set of musical non-function types as nodes and musical functions as types in a multi-graph, and traversing the multi-graph down multiple paths that together arrive at the desired type (a list of lists of notes).

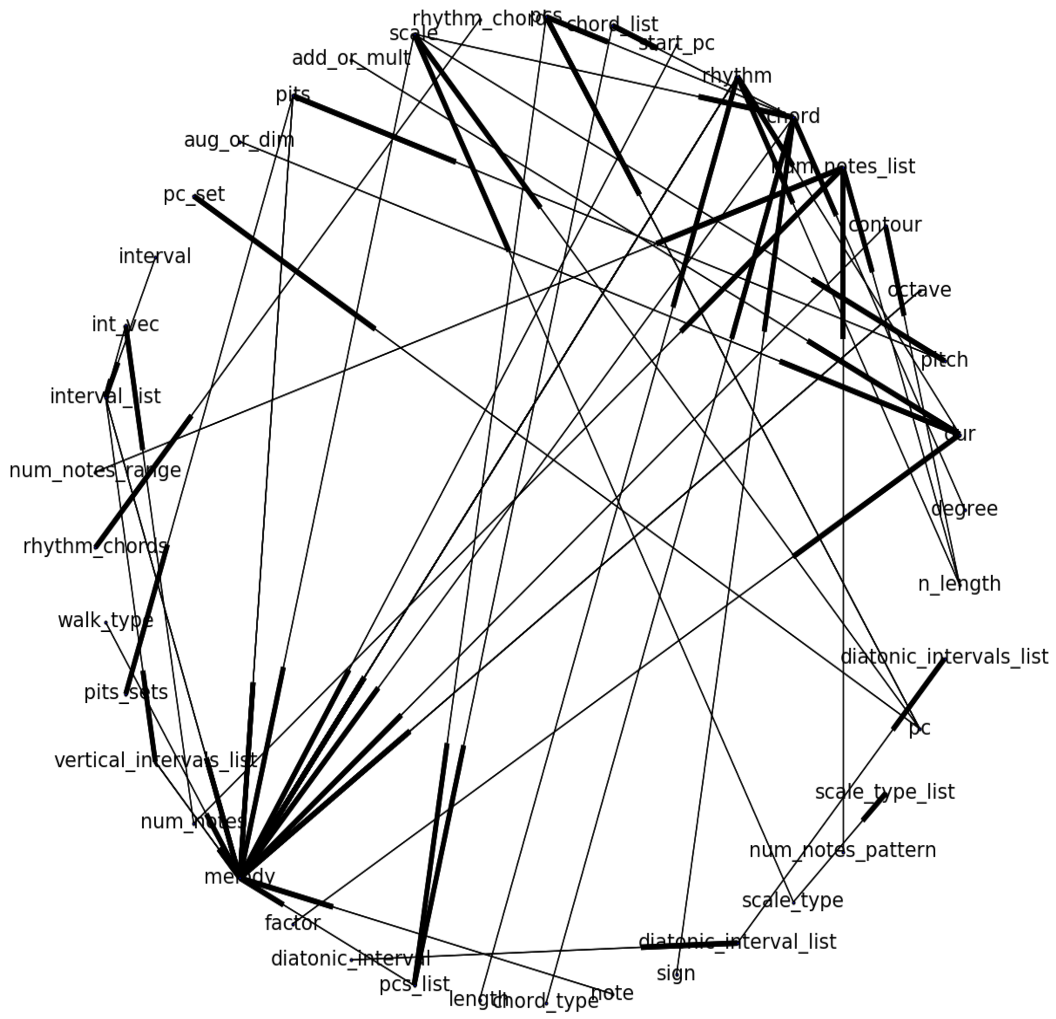

Figure 7. A Partial Graph of the Relationships between Types

Acknowledgements

This research was supported by the NSF under grant CCF-1302327.

References

[1] Temperly, David. (2006). *Music and Probability.* The MIT Press.

[2] Anderson, Gene. (1983). Pythagoras and the Origin of Music Theory. *Indiana Theory Review*, Vol. 6, No. 3.

[3] Bartel, Dietrich. (1997). *Musica Poetica.* University of Nebraska Press.

[4] Gillibert, Jean and Retore, Christian. (2014). Category Theory, Logic and Formal Linguistics: Some Connections, Old and New. *Journal of Applied Logic* Vol 12, No 1, pp 1-13.

[5] Doer, Rita. (1995). *A Lexical Semantic and Statistical Approach to Lexical Collocation Extraction for Natural Language Generation.* (Doctoral Dissertation). Retrieved from dl.acm.org

[6] Quick, Donya and Hudak, Paul. (2013). Grammar Based Automated Music Composition in Haskell. *Proceedings of the first ACM SIGPLAN workshop on Functional art, music, modeling and design*, pp 59-70.

[7] Montague, Richard. (1970). .Universal grammar. *Theoria 36*, 373-398. (reprinted in Thomason, 1974)

[8] Ganroth-Wilding, MT. (2013). *Harmonic Analysis of Music Using Combinatory Categorial Grammar.* (Doctoral thesis). Retrieved from era.lib.ed.ac.uk.

[9] Fowler, Timothy. (2016). *Lambek Categorial Grammars for Practical/ Parsing* (Doctoral Dissertation). Retrieved from cs.toronto.edu

[10] Nauert, Paul. (2003). Field Notes: A Study of Fixed-Pitch Formations. *Perspectives of New Music*, Vol. 41, No. 1.

[11] Brown, A. et al. (2009). Generation in Context: An Exploratory Method for Musical Enquiry. *2nd International Conference on Music Communication Science ICoMCS2.*

[12] Osera and Zdancewic. (2016). Type- and Example-Driven Program Synthesis. *Symposium on Trends in Functional Programming.*

[13] Laitz, Steven, and Bartlette, Christopher. (2009). *Graduate Review of Tonal Theory: A Recasting of Common-Practice Harmony, Form, and Counterpoint.* Oxford University Press.